Apuntes de
Estructuras de Datos y Algoritmos

(Segunda edición, versión 3, 2020)

Javier Campos

Departamento de Informática e Ingeniería de Sistemas

Universidad de Zaragoza

Este documento está sujeto a una licencia de uso Creative Commons. No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

Tabla de contenidos

Tabla de contenidos ... iii

Prólogo de la primera edición ... v

Prólogo de la segunda edición ... vii

Tema I: Programación con Tipos Abstractos de Datos ... 1

- Lección 1: Tipos Abstractos de Datos (TAD) ... 3
- Lección 2: Especificación de TAD ... 17
- Lección 3: Implementación de TAD ... 25
- Lección 4: TAD genéricos ... 35
- Lección 5: TAD fundamentales ... 47

Tema II: Tipos de datos lineales ... 53

- Lección 6: El TAD pila genérica. Definición e implementación estática ... 55
- Lección 7: Datos puntero y estructuras dinámicas de datos ... 61
- Lección 8: Implementación dinámica de pilas ... 65
- Lección 9: El TAD cola genérica ... 75
- Lección 10: El TAD diccionario. Implementación con listas enlazadas ordenadas ... 87

Tema III: Tipos de datos arborescentes ... 97

- Lección 11: Introducción a los árboles ... 99
- Lección 12: Árboles binarios ... 103
- Lección 13: Árboles binarios de búsqueda ... 115
- Lección 14: Árboles AVL ... 127
- Lección 15: Árboles *n*–arios ... 139
- Lección 16: Árboles *n*–arios de búsqueda ... 149
- Lección 17: Árboles lexicográficos (o *tries*) ... 159
- Lección 18: Colas con prioridad, montículos y el *heapsort* ... 167

Tema IV: Tipos de datos funcionales ... 175

- Lección 19: El TAD tabla y las tablas dispersas (*hash*) ... 177
- Lección 20: Tablas multidimensionales ... 189

Anexos: Material adicional ... 191

- Anexo 1: El TAD grafo y su representación en memoria ... 193
- Anexo 2: Algoritmos de vuelta atrás y árboles de juego ... 199
- Anexo 3: Introducción a los algoritmos voraces ... 209
- Anexo 4: Especificación algebraica de TAD ... 217
- Anexo 5: Transformación de algoritmos recursivos en iterativos ... 235
- Anexo 6: Chuletas de sintaxis de especificación y pseudocódigo ... 245

Bibliografía ... 253

Prólogo de la primera edición

La **abstracción de acciones** es la base de la metodología de **diseño descendente por refinamientos sucesivos**, útil para la resolución de pequeños problemas de tratamiento de información. Sin embargo, para afrontar la construcción de programas en media y gran escala es necesaria una metodología de **diseño modular**, que permita la partición del trabajo en unidades de programa que puedan ser desarrolladas independientemente del resto. El propósito de estos apuntes es presentar los principios básicos de una metodología de diseño modular basada en la **abstracción de datos**.

Este material ha sido elaborado para servir como soporte de la asignatura ***Estructuras de datos y algoritmos***, que se imparte en el tercer semestre de los estudios de Ingeniería Informática en el Centro Politécnico Superior[1] de la Universidad de Zaragoza. Los alumnos que cursan dicha asignatura han seguido previamente dos semestres de programación en los que han debido aprender a especificar formalmente y diseñar programas en pequeña escala, utilizando tipos de datos sencillos (como los predefinidos en un lenguaje de programación de la familia del Pascal); los alumnos conocen técnicas de diseño recursivo e iterativo, así como las herramientas básicas para poder medir la eficiencia de los algoritmos (atendiendo a su tiempo de ejecución). No obstante, el material presentado puede ser útil también para un segundo nivel en todos aquellos planes de estudios en los que se incluyan dos cursos de programación de computadores.

Pueden encontrarse en las librerías varios trabajos (muchos de ellos ya clásicos) con títulos similares o iguales a éste. Sin embargo, y ésta es la razón para la existencia de uno nuevo, la aproximación al tema que se pretende desarrollar es bien diferente. De hecho, el título que el autor habría elegido, en caso de no haber optado por mantener el nombre de la asignatura antes mencionada, hubiese sido más bien "Tipos abstractos de datos y algoritmos" o mejor "Introducción a la programación con tipos abstractos de datos". La diferencia estriba en el énfasis que se pretende dar en las páginas que siguen a la especificación formal de los tipos (de ahí la expresión "tipos abstractos de datos") como herramienta fundamental para el diseño modular de programas, en lugar de limitarse a presentar las estructuras de datos necesarias para representar los valores de los tipos definidos.

El comentario anterior no debe hacer pensar al lector que el material que sigue es original del autor. Nada más lejos de la realidad. Únicamente nos hemos limitado a enlazar las excelentes aproximaciones existentes en la literatura a la definición y conceptos relacionados con los tipos abstractos de datos y su especificación algebraica (véanse, por ejemplo, los dos últimos capítulos de la obra de Ricardo Peña titulada *Diseño de Programas. Formalismo y Abstracción*) con los trabajos más clásicos sobre estructuras de datos y algoritmos de manipulación (como, por ejemplo, *Estructuras de Datos y Algoritmos*, de Aho, Hopcroft y Ullman).

Los apuntes están estructurados en lecciones, agrupadas en grandes temas. En el primero de ellos, titulado "Tipos abstractos de datos", se presentan los conceptos fundamentales sobre los tipos abstractos de datos, su especificación formal (algebraica) y su utilización en el diseño modular de programas.

El segundo tema, "Tipos de datos lineales", introduce tres de los tipos abstractos lineales más representativos y útiles en programación: las pilas, las colas y las listas con acceso por posición. Para cada nuevo tipo presentado se incluyen su especificación formal, una o varias soluciones para la representación de sus valores, la implementación de las operaciones más importantes, su coste computacional y algunos ejemplos de aplicación.

El tercer tema, titulado "Árboles y esquemas algorítmicos", incluye los detalles sobre algunos de los tipos de árboles más frecuentemente utilizados, como los árboles binarios, árboles ordenados, árboles de búsqueda, montículos… y ejemplos de aplicación. Además, se introducen los algoritmos de vuelta atrás y las heurísticas voraces.

Los dos últimos temas, sobre "Tipos de datos funcionales" (o tablas) e "Introducción a los grafos", no se desarrollan con la misma extensión que los anteriores por razones diferentes. En el caso de las tablas, tras las definiciones formales convenientes, se hace hincapié en la representación mediante tablas dispersas basadas en la utilización de una función de localización (*hashing*, en inglés) y en las tablas multidimensionales representadas

[1] Hoy Escuela de Ingeniería y Arquitectura.

mediante estructuras de listas múltiples, pues otras representaciones posibles basadas en listas lineales o árboles de búsqueda no precisan mayor explicación tras el estudio de los temas previos.

En cuanto a los grafos, los alumnos de Ingeniería Informática (a quienes va dirigida preferentemente esta obra) han cursado previamente una asignatura titulada "Matemática discreta", en la que se les ha presentado el concepto de grafo y una buena colección de algoritmos para su manipulación. Por ello, y atendiendo a razones de completitud, se presentan sólo las especificaciones formales y varias alternativas de representación, junto a algunas consideraciones sobre el efecto que la elección de la representación tiene en el coste de los algoritmos de manipulación.

Por último, un comentario sobre las notaciones empleadas y los lenguajes de programación que pueden servir como soporte de prácticas. Para la especificación algebraica de tipos abstractos, se utiliza una sintaxis similar a la del lenguaje OBJ, pero en español. En cuanto a los módulos, estructuras de datos y algoritmos, se emplea una notación algorítmica, también en español, que consiste en una extensión modular de la notación utilizada en los apuntes sobre Introducción a la programación, elaborados por Javier Martínez y Javier Campos como soporte a la asignatura de igual nombre existente en el currículum de Ingeniería Informática del CPS.

En cuanto al lenguaje de programación soporte de las prácticas, el autor desaconseja la utilización de las extensiones modulares de Pascal (incluido el Modula 2), pues carecen de la posibilidad de definir tipos opacos y tipos genéricos, siendo ambos mecanismos fundamentales en la metodología desarrollada. Así, un lenguaje apropiado resulta ser el Ada, dotado de la posibilidad de definición de tipos opacos y tipos genéricos, con una sintaxis y semántica bien pensadas y una dificultad de aprendizaje similar al Pascal, si se limita su presentación a la parte secuencial. Otras alternativas pueden encontrarse en lenguajes de programación orientados a objetos (como, por ejemplo, C++), dada la cercanía de los conceptos de "clase" y "tipo abstracto de dato".

Javier Campos Laclaustra

Zaragoza, 30 de marzo de 1995

Prólogo de la segunda edición

Pasados ya más de veinte años desde la publicación de la primera edición de estos apuntes, gracias en aquella ocasión a *Prensas Universitarias de Zaragoza*, se presenta aquí una reedición y actualización que venía siendo necesaria desde hace ya varios años.

El cambio de plan de estudios de la anterior Ingeniería en Informática, de cinco años de duración, al actual Grado de Ingeniería Informática, de cuatro años, ha conllevado la reducción de créditos de algunas asignaturas y con ella la de horas de clase y, por tanto, de contenidos. Es el caso de la asignatura de Estructuras de Datos y Algoritmos, de segundo curso, tercer cuatrimestre del Grado, en la Escuela de Ingeniería y Arquitectura de la Universidad de Zaragoza.

En esta nueva edición, se han eliminado del temario las lecciones sobre especificación formal –algebraica– de tipos abstractos de datos y, en su lugar, se ha optado por utilizar una especificación no formal, en lenguaje natural. Así, las anteriores lecciones sobre especificación algebraica se han trasladado a un Anexo.

Además, se han introducido algunos pequeños cambios o correcciones en algunos aspectos de los algoritmos presentados. Por ejemplo, cabe destacar la sustitución de los mecanismos utilizados para el paso de parámetros en procedimientos y funciones. En la primera edición, inspirada por el lenguaje Pascal, se utilizaba el paso por valor y por referencia. En ésta, más inspirada por el lenguaje Ada, se utilizan mecanismos de más alto nivel de abstracción, es decir, más cercanos al razonamiento del diseñador de algoritmos y menos a los detalles de implementación, como son los pasos de parámetros de entrada, salida o entrada/salida.

Por lo demás, algunas otras de las lecciones de la primera edición han sido desplazadas también a Anexos, como las de transformación de algoritmos recursivos en iterativos, la especificación y representación en memoria del TAD grafo, y las lecciones de introducción a esquemas algorítmicos sencillos, como los algoritmos de vuelta atrás o los algoritmos voraces. Por el contrario, se ha introducido una nueva lección sobre árboles lexicográficos.

También se han eliminado los ejercicios propuestos en las últimas páginas de la primera edición. Los estudiantes actuales de la asignatura en la Escuela de Ingeniería y Arquitectura de Zaragoza disponen de suficiente material de trabajo, enunciados de ejercicios y de exámenes de convocatorias anteriores, en la página web de la asignatura: http://webdiis.unizar.es/asignaturas/EDA/.

Aún con estos recortes y pequeños cambios, los objetivos fundamentales enumerados en el prólogo de la primera edición se mantienen inalterados, y por eso se ha optado por incluir dicho prólogo en las páginas anteriores.

Para terminar, debo agradecer a los profesores del Departamento de Informática e Ingeniería de Sistemas de la Universidad de Zaragoza que en todos estos años me han acompañado en la impartición de la asignatura: Elvira Mayordomo, Pedro Fernández, Miguel Ángel Villarroel, Elsa García, Maite Lozano, Pablo López, Javier Martínez, Luis Carlos Gallego, Yolanda Villate, Jorge Júlvez, Ángel Luis Garrido, Miguel Flores, Fernando Tricas y Jorge Bernad. Algunos de los materiales e ideas aquí presentados han sido inspirados por ellos.

Javier Campos Laclaustra

Zaragoza, 15 de noviembre de 2018

TEMA I

Programación con Tipos Abstractos de Datos

Lección 1

Tipos Abstractos de Datos (TAD)

Indice

1. Concepto de abstracción
2. Definición de TAD
3. Programación con TAD
4. Ventajas de la programación con TAD

1. Concepto de abstracción

El concepto de **abstracción** en el proceso de comprensión de un problema lleva consigo el **destacar los detalles importantes** e ignorar los irrelevantes.

Los siguientes son ejemplos de abstracción relacionados de alguna manera con la programación y los lenguajes:

- los lenguajes de alto nivel persiguen abstraerse de los detalles de los ensambladores;
- los procedimientos y funciones permiten la abstracción de un conjunto de instrucciones;
- la especificación formal es una abstracción de cómo resolver un problema.

El uso de la abstracción en programación sugiere un método jerárquico de diseño (normalmente descendente) que implica la consideración de varios **niveles de detalle**.

En programación cabe distinguir entre las dos formas siguientes de abstracción:

- **Abstracción de acciones** (procedural o funcional): una acción (o valor) virtual parametrizada (ocultación de información: datos locales y secuencia de instrucciones); separación del qué (especificación) y el cómo (implementación).
- **Abstracción de datos**, en la que nos vamos a centrar en estos apuntes, si bien conocemos ya algunos ejemplos en los lenguajes de programación usuales:
  - Tipos simples estándar (enteros, booleanos, reales)
    independencia de la máquina
    representación invisible
    manipulación con un conjunto de operaciones prefijadas
  - Tipos simples definidos por el programador (enumeraciones)
    se eleva el "nivel del lenguaje" (ejemplo: `enum Color {rojo, verde, azul};`)
    conjunto de operaciones predefinidas
  - Tipos estructurados definidos por el programador
    constructores **genéricos** de tipos que debe completar el programador (vectores y tuplas o registros)
    (ejemplos: `int T[100]; struct Nif {int dni; char letra;};`)
    operaciones predefinidas para tipos estructurados como los anteriores (ejemplo, el acceso a T[0])

2. Definición de TAD

La siguiente puede considerarse una definición informal de un **tipo abstracto de datos** (TAD, en lo que sigue): **un TAD es una colección de valores y de operaciones** definidos mediante una **especificación independiente de cualquier representación**.

Ejemplos ya conocidos son los booleanos y naturales. A continuación, se presentan algunas de las operaciones importantes sobre estos tipos, que ya estamos acostumbrados a manejar de forma independiente de la representación de los valores de los tipos correspondientes (se utiliza una notación que se presentará más adelante, si bien resulta fácil de interpretar sin más comentarios previos).

```
espec boolnat
  géneros bool,nat
  operaciones
    verdad: -> bool
    falso: -> bool
    ¬_: bool -> bool
    _∧_,_∨_: bool bool -> bool

    0: -> nat
    suc: nat -> nat
    _+_,_*_: nat nat -> nat
    _≤_,_>_: nat nat -> bool
fespec
```

Algunos lenguajes no permiten decidir las operaciones que se desea utilizar para cada tipo definido por el programador. Es decir, hay lenguajes que no permiten definir tipos, sólo representar unos tipos con otros. En esos lenguajes, si por ejemplo quisiéramos utilizar secuencias de datos con una cierta colección de operaciones, tendríamos que representarlas con otro tipo (vector, fichero…). Además, la representación sería visible en todo el ámbito del programa.

En cambio, la mayor parte de lenguajes en la actualidad sí permiten decidir con precisión qué operaciones se podrán utilizar para los valores de un tipo definido por el programador. Es en esos casos en los que la programación con TAD tiene sentido y adquiere importancia.

La programación con TAD requiere dos pasos:

Paso 1. **Definición del tipo**. Establecer la interfaz con el (programador) usuario del tipo: decidir las operaciones necesarias/adecuadas para manipular los valores y especificarlas (tiene una parte sintáctica y otra semántica).

Paso 2. **Implementación del tipo**. Elegir la representación de los valores e implementar las operaciones.

El concepto fundamental subyacente bajo la programación con TAD es la **encapsulación**. La encapsulación consiste básicamente en:

- la **privacidad** de la representación (el usuario no conoce sus detalles), y
- la **protección** del tipo (el usuario sólo puede utilizar las operaciones previstas).

Ejercicio: pensar en otros TAD ya manipulados, como, por ejemplo, las cadenas de caracteres, los pares ordenados de naturales, las fechas, etc. ¿Cuáles son las operaciones básicas que pueden necesitarse para manipular estos tipos?

Un TAD es una abstracción porque distingue entre:

- los detalles importantes: la interfaz que debe conocer el usuario (comportamiento observable), es decir, la definición o especificación del tipo (nombre del tipo y los encabezamientos y especificación de las operaciones básicas), y
- los detalles ocultables (irrelevantes en ese nivel): la implementación del tipo, es decir, la elección de la representación de los valores y la implementación de las operaciones básicas.

Los diferentes valores del tipo (dominio) definen un cierto conjunto sobre el que se definen una serie de operaciones.

Se denomina **especificación** de un tipo abstracto de datos a la definición precisa de dicho tipo, es decir, la definición del dominio de valores y de las operaciones de manipulación de los mismos. Esta definición puede hacerse utilizando un lenguaje formal (normalmente algebraico) o bien usando el lenguaje natural para expresar la semántica de las operaciones del tipo.

Como veremos en la siguiente lección, los TAD están definidos fundamentalmente por la semántica de sus operaciones y no por los identificadores con que nos refiramos a ellas. Por ejemplo, para el tipo `booleano`, no es tan importante que sus valores posibles se llamen `verdad` y `falso` como que existan dos valores, v_1 y v_2, tres operaciones internas definidas sobre ellos, una monoaria (`not`) y dos binarias (`and` y `or`), verificando una serie de propiedades ($not(v_1)=v_2$; $not(v_2)=v_1$; etc.).

A título de ejemplo, se presenta a continuación la especificación de un tipo abstracto de datos denominado `conjcar` (una definición del tipo conjunto de caracteres con algunas operaciones básicas de manipulación), de dos formas distintas: mediante un lenguaje algebraico (especificación formal) y mediante el lenguaje natural (especificación no formal).

Especificación formal (nótese que la semántica de las operaciones se expresa mediante ecuaciones):

```
espec conjuntos_de_caracteres
  usa booleanos,caracteres,naturales
  género conjcar
  operaciones
    ∅: -> conjcar
    poner: carácter conjcar -> conjcar
    quitar: carácter conjcar -> conjcar
    _∈_: carácter conjcar -> booleano
    _∪_: conjcar conjcar -> conjcar
    _∩_: conjcar conjcar -> conjcar
    vacío?: conjcar -> booleano
    cardinal: conjcar -> natural
  ecuaciones A,B: conjcar; c,c1,c2: carácter
    (c1=c2) => poner(c1,poner(c2,A)) = poner(c2,A)
    (c1≠c2) => poner(c1,poner(c2,A)) = poner(c2,poner(c1,A))

    quitar(c,∅) = ∅
    (c1=c2) => quitar(c1,poner(c2,A)) = quitar(c1,A)
    (c1≠c2) => quitar(c1,poner(c2,A)) = poner(c2,quitar(c1,A))

    c∈∅ = falso
    c1∈poner(c2,A) = (c1=c2) ∨ (c1∈A)

    A∪∅ = A
    A∪poner(c,B) = poner(c,A∪B)

    A∩∅ = ∅
    (c∈A) => A∩poner(c,B) = poner(c,A∩B)
    ¬(c∈A) => A∩poner(c,B) = A∩B

    vacío?(∅) = verdad
    vacío?(poner(c,A)) = falso

    cardinal(∅) = 0
    cardinal(poner(c,A)) = suc(cardinal(quitar(c,A)))
fespec
```

Especificación no formal (nótese que en color verde aparece la semántica, escrita en lenguaje natural):

```
espec conjuntos_de_caracteres
usa caracteres, booleanos, naturales
género conjcar
  {Los valores del TAD conjcar representan conjuntos de caracteres,
  sin elementos repetidos, y en los que no es relevante el orden en el que
  los caracteres se añaden al conjunto.}
operaciones
  creaVacío: -> conjcar
  {Devuelve un conjunto de caracteres vacío, es decir un conjunto
  que no contiene ningún carácter}

  poner: carácter e, conjcar c -> conjcar
  {Si e está en c, devuelve un conjunto igual a c;
  si e no está en c, devuelve el conjunto resultante de añadir e a c}

  quitar: carácter e, conjcar c -> conjcar
  {Si e está en c, devuelve el conjunto resultante de eliminar e de c;
  si e no está en c, devuelve un conjunto igual a c}

  pertenece: carácter e, conjcar c -> booleano
  {Devuelve verdad si y sólo si e está en c}

  unión: conjcar c1, conjcar c2 -> conjcar
  {Devuelve un conjunto que contiene todos los caracteres que están en c1 y todos
  los que están en c2}

  intersección: conjcar c1, conjcar c2 -> conjcar
```

```
{Devuelve un conjunto que contiene únicamente los caracteres que están tanto
en c1 como en c2}

vacío?: conjcar c -> booleano
{Devuelve verdad si y sólo si c no contiene ningún carácter}

cardinal: conjcar c -> natural
{Devuelve el número total de caracteres que contiene c (0 si es vacío)}

{Las cuatro siguientes operaciones son un Iterador. Hablaremos de ello más adelante}

iniciarIterador: conjcar c -> conjcar
{Prepara el iterador para recorrer los caracteres del conjunto c, de forma que el
 siguiente carácter a visitar sea el primero que visitamos (situación de no haber
 visitado ningún carácter)}

existeSiguiente?: conjcar c -> booleano
{Devuelve falso si ya se han visitado todos los caracteres de c, devuelve verdad en
 caso contrario}

parcial siguiente: conjcar c -> carácter
{Devuelve el siguiente carácter de c.
 Parcial: la operación no está definida si no existeSiguiente?(c)}

parcial avanza: conjcar c -> conjcar
{Prepara el iterador para visitar el siguiente carácter de c.
 Parcial: la operación no está definida si no existeSiguiente?(c)}

fespec
```

3. Programación con TAD

La programación en gran escala exige la partición del código en **módulos**, o bien en **clases** (si estamos utilizando características de Orientación a Objetos). En el resto de estos apuntes utilizaremos módulos y no clases.

Un módulo es una unidad del programa que puede ser desarrollada independientemente del resto.

Para ello, la descomposición en módulos debe cumplir unos requisitos:

- Cada módulo debe tener una conexión mínima con el resto: la interfaz (para obtener la relativa independencia en el desarrollo).
- La mayor parte de cambios y mejoras del programa afecten sólo a un número pequeño de módulos.
- El tamaño de cada módulo sea adecuado (si es muy grande es difícil hacer cambios, si es muy pequeño resulta costoso por los trabajos adicionales de especificación, documentación, control de versiones, ...).

La definición e implementación de un TAD puede encapsularse en un módulo:

- La interfaz (conexión con el exterior) es reducida: el nombre del tipo y los encabezamientos de las operaciones (procedimientos y/o funciones). No permite compartir variables entre módulos, ni conocer la estructura interna de la representación de los valores del tipo en el exterior. Un tipo del que sólo se exporta el nombre se denomina **opaco**.
- Puede cambiarse la implementación independientemente (los cambios afectan en muchos casos a un solo módulo).
- El tamaño del módulo suele ser suficientemente grande (implementación de las diferentes operaciones).

A continuación, a título de ejemplo, se desarrolla el módulo `conjuntosDeCaracteres`, que exporta el tipo `conjcar` definido en la sección anterior, utilizando una notación algorítmica en español. Véase el Anexo 6 para conocer la sintaxis de esa notación o pseudocódigo.

```
módulo conjuntosDeCaracteres
importa booleanos,caracteres,naturales
exporta { esta parte contiene la interfaz }

  tipo conjcar[1] {Tipo especificado previamente}
  {Los valores del TAD conjcar representan conjuntos de caracteres,
  sin elementos repetidos, y en los que no es relevante el orden en el que
  los caracteres se añaden al conjunto.}
```

[1] Es un tipo opaco.

```
procedimiento vacío(sal A:conjcar)
{Devuelve un conjunto de caracteres vacío, es decir un conjunto
que no contiene ningún carácter}

función esVacío(A:conjcar) devuelve booleano
{Devuelve verdad si y sólo si A no contiene ningún carácter}

procedimiento poner(ent c:carácter; e/s A:conjcar)
{Si c está en A, devuelve un conjunto igual a A;
si c no está en A, devuelve el conjunto resultante de añadir c a A}

procedimiento quitar(ent c:carácter; e/s A:conjcar)
{Si c está en A, devuelve el conjunto resultante de eliminar c de A;
si c no está en A, devuelve un conjunto igual a A}

función pertenece(c:carácter; A:conjcar) devuelve booleano
{Devuelve verdad si y sólo si c está en A}

procedimiento unión(ent A,B:conjcar; sal C:conjcar)
{Devuelve en C un conjunto que contiene todos los caracteres que están en
 A y todos los que están en B}

procedimiento intersección(ent A,B:conjcar; sal C:conjcar)
{Devuelve en C un conjunto que contiene únicamente los caracteres que
 están tanto en A como en B}

función cardinal(A:conjcar) devuelve natural
{Devuelve el número total de caracteres que contiene c (0 si es vacío)}

{Las tres siguientes operaciones implementan un Iterador.
 Hablaremos de ello más adelante}

procedimiento iniciarIterador(e/s A:conjcar)
{Prepara el iterador para que el siguiente elemento a visitar sea el primer
 carácter de A por visitar, si existe (situación de no haber visitado ningún
 carácter)}

función existeSiguiente(A:conjcar) devuelve booleano
{Devuelve falso si ya se han visitado todos los caracteres de A.
 Devuelve verdad en caso contrario.}

procedimiento siguiente(e/s A:conjcar; sal c:carácter; sal error:booleano)
{Implementa las operaciones "siguiente" y "avanza" de la especificación, es decir:
 Si existeSiguiente(A), error toma el valor falso, c toma el valor del siguiente
 carácter del conjunto, y se avanza el iterador al carácter siguiente del
 conjunto. Si no existeSiguiente(A), error toma el valor verdad, c queda
 indefinido y A queda como estaba.}

implementación
  tipo conjcar = registro
                   elmto: vector[carácter] de booleano;
                   card: natural
                 freg

  procedimiento vacío(sal A:conjcar)
  variable c:carácter
  principio
    A.card:=0;
    para c:=chr(0) hasta chr(255) hacer
      A.elmto[c]:=falso
    fpara
  fin

  función esVacío(A:conjcar) devuelve booleano
  principio
    devuelve(A.card=0)
  fin
```

```
función pertenece(c:carácter; A:conjcar) devuelve booleano
principio
  devuelve(A.elmto[c])
fin

procedimiento poner(ent c:carácter; e/s A:conjcar)
principio
  si not pertenece(c,A) entonces
    A.elmto[c]:=verdad;
    A.card:=A.card+1
  fsi
fin

procedimiento quitar(ent c:carácter; e/s A:conjcar)
principio
  si pertenece(c,A) entonces
    A.elmto[c]:=falso;
    A.card:=A.card-1
  fsi
fin

procedimiento unión(ent A,B:conjcar; sal C:conjcar)
variable x:carácter
principio
  C.card:=0;
  para x:=chr(0) hasta chr(255) hacer
    C.elmto[x]:=A.elmto[x] or B.elmto[x];
    si C.elmto[x] entonces
      C.card:=C.card+1
    fsi
  fpara
fin

procedimiento intersección(ent A,B:conjcar; sal C:conjcar)
variable x:carácter
principio
  C.card:=0;
  para x:=chr(0) hasta chr(255) hacer
    C.elmto[x]:=A.elmto[x] and B.elmto[x];
    si C.elmto[x] entonces
      C.card:=C.card+1
    fsi
  fpara
fin

función cardinal(A:conjcar) devuelve natural
principio
  devuelve(A.card)
fin

procedimiento iniciarIterador(e/s A:conjcar)
principio
  A.iterPos:=0;
  mientrasQue A.iterPos≤255 andthen not A.elmto[chr(A.iterPos)] hacer
    A.iterPos:=A.iterPos+1
  fmq
fin

función existeSiguiente(A:conjcar) devuelve booleano
principio
  devuelve A.iterPos<256
fin

procedimiento siguiente(e/s A:conjcar; sal c:carácter; sal error:booleano)
principio
  si existeSiguiente(A) entonces
    c:=chr(A.iterPos);
    A.iterPos:=A.iterpos+1;
    mientrasQue A.iterPos≤255 andthen not A.elmto[chr(A.iterPos)] hacer
```

```
        A.iterPos:=A.iterPos+1
      fmq;
      error:=falso
    sino
      error:=verdad
    fsi
  fin

fin
```

Desde los años setenta los lenguajes de programación se diseñan para poder soportar el diseño a gran escala de algoritmos y, por tanto, suelen permitir el desarrollo de módulos independientes.

Vamos a presentar un mismo ejemplo (el módulo `conjuntos` de la sección anterior) codificado en dos lenguajes diferentes que adoptan distintas soluciones para la definición de módulos: Ada y C++. Lo hacemos sin utilizar orientación a objetos, es decir, sin utilizar clases.

Veamos ahora la codificación del mismo módulo anterior en Ada (evitamos, para abreviar, escribir de nuevo la documentación del código; tampoco incluimos las operaciones del iterador):

```
package conjuntos is
  type conjcar is private;

  procedure vacio(A:out conjcar);
  function esVacio(A:in conjcar) return boolean;
  procedure poner(c:in character; A:in out conjcar);
  procedure quitar(c:in character; A:in out conjcar);
  function pertenece(c:in character; A:in conjcar) return boolean;
  procedure union(A,B:in conjcar; C:out conjcar);
  procedure interseccion(A,B:in conjcar; C:out conjcar);
  function cardinal(A:in conjcar) return integer;
private
  type elementos is array(character) of boolean;
  type conjcar is
    record
      elmto:elementos;
      card:integer;
    end record;
end conjuntos;

package body conjuntos is
  procedure vacio(A:out conjcar) is
  begin
    A.card:=0;
    for c in character loop
      A.elmto(c):=false;
    end loop;
  end vacio;

  function esVacio(A:in conjcar) return boolean is
  begin
    return A.card=0;
  end esVacio;

  function pertenece(c:in character; A:in conjcar) return boolean is
  begin
    return A.elmto(c);
  end pertenece;

  procedure poner(c:in character; A:in out conjcar) is
  begin
    if not pertenece(c,A) then
      A.elmto(c):=true;
      A.card:=A.card+1;
    end if;
  end poner;

  procedure quitar(c:in character; A:in out conjcar) is
```

```
    begin
      if pertenece(c,A) then
        A.elmto(c):=false;
        A.card:=A.card-1;
      end if;
    end quitar;

    procedure union(A,B:in conjcar; C:out conjcar) is
      SOL:conjcar;
    begin
      SOL.card:=0;
      for x in character loop
        SOL.elmto(x):=A.elmto(x) or B.elmto(x);
        if SOL.elmto(x) then
          SOL.card:=SOL.card+1;
        end if;
      end loop;
      C:=SOL;
    end union;

    procedure interseccion(A,B:in conjcar; C:out conjcar) is
      SOL:conjcar;
    begin
      SOL.card:=0;
      for x in character loop
        SOL.elmto(x):=A.elmto(x) and B.elmto(x);
        if SOL.elmto(x) then
          SOL.card:=SOL.card+1;
        end if;
      end loop;
      C:=SOL;
    end interseccion;

    function cardinal(A:in conjcar) return integer is
    begin
      return A.card;
    end cardinal;

  end conjuntos;
```

Como puede verse, se han escrito dos módulos diferente ("packages", en Ada). El primero de ellos, denominado **módulo de declaración**, contiene la definición del tipo de dato (nombre del tipo y encabezamientos de los algoritmos), pero **también contiene la representación concreta** del tipo de dato, bajo la palabra clave **`private`**.

En Ada, la información incluida tras la palabra clave **`private`** en el módulo de declaración no puede ser utilizada por el programador usuario del módulo. La representación del tipo no es accesible en el exterior.

El segundo módulo (**`package body`**), denominado **módulo de implementación**, contiene los detalles de implementación de los algoritmos declarados en el módulo anterior.

Comentar, finalmente, que cuando el programador de un módulo auxiliar entrega, a modo de documentación, la definición de éste a otro programador "usuario" del módulo, no precisa incluir la parte contenida bajo la palabra clave **`private`** (la representación del tipo). Esto es así porque el "usuario" no necesita conocer la representación para manipular datos del tipo definido (salvo quizás por aspectos de eficiencia de los algoritmos desarrollados, que pueden depender de la representación elegida).

En C++, una implementación posible (sin utilizar clases) es la siguiente (de nuevo, evitamos comentar el código). Suponemos que el lector posee una cierta práctica en el uso del lenguaje C++ y no aportamos más explicaciones:

Archivo conjcar.h:

```
#ifndef _CONJCAR_
#define _CONJCAR_

const int TOTAL_CARACTERES = 128;

// Interfaz del TAD. Pre-declaraciones:
```

```
struct Conjcar;
void vacio(Conjcar& c);
bool esVacio(const Conjcar& c);
void poner(char caracter, Conjcar& c);
void quitar(char caracter, Conjcar& c);
bool pertenece(char caracter, const Conjcar& c);
void unionConjcar(const Conjcar& c1, const Conjcar& c2, Conjcar& c);
void intersecConjcar(const Conjcar& c1, const Conjcar& c2, Conjcar& c);
int cardinal(const Conjcar& c);
void iniciarIterador(Conjcar& c);
bool existeSiguiente(const Conjcar& c);
bool siguiente(Conjcar& c, char& caracter);


// Declaración

struct Conjcar {
    friend void vacio(Conjcar& c);
    friend bool esVacio(const Conjcar& c);
    friend void poner(char caracter, Conjcar& c);
    friend void quitar(char caracter, Conjcar& c);
    friend bool pertenece(char caracter, const Conjcar& c);
    friend void unionConjcar(const Conjcar& c1, const Conjcar& c2, Conjcar& c);
    friend void intersecConjcar(const Conjcar& c1, const Conjcar& c2,
                                Conjcar& c);
    friend int cardinal(const Conjcar& c);
    friend void iniciarIterador(Conjcar& c);
    friend bool existeSiguiente(const Conjcar& c);
    friend bool siguiente(Conjcar& c, char& caracter);

    // Representación de los valores del TAD

    private:
        bool conjunto [TOTAL_CARACTERES];
        int card;
};

#endif
```

Archivo conjcar.cpp:

```
#include "conjcar.h"

void vacio(Conjcar& c) {
    for (int i = 0; i < TOTAL_CARACTERES; i++) {
        c.conjunto[i] = false;
    }
    c.card = 0;
};

bool esVacio(const Conjcar& c) {
    return c.card == 0;
};

void poner(char caracter, Conjcar& c) {
    if (!pertenece(caracter,c)) {
         c.conjunto[int(caracter)] = true;
         c.card = c.card +1;
    }
};

void quitar(char caracter, Conjcar& c) {
    if (pertenece(caracter,c)) {
        c.conjunto[int(caracter)] = false;
        c.card = c.card - 1;
    }
};
```

```
bool pertenece(char caracter, const Conjcar& c) {
    return 0<=caracter && caracter<TOTAL_CARACTERES && c.conjunto[int(caracter)];
};

void unionConjcar(const Conjcar& c1, const Conjcar& c2, Conjcar& c) {
    c.card = 0;
    for (int i = 0; i < TOTAL_CARACTERES; i++) {
        c.conjunto[i] = c1.conjunto[i] || c2.conjunto[i];
        if (c.conjunto[i]) {
            c.card = c.card + 1;
        }
    }
};

void intersecConjcar(const Conjcar& c1, const Conjcar& c2, Conjcar& c) {
    c.card = 0;
    for (int i = 0; i < TOTAL_CARACTERES; i++) {
        c.conjunto[i] = c1.conjunto[i] && c2.conjunto[i];
        if (c.conjunto[i]) {
            c.card = c.card + 1;
        }
    }
};

int cardinal(const Conjcar& c) {
    return c.card;
};

void iniciarIterador(Conjcar& c) {
    c.iterPos = 0;
    while(c.iterPos < TOTAL_CARACTERES && !c.conjunto[c.iterPos]) {
        c.iterPos = c.iterPos + 1;
    }
}

bool existeSiguiente(const Conjcar& c) {
    return c.iterPos < TOTAL_CARACTERES;
}

bool siguiente(Conjcar& c, char& caracter) {
    if (existeSiguiente(c)) {
        caracter = char(c.iterPos);
        c.iterPos = c.iterPos + 1;
        while(c.iterPos < TOTAL_CARACTERES && !c.conjunto[c.iterPos]) {
            c.iterPos = c.iterPos + 1;
        }
        return true;
    }
    else {
        return false;
    }
}
```

4. Ventajas de la programación con TAD

La utilización de la metodología de diseño descendente, mediante refinamientos sucesivos, facilita el desarrollo de programas de “tamaño pequeño” (**programación a pequeña escala**). Dicha metodología resulta sin embargo insuficiente para el diseño de programas de “medio o gran tamaño” (programación a media o gran escala).

El diseño por refinamientos sucesivos se basa en la **abstracción de acciones** (procedural o funcional).

Inconvenientes de ese mecanismo de abstracción para programar en gran escala son:

- Los tipos de datos utilizados son los predefinidos en el lenguaje, son tipos concretos o de bajo nivel, por tanto, existe un desequilibrio: acciones abstractas (procedimientos o funciones) o de alto nivel que manipulan datos concretos o de bajo nivel.

- Las decisiones sobre representación de datos se toman al principio; deberían aplazarse hasta que se conozcan las operaciones necesarias para cada tipo.
- En algoritmos de alto nivel hay que utilizar detalles de bajo nivel sobre los datos (se oscurecen los rasgos importantes de esos algoritmos).

La clásica "ecuación" de N. Wirth:

programas = datos + algoritmos

es bastante representativa de la metodología de diseño basada en la abstracción de acciones. Esa ecuación podría refinarse en la siguiente forma:

programas = datos + (algoritmos de datos + algoritmos de control)

entendiendo por "algoritmos de datos" los algoritmos de más bajo nivel encargados de la manipulación de las estructuras de datos y por "algoritmos de control" a la parte del algoritmo que representa el método de solución del problema (independiente, hasta cierto punto, de las estructuras de datos seleccionadas).

Los problemas mencionados se resuelven con la metodología de programación modular basada en TAD. Esta metodología se puede resumir en la ecuación siguiente, en la que se han agrupado los términos "datos" y "algoritmos de datos" en uno solo denominado "implementación de TAD":

programas = implementación de TAD + algoritmos de control

Veamos un ejemplo.

Ejercicio: *Diseñar un programa que lea una secuencia de enteros de un fichero y escriba en pantalla cada entero distinto leído junto con su frecuencia de aparición, en orden de frecuencias decrecientes.*

La metodología de programación basada en TAD sugiere posponer la decisión de cómo se almacenarán los enteros leídos y sus frecuencias y suponer que existe un TAD llamado `tabla` para dicho almacenamiento. Las operaciones necesarias en ese TAD se conocerán tras haber diseñado el programa principal (módulo usuario del TAD).

```
procedimiento estadística
importa tablas
variables
  f:fichero de entero; nombre:cadena;
  t:tabla; dato,orden,frec:entero
principio
  escribir('Nombre del fichero: ');
  leer(nombre);
  asociar(f,nombre);
  iniciarlectura(f);
  inicializar(t);
  mientrasQue not finFichero(f) hacer
    leer(f,dato);
    añadir(t,dato)
  fmq;
  disociar(f);
  para orden:=1 hasta total(t) hacer
    info(t,orden,dato,frec);
    escribir('entero: ',dato,' frecuencia: ',frec)
  fpara
fin
```

El algoritmo recoge sólo los aspectos esenciales. Falta implementar el tipo `tabla`, del cual se conocen ya su nombre y requisitos de las operaciones, es decir, se conoce la interfaz:

```
módulo tablas
exporta
  tipo tabla  { tabla de frecuencias de enteros }

  procedimiento inicializar(sal t:tabla)
```

```
    { Crea una tabla vacía t de frecuencias }

    procedimiento añadir(e/s t:tabla; ent n:entero)
    { Modifica t incrementando en 1 la frecuencia de n }

    función total(t:tabla) devuelve entero
    { Devuelve el nº de enteros distintos en la tabla t }

    procedimiento info(ent t:tabla; ent i:entero; sal n,frec:entero)
    { Al terminar, n es el entero que ocupa el i-ésimo lugar en la tabla t,
      en orden de frecuencias decrecientes, y frec es su frecuencia }

  implementación
    ...
  fin
```

El siguiente refinamiento lleva asociada la elección de la representación del TAD tabla y la implementación de las operaciones. En la elección de la representación influyen las operaciones que deben implementarse (`añadir` e `info`, fundamentalmente), pues debe buscarse la máxima eficiencia. Hay muchas soluciones posibles pero la elección no modificará en nada el código del algoritmo principal (quizá únicamente su eficiencia).

Una metodología de programación en media o gran escala puede basarse en los **refinamientos sucesivos de TAD**:

- Primera fase: decidir las interfaces de todos los módulos (cada módulo define un TAD). Para ello no es necesario detallar la representación e implementación de las operaciones, sino que basta con decidir si para representar/implementar cada TAD se precisan otros TAD de más bajo nivel, para los que a su vez hay que definir las interfaces correspondientes.
- Segunda fase: distribución del trabajo de implementación de cada módulo entre los diversos programadores del equipo (cada programador detalla uno o más módulos). Para la implementación detallada de cada módulo sólo se necesitan del resto las interfaces.

Para terminar, enumeramos algunos criterios de calidad del software y a continuación algunas de las ventajas de la programación con TAD con respecto a esos criterios.

Criterios de calidad del software:

- Corrección → debe realizar exactamente las tareas definidas por su especificación.
- Legibilidad → debe ser fácil de leer y lo más sencillo posible. Contribuyen la abstracción y la codificación con comentarios, sangrados, etc.
- Extensibilidad → facilidad para adaptarse a cambios en su especificación.
- Robustez → capacidad de funcionar correctamente incluso en situaciones anormales.
- Eficiencia → hace un buen uso de los recursos, tales como el tiempo, espacio (memoria y disco), etc.
- Facilidad de uso → la utilidad de un sistema está relacionado con su facilidad de uso.
- Portabilidad → facilidad con la que puede ser transportado a diferentes sistemas físicos o lógicos.
- Verificabilidad → facilidad de verificación de un software, su capacidad para soportar los procedimientos de validación, juegos de test, ensayo o pruebas.
- Reutilización → capacidad de ser reutilizados en nuevas aplicaciones o desarrollos.
- Integridad → capacidad de proteger sus propios componentes frente a accesos o usos indebidos.
- Compatibilidad → facilidad para ser combinados con otros y usados en diferentes plataformas hardware o software.

Algunas ventajas de la programación con TAD para varios de los criterios de calidad del software antes enumerados:

- Abstracción:
  - La complejidad del problema se diluye. Los módulos en los que se descompone el problema serán de menor complejidad.
  - Se pueden implementar los TAD sólo a partir de la especificación, sin saber para qué se van a usar → Reusabilidad.
- Corrección:

    - Los TAD pueden ser desarrollados y probados de forma independiente.
    - Se pueden utilizar los TAD sólo conociendo la especificación. Facilita la integración de módulos.
- Eficiencia:
    - La implementación puede retrasarse hasta conocer las restricciones de eficiencia sobre sus operaciones.
    - Para un TAD podemos contar con diferentes implementaciones válidas y optar por la más eficiente y adecuada a las restricciones a cumplir.
- Legibilidad:
    - La especificación de un TAD es suficiente para entender su significado y comportamiento.
    - Un TAD tiene un tamaño y complejidad acotados que facilita su legibilidad.
- Modificabilidad y mantenimiento:
    - Tanto durante el periodo de desarrollo y pruebas, como durante en el periodo de mantenimiento y de vida del software.
    - Cambios localizados y acotados.
    - Cambios que no afecten a la especificación no afectarán a los programas que usen el TAD.
- Organización:
    - Facilita el reparto de tareas y la comunicación en un grupo de programadores.
    - El equipo desarrolla en paralelo las múltiples partes o módulos del sistema.
- Reusabilidad:
    - TAD reutilizables en otros contextos con pocos o ningún cambio (¡escoger bien el conjunto de operaciones!).
- Seguridad:
    - Imposibilidad de manipular directamente la representación interna de los datos u objetos del tipo.
    - Impide el mal uso y la generación de valores incorrectos.

Lección 2

Especificación de TAD

Indice

1. Características generales de una especificación
2. Especificación algebraica
3. Especificación no formal

1. Características generales de una especificación

Como ya se ha indicado, un Tipo Abstracto de Datos (TAD) es un conjunto de valores y de operaciones definidos mediante una especificación independiente de cualquier implementación.

La definición o especificación del TAD es el primero de los pasos en la programación con TAD.

La especificación de un TAD consiste en establecer la interfaz con el usuario del tipo ("lo que necesita saber el programador-usuario-del-TAD"), es decir, describir qué es el TAD sin decir cómo se implementa:

- Definir su dominio de valores, y
- decidir la lista de operaciones necesarias y especificarlas, o lo que es lo mismo, para cada operación describir la información de entrada, la información de salida, y el comportamiento o efecto de la operación.

De alguna manera, definir un TAD consiste en establecer las propiedades de sus operaciones en relación con los datos de su dominio.

Una especificación, para ser útil, debe ser:

- Precisa o no ambigua: para evitar problemas de interpretación.
- Concisa: decir lo imprescindible.
- General: adaptable a diferentes contextos.
- Legible: para todos y que todos entiendan lo mismo.

Siendo no ambigua, la especificación define un único tipo de datos, es decir, define totalmente su comportamiento.

Esencialmente hay dos formas de especificar un TAD: algebraica (o formal) y no formal.

2. Especificación algebraica

La especificación algebraica es una técnica formal para especificar TAD.

El objetivo es definir sin ambigüedades un tipo de datos (conjunto de valores y efecto de cada operación permitida).

Las ventajas de especificar algebraicamente TAD son:

- permite definir tipos independientemente de cualquier posible representación y razonar sobre la corrección de una representación/implementación;
- unanimidad de la interpretación del tipo por los distintos programadores usuarios del mismo;
- deducir propiedades satisfechas por cualquier implementación correcta del tipo y, en consecuencia, posibilidad de verificar formalmente los módulos que usan el tipo.

En esta sección se introduce la sintaxis habitual de las especificaciones algebraicas de tipos, si bien, no se entra en profundidad en aspectos de su semántica puesto que en lo que sigue emplearemos una especificación no formal para definir tipos.

Las especificaciones algebraicas se componen de una signatura y de un conjunto de ecuaciones.

La **signatura** de una especificación algebraica define los **géneros** o nombres de los nuevos tipos especificados, los nombres de las **operaciones** y sus **perfiles** (es decir, su dominio o aridad y su rango o coaridad).

Se suele utilizar una **notación funcional** para definir las operaciones de los TAD, es decir, una operación es una función que toma **como parámetros cero o más valores** de diversos tipos y produce **como resultado un solo valor** de otro tipo.

Por ejemplo, la signatura del TAD `tabla` mencionado en la lección anterior es la siguiente:

```
espec tablas
  usa naturales,enteros
  género tabla
  operaciones
    inicializar: -> tabla
    añadir: tabla entero -> tabla
    total: tabla -> natural
    infoEnt: tabla natural -> entero
    infoFrec: tabla natural -> natural
fespec
```

La cláusula **`usa`** se utiliza para importar las definiciones hechas en otras especificaciones (definición de los géneros `natural` y `entero`, en este caso). El género especificado en la signatura anterior es `tabla`. En general, una especificación puede contener la definición de varios géneros.

En el ejemplo, se han definido los nombre y perfiles de cinco operaciones. Nótese lo siguiente:

- Hay operaciones 0-arias, es decir, con cero parámetros (como `inicializar`). Estas operaciones se denominan **constantes** del tipo resultado (en el ejemplo, de tipo `tabla`). Su implementación puede consistir en una constante de un tipo predefinido en el lenguaje utilizado, en una función sin parámetros o en un procedimiento con un solo parámetro de salida, al que da valor el procedimiento. En el ejemplo, la constante `inicializar` se implementó (en la lección anterior) de esta última forma.
- La traducción de la notación algebraica (funcional) a la **imperativa** (la que utilizamos habitualmente, dotada de procedimientos y funciones) es normalmente inmediata. Por ejemplo, la operación `total` da lugar a una función con igual perfil. En el caso de la operación `añadir`, se optó en la implementación de la lección anterior por un procedimiento en el que el parámetro `t` de tipo `tabla` es un parámetro de entrada y salida, y por tanto hace un doble papel: el de uno de los parámetros de la aridad y el de resultado.
- La restricción a sólo un resultado por función no es importante. En la práctica, varias operaciones con la misma aridad y distinto resultado pueden combinarse en un solo procedimiento con varios parámetros de salida (correspondiente a los resultados). Por ejemplo, las operaciones `infoEnt` e `infoFrec` se implementaron con el procedimiento `info`.

Veamos, a continuación, una posible signatura de los tipos “booleano” y “naturales con el cero”.

```
espec boolnat
  géneros booleano,natural
  operaciones
    verdad,falso: -> booleano
    ¬_: booleano -> booleano
    _∧_,_∨_: booleano booleano -> booleano
    0,1: -> natural
    suc: natural -> natural
    _+_,_*_: natural natural -> natural
    _≤_,_>_: natural natural -> booleano
fespec
```

En este ejemplo, `verdad`, `falso`, `0` y `1` son constantes. La operación `suc` es prefija, es decir, el nombre de la operación precede a los operandos y éstos van entre paréntesis y separados por comas. Para indicar operaciones prefijas sin paréntesis o infijas, indicaremos mediante el símbolo ‘_’ la posición de los argumentos con respecto al nombre de la operación (ejemplos: `¬_`, `_+_`, …).

Para cada género existe un conjunto de términos bien formados (es decir, sintácticamente correctos) o, simplemente, **términos**. La signatura especifica cómo se construyen los términos. De manera informal, **cada constante es un término y la aplicación de un símbolo de operación a un número apropiado de términos de géneros adecuados es también un término**. En el caso de definir operaciones con notación infija se precisa además la utilización de paréntesis para construir los términos. Los siguientes son tres ejemplos de términos bien formados:

```
1
(suc(1+suc(0))*1)≤1
```

```
((verdad∨falso) ∧ (¬falso)) ∧ (0>suc(suc(1)))
```

A la especificación algebraica de un TAD se le puede atribuir una **semántica** o significado. Dicho significado consiste en considerar que **cada término bien formado denota un valor del tipo** al que pertenece la expresión construida. Por ejemplo, `0`, `1`, `suc(0)`, `0+1`, son valores del tipo `natural`, mientras que `verdad`, `falso`, `¬falso`, `(suc(1+suc(0))*1)≤1`, son valores de tipo `booleano`.

En la denominada **semántica inicial** del TAD natural definido previamente, cada término bien formado de tipo natural denota un valor **diferente** de dicho tipo. Sin embargo, es posible que según la idea intuitiva del programador sobre el tipo que está construyendo, varios términos bien formados diferentes deban corresponder a un mismo valor. Por ejemplo, los términos `1`, `suc(0)`, `0+1`, `1+0` y `1*1`, corresponden a la idea abstracta “uno” que todos conocemos y por tanto deberían tener un mismo significado. Sin embargo, la semántica inicial considera, por defecto, valores distintos aquéllos que se construyen con términos distintos.

Considerando la semántica inicial, la siguiente especificación construye exactamente el tipo de los números naturales (con el cero):

```
espec naturales_1
  género natural
  operaciones
    0: -> natural
    suc: natural -> natural
fespec
```

Los únicos valores que pueden construirse son `0`, `suc(0)`, `suc(suc(0))`, `suc(suc(suc(0)))`, etcétera. Cada término denota un valor diferente, que corresponde a la idea intuitiva de un natural diferente.

Si queremos añadir la operación “suma” a la especificación anterior, puede intentarse en la forma siguiente:

```
espec naturales_2
  género natural
  operaciones
    0: -> natural
    suc: natural -> natural
    _+_: natural natural -> natural
fespec
```

Sin embargo, esta especificación construye un tipo que no corresponde con lo que llamamos “naturales” puesto que, por ejemplo, los términos `suc(0)` y `0+suc(0)` denotan, en principio, valores diferentes (algo contrario a nuestro conocimiento sobre cómo los números naturales deberían comportarse).

La forma de expresar en una especificación que **varios términos corresponden a un mismo valor** y tienen, por tanto, un mismo significado es añadir **ecuaciones**:

```
término_1 = término_2
```

Donde `término_1` y `término_2` son términos bien formados de un mismo género.

Para poder expresar el hecho de que un número grande o infinito de términos bien formados tienen el mismo valor se pueden introducir **variables** en las ecuaciones. Se entiende que en cada ecuación con variables, éstas están (implícitamente) cuantificadas universalmente (∀).

```
espec naturales_3
  género natural
  operaciones
    0: -> natural
    suc: natural -> natural
    _+_: natural natural -> natural
  ecuaciones x,y:natural
    x+0 = x
    x+suc(y) = suc(x+y)
fespec
```

2. Especificación no formal

Las especificaciones no formales de TAD expresan el **dominio de valores** del TAD y la **semántica de las operaciones** con **lenguaje natural**, estableciendo todas las propiedades que las definen, de forma **independiente de cualquier posible representación de los valores o implementación del TAD**.

A pesar de la carencia de formalismo, una especificación no formal de un TAD se considera igualmente un "contrato público" entre programadores, aquéllos que implementan el tipo y los que lo utilizan para resolver problemas.

Por tanto, las especificaciones no formales, al igual que las algebraicas, deben ser precisas, generales, legibles, no ambiguas y definir totalmente el comportamiento del TAD y sus operaciones.

Utilizaremos una sintaxis parecida a la vista para las especificaciones algebraicas, mediante una **notación funcional**:

- Cada operación toma como parámetros 0 o *N* valores (definen el dominio o aridad de la operación). Se denominan constantes las operaciones con 0 parámetros.
- Cada operación produce un solo valor resultado (define el rango o coaridad de la operación).

Las diferencias con respecto a la especificación algebraica son:

- daremos un nombre a cada uno de los parámetros del dominio de cada operación, y
- en lugar de utilizar ecuaciones para expresar la semántica de las mismas, utilizaremos descripciones textuales, es decir, en lenguaje natural.

Una especificación no formal constará de:

- Una **parte sintáctica** denominada signatura, que define
  - los géneros o **nombres de los nuevos tipos**,
  - los **nombres de las operaciones**, y
  - los perfiles de las operaciones, es decir, su dominio (incluidos los nombres de los parámetros) y rango.
- Una **parte semántica**:
  - las descripciones textuales del dominio de valores del TAD y del comportamiento de las operaciones, escritas en lenguaje natural, pero de forma precisa, general, legible, y no ambigua.

En el Anexo 6 incluimos este resumen de la sintaxis de la escritura no formal de especificaciones:

espec nombreEspecificación

[**usa** especificación1, especificación2, …]

[parámetro formal
    género nombreGénero1, …
    [operación
        [parcial] [_]nombreOperación**[_]:** [dominio] **->** rango
            {Descripción del dominio y el rango de la operación, y en su caso de las situaciones que hacen la operación parcial}
        ….
    ]
  fpf]

género nombreGénero1, … *{descripción del dominio de valores del TAD}*

operaciones

    [parcial] nombreOperación**:** [dominio] **->** rango
        {Descripción del dominio y el rango de la operación.
        *[**Parcial:** descripción de las situaciones que hacen la operación parcial]}*
        ...
    [parcial] nombreOperación_**:** géneroArg nombreArg **->** rango
        {Descripción del dominio y el rango de la operación.
        *[**Parcial:** descripción de las situaciones que hacen la operación parcial]}*

...
[parcial] _nombreOperación_ **:**
géneroArg1 nombreArg1**,** géneroArg2 nombreArg2 **->** rango
{Descripción del dominio y el rango de la operación.
*[**Parcial:** descripción de las situaciones que hacen la operación parcial]}*
...
fespec

Donde:

- **[ ]** Significa que lo que está entre los corchetes es **opcional**, puede que aparezca o no.
- **Dominio** es una lista de elementos separados por ',' y donde cada elemento describe un argumento de la operación, indicando el género (o nombre de tipo) y nombre del argumento.
- **Rango** es un género, el nombre del tipo resultado de la operación.
- El símbolo **'_'** indica la **posición de los argumentos** respecto al nombre de la operación. Se utiliza para indicar operaciones con notación prefija sin paréntesis o con notación infija (ejemplos: ¬_ , _≤_).
- El **parámetro formal** opcional que aparece encima de la definición del género lo utilizaremos para definir **TAD genéricos**. Los presentaremos en detalle en la lección siguiente.

Insistimos que en las descripciones (parte semántica, *escrita en color verde*) **no puede aparecer ningún detalle sobre la implementación**, es decir, sobre la forma en que se almacenan los valores del TAD (del estilo de "usaremos un vector para…") ni sobre cómo se deben implementar las operaciones (del estilo de "recorreremos el vector para…").

Nótese que la sintaxis de escritura de especificaciones presentada incluye:

- **descripción del dominio de valores del TAD** (situamos esta descripción al lado del nombre del nuevo género definido);
- **para cada operación**, junto a su perfil, debe describirse completamente:
  - información de entrada y los prerrequisitos que deban cumplirse para usar la operación (**precondición**),
  - comportamiento o efecto de la operación al aplicarse sobre las entradas e indicando qué resultado se genera (**poscondición**);
- **situaciones indeseadas o de error**:
  - cuando existen casos para los cuales no existe un valor válido que pueda representar el resultado de la operación;
  - se considerarán **operaciones parciales** y se indicarán en la especificación.

Por ejemplo, una especificación no formal del TAD pila (genérico, es decir, pilas de datos de un tipo elemento cualquiera) es la siguiente:

espec pilasGenéricas
usa booleanos
parámetro formal
género elemento
fpf
género pila
{Los valores del TAD pila representan secuencias de elementos con acceso LIFO (last in, first out), esto es, el último elemento añadido (o apilado) será el primero en ser borrado (o desapilado)}
operaciones
pilaVacía: -> pila
{Devuelve una pila vacía, sin elementos}

apilar: pila p, elemento e -> pila
{Devuelve la pila resultante de añadir e a p}

desapilar: pila p -> pila
{Si p es no vacía, devuelve la pila resultante de eliminar de p el último elemento que fue apilado.
Si p es vacía, devuelve una pila igual a p}

parcial cima: pila p -> elemento
{Devuelve el último elemento apilado en p.
Parcial: *la operación no está definida si p es vacía}*

esVacía?: pila p -> bool
{Devuelve verdad si y sólo si p no tiene elementos}
fespec

Nótese la necesidad de definir como **parcial** la operación **cima** para evitar la **situación de error** que se produciría al intentar devolver el valor del elemento situado en la cima de una pila vacía, puesto que éste es inexistente.

Es decir, las operaciones parciales permiten describir funciones en las que el dominio pueda ser restringido (no todos los valores del tipo), para evitar situaciones indeseadas o de error.

Veamos en otro ejemplo una posible especificación del TAD fecha, con unas pocas operaciones elementales:

espec fechas
usa enteros, booleanos
género fecha
{Los valores del TAD fecha representan fechas válidas según las reglas del calendario gregoriano.}
operaciones
parcial crear: entero d, entero m, entero a -> fecha
{Dados los tres valores enteros, se obtiene una fecha compuesta con los tres valores dados usados como día, mes y año respectivamente.
Parcial: *$1 \leq d \leq 31$ y $1 \leq m \leq 12$ y además deben formar una fecha válida según el calendario gregoriano.}*

día: fecha f -> entero
{Dada una fecha f, se obtiene el entero que corresponde al día en la fecha f.}

mes: fecha f -> entero
{Dada una fecha f, se obtiene el entero que corresponde al mes en la fecha f.}

año: fecha f -> entero
{Dada una fecha f, se obtiene el entero que corresponde al año en la fecha f.}

iguales: fecha f1, fecha f2 -> booleano
{Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y sólo si la fecha f1 es igual que la fecha f2, es decir, corresponden al mismo día, mes y año.}

anterior: fecha f1, fecha f2 -> booleano
{Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y sólo si la fecha f1 es cronológicamente anterior a la fecha f2.}

posterior: fecha f1, fecha f2 -> booleano
{Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y sólo si la fecha f1 es cronológicamente posterior a la fecha f2.}

fespec

Finalmente, un ejemplo de especificación del TAD tabla de frecuencias de enteros que aparece en el ejercicio de la lección anterior:

espec tablas
usa naturales, enteros
género tabla
{Los valores del TAD tablas de frecuencia representan colecciones de números enteros tales que:
- no se almacenan enteros repetidos, pero si se registra cuántas veces se ha introducido cada entero (su frecuencia)

- las operaciones permiten obtener la información de un entero o su frecuencia según su puesto en el orden decreciente por valores de frecuencia}

operaciones

inicializar: -> tabla

{Devuelve una tabla vacía, es decir, que no contiene datos para ningún número entero}

añadir: tabla t, entero e -> tabla

{Si e no está t, devuelve la tabla resultante de añadir e a t con número de apariciones igual a 1. Si e está en t, devuelve la tabla resultante de incrementar en 1 el número de apariciones de e (su frecuencia) en t}

total: tabla t -> natural

{Devuelve el número total de enteros para los que t contiene información}

parcial infoEnt: tabla t, natural n -> entero

{Devuelve el entero que corresponde al n-ésimo entero en la tabla t según el orden en número de apariciones decreciente.
Parcial: la operación no está definida si total(t)<n}

parcial infoFrec: tabla t, natural n -> natural

{Devuelve el natural que corresponde al número de apariciones del n-ésimo entero en la tabla t según el orden en número de apariciones decreciente.
Parcial: la operación no está definida si total(t)<n}

fespec

Lección 3

Implementación de TAD

Indice

1. Características de la implementación de un TAD
2. Implementación modular
   2.1. Pseudocódigo
   2.2. C++

1. Características de la implementación de un TAD

Como ya dijimos anteriormente, la implementación de un TAD consiste en:

- elegir la **representación de los valores del tipo**, usando otros TAD previamente definidos e implementados y/o tipos o constructores de tipos predefinidos en el lenguaje de implementación, e
- **implementar las operaciones**.

Una buena implementación debe tener las siguientes características:

- Estructurada, para facilitar su desarrollo.
- Eficiente, para optimizar el uso de recursos: tiempo, espacio.
- Legible, para facilitar su modificación y mantenimiento.
- Segura y robusta.
- Correcta, verificable, fácil de usar.
- Garantizar la encapsulación.

Como ya se dijo, la **encapsulación** es el concepto fundamental subyacente bajo la programación con TAD. La encapsulación consiste básicamente en garantizar:

- la **privacidad** de la representación: el usuario del TAD no conoce los detalles de representación de los valores del TAD, o no necesita conocerlos, y
- la **protección** del tipo: el usuario del TAD sólo puede utilizar las operaciones previstas y hechas públicas en la interfaz del TAD.

2. Implementación modular

En estos apuntes se va a implementar los TAD con módulos y no con clases (propias de la Programación Orientada a Objetos), así que presentamos en primer lugar nuestra sintaxis en pseudocódigo para esos módulos y posteriormente una posible codificación en C++ (otras variantes son posibles en el mismo lenguaje).

2.1. Pseudocódigo

Un buen lenguaje de programación con TAD debe facilitar la encapsulación. Para ello, debe separar la interfaz, o parte pública, que incluya la declaración del TAD, de la implementación, o parte privada, que encapsule los detalles de representación de los valores del tipo e implementación de las operaciones.

En el caso de tener que utilizar un lenguaje que no garantice la encapsulación, es responsabilidad exclusiva del programador el cumplir con la encapsulación, cumpliendo las restricciones de acceso a los detalles de implementación que un buen lenguaje de programación con TAD garantizaría.

En estos apuntes utilizaremos un **pseudocódigo en castellano** (ver el Anexo 6) en el que los módulos para implementar los TAD tendrán la siguiente sintaxis:

```
módulo <nombre del módulo>
importa <lista de módulos que necesita usar>
exporta
  {parte pública: definición de constantes, nombres de tipos, encabezamientos
   de procedimientos y funciones}
  ...
implementación
  {parte privada: se incluyen las definiciones de los tipos cuyos nombres aparecen
   en la parte pública, otros tipos privados, el código de procedimientos
   y funciones...}
  ...
fin
```

De esta forma, para implementar un TAD, en primer lugar, hay que definir todo lo que aparecerá en la **parte pública** o **interfaz** del módulo, es decir, lo que el módulo **exporta**:

- los identificadores válidos de constantes (si son necesarias), tipos, y operaciones (procedimientos y/o funciones);
- los perfiles o cabeceras de cada operación (sea procedimiento o función): parámetros de entrada, parámetros de salida y/o parámetros de entrada y salida; y
- la comunicación de las situaciones de error (en estos apuntes utilizaremos para ello más parámetros de salida).

Una vez decidida la parte pública, se pueden realizar independientemente (por parte de distintos programadores incluso) la implementación del TAD y su utilización en otros módulos o en programas principales.

En segundo lugar, y ya en lo referido a la implementación del módulo, hay que decidir la **representación interna** del TAD, es decir, **cómo representar los valores del tipo** de datos especificado, basándose en: tipos básicos predefinidos, constructores básicos predefinidos (como vectores y registros), y/u otros TAD definidos previamente.

La representación interna deberá permitir implementar las operaciones definidas para el tipo de forma eficiente, tanto en relación con su coste en memoria como en tiempo.

Además, la representación interna deberá permanecer oculta, encapsulada. Es decir, el uso del nuevo tipo solo será posible mediante las operaciones definidas en la interfaz del tipo.

En tercer lugar, también dentro de la parte de implementación del módulo, hay que **implementar cada operación** de la interfaz del TAD, de acuerdo a la representación interna definida para los valores del TAD, y las operaciones auxiliares que resulten de interés o de utilidad (siendo éstas inaccesibles para los programadores usuarios del módulo).

Para la implementación de cada operación, las operaciones 0-arias o constantes se pueden implementar, según convenga, como constantes predefinidas en algún tipo de dato del lenguaje, como procedimientos o como funciones sin parámetros. Las demás operaciones se implementan como procedimientos o funciones, según convenga.

Como se ha dicho anteriormente, varias operaciones con el mismo dominio y distinto rango o resultado pueden combinarse en un solo procedimiento con varios parámetros de salida, correspondiente a los resultados. Esta posibilidad es especialmente recomendable si esas operaciones van a utilizarse a menudo de forma conjunta y la implementación conjunta reduce el coste en tiempo de obtener los resultados.

En lo concerniente a las operaciones parciales (es decir, con situaciones de error), cabe la posibilidad de utilizar los mecanismos de manejo de excepciones del lenguaje de implementación, si los tiene. En estos apuntes, no vamos a entrar en el uso de esos posibles mecanismos, sino que utilizaremos el método, más rudimentario pero utilizable en cualquier lenguaje de programación, de añadir parámetros de salida sobre el error ocurrido a los procedimientos.

Como regla general, si una operación sólo tiene que devolver un dato resultado, podrá implementarse tanto con un procedimiento (incluyendo el correspondiente parámetro de salida, o eventualmente de entrada y salida) como con una función (que devuelve el resultado asociado a su nombre). Por el contrario, si una operación tiene que devolver 0, 2 o más

resultados, tendrá que optarse por un procedimiento que incluya tantos parámetros de salida, o eventualmente de entrada y salida, como resultados.

En cuanto a las implicaciones que el utilizar un parámetro de entrada y otro de salida o un único parámetro de entrada y salida tiene en el almacenamiento de los datos en memoria, se puede decidir que:

a) el resultado sea la actualización de uno de los parámetros del dominio:
   - almacenamos el parámetro del dominio de entrada y el resultado en un único parámetro de entrada y salida,
   - sólo posible con procedimientos (puesto que vamos a suponer que las funciones sólo tienen parámetros de entrada),
   - se evita ocupar nueva memoria para los datos resultado y el tiempo de copiar toda la parte de los datos que no resulta modificada (es más eficiente, en tiempo y memoria),
   - combinando su uso con el de una operación de copiar (o duplicar), siempre se podrán generar nuevas copias separadas de los datos (valores previo y posterior a la modificación por la operación).

b) el resultado sea una copia distinta en memoria del parámetro del dominio:
   - el parámetro del dominio será de entrada y el resultado será de salida (o el valor devuelto por una función),
   - se ocupa memoria adicional, independiente, para dato y resultado,
   - combinando su uso con el de una operación copiar (o duplicar), siempre se podrá hacer que el nuevo valor sustituya al original (en memoria).

Veamos como ejemplo la implementación del TAD fecha especificado en una lección anterior.

```
módulo fechas
exporta
  tipo fecha
       {Los valores del TAD fecha representan fechas válidas según las reglas
        del calendario gregoriano.}

  procedimiento crear(ent d,m,a:entero; sal f:fecha; sal error:booleano)
  {Dados los tres valores enteros d,m,a, si forman una fecha válida según el calendario
   gregoriano, se devuelve en f la fecha compuesta con los tres valores dados usados
   como día, mes y año respectivamente, y error devuelve falso.
   Si d,m,a no forman una fecha válida, error devuelve verdad.}

  función día(f:fecha) devuelve entero
  {Dada una fecha f, se obtiene el entero que corresponde al día en la fecha f.}

  función mes(f:fecha) devuelve entero
  {Dada una fecha f, se obtiene el entero que corresponde al mes en la fecha f.}

  función año(f:fecha) devuelve entero
  {Dada una fecha f, se obtiene el entero que corresponde al año en la fecha f.}

  función iguales(f1,f2:fecha) devuelve booleano
  {Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y sólo si la
   fecha f1 es igual que la fecha f2, es decir, corresponden al mismo día, mes y
   año.}

  función anterior(f1,f2:fecha) devuelve booleano
  {Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y sólo si la
   fecha f1 es cronológicamente anterior a la fecha f2.}
  función posterior(f1,f2:fecha) devuelve booleano
  {Dadas dos fechas f1 y f2, se obtiene un booleano con valor verdad si y sólo si la
   fecha f1 es cronológicamente posterior a la fecha f2.}
```

```
implementación
  tipo fecha = registro
                 elDía,elMes,elAño:entero
               freg

  procedimiento crear(ent d,m,a:entero; sal f:fecha; sal error:booleano)
  principio
    si d<1 or d>31 or m<1 or m>12 or a<1583 or
       (d=31 and (m=2 or m=4 or m=6 or m=9 or m=11)) or (m=2 and d=30) entonces
      error:=verdad
    sino
      si m=2 and d=29 and
         ((a mod 4/=0) or (a mod 4=0 and a mod 100=0 and a mod 400/=0)) entonces
        error:=verdad
      sino
        f.elDía:=d;
        f.elMes:=m;
        f.elAño:=a;
        error:=falso
      fsi
    fsi
  fin

  función día(f:fecha) devuelve entero
  principio
    devuelve f.elDía
  fin

  función mes(f:fecha) devuelve entero
  principio
    devuelve f.elMes
  fin

  función año(f:fecha) devuelve entero
  principio
    devuelve f.elAño
  fin

  función iguales(f1,f2:fecha) devuelve booleano
  principio
    devuelve ((f1.elAño=f2.elAño) and (f1.elMes=f2.elMes) and (f1.elDía=f2.elDía))
  fin

  función anterior(f1,f2:fecha) devuelve booleano
  principio
    devuelve (f1.elAño<f2.elAño) or
             ((f1.elAño=f2.elAño) and (f1.elMes<f2.elMes)) or
             ((f1.elAño=f2.elAño) and (f1.elMes=f2.elMes) and (f1.elDía<f2.elDía))
  fin

  función posterior(f1,f2:fecha) devuelve booleano
```

```
  principio
    devuelve not( iguales(f1,f2) or anterior(f1,f2) )
  fin
fin
```

Como comentarios adicionales:

- Dada una especificación de TAD hay muchas implementaciones válidas.
- Un cambio de implementación de un TAD debe ser transparente a los programas que lo utilizan.
- Cuando se implementa un TAD, se está construyendo una interpretación de la especificación. La implementación de un TAD debe corresponderse con la especificación, manteniendo sus propiedades, si es posible sin introducir ***"basura"*** ni ***"confusión"***.
    - Basura: la representación de datos elegida para el TAD hace que sean representables más valores de los especificados (llamados basura).
    - Confusión: la representación de datos elegida para el TAD hace que varios de los valores especificados tengan una misma representación (se confunden).

  Si no queda más remedio que introducir basura o confusión, deberán estar documentadas para que quien use la implementación del TAD sepa exactamente qué se le está ofreciendo.

Ejemplos de basura y confusión para el TAD fecha:

- Basura: que un dato fecha pueda tomar valores de fechas no válidas: 31-2-2011, 2-15-2011…
- Confusión: que varios valores válidos se representen exactamente igual y por tanto sean indistinguibles: 31-1-1920 y 31-1-2020 representados ambos como 31-1-20.

Hay veces que la confusión es inevitable, pero como se ha dicho debe estar documentada. Por ejemplo, si se trata de representar un dominio de valores de cardinal infinito, como los números enteros, utilizando una representación en memoria acotada (por ejemplo de 4 bytes), lo cual es inevitable.

Como último ejemplo, veamos la implementación del TAD tabla de frecuencias mencionado en el ejercicio final de la lección 1 y especificado en la lección 2.

```
módulo tablas
exporta

  constante maxnumdatos = 1000

  tipo tabla
       {Tabla de frecuencias de enteros según el enunciado de la lección 1.
        Implementación limitada a tablas con un tamaño máximo de maxnumdatos enteros
        distintos.}

  procedimiento inicializar(sal t:tabla)
  {Crea una tabla vacía t de frecuencias}

  procedimiento añadir(e/s t:tabla; ent e:entero; sal error:booleano)
  {Modifica t incrementando en 1 la frecuencia de e.
   La implementación limita a maxnumdatos el nº de datos distintos,
   por tanto, si e no cabe en la tabla, devuelve error=verdad.}

  función total(t:tabla) devuelve natural
  {Devuelve el nº de enteros distintos en la tabla t.}

  procedimiento info(ent t:tabla; ent n:natural;
                     sal e:entero; sal m:natural; sal error:booleano)
  {Devuelve en e el entero que ocupa el n-ésimo lugar en la tabla t,
   en orden de frecuencias decrecientes, y m es su frecuencia.
   Si no existe ese entero (i.e. total(t)<n), devuelve 0 en ambos y error=verdad.}

implementación

  tipos
    dato = registro
             número:entero;
```

```
          frecuencia:natural
        freg;
  elementos = vector[1..maxnumdatos] de dato
  tabla = registro
            elmto:elementos;
            total:natural
          freg
  {Guarda los datos ordenados por frecuencias de mayor a menor.}

procedimiento inicializar(sal t:tabla)
{Crea una tabla vacía t de frecuencias}
principio
  t.total:=0
fin

procedimiento añadir(e/s t:tabla; ent e:entero; sal error:booleano)
{Modifica t incrementando en 1 la frecuencia de e.
 La implementación limita a maxnumdatos el nº de datos distintos,
 por tanto, si e no cabe en la tabla, devuelve error=verdad.}
variables
  i:natural:=0;
  éxito:booleano:=falso;
  aux:entero
principio
  {buscar e en t}
  mientrasQue not éxito and i<t.total hacer
    i:=i+1;
    éxito:=t.elmto[i].número=e
  fmq;
  si éxito entonces
  {e ya estaba -> incrementar su frecuencia y recolocarlo}
    t.elmto[i].frecuencia:=t.elmto[i].frecuencia+1;
    mientrasQue i>1 andThen t.elmto[i].frecuencia>t.elmto[i-1].frecuencia hacer
      aux:=t.elmto[i].número;
      t.elmto[i].número:=t.elmto(i-1).número;
      t.elmto[i-1].número:=aux;
      aux:=t.elmto[i].frecuencia;
      t.elmto[i].frecuencia:=t.elmto[i-1].frecuencia;
      t.elmto[i-1].frecuencia:=aux
    fmq;
    error:=falso
  sino
  {e no estaba -> añadirlo al final de la tabla, si cabe}
    si t.total<maxnumdatos entonces
      t.total:=t.total+1;
      t.elmto[t.total].número:=e;
      t.elmto[t.total].frecuencia:=1;
      error:=falso
    sino
      error:=verdad
    fsi
  fsi
fin

función total(t:tabla) devuelve natural
{Devuelve el nº de enteros distintos en la tabla t.}
principio
  devuelve t.total
fin

procedimiento info(ent t:tabla; ent n:natural;
                   sal e:entero; sal m:natural; sal error:booleano)
{Devuelve en e el entero que ocupa el n-ésimo lugar en la tabla t,
 en orden de frecuencias decrecientes, y m es su frecuencia.
 Si no existe ese entero (i.e. total(t)<n), devuelve 0 en ambos y error=verdad.}
principio
  si n<=total(t) entonces
    e:=t.elmto[n].número;
    m:=t.elmto[n].frecuencia;
```

```
      error:=falso
    sino
      e:=0;
      m:=0;
      error:=verdad
    fsi
  fin

fin
```

2.2. C++

Para implementar un TAD en C++ hay esencialmente dos posibilidades: utilizar el concepto de **clase** de la programación orientada a objetos e implementar cada TAD en una clase, o bien utilizar **registros** o tuplas (constructor de tipos ***struct***). Dado que en las asignaturas anteriores todavía no se ha estudiado la orientación a objetos y que ese es uno de los objetivos de la asignatura de programación siguiente a ésta en el plan de estudios, optaremos por la representación con registros.

La representación de los datos en el registro será privada (***private***) y para poder acceder a ella las operaciones del TAD serán funciones amigas (***friend***).

Las operaciones que en pseudocódigo aparecen como procedimientos se pueden codificar con funciones ***void***. Las operaciones que en pseudocódigo aparecen como funciones se pueden codificar con funciones que devuelven un dato.

Los **parámetros de entrada** se pueden codificar con

- parámetros de entrada (**transmisión por valor**), útiles si ocupan poca memoria;
- parámetros **constantes transmitidos por referencia**, útiles si ocupan mucha memoria.

Los **parámetros de salida** o **de entrada y salida** se codifican con parámetros **transmitidos por referencia**.

Veamos en primer lugar una codificación en C++ del ejemplo del TAD fecha.

Fichero fecha.h:

```
#ifndef _FECHA_H
#define _FECHA_H

// Interfaz del TAD fecha. Pre-declaraciones:

/* Los valores del TAD fecha representan fechas válidas
 * según las reglas del calendario gregoriano (adoptado en 1583) */
struct Fecha;

/* Dados los tres valores enteros dia, mes y anyo, se devuelve en f
 * la fecha compuesta por ellos.
 * Parcial: se precisa que 1≤dia≤31, 1≤mes≤12, 1583≤anyo, y además
 * que dia, mes y anyo formen una fecha válida según el calendario
 * gregoriano; de lo contrario, error devuelve el valor falso */
void crear(int dia, int mes, int anyo, Fecha& f, bool& error);

/* Devuelve el dia de la fecha */
int dia(const Fecha& f);

/* Devuelve el mes de la fecha */
int mes(const Fecha& f);

/* Devuelve el año de la fecha */
int anyo(const Fecha& f);

/* Devuelve verdad si y sólo si f1 y f2 son la misma fecha */
bool iguales(const Fecha& f1, const Fecha& f2);

/* Devuelve verdad si y sólo si la fecha f1 es cronológicamente
 * anterior a la fecha f2 */
bool anterior(const Fecha& f1, const Fecha& f2);
```

```
/* Devuelve verdad si y sólo si la fecha f1 es cronológicamente
 * posterior a la fecha f2 */
bool posterior(const Fecha& f1, const Fecha& f2);
// Declaración

struct Fecha {
     friend void crear(int dia, int mes, int anyo, Fecha& f, bool& error) ;
     friend int dia(const Fecha& f);
     friend int mes(const Fecha& f);
     friend int anyo(const Fecha& f);
     friend bool iguales(const Fecha& f1, const Fecha& f2) ;
     friend bool anterior(const Fecha& f1, const Fecha& f2);
     friend bool posterior(const Fecha& f1, const Fecha& f2);

     private:
     // Representación de los valores del TAD.
       int elDia;
       int elMes;
       int elAnyo;
};

#endif
```

Fichero fecha.cpp:

```
#include "fecha.h"

// Implementacion de las operaciones del TAD fecha.

   // Operaciones auxiliares sobre enteros.

   // Devuelve verdad si y sólo si el año a es bisiesto.
   bool esBisiesto(int a) {
     return (a % 4 == 0 && !(a % 100 == 0 && a % 400 == 0));
   }

   // Devuelve verdad si y sólo si (d,m,a) representan una fecha válida.
   bool esFechaValida(int d, int m, int a) {
     bool valida = 1583 <= a && 1 <= m && m <= 12;
     if (valida) {
       switch (m) {
         case 4: case 6: case 9: case 11:
           valida = 1 <= d && d <= 30;
           break;
         case 1: case 3: case 5: case 7: case 8: case 10: case 12:
           valida = 1 <= d && d <= 31;
           break;
         case 2:
           if (!esBisiesto(a)) {
             valida = 1 <= d && d <= 28;
           }
           else
             valida = 1 <= d && d <= 29;
           break;
       }
     }
     return valida;
   }

   void crear(int dia, int mes, int anyo, Fecha& f, bool& error) {
     if (esFechaValida(dia, mes, anyo)) {
       f.elDia = dia;
       f.elMes = mes;
       f.elAnyo = anyo;
       error = false;
     }
     else
       error = true;
```

```
    };

    int dia(const Fecha& f) {
        return f.elDia;
    };

    int mes(const Fecha& f) {
        return f.elMes;
    };

    int anyo(const Fecha& f) {
        return f.elAnyo;
    };

    bool iguales(const Fecha& f1, const Fecha& f2) {
        return f1.elDia == f2.elDia && f1.elMes == f2.elMes &&
                                                  f1.elAnyo == f2.elAnyo;
    };

    bool anterior(const Fecha& f1, const Fecha& f2) {
        return f1.elAnyo <  f2.elAnyo ||
           (f1.elAnyo == f2.elAnyo && f1.elMes < f2.elMes) ||
           (f1.elAnyo == f2.elAnyo && f1.elMes == f2.elMes && f1.elDia < f2.elDia);
    };

    bool posterior(const Fecha& f1, const Fecha& f2) {
        return !iguales(f1,f2) && !anterior(f1,f2);
    };
```

Y finalmente, para el ejemplo de las tablas de frecuencias.

Fichero tabla_frec.h:

```
#ifndef _TABLA_FREC_H
#define _TABLA_FREC_H

// Interfaz del TAD tabla de frecuencias. Pre-declaraciones:

const int MAX_NUM_DATOS = 1000;

struct Tabla;

void inicializar(Tabla& t);
bool anyadir(Tabla& t, int n);
int total(const Tabla& t);
int infoEnt(const Tabla& t, int n);
int infoFrec(const Tabla& t, int n);

// Declaración

struct Tabla {
    friend void inicializar(Tabla& t);
    friend bool anyadir(Tabla& t, int n);
    friend int total(const Tabla& t);
    friend int infoEnt(const Tabla& t, int n);
    friend int infoFrec(const Tabla& t, int n);

// Representación de los valores del TAD
    private:
        struct Frecuencia {
                    int numero;
                    int frec;
        };

        Frecuencia elementos [MAX_NUM_DATOS];
        int numElementos;
};
```

```
#endif
```

Fichero tabla_frec.cpp:

```
#include "tabla_frec.h"

void inicializar(Tabla& t){
    t.numElementos = 0;
};

bool anyadir(Tabla& t, int n){
    //encontrar n en tabla
    int pos = 0;
    bool encontrado = false;
    while (!encontrado && pos < t.numElementos){
        if (t.elementos[pos].numero == n){
            encontrado = true;
        }
        else{
            pos = pos + 1;
        }
    }
    if (!encontrado){
        pos = - 1;
    }
    bool error = false;
    //si está n en tabla, recolocar
    if (pos != -1) {
        t.elementos[pos].frec = t.elementos[pos].frec + 1;
        while (pos > 0 && t.elementos[pos].frec > t.elementos[pos-1].frec) {
            Frecuencia aux = t.elementos[pos];
            t.elementos[pos] = t.elementos[pos-1];
            t.elementos[pos-1] = aux;
            pos = pos - 1;
        }
    }
    else { //si no esá, introducir al final si se puede
        if (t.numElementos < MAX_NUM_DATOS) {
            t.elementos[t.numElementos].numero = n;
            t.elementos[t.numElementos].frec = 1;
            t.numElementos = t.numElementos + 1;
        }
        else {
            error = true;
        }
    }
    return error;
};

int total(const Tabla& t) {
    return t.numElementos;
};

int infoEnt(const Tabla& t, int n) {
    if (1 <= n && n <= t.numElementos) {
        return t.elementos[n-1].numero;
    } else {
        return 0;
    }
};

int infoFrec(const Tabla& t, int n) {
    if (1 <= n && n <= t.numElementos) {
        return t.elementos[n-1].frec;
    } else {
        return 0;
    }
};
```

Lección 4

TAD genéricos

Indice

1. Concepto de genericidad
2. TAD genéricos
3. Implementación en C++

1. Concepto de genericidad

La genericidad es un mecanismo que permite escribir fragmentos (subprogramas, módulos, clases…) de **código genérico**, es decir, código que incluye referencias a uno o varios nombres de tipos de datos (o clases) e incluso a nombres de procedimientos o funciones que manipulan valores de esos tipos, sin que exista una declaración de los mismos, es decir, sin concretar qué tipos, qué procedimientos o qué funciones son los utilizados. Esos nombres de tipos, de procedimientos o de funciones se denominan parámetros de tipo, de procedimiento o de función. Además, es posible especificar restricciones para los parámetros de tipo del código genérico, como, por ejemplo, que el tipo en el que se concrete luego ese parámetro sea un tipo discreto.

Particularizar (o concretar) el código genérico consiste en indicar los tipos concretos (previamente definidos) en los que se convierten los parámetros de tipos. Si además de parámetros de tipo hay parámetros de procedimiento o de función, es necesario también indicar los procedimientos o funciones concretos (previamente implementados) en los que se convierten los parámetros de procedimiento o función.

Con la particularización del código genérico, se obtiene **código concreto**, que puede ser ejecutado.

En un lenguaje de alto nivel que incorpora adecuadamente la genericidad, el compilador es capaz de generar **código objeto genérico**. Y es en la fase de edición de enlaces (*linking*) en la que se realiza la sustitución de los nombres de los parámetros de tipo (o de procedimiento o función) por los nombres de los tipos (procedimientos o funciones) concretos, dando lugar a código ejecutable. No obstante, la variedad de paradigmas y lenguajes existentes ha dado lugar a muy diversas formas de implementar la genericidad.

La genericidad facilita la **reutilización** de algoritmos. Por ejemplo, si se implementa un algoritmo de ordenación de vectores por el método de ordenación rápida de Hoare (*quicksort*), los tipos de datos de los índices y de los elementos del vector no afectan al algoritmo, siempre que se disponga de una función de orden que diga si un elemento es mayor que otro. Un tipo genérico *vector* puede especificarse así:

```
espec vectoresGenéricos
  usa booleanos
  parámetros formales
    géneros índice, elemento {dos parámetros de tipo: el tipo de los índices y el de los datos del vector}
    operación
      _>_: elemento e1, elemento e2 -> booleano { “>” es una relación de orden }
  fpf
  género vector {vector es una colección de datos de tipo elemento indexados con los valores del tipo índice}
  operaciones
    modifica: vector v, índice i, elemento e -> vector
    {Devuelve el vector resultante de modificar v asignando el valor e a la componente de índice i de v.}

    valor: vector v, índice i -> elemento
    {Devuelve el valor del elemento de índice i en el vector v.
      Si al elemento de índice i en v no se le ha asignado previamente ningún valor con la operación modifica,
      entonces devuelve un valor cualquiera del tipo elemento.}
fespec
```

Un **algoritmo** se denomina **genérico** si manipula TAD genéricos. Por ejemplo, puede desarrollarse el algoritmo de ordenación rápida (*quicksort*) para vectores genéricos. Bastará con concretar los géneros *índice* y *elemento* y la operación ">" para contar con un ejemplar del TAD genérico *vector* (TAD concreto) y con un algoritmo concreto de ordenación para ese tipo.

Algunos lenguajes imperativos (por ejemplo, Ada) soportan la genericidad. A continuación, y a título de ejemplo, se incluye un algoritmo genérico de ordenación rápida en Ada.

```
generic -- módulo de declaración; este módulo genérico ofrece un procedimiento de
        -- ordenación de vectores genéricos
  type indice is (<>);      -- cualquier tipo discreto (es un parámetro de tipo)
  type elemento is private; -- cualquier tipo (es otro parámetro de tipo)
  type vector is array(indice range <>) of elemento;
  with function ">"(a,b:elemento) return boolean;
package ordenacion_g is
  procedure ordena(v:in out vector);
end; -- del módulo de declaración

package body ordenacion_g is -- módulo de implementación

  procedure ordena(v:in out vector) is
    i,j:indice; m,t:elemento; n:integer;
  begin
    -- inicialización
    i:=v'first;
    j:=v'last;
    n:=indice'pos(i);
    n:=n+indice'pos(j);
    n:=n/2;
    m:=v(indice'val(n));
    -- partición del vector en dos
    while i<=j loop
      while m>v(i) loop
        i:=indice'succ(i);
      end loop;
      while v(j)>m loop
        j:=indice'pred(j);
      end loop;
      if i<=j then
        t:=v(i);
        v(i):=v(j);
        v(j):=t;
        i:=indice'succ(i);
        j:=indice'pred(j);
      end if;
    end loop;
    -- recursión en los dos subvectores
    if v'first<j then
      ordena(v(v'first..j));
    end if;
    if i<v'last then
      ordena(v(i..v'last));
    end if;
  end ordena;

end ordenacion_g; -- del módulo de implementación
```

La utilización de un ejemplar concreto del algoritmo genérico anterior se puede realizar de la siguiente forma:

```
with ordenacion_g;
procedure titi is
  type color is (rojo,azul,gris);
  type dia is (lu,ma,mi,ju,vi,sa,do);
  type vect is array(dia range <>) of color;
  x:vect(ma..vi):=(gris,azul,rojo,gris);
  package o is new ordenacion_g(dia,color,vect,">");
begin
  ...
```

```
    o.ordena(x);
    ...
  end titi;
```

2. TAD genéricos

En el diseño descendente por refinamientos sucesivos basado en la abstracción de acciones se utiliza la parametrización de procedimientos y funciones (parámetros formales que en cada ejecución se concretan en los parámetros reales).

De forma análoga, en el diseño basado en la abstracción de datos se pueden definir **TAD genéricos** (o tipos parametrizados) con algunas características indefinidas. Esas características pueden ser concretadas posteriormente de diversas formas según las necesidades, obteniendo **ejemplares** de **TAD concretos** distintos.

Las características indefinidas son **parámetros formales** (igual que en los procedimientos y funciones) que pueden corresponder a otros tipos, operaciones y constantes. Los parámetros formales se concretan mediante **argumentos** o parámetros reales (otros tipos, operaciones o constantes).

Ejemplo sencillo que generaliza (haciéndolo más reutilizable) el TAD conjunto de caracteres visto en una lección anterior:

espec conjuntosGenéricos
usa booleanos,naturales
parámetro formal
  género elemento *{es un parámetro de tipo, es decir, se trata del nombre de un tipo cualquiera}*
fpf
género conjgen
{Los valores del TAD conjgen representan conjuntos de elementos, sin elementos repetidos, y en los que no es relevante el orden en el que los elementos se añaden al conjunto.}

operaciones
  vacío: -> conjgen
  {Devuelve un conjunto de elementos vacío, es decir un conjunto que no contiene ningún elemento.}

  poner: elemento e, conjgen c -> conjgen
  *{Si e está en c, devuelve un conjunto igual a c;
  si e no está en c, devuelve el conjunto resultante de añadir e a c.}*

  quitar: elemento e, conjgen c -> conjgen
  *{Si e está en c, devuelve el conjunto resultante de eliminar e de c;
  si e no está en c, devuelve un conjunto igual a c.}*

  _∪_: conjgen c1, conjgen c2 -> conjgen
  {Devuelve un conjunto que contiene todos los caracteres que están en c1 y todos los que están en c2.}

  _∩_: conjgen c1, conjgen c2 -> conjgen
  {Devuelve un conjunto que contiene únicamente los caracteres que están tanto en c1 como en c2.}

  _∈_: elemento e, conjgen c -> booleano
  {Devuelve verdad si y sólo si e está en c.}

  esVacío: conjgen c -> booleano
  {Devuelve verdad si y sólo si c no contiene ningún elemento.}

  cardinal: conjgen c -> natural
  {Devuelve el número total de elementos que contiene c (0 si es vacío)}
fespec

Supongamos que en el desarrollo del software de una máquina expendedora de fruta fresca llegamos a la conclusión de que resultará útil especificar e implementar un TAD para gestionar la información de una colección de monedas, o monedero, y otro para gestionar la información de una colección de frutas, o frutero.

En primer lugar, tendremos que especificar el TAD moneda:

espec monedas
usa naturales
género moneda
{Los valores del TAD moneda representan valores posibles de una moneda}

operaciones
c1: -> moneda
{devuelve una moneda de 1 céntimo}

c10: -> moneda
{devuelve una moneda de 10 céntimos}

c50: -> moneda
{devuelve una moneda de 50 céntimos}

e1: -> moneda
{devuelve una moneda de 1 euro}

precio: moneda m -> natural
{devuelve el valor de la moneda m en céntimos}
fespec

Para, a continuación, especificar las colecciones de monedas con las operaciones que se ha decidido definir:

espec monederos
usa monedas, naturales
género monedero
{Los valores del TAD monedero representan valores posibles de un multiconjunto (saco) de monedas}

operaciones
vacio: -> monedero
{devuelve un monedero vacío, sin monedas}

meter: monedero s, moneda m -> monedero
{devuelve el monedero resultante de añadir la moneda m a s}

sacar: monedero s, moneda m -> monedero
*{devuelve el monedero resultante de extraer la moneda m de s;
si no hay ninguna moneda m en s, devuelve un monedero igual a s}*

cuántas: monedero s, moneda m -> natural
{devuelve el nº de monedas de valor m en s}

valor: monedero s -> natural
{devuelve la suma de los valores de todas las monedas de s}
fespec

El pseudocódigo resultante (una posible implementación de los TAD moneda y monedero) sería:

```
módulo monedas
exporta
  tipo moneda=(c1,c10,c50,e1)
  función precio(m:moneda) devuelve natural
implementación
  función precio(m:moneda) devuelve natural
  principio
    selección
      m=c1: devuelve 1;
```

```
      m=c10: devuelve 10;
      m=c50: devuelve 50;
      m=e1: devuelve 100;
    fselec
  fin
fin

módulo monederos
importa monedas
exporta
  tipo monedero
  procedimiento vacio(sal s:monedero)
  procedimiento meter(e/s s:monedero; ent m:moneda)
  procedimiento sacar(e/s s:monedero; ent m:moneda)
  función cuántas(s:monedero; m:moneda) devuelve natural
  función valor(s:monedero) devuelve natural
implementación
  tipo monedero=vector[moneda] de natural

  procedimiento vacio(sal s:monedero)
  variable m:moneda
  principio
    para m:=c1 hasta e1 hacer
      s[m]:=0
    fpara
  fin

  procedimiento meter(e/s s:monedero; ent m:moneda)
  principio
    s[m]:=s[m]+1
  fin

  procedimiento sacar(e/s s:monedero; ent m:moneda)
  principio
    si s[m]>0 entonces
      s[m]:=s[m]-1
    fsi
  fin

  función cuántas(s:monedero; m:moneda) devuelve natural
  principio
    devuelve s[m]
  fin

  función valor(s:monedero) devuelve natural
  variables v:natural; m:moneda
  principio
    v:=0;
    para m:=c1 hasta e1 hacer
      v:=v+cuántas(s,m)*precio(m)
    fpara;
    devuelve v
```

```
  fin
fin
```

A continuación, hagamos lo mismo con los TAD fruta y frutero (colección de frutas).
Sus especificaciones:

espec frutas
usa naturales
género fruta
{Los valores del TAD fruta representan valores posibles de una fruta}
operaciones
pera: **->** fruta
{devuelve una pera}
manzana: **->** fruta
{devuelve una manzana}
limón: **->** fruta
{devuelve un limón}
pomelo: **->** fruta
{devuelve un pomelo}
papaya: **->** fruta
{devuelve una papaya}
precio: fruta f **->** natural
{devuelve el valor de la fruta f}
fespec

espec frutero
usa frutas, naturales
género frutero
{Los valores del TAD frutero representan valores posibles de un multiconjunto (saco) de frutas}
operaciones
vacio: **->** frutero
{devuelve un frutero vacío, sin frutas}
meter: frutero s, fruta f **->** frutero
{devuelve el frutero resultante de añadir la fruta f a s}
sacar: frutero s, fruta f **->** frutero
{devuelve el frutero resultante de extraer la fruta f de s;
si no hay ninguna fruta f en s, devuelve un frutero igual a s}
cuántas: frutero s, fruta f **->** natural
{cuenta cuántas frutas de valor f hay en s}
valor: frutero s **->** natural
{devuelve la suma del precio de todas las frutas de s}
fespec

La implementación en pseudocódigo de frutas y fruteros sería idéntica a la de monedas y monederos, sustituyendo “moneda” por “fruta” y “monedero” por “frutero”. No la repetimos.

Con objeto de **ahorrar tiempo** y de realizar una **especificación** y una **implementación** más **reutilizables**, procedemos a **especificar e implementar un TAD genérico**, saco, que puede hacer el papel (concretándolo o particularizándolo cuando se necesite) tanto de frutero como de monedero:

espec sacosGenéricos
usa naturales
parámetro formal
  género elemento
  operación
    precio: elemento e -> natural
    _=_ : elemento e1, elemento e2 -> booleano
  {el parámetro es un tipo no definido (elemento) al que se le exigirá tener definidas
  una operación con el perfil que tiene la operación precio y otra operación de igualdad}
fpf

género saco
{Los valores del TAD genérico saco representan valores posibles de un multiconjunto de elementos}

operaciones
vacio: -> saco
{devuelve un saco vacío, sin elementos}

meter: saco s , elemento e -> saco
{devuelve el saco resultante de añadir el elemento e a s}

sacar: saco s , elemento e -> saco
*{devuelve el saco resultante de extraer el elemento e de s;
si no hay ningún elemento e en s, devuelve un saco igual a s}*

cuántas: saco s , elemento e -> natural
{cuenta cuántas unidades del elemento e hay en s}

valor: saco s -> natural
{devuelve la suma del precio de todos los elementos de s; para su cálculo será preciso usar la operación precio, que deberá estar definida para los datos de tipo elemento}
fespec

La implementación genérica del TAD (genérico) saco tendría la forma siguiente:

```
módulo genérico sacosGen
parámetros
  tipo elemento
  con función precio(e:elemento) devuelve natural
  con función "="(e1,e2:elemento) devuelve booleano
exporta
  constante maxNum = 1000000
  tipo saco
  procedimiento vacio(sal s:saco)
  procedimiento meter(e/s s:saco; ent e:elemento)
  procedimiento sacar(e/s s:saco; ent e:elemento)
  función cuántas(s:saco; e:elemento) devuelve natural
  función valor(s:saco) devuelve natural
implementación
  tipo unElemto = registro
                    elElemto:elemento;
                    numVecesRepetido:natural
                  freg
       elementos = vector[1..maxNum] de unElemto
       saco = registro
                losElementos:elementos;
                numDistintos:natural
              freg
  procedimiento vacio(sal s:saco)
  variable e:elemento
  principio
    s.numDistintos:=0
  fin
  ...
  {Dejamos como ejercicio escribir el resto de la implementación, teniendo en
   cuenta que el tipo "elemento" es un parámetro formal, es decir, es sólo un
```

```
    nombre de tipo y no sabemos más características de él, salvo que tendrá
    siempre definida una función "precio" con el perfil indicado.}
fin
```

Finalmente, para poder **usar** un TAD genérico, como el TAD saco, es necesario antes **concretarlo** o particularizarlo, indicando con qué tipo se corresponde su parámetro formal elemento.

```
procedimiento tití
importa monedas,frutas,sacosGen
{En los módulos 'monedas' y 'frutas' están definidos los tipos moneda, fruta, y
 una función 'precio' para monedas y otra para frutas}
  módulo monedero concreta sacosGen(moneda,precio);
  módulo frutero concreta sacosGen(fruta,precio);
  variables m:monedero.saco; f:frutero.saco
principio
  ...
  vacío(m);
  meter(m,e1);
  vacío(f);
  meter(f,pera)
  ...
fin
```

3. Implementación en C++

C++ no implementa realmente la genericidad, es decir, no permite obtener código objeto (compilado) que sea genérico.

La técnica habitual en este lenguaje para simular la genericidad son las **plantillas** (*templates*).

El compilador hace simplemente una sustitución del texto del parámetro formal por el parámetro actual (es decir, por el tipo concreto). Por tanto, no se genera código objeto genérico sino que se genera un código objeto distinto para cada particularización del parámetro formal.

Además, se debe incluir la implementación de las operaciones en el mismo archivo de cabecera.

Así, una posible implementación en C++ del TAD genérico saco visto anteriormente, es la que sigue.

Fichero fruta.h, con la implementación del TAD fruta:

```
#ifndef _FRUTA_H_
#define _FRUTA_H_

enum Fruta { pera, manzana, limon, pomelo, papaya };

int precio(Fruta f){
    switch (f){
        case pera:
            return 2;
            break;
        case manzana:
            return 1;
            break;
        case limon:
            return 3;
            break;
```

```
        case pomelo:
            return 4;
            break;
        case papaya:
            return 5;
            break;
        default:
            return 0;
    }
}

#endif
```

Fichero moneda.h, con la implementación del TAD moneda:

```
#ifndef _MONEDA_H_
#define _MONEDA_H_

enum Moneda { c1, c10, c50, e1 };

int precio(Moneda m){
    switch (m){
        case c1:
            return 1;
            break;
        case c10:
            return 10;
            break;
        case c50:
            return 50;
            break;
        case e1:
            return 100;
            break;
        default:
            return 0;
    }
}

#endif
```

Fichero saco.h con la implementación del TAD genérico saco:

```
#ifndef _SACO_H_
#define _SACO_H_
// Interfaz del TAD. Pre-declaraciones:
const int MAX_NUM_ELEMENTOS = 1000;

template<typename Elemento> struct Saco;
/* El tipo Elemento requerirá tener las funciones:
 * int precio(const Elemento& e);
 * bool operator==(const Elemento& e1, const Elemento& e2);
```

```
 */

template<typename Elemento> void vacio(Saco<Elemento>& s);
template<typename Elemento> bool meter(Saco<Elemento>& s, const Elemento& e);
template<typename Elemento> void sacar(Saco<Elemento>& s, const Elemento& e);
template<typename Elemento> int cuantos(const Saco<Elemento>& s, const Elemento& e);
template<typename Elemento> int valor(const Saco<Elemento>& s);

// Operaciones NO pertenecientes al Interfaz del TAD (internas)
// por tanto, no deben utilizarse fuera de esta implementación:
template<typename Elemento> int buscar(const Saco<Elemento>& s, const Elemento& e);

// Declaración:
template<typename Elemento> struct Saco {
    friend void vacio<Elemento>(Saco<Elemento>& s);
    friend bool meter<Elemento>(Saco<Elemento>& s, const Elemento& e);
    friend void sacar<Elemento>(Saco<Elemento>& s, const Elemento& e);
    friend int cuantos<Elemento>(const Saco<Elemento>& s, const Elemento& e);
    friend int valor<Elemento>(const Saco<Elemento>& s);
   //operación interna:
    friend int buscar<Elemento>(const Saco<Elemento>& s, const Elemento& e);

private:
    struct Repeticiones {
        Elemento dato;
        int numRep;
    };
    Repeticiones elementos [MAX_NUM_ELEMENTOS];
    int total;
};

// Implementación de las operaciones
template<typename Elemento> void vacio(Saco<Elemento>& s) {
    s.total = 0;
}

template<typename Elemento> int buscar(const Saco<Elemento>& s, const Elemento& e) {
    int pos = 0;
    while (pos < s.total && s.elementos[pos].dato != e) {
        pos = pos + 1;
    }
    if (pos == s.total) {
        pos = -1;
    }
    return pos;
}

template<typename Elemento> bool meter(Saco<Elemento>& s, const Elemento& e) {
    bool sePuede = true;
    int pos = buscar(s,e);
    if (pos != -1) {
        s.elementos[pos].numRep = s.elementos[pos].numRep + 1;
```

```
    }
    else {
        sePuede = s.total < MAX_NUM_ELEMENTOS;
        if (sePuede) {
            s.elementos[s.total].dato = e;
            s.elementos[s.total].numRep = 1;
            s.total = s.total + 1;
        }
    }
    return sePuede;
}

template<typename Elemento> void sacar(Saco<Elemento>& s, const Elemento& e) {
    int pos = buscar(s,e);
    if (pos != -1) {
        if (s.elementos[pos].numRep > 1) {
            s.elementos[pos].numRep = s.elementos[pos].numRep - 1;
        }
        else { // al sacar, quedan 0 unidades de ese elemento
            if (s.total == 1) {
                s.total = 0;
            }
            else { // sustituirlo por el ultimo de la tabla
                s.total = s.total - 1;
                s.elementos[pos].dato = s.elementos[s.total].dato;
                s.elementos[pos].numRep = s.elementos[s.total].numRep;
            }
        }
    }
}

template<typename Elemento> int cuantos(const Saco<Elemento>& s, const Elemento& e){
    int pos = buscar(s,e);
    if (pos != -1) {
        return s.elementos[pos].numRep;
    }
    else {
        return 0;
    }
}

template<typename Elemento> int valor(const Saco<Elemento>& s) {
    int res = 0;
    for (int i = 0; i < s.total; i++) {
        res = res + s.elementos[i].numRep * precio(s.elementos[i].dato);
    }
    return res;
}

#endif
```

Y un ejemplo sencillo de utilización del TAD genérico anterior, en el siguiente fichero main.cpp:

```
#include <iostream>
#include "fruta.h"
#include "moneda.h"
#include "saco.h"
using namespace std;

int main() {
    bool todoBien = true;
    Saco<Fruta> frutero;
    vacio(frutero);
    todoBien = meter(frutero,pera);
    todoBien = meter(frutero,limon);
    todoBien = meter(frutero,pomelo);
    todoBien = meter(frutero,pera);
    todoBien = meter(frutero,pera);
    todoBien = meter(frutero,pera);
    if (!todoBien) {
        cout << "ojo, ha habido un error frutal" << endl;
    }
    cout << "numero de peras: " << cuantos(frutero,pera) << endl;
    cout << "valor total: " << valor(frutero) << endl;

    todoBien = true;
    Saco<Moneda> monedero;
    vacio(monedero);
    todoBien = meter(monedero,c1);
    todoBien = meter(monedero,c10);
    todoBien = meter(monedero,c1);
    todoBien = meter(monedero,e1);
    if (!todoBien) {
        cout << "ojo, ha habido un error monetario" << endl;
    }
    cout << "numero de monedas de 1 centimo: " << cuantos(monedero,c1) << endl;
    cout << "valor total: " << valor(monedero) << endl;

}
```

Lección 5

TAD fundamentales

Indice

1. Contenedores
2. Iteradores
3. La biblioteca *Standard Template Library* (STL)

1. Contenedores

La mayor parte de TAD que vamos a definir e implementar son **contenedores** de datos, es decir, **agregaciones de varios datos** (frecuentemente muchos) en una unidad conceptual, el contenedor, incluyendo también en ese contenedor las **operaciones para manipularlos**.

Se trata de **TAD fundamentales** y algoritmos de uso frecuente, que todo ingeniero en informática debe conocer y saber implementar y utilizar para poder diseñar soluciones en los nuevos contextos o problemas. En su mayor parte serán TAD **genéricos**, de forma que lo importante no será tanto el tipo concreto de sus elementos, sino la colección de operaciones para su manipulación y su semántica.

Veremos algunos contenedores que incluirán a**gregaciones de elementos de un mismo tipo**, **sin** necesidad de almacenar **relaciones** de orden, de secuencia, de jerarquía, o de otro tipo, **entre ellos**, más allá de la relación de pertenencia a un mismo contenedor. Ejemplos típicos de estos contenedores son:

- Conjuntos de elementos
- Multiconjuntos de elementos
- Diccionarios (también denominados mapas o tablas)
- Etc.

Veremos otros contenedores que incluirán **agregaciones de elementos de un mismo tipo** que permitirán reflejar las **relaciones o interacciones entre los elementos**; relaciones de orden, de secuencia, de jerarquía, de incidencia previa o posterior, que se dan en la realidad y permiten plantear soluciones sencillas y eficientes. Ejemplos típicos de estos contenedores son:

- Colas (de clientes esperando delante de una ventanilla para recibir un servicio, por ejemplo)
- Pilas (de libros colocados encima de una mesa)
- Listas (de tareas pendientes)
- Árboles que representan jerarquías o clasificaciones (como el árbol de directorios y ficheros en un sistema operativo)
- Grafos (que representan conectividad o dependencias entre elementos)
- Etc.

Pasamos a describir muy brevemente algunas características de unos cuantos de estos contenedores importantes.

Un **conjunto** es un TAD genérico cuyo dominio de valores es una colección de 0, 1 o más elementos. No se admiten elementos repetidos en un conjunto, es decir, un elemento está o no está en un conjunto dado. Las operaciones típicas de manipulación de conjuntos incluyen:

- crear: crea un conjunto vacío, sin elementos
- añadir: dado un elemento, si no pertenece al conjunto, lo añade

- ¿pertenece?: dado un elemento comprueba si se encuentra en el conjunto
- quitar: dado un elemento, si pertenece al conjunto lo elimina
- cardinal: devuelve el número de elementos en el conjunto
- ¿esVacío?: comprueba si el conjunto está vacío (no contiene ningún elemento)
- eliminarTodos: elimina todos los elementos del conjunto, dejándolo vacío
- unión de conjuntos
- diferencia de conjuntos
- intersección de conjuntos
- …

En algunas de esas operaciones pueden producirse situaciones de error, que habrá que especificar convenientemente y tener en cuenta posteriormente al implementar el TAD. Por ejemplo, ¿qué sentido tiene intentar añadir a un conjunto un elemento que no pertenezca al tipo de elementos del conjunto?

Un **multiconjunto**, también denominado saco o bolsa, es un TAD genérico cuyo dominio de valores es una colección de 0, 1 o más elementos; a diferencia de lo que ocurre en un conjunto, en un multiconjunto cada elemento puede estar repetido un determinado número de veces (recordar el TAD genérico saco especificado e implementado en la lección anterior). Las operaciones típicas de manipulación de multiconjuntos incluyen:

- crear: crea un multiconjunto vacío, sin elementos
- añadir: añade un ejemplar de un elemento al multiconjunto
- ¿pertenece?: dado un elemento, comprueba si se encuentra en el multiconjunto
- multiplicidad: dado un elemento, devuelve el número de ejemplares del elemento en el multiconjunto
- quitar: dado un elemento, si existe en el multiconjunto, elimina una de sus apariciones
- cardinal: devuelve el número total de ejemplares de elementos en el multiconjunto (teniendo en cuenta la multiplicidad)
- ¿esVacío?: comprueba si el multiconjunto está vacío (no contiene ningún elemento)
- eliminarTodos: elimina todos los elementos del multiconjunto, dejándolo vacío
- unión de multiconjuntos
- diferencia de multiconjuntos
- intersección de multiconjuntos
- …

De nuevo, como en el caso de los conjuntos, pueden producirse algunas situaciones de error que conviene prever.

Un **diccionario** (o mapa o tabla) es un TAD genérico cuyo dominio de valores es una colección de 0, 1 o más elementos formados por pares <clave, valor>. Tiene, por tanto, dos parámetros formales de tipo: el tipo clave y el tipo valor. Además, en un diccionario no puede haber dos parejas con la misma clave. Es decir, las claves en un diccionario son únicas. Una forma alternativa de definirlos es como una función que asocia datos de tipo valor a datos de tipo clave, es decir, cada clave perteneciente al dominio del diccionario, tiene asociado un valor. Por eso al TAD diccionario se le denomina también TAD funcional o asociativo. Las operaciones típicas de manipulación de diccionarios son operaciones de acceso por clave:

- crear: crea un diccionario vacío, sin elementos
- añadir: dada una clave y un valor, asigna el valor a la clave en el diccionario
- ¿pertenece?: dada una clave, devuelve un booleano que indica si la clave está en el diccionario
- obtenerValor: dada una clave, devuelve el valor asociado a ella en el diccionario
- quitar: dada una clave, la borra del diccionario junto con su valor asociado
- cardinal: devuelve el número de elementos (parejas <clave,valor>) en el diccionario
- ¿esVacío?: comprueba si el diccionario está vacío (no contiene ningún elemento)
- …

Convendrá detectar y especificar las posibles situaciones de error. Por ejemplo: si se intentan obtener valores de claves no existentes en el diccionario.

Existe un gran abanico de **TAD lineales**, o **listas**, que se caracterizan porque su dominio de valores son las secuencias de elementos de un cierto tipo: $<e_1, e_2, \ldots e_n>$, de longitud arbitraria, e incluyendo la secuencia vacía < >. Nótese que los elementos de una secuencia tienen un orden dado por su posición en la secuencia: existe un primero, un segundo, …, un último elemento de la secuencia, salvo que la secuencia sea vacía. Con ese mismo dominio de valores (genérico) se pueden definir multitud de TAD dependiendo de las restricciones adicionales que puedan imponerse sobre los valores de la secuencia y también sobre el conjunto de operaciones de manipulación que se consideren:

- Secuencias con o sin elementos repetidos.
- Datos de la secuencia ordenados con respecto a sus valores (por ejemplo, por orden alfabético) o no.
- Datos de la secuencia ordenados con respecto a criterios diferentes a los valores de los datos (por ejemplo, el orden de llegada o inserción).
- Un subconjunto de operaciones muy habituales son las de crear (la secuencia vacía), añadir (un elemento a la secuencia), averiguar si es vacía la secuencia, quitar un elemento de la secuencia, saber si un elemento pertenece a la secuencia, tamaño o longitud de la secuencia, etc.
- Además, puede haber operaciones de acceso y recorrido, siguiendo el orden de la secuencia de elementos (los denominados *iteradores*, de los que hablaremos enseguida).
- Evidentemente, la semántica del par de operaciones *añadir elemento*/*quitar elemento* puede cambiar, dando lugar a distintos TAD. Pueden ser operaciones:
    - limitadas a determinados extremos de la secuencia (TAD pila, cola, bicola…),
    - relativas a la posición en la secuencia (listas con acceso por posición),
    - relativas al último elemento accedido o insertado (listas con punto de interés), …

Otra gran familia de TAD es la de **tipos arborescentes**, cuyo dominio son colecciones de datos que no se representan de manera directa como una secuencia, sino en una **jerarquía de árbol**. Las variantes incluyen, entre otras muchas, el grado del árbol (binarios o *n*–arios), la ordenación de los elementos en los vértices o nodos del árbol con respecto a algún criterio relacionado o no con sus valores, las operaciones de manipulación (desde las de un simple diccionario hasta conjuntos de operaciones con semántica más dependiente del dominio de aplicación, o de la posición de los elementos en el árbol, etc.)

También pueden definirse **distintos TAD grafo**, para almacenar y manipular diferentes tipos de grafos: dirigidos o no, con pesos en las aristas o no, bipartitos o no, etc. De nuevo, la panoplia de operaciones es amplia y depende mucho del tipo de grafo y del dominio de aplicación (algoritmos de cálculo de distancias, de caminos mínimos, de cálculo de árboles de recubrimiento, etc).

2. Iteradores

Con las operaciones típicas que hemos mencionado para los contenedores (conjunto, multiconjunto, diccionario, etc.),

- ¿cómo podemos (por ejemplo) mostrar todos los elementos del contenedor por pantalla?
- ¿cómo podemos (por ejemplo) conocer todos los elementos del contenedor que cumplan cierta condición?

Sencillamente, teniendo en cuenta que la especificación del TAD se realiza sin conocer los detalles de la implementación (en particular, sin saber cómo están almacenados los valores del TAD contenedor), no podemos.

Es necesario añadir a cualquier contenedor, cuyos datos se precise "recorrer" en algún momento, un **conjunto pequeño de operaciones que puedan servir para acceder a todos y cada uno de los elementos del contenedor una vez y sólo una y de forma eficiente**, y sin saber cómo están almacenados los elementos del contenedor. Ese conjunto de operaciones se denomina **iterador**. Disponer de un iterador para un TAD permite implementar posteriormente algoritmos de recorrido o búsqueda.

Para definir un iterador, puede suponerse que, además de los elementos almacenados en el contenedor y sus posibles relaciones, existe un **cursor** o índice que señala a un elemento del contenedor, o a ninguno si el cursor ya ha recorrido y señalado antes a todos los elementos. Se necesitan las siguientes operaciones para, utilizando ese cursor, poder recorrer todos los elementos del contenedor:

- ***iniciarIterador***: prepara el cursor para que el siguiente elemento señalado (o a visitar) sea el primero del contenedor (situación de no haber visitado ningún elemento), es decir, coloca el cursor señalando a un primer elemento del contenedor (en el caso de un TAD lineal, suele ser el primero de la secuencia);
- ***¿existeSiguiente?***: devuelve falso si ya se ha visitado el último elemento del contenedor (el cursor no señala a ningún elemento), devuelve cierto en caso contrario;
- ***siguiente***: devuelve el elemento señalado por el cursor; es una operación parcial: no está definida si no ¿existeSiguiente?;
- ***avanza***: avanza el cursor para señalar a otro elemento del contenedor que todavía no se haya visitado, es decir, que no haya sido señalado antes por el cursor; es una operación parcial: no está definida si no ¿existeSiguiente?

Es habitual implementar las operaciones *siguiente* y *avanza* juntas por razones de eficiencia y porque en el uso del iterador para recorrer los elementos del contenedor siempre sería necesario invocar una tras otra:

```
procedimiento iniciarIterador(e/s c:tipo_del_contenedor)
{Prepara el cursor del contenedor c para que el elemento señalado sea el primero
 del contenedor (situación de no haber visitado antes ningún elemento)}
función existeSiguiente(c:tipo_del_contenedor) devuelve booleano
{Devuelve falso si ya se ha visitado el último elemento del contenedor
 (el cursor no señala a ningún elemento), devuelve cierto en caso contrario}
procedimiento siguiente(e/s c:tipo_del_contenedor;
                        sal e:elemento_base_del_contenedor; sal error:booleano)
{Si no ¿existeSiguente?(c) entonces devuelve el valor verdad en el parámetro error;
 En caso contrario:
   1. devuelve falso en el parámetro error;
   2. devuelve el valor del elemento señalado por el cursor en el parámetro e;
   3. avanza el cursor para señalar a otro elemento del contenedor que todavía no se haya
      visitado}
```

Un uso típico para el que el iterador está diseñado es el siguiente (recorrido de los datos del contenedor):

```
iniciarIterador(c);
mientrasQue existeSiguiente(c) hacer
  siguiente(c,e,error);
  <utilizar el valor de e para algo>
fmq
```

Nunca se debe modificar el contenedor de datos (con las operaciones de añadir, borrar o modificar de que disponga) mientras se recorren sus elementos con las operaciones de un iterador.

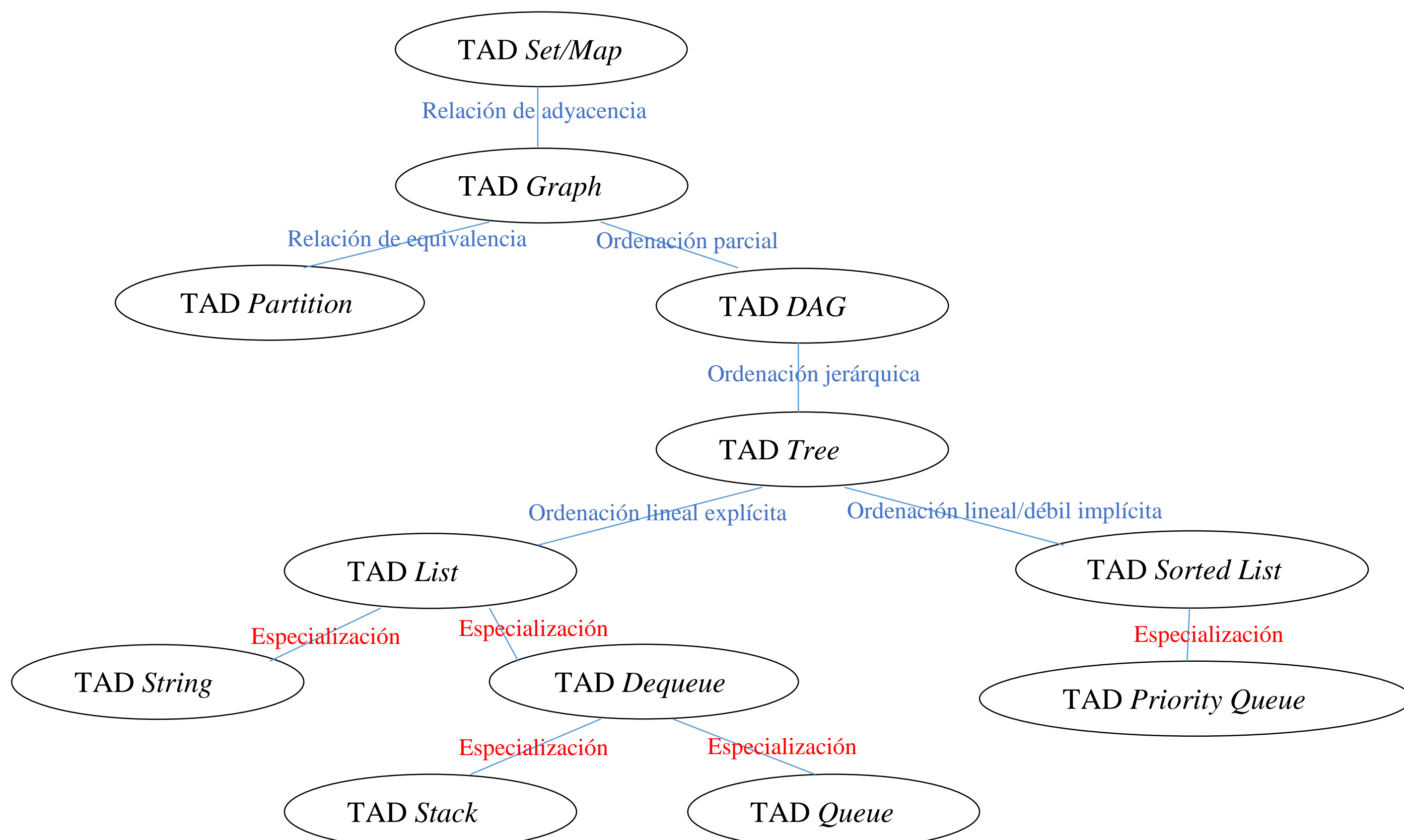
3. La biblioteca *Standard Template Library* (STL)

La *Standard Template Library* (STL) es una biblioteca de C++ que proporciona TAD, estructuras de datos y algoritmos de uso frecuente.

Es un buen ejemplo de definición e implementación de TAD reutilizables. Aun así, no siempre encontraremos precisamente lo que necesitemos. Por ejemplo, para tener código general y muy reutilizable es posible que la STL esté sacrificando eficiencia y utilizando más recursos de los aceptables.

No existe una biblioteca equivalente disponible en todos los lenguajes de programación, pero sí en muchos de ellos.

Este es un fragmento de la estructura de TAD en la STL. Puede verse que incluye conjuntos, mapas (diccionarios), grafos de varias clases, árboles, listas, pilas, colas, bicolas…



No es objetivo de estos apuntes mostrar los detalles ni enseñar a utilizar la STL, ni otras bibliotecas similares. Por el contrario, pretendemos guiar en el aprendizaje del diseño o definición de TAD para que sean reutilizables, eficientes y robustos, y en el aprendizaje de su implementación, garantizando dichas propiedades.

TEMA II

Tipos de datos lineales

Lección 6

El TAD pila genérica. Definición e implementación estática

Indice

1. Concepto de pila y especificación
2. Representación estática e implementación de operaciones
3. Representación de varias pilas en un vector

1. Concepto de pila y especificación

La pila, como el resto de tipos que veremos en este tema, es una **secuencia de elementos de un cierto tipo** (que se concretará para cada aplicación) **dispuestos en una dimensión**. Se dice, por tanto, que es un **tipo lineal** de datos.

En el tipo pila hay un valor especial que denominamos **pila vacía**. Además, en una pila se pueden añadir y quitar elementos. La operación de añadir se denomina **apilar** y la de quitar se denomina **desapilar**. Además, si la pila no es vacía, podemos observar el último elemento apilado, que denominamos **cima** de la pila.

El **comportamiento intuitivo** de una pila es el siguiente:

- la cima de una pila no vacía es el último elemento apilado, y
- al desapilar de una pila no vacía desaparece el último elemento apilado.

Las pilas se denominan también estructuras **LIFO** (del inglés, *Last In, First Out*), nombre que hace referencia al modo en que se añaden y quitan sus elementos.

La **especificación** del TAD pila (genérico), que ya fue introducida en una lección anterior, es la siguiente es la siguiente:

espec pilasGenéricas
usa booleanos, naturales
parámetro formal
  género elemento
fpf
género pila
  {Los valores del TAD pila representan secuencias de datos de tipo "elemento" con acceso LIFO (last in, first out), esto es, el último elemento añadido (o apilado) será el primero en ser borrado (o desapilado)}
operaciones
  pilaVacía: -> pila
  {Devuelve una pila vacía, sin elementos}

  apilar: pila p, elemento e -> pila
  {Devuelve la pila resultante de añadir e a p}

  desapilar: pila p -> pila
  *{Si p es no vacía, devuelve la pila resultante de eliminar de p el último elemento que fue apilado.
  Si p es vacía, devuelve una pila igual a p}*

  parcial cima: pila p -> elemento
  *{Devuelve el último elemento apilado en p.
  Parcial: la operación no está definida si p es vacía}*

  esVacía?: pila p -> bool
  {Devuelve verdad si y sólo si p no tiene elementos}

  altura: pila p -> natural
  {Devuelve el nº de elementos de p}

{A cualquiera de los contenedores o colecciones de datos que veamos, será conveniente añadirles un Iterador. Las cuatro siguientes operaciones son un Iterador definido sobre las pilas}
iniciarIterador: pila p -> pila
{Prepara el iterador para que el siguiente elemento a visitar sea el último de la pila p, si existe (situación de no haber visitado ningún elemento)}

existeSiguiente?: pila p -> booleano
{Devuelve falso si ya se han visitado todos los elementos de p, devuelve verdad en caso contrario}

parcial siguiente: pila p -> elemento
{Devuelve el siguiente elemento de p.
Parcial: la operación no está definida si no existeSiguiente?(p)}

parcial avanza: pila p -> pila
{Devuelve la pila resultante de avanzar el iterador en p.
Parcial: la operación no está definida si no existeSiguiente?(p)}

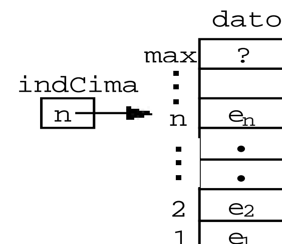
fespec

2. Representación estática e implementación de operaciones

La representación estática o **contigua** de una pila consiste en un vector con espacio para un máximo `max` de elementos y un contador `cima`, `0 ≤ cima ≤ max`, que indica el número de elementos válidos almacenados en el vector. Es decir:

```
constante max = ...
tipo pila = registro
              dato: vector[1..max] de elemento;
              indCima,iter: 0..max
             freg
```

Donde `elemento` es un tipo previamente definido. Si la pila `p` es no vacía, el elemento almacenado en `p.dato[p.indCima]` corresponde a la cima de la pila, mientras que el elemento `p.dato[1]` es el fondo de la pila (el elemento que fue apilado en primer lugar). La pila vacía se representa con `p.indCima=0`. El campo `iter` se utilizará para implementar las operaciones del iterador.



A continuación, presentamos una implementación del tipo:

```
módulo genérico pilas
parámetro
  tipo elemento
exporta
  constante max = 100¹ {Es la altura máxima de cualquier pila representable en el
                         tipo pila, limitada por una decisión de implementación.}
  tipo pila
    {Los valores del TAD pila representan secuencias de longitud menor o igual que
    el valor de la constante 'max' de elementos con acceso LIFO (last in, first
    out), esto es, el último elemento añadido (o apilado) será el primero en ser
    borrado (o desapilado)}

  procedimiento crearVacía(sal p:pila)
  {Devuelve una pila vacía, sin elementos}

  procedimiento apilar(e/s p:pila; ent e:elemento; sal error:booleano)
  {Si altura(p)<max), entonces devuelve en el mismo parámetro p la pila resultante
   de añadir e a p, y error=falso. En caso contrario, devuelve error=verdad y no
   modifica p.}
```

[1] Es un valor arbitrario.

```
procedimiento desapilar(e/s p:pila)
{Si p es no vacía, devuelve la pila resultante de eliminar de p el último elemento
 que fue apilado. Si p es vacía, la deja igual}

función cima(p:pila) devuelve elemento
{Pre: p es no vacía} {Devuelve el último elemento apilado en p}

función esVacía(p:pila) devuelve booleano
{Devuelve verdad si y sólo si p no tiene elementos}

función altura(p:pila) devuelve natural
{Devuelve el nº de elementos de p, 0 si no tiene elementos}

procedimiento iniciarIterador(e/s p:pila)
{Prepara el cursor del iterador para que el siguiente elemento a visitar sea la
 cima de la pila p, si existe (situación de no haber visitado ningún elemento)}

función existeSiguiente(p:pila) devuelve booleano
{Devuelve falso si ya se han visitado todos los elementos de p.
 Devuelve verdad en caso contrario.}

procedimiento siguiente(e/s p:pila; sal e:elemento; sal error:booleano)
{Implementa las operaciones "siguiente" y "avanza" de la especificación, es decir:
 Si existeSiguiente(p), error toma el valor falso, e toma el valor del siguiente
 elemento de la pila, y se avanza el cursor del iterador al elemento siguiente
 de la pila.
 Si no existeSiguiente(p), error toma el valor verdad, e queda indefinido y p
 queda como estaba.}

implementación
  tipo pila = registro
                dato: vector[1..max] de elemento;
                indCima,iter: 0..max
              freg

  procedimiento crearVacía(sal p:pila)
  {Devuelve una pila vacía, sin elementos}
  principio
    p.indCima:=0
  fin

  procedimiento apilar(e/s p:pila; ent e:elemento; sal error:booleano)
  {Si altura(p)<max), entonces devuelve en el mismo parámetro p la pila resultante
   de añadir e a p, y error=falso. En caso contrario, devuelve error=verdad y no
   modifica p.}
  principio
    si p.indCima<max entonces
      error:=falso;
      p.indCima:=p.indCima+1;
      p.dato[p.indCima]:=e
    sino
      error:=verdad
    fsi
  fin

  procedimiento desapilar(e/s p:pila)
  {Si p es no vacía, devuelve la pila resultante de eliminar de p el último elemento
   que fue apilado. Si p es vacía, la deja igual}
  principio
    si p.indCima>0 entonces
      p.indCima:=p.indCima-1
    fsi
  fin

  función cima(p:pila) devuelve elemento
  {Pre: p es no vacía} {Devuelve el último elemento apilado en p}
```

```
principio[2]
  devuelve(p.dato[p.indCima])
fin

función esVacía(p:pila) devuelve booleano
{Devuelve verdad si y sólo si p no tiene elementos}
principio
  devuelve(p.indCima=0)
fin

función altura(p:pila) devuelve natural
{Devuelve el nº de elementos de p, 0 si no tiene elementos}
principio
  devuelve p.indCima
fin

procedimiento iniciarIterador(e/s p:pila)
{Prepara el iterador para que el siguiente elemento a visitar sea la
 cima de la pila p, si existe (situación de no haber visitado ningún elemento)}
principio
  p.iter:=p.indCima
fin

función existeSiguiente(p:pila) devuelve booleano
{Devuelve falso si ya se han visitado todos los elementos de p.
 Devuelve verdad en caso contrario.}
principio
  devuelve (p.iter>0)
fin

procedimiento siguiente(e/s p:pila; sal e:elemento; sal error:booleano)
{Implementa las operaciones "siguiente" y "avanza" de la especificación, es decir:
 Si existeSiguiente(p), error toma el valor falso, e toma el valor del siguiente
 elemento de la pila, y se avanza el iterador al elemento siguiente de la pila.
 Si no existeSiguiente(p), error toma el valor verdad, e queda indefinido y p
 queda como estaba.}
principio
  si existeSiguiente(p) entonces
    error:=falso;
    e:=p.dato[p.iter];
    p.iter:=p.iter-1
  sino
    error:=verdad
  fsi
fin

fin {del módulo}
```

Como se indica, el código de la función cima no es robusto, no chequea si se cumple la precondición de pila no vacía. Si se desea hacer robusto, hay que optar, por ejemplo, por la siguiente implementación:

```
procedimiento cima(ent p:pila; sal e:elemento; sal error:booleano)
{Si p es vacía, error toma el valor verdad y se deja e indefinido.
 Si p no es vacía, error toma el valor falso y e toma el valor de la cima de p.}
principio
  si esVacía(p) entonces
    error:=verdad
  sino
    error:=falso;
    e:=p.dato[p.indCima]
  fsi
fin
```

[2] Ojo, no es código robusto. Es decir, el código no chequea si se cumple la precondición de pila no vacía.

El coste en tiempo de todas las operaciones es $O(1)$, es decir, independiente de la altura de la pila.

El mayor inconveniente de esta representación (estática) es que debe reservarse espacio (tamaño del vector) para el máximo previsto de elementos, aunque luego no se alcance al ejecutar el programa usuario. Además, **la operación `apilar` hay que implementarla como si fuese una operación parcial**, puesto que al intentar apilar un dato cuando `indCima = max`, se produce un error. A pesar de este problema, consideramos válida esta representación, de la misma forma que en los tipos predefinidos, como los enteros o los reales, toda implementación impone restricciones de tamaño o redondeo que no están presentes en la especificación del tipo (no se puede representar un tipo con un conjunto de valores de cardinal infinito en un espacio finito).

Además de las operaciones especificadas e implementadas anteriormente, toda implementación de un contenedor o colección de datos, debería incluir además operaciones básicas para **duplicar** la representación de un valor del TAD, es decir, hacer una copia de una variable del TAD en otra distinta y de **comparación de igualdad** entre dos valores del TAD. Para el tipo pila implementado aquí, serían las siguientes:

```
procedimiento duplicar(ent pEnt:pila; sal pSal:pila)
{Hace una copia en pSal de la pila almacenada en pEnt.}
Variable i:natural
principio
  pSal.indCima:=pEnt.indCima;
  si not esVacía(pEnt) entonces
    para i:= 1 hasta pEnt.indCima hacer
      pSal.dato[i]:=pEnt.dato[i]  (*)
    fpara
  fsi
fin

función iguales(p1,p2:pila) devuelve booleano
{Devuelve verdad si y sólo si p1 y p2 almacenan la misma pila.}
variables igual:booleano; i:natural
principio
  si esVacía(p1) and esVacía(p2) entonces
    devuelve verdad
  sino_si altura(p1)≠altura(p2) entonces
    devuelve falso
  sino {ambas pilas tienen el mismo número, no nulo, de elementos}
    igual:=verdad;
    i:=p1.indCima;
    mientrasQue igual and i>0 hacer
      igual:= (p1.dato[i]=p2.dato[1]); (**)
      i:=i-1
    fmq;
    devuelve igual
  fsi
fin
```

Debemos destacar que en la instrucción `(**)` de esta última función, estamos comparando dos datos de tipo "elemento", que no sabemos cuál es. Si no es un tipo predefinido (que permita esa comparación de igualdad con el operador "="), deberemos usar una función "`iguales(e1,e2)`", en este caso "`iguales(p1.dato[i],p2.dato[i])`" que debería existir para el tipo de dato "elemento", y que sirva para comparar valores de ese tipo.

Algo similar sucede con las operaciones en las que asignamos un elemento a una componente del vector campo "dato" del tipo "pila". Por ejemplo, cuando en el código de la operación duplicar escribimos `(*)`: `pSal.dato[i]:=pEnt.dato[i]`. Si el tipo elemento se particulariza en un tipo no predefinido, la operación de asignación (":=") entre datos de tipo "elemento" debería sustituirse por una llamada a un procedimiento para "duplicar elementos", similar al procedimiento de duplicar pila que hemos implementado aquí.

Una forma de resolver este problema es exigir al parámetro formal de tipo "elemento" que disponga de sendas operaciones ":=" e "=" (podrían llamarse alternativamente "duplicar" e "iguales") para duplicar su representación (lo que habitualmente llamamos "asignación entre variables") y para realizar la comparación de igualdad entre elementos,

respectivamente. En ese caso, el inicio del módulo genérico de implementación de pilas sería el siguiente, exigiendo que el parámetro formal de tipo “elemento” disponga de esas dos operaciones:

```
módulo genérico pilas
parámetros
  tipo elemento
  con procedimiento duplicar(ent eEnt:elemento; sal eSal:elemento)
      {procedimiento que duplica la representación de eEnt en eSal}
  con función iguales(e1,e2:elemento) devuelve booleano
      {función que devuelve verdad si y sólo si e1 y e2 almacenan el mismo elemento}
exporta
  ...
```

3. Representación de varias pilas en un vector

Cuando un programa necesita utilizar varias pilas, la representación de un vector para cada pila puede resultar muy costosa en espacio. Puede optarse por almacenar varias pilas en un solo vector.

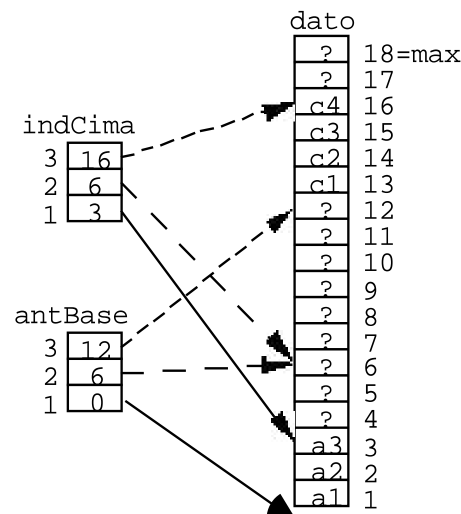
Si el número de pilas a representar en un vector de componentes `1..max` es dos, la solución es fácil. Basta con apilar los elementos de una pila desde la componente `1` en adelante y los elementos de la otra desde la componente `max` hacia atrás.

Si el número de pilas a representar, `n`, es superior a dos, la solución no es tan fácil. Es necesario dividir las `max` componentes en `n` segmentos y alojar los elementos de cada pila en un segmento.

```
constantes max = 100; {nº máximo de elementos simultáneamente en todas las pilas}
           n = 3; {nº máximo de pilas representadas en el vector}
tipo nPilas = registro
                dato: vector[1..max] de elemento;
                antBase: vector[1..n] de 0..max;
                indCima: vector[1..n] de 0..max
              freg
```

En la figura puede verse el caso de `n = 3`, para un espacio total de `18` elementos.

La primera pila representada tiene tres elementos, `a1`, `a2` y `a3`. La cima de la pila es `a3`. El índice anterior al que ocupa la base de la pila es el `0` y está almacenado en la primera componente del vector `antBase`. La posición de la cima, `3`, está almacenada en la primera componente del vector `indCima`.



La segunda pila es la pila vacía. El índice anterior al que ocuparía la base en caso de no ser la pila vacía es el `6` (almacenado en la segunda componente del vector `antBase`).

El hecho de que la pila es vacía se representa almacenando en el lugar destinado al índice de su cima (la segunda componente del vector `indCima`) el mismo valor `6`. En forma análoga se ha almacenado la tercera pila, que consta de los elementos `c1`, `c2`, `c3` y `c4`. El anterior al índice de la base es el `12` y el índice de la cima es el `16`.

La implementación de las operaciones es sencilla siempre que las pilas no sobrepasen la capacidad del segmento correspondiente. Si alguna pila sobrepasa la longitud de su segmento, se hace necesario **reorganizar el vector**, adjudicando mayor espacio a esa pila. Por ejemplo, es posible hacer que el espacio libre sobre cada pila sea proporcional al tamaño actual de la misma (en el supuesto de que las pilas más grandes tienen mayor probabilidad de crecer antes, se retrasa al máximo el instante de la siguiente reorganización).

Lección 7

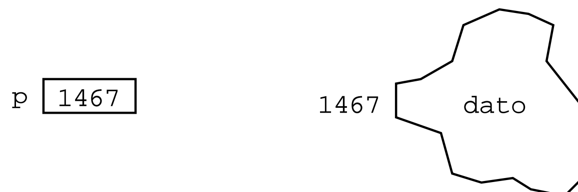
Datos puntero y estructuras dinámicas de datos

Indice

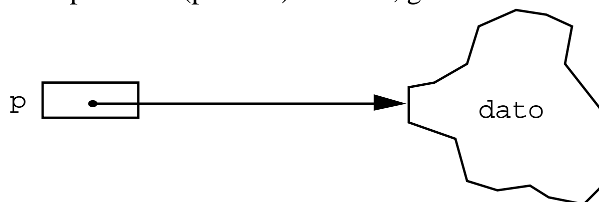
1. Datos puntero y datos dinámicos
2. Estructuras de datos recursivas: representación mediante punteros y datos dinámicos
3. Punteros y datos dinámicos en C++

1. Datos puntero y datos dinámicos

Un **puntero** es un dato cuyo valor es la **dirección en memoria** de otro determinado dato.



El valor de p puede verse como un apuntador (puntero) del dato, gráficamente se representa así:



La dirección almacenada en un puntero va a ser transparente al programador. A éste sólo le interesa conocer el valor de un **puntero como referencia a un determinado dato**.

Los punteros con los que vamos a trabajar están **especializados** para servir como referencia de datos de un **determinado tipo**.

Nuestro lenguaje algorítmico estará dotado del siguiente **mecanismo constructor de datos puntero**:

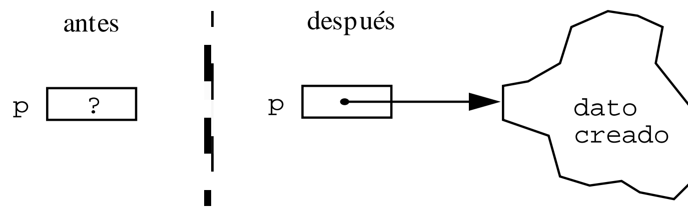
```
tipos tx = ...
      ...
      tpPuntTx = ↑tx
      ...
variables p,q:tpPuntTx
```

Un puntero se puede utilizar para servir de **referencia de datos creados dinámicamente** (es decir, durante la ejecución del algoritmo[3]) mediante la instrucción:

```
nuevoDato(p)
```

que tiene como efecto la reserva de espacio para almacenar una nueva variable del tipo apuntado por el puntero `p`, es decir, del tipo `tx`; esta variable queda apuntada por el puntero `p`:

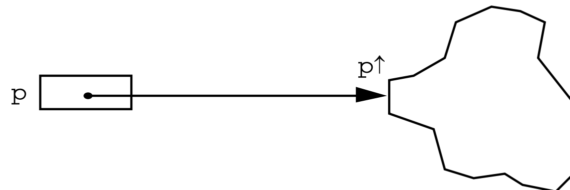
[3] Se denominan **datos estáticos** los almacenados en las variables declaradas al principio de los algoritmos (variables estáticas) y **datos dinámicos** los referenciados por un puntero (durante la ejecución del algoritmo). Los datos estáticos existen durante todo el tiempo de ejecución de su tiempo de vida; se les asigna una zona de la memoria principal al comenzar la ejecución y se libera esa memoria al terminar la ejecución.



Un dato creado dinámicamente **no tiene** ningún identificador asociado como **nombre**. Se puede hacer referencia a él a través de algún puntero, `p`, que apunte al dato, de acuerdo con la siguiente sintaxis:

```
p↑
```

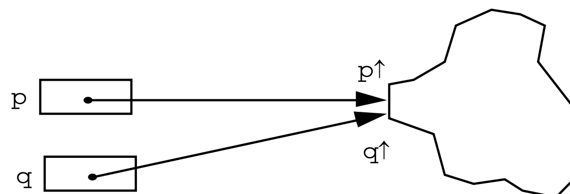
que significa “dato apuntado por el puntero `p`”.



A una variable de tipo puntero se le puede **asignar** el valor de otra variable de tipo puntero (ambos deben ser punteros de datos del mismo tipo):

```
q:=p
```

Produciendo como efecto:



En este caso, el dato creado dinámicamente puede ser accedido de dos formas:

```
p↑
q↑
```

Existe un valor constante, `nil`, que debe ser asignado a todo puntero que se desee que no apunte a ningún dato. A cualquier variable de tipo puntero (independientemente del tipo del dato al que apunte) puede ser asignado dicho valor:

```
p:=nil
```

La acción opuesta a `nuevoDato` es `disponer`:

```
disponer(p)
```

Se llama “**recolección de basura**” a la gestión manual o automática de la memoria dinámica. La instrucción disponer es la que permite al programador responsabilizarse de la recolección (manual) de basura. Esta instrucción libera la zona de memoria que ocupaba el dato apuntado por el puntero `p`. Hay lenguajes que permiten la gestión manual (como el C++) y otros que no (como el *Java*).

Finalmente, dos datos de tipo puntero (a datos del mismo tipo) pueden ser comparados con los operadores relacionales de igualdad (=) o desigualdad (≠).

2. Estructuras de datos recursivas: representación mediante punteros y datos dinámicos

En un primer curso de programación se suelen estudiar sólo **datos estáticos**: datos simples (estándar, como los booleanos, carácter, enteros o reales; enumerados o subrangos) y estructurados (vectores o registros). El correspondiente procesador (compilador o intérprete) determina para ellos una **representación fija** (estructura y tamaño).

Veremos a continuación cómo pueden utilizarse los datos puntero para definir **estructuras dinámicas**, que puedan variar durante la ejecución del algoritmo.

Al igual que se pueden definir algoritmos recursivos cabría pensar en la definición de estructuras de datos recursivas para representar datos definido de forma recursiva como, por ejemplo, los siguientes:

Ejemplo 1: Una cadena de caracteres no vacía se compone de un carácter seguido del resto, otra cadena de caracteres.

```
tipo cadena = registro
                vacía:booleano;
                primero:carácter;
                resto:cadena*
              freg
```

***¡No lo permitimos!**

Ejemplo 2: Una expresión aritmética es un operador (+, -, * ó /) infijo y dos operandos; un operando es o bien un número real o una nueva expresión aritmética.

```
tipos operador = (suma,resta,producto,división);
      expresión = registro
                    elOperador:operador;
                    operando1,operando2:operando
                  freg
      operando = registro
                   esNúmero:booleano;
                   valor:real;
                   subexpresión:expresión*
                 freg
```

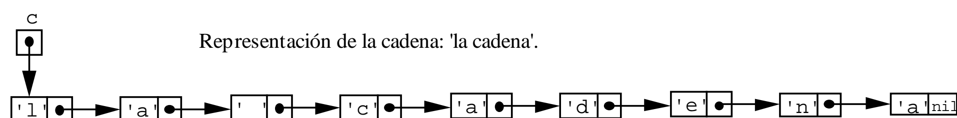
***¡No lo permitimos!**

Las definiciones recursivas anteriores no son directamente codificables con los lenguajes de programación habituales. La razón es que a partir de tales definiciones un procesador no es capaz de averiguar el espacio de memoria necesario (ni su estructuración) que corresponde a una variable estática del tipo correspondiente.

La solución es utilizar datos puntero para encadenar las diferentes partes de la estructura de datos, de forma que la forma y el tamaño de dicha estructura (dinámica) pueda variar durante la ejecución del algoritmo.

Así, los datos cadena pueden representarse de la siguiente forma:

```
tipos cadena = ↑cad;
      cad = registro
              primero:carácter;
              resto:cadena
            freg
```



Las expresiones aritméticas definidas anteriormente admiten la siguiente representación:

```
tipos operador = (suma,resta,producto,división);
      expresión = registro
                    elOperador:operador;
```

```
                        operando1,operando2:operando
                      freg
        operando = registro
                     esNúmero:booleano;
                     valor:real;
                     subexpresión:↑expresión
                   freg
```

3. Punteros y datos dinámicos en C++

A continuación, se presenta, a título de ejemplo, la equivalencia entre nuestra notación algorítmica y el lenguaje C++:

```
tipo Persona = registro
                 nombre:cadena;
                 edad:entero
               freg;
     punteroPersona = ↑Persona

variable p,q:punteroPersona


nuevoDato(p);
p↑.nombre:="Pepe";
p↑.edad:=23;

disponer(p);

q:=nil;
```

```
struct Persona {
  string nombre;
  int edad;
};


Persona* p;
Persona* q;

p = new Persona;
p->nombre = "Pepe"; (*)
p->edad = 23;

delete p;

q = nullptr;
```

(*) Es lo mismo que:

```
(*p).nombre = "Pepe";
```

En cuanto a la estructura de datos para almacenar cadenas de caracteres (si el lenguaje no dispusiera de ese tipo ya predefinido), una solución elemental similar a la escrita en la sección anterior en pseudocódigo es:

```
struct Cadena {
      char primero;
      Cadena* resto;
}
```

Lección 8

Implementación dinámica de pilas

Indice

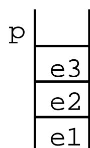
1. Representación dinámica de una pila e implementación de operaciones
2. Codificación en C++ (fragmento)
3. Ejemplo de aplicación del TAD pila: evaluación de expresiones postfijas
4. Otro ejemplo de aplicación: traducción de expresiones infijas a postfijas

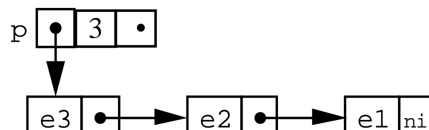
1. Representación dinámica de una pila e implementación de operaciones

El tipo pila se representa mediante un puntero que apunta a un dato creado dinámicamente donde está el elemento correspondiente a la cima de la pila. En dicho dato dinámico se almacena, además del elemento, un puntero que apunta a otro dato dinámico donde está el elemento anterior a la cima, y así sucesivamente, hasta llegar al dato que almacena el elemento más profundo de la pila, cuyo puntero correspondiente tendrá el valor `nil`. Esta representación se denomina **encadenamiento mediante punteros** (o lista enlazada) de los elementos de la pila. Si la pila está vacía, el puntero inicial tendrá directamente el valor `nil`.

A esta representación básica, puede añadirse fácilmente un campo adicional para almacenar la altura de la pila y otro para implementar un iterador, tal como se hizo en la representación estática de pilas.

```
tipos ptDato = ↑unDato;
      unDato = registro
                  dato:elemento;
                  sig:ptDato
                freg
      pila = registro
                cim:ptDato;
                alt:natural;
                iter:ptDato
              freg
```





La implementación completa del TAD se incluye en el siguiente módulo:

```
módulo genérico pilas
parámetro
  tipo elemento
exporta
  tipo pila
   {Los valores del TAD pila representan secuencias de elementos con acceso LIFO
   (last in, first out), esto es, el último elemento añadido (o apilado) será el
   primero en ser borrado (o desapilado)}

  procedimiento crearVacía(sal p:pila)
  {Devuelve una pila vacía, sin elementos}

  procedimiento apilar(e/s p:pila; ent e:elemento)
  {Devuelve la pila resultante de añadir e a p}

  procedimiento desapilar(e/s p:pila)
  {Si p es no vacía, devuelve la pila resultante de eliminar de p el último elemento
   que fue apilado. Si p es vacía, la deja igual}
```

```
  función cima(p:pila) devuelve elemento
  {Pre: p es no vacía} {Devuelve el último elemento apilado en p}

  función esVacía(p:pila) devuelve booleano
  {Devuelve verdad si y sólo si p no tiene elementos}

  función altura(p:pila) devuelve natural
  {Devuelve el nº de elementos de p, 0 si no tiene elementos}

  procedimiento duplicar(ent pilaEnt:pila; sal pilaSal:pila)
  {Hace una copia en pilaSal de la pila almacenada en pilaEnt.}

  función iguales(pila1,pila2:pila) devuelve booleano
  {Devuelve verdad si y sólo si pila1 y pila2 almacenan la misma pila.}

  procedimiento liberar(e/s p:pila)
  {Devuelve la pila vacía, liberando previamente toda la memoria que ocupa la pila
  de entrada p.}

  procedimiento iniciarIterador(e/s p:pila)
  {Prepara el puntero del iterador para que el siguiente elemento a visitar sea la
   cima de la pila p, si existe (situación de no haber visitado ningún elemento)}

  función existeSiguiente(p:pila) devuelve booleano
  {Devuelve falso si ya se han visitado todos los elementos de p.
   Devuelve verdad en caso contrario.}

  procedimiento siguiente(e/s p:pila; sal e:elemento; sal error:booleano)
  {Implementa las operaciones "siguiente" y "avanza" de la especificación, es decir:
   Si existeSiguiente(p), error toma el valor falso, e toma el valor del siguiente
   elemento de la pila, y se avanza el iterador al elemento siguiente de
   la pila.
   Si no existeSiguiente(p), error toma el valor verdad, e queda indefinido y p
   queda como estaba.}

implementación

  tipos ptDato = ↑unDato;
        unDato = registro
                   dato:elemento;
                   sig:ptDato
                 freg
        pila = registro
                 cim:ptDato;
                 alt:natural;
                 iter:ptDato
               freg

  procedimiento crearVacía(sal p:pila)
  {Devuelve una pila vacía, sin elementos}
  principio
    p.cim:=nil:
    p.alt:=0;
  fin

  procedimiento apilar(e/s p:pila; ent e:elemento)
  {Devuelve la pila resultante de añadir e a p}
  variable aux:ptDato
  principio
    aux:=p.cim;
    nuevoDato(p.cim);
    p.cim↑.dato:=e;
    p.cim↑.sig:=aux;
    p.alt:=p.alt+1
  fin
```

```
procedimiento desapilar(e/s p:pila)
{Si p es no vacía, devuelve la pila resultante de eliminar de p el último elemento
 que fue apilado. Si p es vacía, la deja igual}
variable aux:ptDato
principio
  si p.alt≠0 entonces
    aux:=p.cim;
    p.cim:=p.cim↑.sig;
    disponer(aux);
    p.alt:=p.alt-1
  fsi
fin

función cima(p:pila) devuelve elemento
{Pre: p es no vacía} {Devuelve el último elemento apilado en p}
principio
  devuelve p.cim↑.dato
fin

función esVacía(p:pila) devuelve booleano
{Devuelve verdad si y sólo si p no tiene elementos}
principio
  devuelve (p.alt=0)
fin

función altura(p:pila) devuelve natural
{Devuelve el nº de elementos de p, 0 si no tiene elementos}
principio
  devuelve p.alt
fin

procedimiento duplicar(ent pilaEnt:pila; sal pilaSal:pila)
{Hace una copia en pilaSal de la pila almacenada en pilaEnt.}
variables ptSal,ptEnt:ptDato
principio
  si esVacía(pilaEnt) entonces
    crearVacía(pilaSal);
  sino
    ptEnt:=pilaEnt.cim;
    nuevoDato(pilaSal.cim);
    pilaSal.cim↑.dato:=ptEnt↑.dato;
    ptSal:=pilaSal.cim;
    ptEnt:=ptEnt↑.sig;
    mientrasQue ptEnt≠nil hacer
      nuevoDato(ptSal↑.sig);
      ptSal:=ptSal↑.sig;
      ptSal↑.dato:=ptEnt↑.dato;
      ptEnt:=ptEnt↑.sig
    fmq;
    ptSal↑.sig:=nil;
    pilaSal.alt:=pilaEnt.alt
  fsi
fin

función iguales(pila1,pila2:pila) devuelve booleano
{Devuelve verdad si y sólo si pila1 y pila2 almacenan la misma pila.}
variables pt1,pt2:ptDato; iguales:booleano:=verdad
principio
  si pila1.alt≠pila2.alt entonces
    devuelve falso;
  sino
    pt1:=pila1.cim;
    pt2:=pila2.cim;
    mientrasQue pt1≠nil and iguales hacer
      iguales:=pt1↑.dato=pt2↑.dato;
```

```
        pt1:=pt1↑.sig;
        pt2:=pt2↑.sig
      fmq;
      devuelve iguales
    fsi
  fin

  procedimiento liberar(e/s p:pila)
  {Devuelve la pila vacía, liberando previamente toda la memoria que ocupa la pila
  de entrada p.}
  variable aux:ptDato
  principio
    aux:=p.cim;
    mientrasQue aux≠nil hacer
      p.cim:=p.cim↑.sig;
      disponer(aux);
      aux:=p.cim
    fmq;
    p.alt:=0
  fin

  procedimiento iniciarIterador(e/s p:pila)
  {Prepara el iterador para que el siguiente elemento a visitar sea la
   cima de la pila p, si existe (situación de no haber visitado ningún elemento)}
  principio
    p.iter:=p.cima
  fin

  función existeSiguiente(p:pila) devuelve booleano
  {Devuelve falso si ya se han visitado todos los elementos de p.
   Devuelve verdad en caso contrario.}
  principio
    devuelve p.iter≠nil
  fin

  procedimiento siguiente(e/s p:pila; sal e:elemento; sal error:booleano)
  {Implementa las operaciones "siguiente" y "avanza" de la especificación, es decir:
   Si existeSiguiente(p), error toma el valor falso, e toma el valor del siguiente
   elemento de la pila, y se avanza el iterador al elemento siguiente de
   la pila.
   Si no existeSiguiente(p), error toma el valor verdad, e queda indefinido y p
   queda como estaba.}
  principio
    si existeSiguiente(p) entonces
      error:=falso;
      e:=p.iter↑.dato;
      p.iter:=p.iter↑.sig
    sino
      error:=verdad
    fsi
  fin

fin
```

Alternativamente, la función parcial "cima" se puede implementar de forma robusta, con el siguiente procedimiento:

```
  procedimiento cima(ent p:pila; sal e:elemento; sal error:booleano)
  {Si p es vacía, error toma el valor verdad y se deja e indefinido.
   Si p no es vacía, error toma el valor falso y e toma el valor de la cima de p.}
  principio
    si p.alt=0 entonces
      error:=verdad
    sino
      error:=falso;
      e:=p.cim↑.dato
```

```
        fsi
      fin
```

Un problema adicional generado por la elección de una representación dinámica para los valores de un TAD es la modificación de la semántica del **operador de asignación**, ':=', de un valor a una variable del TAD. Si en un algoritmo usuario de un TAD cuyos valores se representan dinámicamente se utiliza el operador de asignación, **no se duplica la representación dinámica** del valor asignado, sino que simplemente **se duplica la forma de acceder a ella (el puntero)**. Para evitar los problemas que esa utilización del operador de asignación puede plantear, hemos incluido la operación de "duplicar", que hace el papel de la asignación (en algunos lenguajes de programación, en la definición de un TAD es posible prohibir la asignación a variables del tipo utilizando el operador estándar ':=').

Finalmente, puede ser de utilidad en toda representación dinámica de los valores de un TAD, y en particular en el caso de las pilas, el disponer de una operación que libere toda la memoria dinámica utilizada para el almacenamiento de un valor, dejando indefinido o con algún valor destacado (la pila vacía, en el caso del TAD pila) el valor de la correspondiente variable del tipo. Por eso se ha incluido la operación "liberar".

Al igual que con la representación estática en base a un vector que vimos en la lección anterior, todas las operaciones tienen un tiempo de ejecución de $O(1)$, salvo las de duplicar, liberar y la comparación de igualdad. En el caso de esta representación dinámica, no hay límite a priori (en tiempo de compilación) para la altura de la pila. La limitación viene dada, en tiempo de ejecución, por el tamaño destinado a la memoria dinámica. Hay un coste adicional en memoria con respecto a la representación estática: el espacio ocupado por los punteros que sirven para encadenar los elementos.

2. Codificación en C++ (fragmento)

Incluimos la parte inicial de una posible codificación en C++ del TAD pila genérica con representación dinámica (enlazada con punteros). No utilizaremos clases (orientación a objetos) sino registros de C++. Los detalles de implementación de las operaciones quedan como ejercicio.

```cpp
// Interfaz del TAD. Pre-declaraciones:
template <typename Elem> struct Pila;

template <typename Elem> void crearVacia(Pila<Elem>& p);
template <typename Elem> void apilar(Pila<Elem>& p, const Elem& dato);
template <typename Elem> void desapilar(Pila<Elem>& p);
template <typename Elem> void cima(const Pila<Elem>& p, Elem& dato, bool& error);
template <typename Elem> bool esVacia(const Pila<Elem>& p);
template <typename Elem> int altura(const Pila<Elem>& p);

template <typename Elem> void duplicar(const Pila<Elem>& pOrigen, Pila<Elem>& pDestino);
template <typename Elem> bool operator==(const Pila<Elem>& p1, const Pila<Elem>& p2);
template <typename Elem> void liberar(Pila<Elem>& p);

template <typename Elem> void iniciarIterador(Pila<Elem>& p);
template <typename Elem> bool existeSiguiente(const Pila<Elem>& p);
template <typename Elem> bool siguiente(Pila<Elem>& p, Elem& dato);

// Declaración
template <typename Elem> struct Pila{

  friend void crearVacia<Elem>(Pila<Elem>& p);
  friend void apilar<Elem>(Pila<Elem>& p, const Elem& dato);
  friend void desapilar<Elem>(Pila<Elem>& p);
  friend void cima<Elem>(const Pila<Elem>& p, Elem& dato, bool& error);
  friend bool esVacia<Elem>(const Pila<Elem>& p);
  friend int altura<Elem>(const Pila<Elem>& p);

  friend void duplicar<Elem>(const Pila<Elem>& pOrigen, Pila<Elem>& pDestino);
  friend bool operator==<Elem>(const Pila<Elem>& p1, const Pila<Elem>& p2);
  friend void liberar<Elem>(Pila<Elem>& p);
```

```
  friend void iniciarIterador<Elem>(Pila<Elem>& p);
  friend bool existeSiguiente<Elem>(const Pila<Elem>& p);
  friend bool siguiente<Elem>(Pila<Elem>& p, Elem& dato);

// Representación de los valores del TAD
private:
  struct Nodo{
    Elem valor;
    Nodo* sig;
  };

  Nodo* laCima;
  int numDatos;
  Nodo* iter;
};

// Implementación de las operaciones
template<typename Elem> void crearVacia(Pila<Elem>& p){
  p.numDatos = 0;
  p.laCima = nullptr;
}

template <typename Elem> void apilar(Pila<Elem>& p, const Elem& dato){
  typename Pila<Elem>::Nodo* aux = new typename Pila<Elem>::Nodo;
  aux->valor=dato;
  aux->sig=p.laCima;
  p.laCima=aux;
  p.numDatos++;
}

// etc, etc, implementación de las demás operaciones (ejercicio)
```

3. Ejemplo de aplicación del TAD pila: evaluación de expresiones postfijas

Una expresión aritmética en **notación postfija** (o notación polaca inversa) es una secuencia formada por símbolos de dos tipos diferentes: **operadores** (para simplificar, consideraremos únicamente los operadores aritméticos binarios `+`, `-`, `*` y `/`) y **operandos** (para simplificar pensaremos en identificadores de una sola letra). Cada **operador se escribe detrás** de sus operandos.

Por ejemplo, a la expresión siguiente, escrita en la notación habitual (**infija**):

`a*b/c`

le corresponde la siguiente expresión en notación postfija:

`ab*c/`

Si en la expresión infija aparecen **paréntesis**, éstos cambian la correspondiente expresión postfija sólo si los paréntesis alteran el orden de prioridad de los operadores:

`a/b+c*d-e*f`, traducido a notación postfija es: `ab/cd*+ef*-`

`a/(b+c)*(d-e)*f`, se traduce en cambio por: `abc+/de-*f*`

Tres **ventaja**s importantes de la notación postfija frente a la convencional infija son las siguientes:

- En notación postfija **nunca son necesarios los paréntesis**.
- En notación postfija **no es necesario definir prioridades** entre operadores.
- Una expresión postfija puede **evaluarse de forma muy sencilla**, como veremos enseguida.

En el siguiente apartado de la lección veremos un algoritmo para traducir expresiones de notación infija a notación postfija. Veamos ahora cómo evaluar una expresión escrita en notación postfija.

Una expresión en notación postfija puede ser **evaluada** haciendo un **recorrido de izquierda a derecha**. Cuando se encuentra un operando, se apila en una **pila de operandos**. Cuando se encuentra un operador, se desapilan dos operandos

de la pila, se realiza la correspondiente operación y el resultado se apila en la pila de operandos. Este método de evaluación es mucho más sencillo que el proceso necesario para evaluar una expresión escrita en notación infija.

A continuación, se presenta una función de evaluación de expresiones en notación postfija:

```
función evaluar(e:expresión) devuelve real
{Devuelve el valor real de la expresión postfija e.}
importa pilasDeSímbolos {es el módulo genérico de pilas particularizado para
           elementos de tipo símbolo, que también se define en este módulo}
variables s,o1,o2,r:símbolo; p:pilaDeSímbolos
principio
  crearVacía(p);
  s:=siguienteSímbolo(e);
  mientrasQue s≠final hacer
    si esOperando(s) entonces
      apilar(p,s)
    sino
      o2:=cima(p);
      desapilar(p);
      o1:=cima(p);
      desapilar(p);
      r:=operar(o1,o2,s);
      apilar(p,r)
    fsi;
    s:=siguienteSímbolo(e)
  fmq;
  devuelve cima(p);
  desapilar(p)
fin
```

Se ha supuesto lo siguiente:

- Se usa el módulo `pilasDeSímbolos`, en el cual se ha definido el tipo `símbolo`, cuyos valores pueden ser operandos, operadores o el valor especial `final`; además, existe la función `esOperando` que devuelve verdad si y sólo si el símbolo enviado como argumento es un operando; existe también la función `operar`, que a partir de dos operandos y un operador devuelve un nuevo operando que es el resultado de realizar la operación.
- El TAD genérico `pila` se supone particularizado para datos de tipo `símbolo` en el módulo `pilasDeSímbolos`, es decir, el parámetro formal `elemento` de la especificación del TAD `pila` debe sustituirse por el tipo `símbolo`.
- Se supone predefinido el tipo `expresión` como una secuencia de `símbolos`. Además, existe la función `siguienteSímbolo` que devuelve el siguiente `símbolo` de una `expresión`.

4. Otro ejemplo de aplicación: traducción de expresiones infijas a postfijas

Una vez resuelto el problema de evaluar una expresión escrita en notación postfija, podemos plantearnos el de realizar la traducción de expresiones infijas a postfijas para, así, tener resuelto el problema de evaluar expresiones infijas.

Comparando una expresión infija con su correspondiente postfija, puede verse en primer lugar que los **operandos mantienen el mismo orden** en ambas notaciones. Por tanto, únicamente hay que "mover" operadores y quitar paréntesis (si existen).

Un primer algoritmo de traducción es el siguiente:

- Añadir a la expresión un par de paréntesis por cada operador. Esto significa añadir paréntesis redundantes con las reglas de prioridad. Por ejemplo, de la expresión:
  `a/b+c*d-e*f`
  pasar a la expresión:
  `(((a/b)+(c*d))-(e*f))`
- Mover todos los operadores de forma que sustituyan a sus correspondientes paréntesis derechos. En el ejemplo anterior:
  `(((ab/(cd*+(ef*-`
- Borrar todos los paréntesis izquierdos. Es decir:

ab/cd*+ef*-

El **problema** del algoritmo anterior es que **requiere dos recorridos** de la expresión para traducirla: el primero para añadir todos los paréntesis redundantes y el segundo para mover los operadores y eliminar paréntesis.

La solución a ese problema se obtiene de la siguiente forma: puesto que el orden de los operandos es el mismo, cada vez que es leído un operando, se escribe en el resultado; cada vez que se lee un operador, hay que decidir si debe escribirse ya en el resultado o almacenarse en un dato auxiliar (veremos que será una pila) hasta el momento de su escritura.

Por ejemplo, para traducir la expresión a+b*c a su correspondiente postfija abc*+, hay que realizar los siguientes pasos:

siguiente símbolo	pila	resultado	comentario
	vacía		inicialización de la pila
a	vacía	a	el operando se escribe ya
+	+	a	el operador se apila porque no hay otro
b	+	ab	el operando se escribe ya
*	+*	ab	* tiene más prioridad que +, luego se apila
c	+*	abc	el operando se escribe ya
final	vacía	abc*+	se vuelca la pila sobre el resultado

Veamos ahora un ejemplo con paréntesis: la expresión a*(b+c)/d.

siguiente símbolo	pila	resultado	comentario
	vacía		inicialización de la pila
a	vacía	a	el operando se escribe ya
*	*	a	el operador se apila porque no hay otro
(	*(	a	el paréntesis izquierdo se apila siempre
b	*(	ab	el operando se escribe ya
+	*(+	ab	a un paréntesis izquierdo sólo lo saca el derecho
c	*(+	abc	el operando se escribe ya
)	*	abc+	se desapila hasta el paréntesis izquierdo, escribiendo el + que se ha desapilado
/	/	abc+*	/ tiene igual prioridad que *, luego se saca * de la pila y se mete /
d	/	abc+*d	el operando se escribe ya
final	vacía	abc+*d/	se vuelca la pila sobre el resultado

En general, el problema se resuelve de la siguiente forma:

- a cada operador y a los paréntesis se les asigna una “prioridad en pila” para cuando están dentro de la pila y otra “prioridad de llegada” para cuando son leídos de la entrada;
- cada vez que se lee un operador o un paréntesis izquierdo, si la pila es vacía, se apila, si no, se compara su prioridad de llegada con la prioridad en pila del símbolo que está en la cima de la pila; debe sacarse el operador de la cima de la pila cuando su prioridad en pila es mayor o igual que la prioridad de llegada del nuevo operador leído, y repetir el proceso con el nuevo operador que queda en la cima;
- cada vez que se lee un paréntesis derecho, debe desapilarse el operador de la cima, escribirse en el resultado y desapilarse también el paréntesis izquierdo que queda en la cima de la pila;
- las prioridades en pila y de llegada de los operadores +, –, * y /, y del paréntesis izquierdo son:

símbolo	prioridad en pila	prioridad de llegada
*, /	2	2
+, -	1	1
(	0	3

El algoritmo que resuelve el problema de la traducción infija a postfija es el siguiente:

```
procedimiento traducir(ent in:expresión; sal post:expresión)
{Traduce la expresión infija in a su correspondiente postfija (post).}
importa pilasDeSímbolos
variables p:pilaDeSímbolos; s:símbolo

  función prioridadDeLaPila(p:pila) devuelve 0..2
  { Pre: p es una pila de símbolos s ∈ {(,+,-,*,/} }
  { Post: p=pilaVacía ⇒ prioridadDeLaPila(p)=0;
          p≠pilaVacía ∧ cima(p)=( ⇒ prioridadDeLaPila(p)=0;
          p≠pilaVacía ∧ cima(p)∈{+,-} ⇒ prioridadDeLaPila(p)=1;
          p≠pilaVacía ∧ cima(p)∈{*,/} ⇒ prioridadDeLaPila(p)=2 }

  función prioridadDeLlegada(s:símbolo) devuelve 1..3
  { Pre: s ∈ {(,+,-,*,/} }
  { Post: s = ( ⇒ prioridadDeLlegada(s)=3;
          (s = +) ∨ (s = -) ⇒ prioridadDeLlegada(s)=1;
          (s = *) ∨ (s = /) ⇒ prioridadDeLlegada(s)=2 }

principio
  crearVacía(p);
  iniciaExpresión(post);
  s:=siguienteSímbolo(in);
  mientrasQue s≠final hacer
    si esOperando(s) entonces
      añadeSímbolo(post,s)
    sino
      si esParéntesisDerecho(s) entonces
        mientrasQue not esParéntesisIzquierdo(cima(p)) hacer
          añadeSímbolo(post,cima(p));
          desapilar(p)
        fmq;
        desapilar(p) {quitar el paréntesis izquierdo}
      sino
        mientrasQue
          prioridadDeLaPila(p)≥prioridadDeLlegada(s) hacer
          añadeSímbolo(post,cima(p));
          desapilar(p)
        fmq;
        apilar(p,s)
      fsi
    fsi;
    s:=siguienteSímbolo(in)
  fmq;
  mientrasQue not esVacía(p) hacer
    añadeSímbolo(post,cima(p));
    desapilar(p)
  fmq;
  añadeSímbolo(post,final)
fin
```

Se ha supuesto lo siguiente:

- Además de la función `esOperando`, existen las funciones booleanas `esParéntesisDerecho` y `esParéntesisIzquierdo`.
- Existen los algoritmos `iniciaExpresión` y `añadeSímbolo` que crean la expresión vacía y añaden un nuevo símbolo a la derecha de una expresión, respectivamente.

Lección 9

El TAD cola genérica

Indice

1. Concepto de cola y especificación
2. Representación dinámica e implementación de operaciones
3. Representación estática circular
4. Ejemplo de aplicación: simulación de una cola de espera

1. Concepto de cola y especificación

Una cola es, al igual que una pila, una secuencia de elementos de un cierto tipo, dispuestos en una dimensión (**tipo lineal** de datos). En el tipo cola hay un valor especial que se denomina **cola vacía**. Además, a una cola se pueden **añadir** y **eliminar** elementos, y puede observarse el **primer elemento** de la cola.

El comportamiento intuitivo de una cola es el siguiente (piénsese, por ejemplo, en la cola que se forma ante una ventanilla de un determinado servicio de la Administración Pública): los nuevos elementos son añadidos siempre por un mismo extremo de la cola (el final) mientras que las eliminaciones se realizan por el extremo opuesto (el principio de la cola). Si se interpreta que el primer elemento de la cola está recibiendo un cierto servicio (está siendo atendido por el funcionario a cargo de la ventanilla), el orden en que los elementos reciben el correspondiente servicio y salen del sistema (son eliminados) debe coincidir con el orden de su llegada (operación de añadir) al sistema. Se dice, por tanto, que las colas son estructuras **FIFO** (del inglés, *First In, First Out*).

La **especificación** del TAD (genérico) cola es la siguiente:

espec colasGenéricas
usa boleanos, naturales
parámetro formal
  género elemento
fpf
género cola *{Los valores del TAD cola representan secuencias de elementos con acceso FIFO (first in, first out), esto es, el primer elemento añadido será el primero en ser borrado}*
operaciones
  crear: -> cola
  {Devuelve una cola vacía, sin elementos}

  encolar: cola c , elemento e -> cola
  {Devuelve la cola resultante de añadir e a c}

  esVacía?: cola c -> booleano
  {Devuelve verdad si y sólo si c no tiene elementos}

  parcial primero: cola c -> elemento
  {Devuelve el primer elemento encolado de los que hay en c. Parcial: la operación no está definida si c es vacía}

  desencolar: cola c -> cola
  *{Si c es no vacía, devuelve la cola resultante de eliminar de c el primer elemento que fue encolado.
  Si c es vacía, devuelve una cola igual a c }*

  longitud: cola c -> natural

{Devuelve el nº de elementos de c}

{Las cuatro siguientes operaciones de un Iterador definido sobre las colas}
iniciarIterador: cola c -> cola
{Prepara el iterador para que el siguiente elemento a visitar sea el primero de la cola c, si existe (situación de no haber visitado ningún elemento)}

existeSiguiente?: cola c -> booleano
{Devuelve falso si ya se han visitado todos los elementos de c, devuelve verdad en caso contrario}

parcial siguiente: cola c -> elemento
*{Devuelve el siguiente elemento de c.
Parcial: la operación no está definida si no existeSiguiente?(c)}*

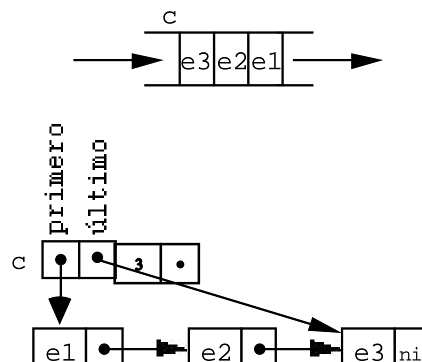
parcial avanza: cola c -> cola
*{Devuelve la cola resultante de avanzar el iterador en c.
Parcial: la operación no está definida si no existeSiguiente?(c)}*

fespec

2. Representación dinámica e implementación de operaciones

La representación dinámica de un dato de tipo cola puede realizarse, de forma análoga a la del tipo pila, **encadenando o enlazando mediante punteros los elementos** de la cola. Para ello, una variable de tipo puntero apunta al dato, creado dinámicamente, en que se almacena el primer elemento de la cola; junto a éste, a su vez, se almacena otro puntero que guarda la dirección del segundo elemento de la cola, etcétera. De esta forma, las operaciones de eliminar y observar el primero de la cola pueden ejecutarse en un tiempo en $O(1)$. Para poder ejecutar en igual tiempo las operaciones de añadir a la cola, es necesaria una variable puntero adicional que apunte siempre al último elemento de la cola (aquél añadido más recientemente). Incluimos además en la representación de la cola, un campo para guardar su longitud y un puntero para implementar el iterador.

```
tipos ptDato = ↑unDato;
      unDato = registro
                 dato:elemento;
                 sig:ptDato
               freg
      cola = registro
               pri,ult:ptDato;
               long:natural;
               iter:ptDato
             freg
```



En la implementación del TAD que sigue, se han añadido a las especificadas en la sección anterior, las operaciones de duplicar colas, igualdad entre dos colas y liberación de memoria dinámica.

```
módulo genérico colasGenéricas
parámetro
  tipo elemento
exporta

  tipo cola {Los valores del TAD cola representan secuencias de elementos con
             acceso FIFO (first in, first out), esto es, el primer elemento añadido
             será el primero en ser borrado.}
```

```
  procedimiento crearVacía(sal c:cola)
  {Devuelve en c la cola vacía, sin elementos}

  procedimiento encolar(e/s c:cola; ent e:elemento)
  {Devuelve en c la cola resultante de añadir e a c}

  función esVacía(c:cola) devuelve booleano
  {Devuelve verdad si y sólo si c no tiene elementos}

  procedimiento primero(ent c:cola; sal e:elemento; sal error:booleano)
  {Si c es no vacía, devuelve en e el primer elemento añadido a c y error=falso.
   Si c es vacía, devuelve error=verdad y e queda indefinido}

  procedimiento desencolar(e/s c:cola)
  {Si c es no vacía, devuelve en c la cola resultante de eliminar de c el primer
   elemento que fue añadido. Si c es vacía, la deja igual}

  función longitud(c:cola) devuelve natural
  {Devuelve el número de elementos de c}

  procedimiento duplicar(sal colaSal:cola; ent colaEnt:cola)
  {Devuelve en colaSal una cola igual a colaEnt, duplicando la representación
   en memoria}

  función iguales(cola1,cola2:cola) devuelve booleano
  {Devuelve verdad si y sólo si cola1 y cola2 tienen los mismos elementos y en
   las mismas posiciones}

  procedimiento liberar(e/s c:cola)
  {Devuelve en c la cola vacía y además libera la memoria utilizada previamente por c}

  procedimiento iniciarIterador(e/s c:cola)
  {Prepara el iterador para que el siguiente elemento a visitar sea un primer
   elemento de c, si existe (situación de no haber visitado ningún elemento)}

  función existeSiguiente(c:cola) devuelve booleano
  {Devuelve falso si ya se han visitado todos los elementos de c; devuelve cierto en
   caso contrario}

  procedimiento siguiente(e/s c:cola; sal e:elemento; sal error:booleano)
  {Si existe algún elemento de c pendiente de visitar, devuelve en e el siguiente
   elemento a visitar y error=falso, y además avanza el iterador para que a
   continuación se pueda visitar otro elemento de c. Si no quedan elementos pendientes
   de visitar devuelve error=verdad y e queda indefinido}

implementación
  tipos ptDato = ↑unDato;
        unDato = registro
                   dato:elemento;
                   sig:ptDato
                 freg;
        cola = registro
                 pri,ult:ptDato;
                 long:natural;
                 iter:ptDato {se utiliza para implementar el iterador}
               freg

  procedimiento crearVacía(sal c:cola)
  {Devuelve en c la cola vacía, sin elementos}
  principio
    c.pri:=nil;
    c.ult:=nil;
    c.long:=0
  fin

  procedimiento encolar(e/s c:cola; ent e:elemento)
  {Devuelve en c la cola resultante de añadir e a c}
```

```
principio
  si c.long=0 entonces
    nuevoDato(c.ult);
    c.pri:=c.ult
  sino
    nuevoDato(c.ult↑.sig);
    c.ult:=c.ult↑.sig
  fsi;
  c.ult↑.dato:=e;
  c.ult↑.sig:=nil;
  c.long:=c.long+1
fin

función esVacía(c:cola) devuelve booleano
{Devuelve verdad si y sólo si c no tiene elementos}
principio
  devuelve c.pri=nil
fin

procedimiento primero(ent c:cola; sal e:elemento; sal error:booleano)
{Si c es no vacía, devuelve en e el primer elemento añadido a c y error=falso.
 Si c es vacía, devuelve error=verdad y e queda indefinido}
principio
  si esVacía(c) entonces
    error:=verdad
  sino
    error:=falso;
    e:=c.pri↑.dato
  fsi
fin

procedimiento desencolar(e/s c:cola)
{Si c es no vacía, devuelve en c la cola resultante de eliminar de c el primer
 elemento que fue añadido. Si c es vacía, la deja igual}
variable aux:ptDato
principio
  si not esVacía(c) entonces
    aux:=c.pri;
    c.pri:=c.pri↑.sig;
    disponer(aux);
    c.long:=c.long-1;
    si c.long=0 entonces c.ult:=nil fsi
  fsi
fin

función longitud(c:cola) devuelve natural
{Devuelve el número de elementos de c}
principio
  devuelve c.long
fin

procedimiento duplicar(sal colaSal:cola; ent colaEnt:cola)
{Devuelve en colaSal una cola igual a colaEnt, duplicando la representación
 en memoria}
variables ptSal,ptEnt:ptDato
principio
  si esVacía(colaEnt) entonces
    crearVacía(colaSal);
  sino
    ptEnt:=colaEnt.pri;
    nuevoDato(colaSal.pri);
    colaSal.pri↑.dato:=ptEnt↑.dato;
    ptSal:=colaSal.pri;
    ptEnt:=ptEnt↑.sig;
    mientrasQue ptEnt≠nil hacer
      nuevoDato(ptSal↑.sig);
```

```
      ptSal:=ptSal↑.sig;
      ptSal↑.dato:=ptEnt↑.dato;
      ptEnt:=ptEnt↑.sig
    fmq;
    ptSal↑.sig:=nil;
    colaSal.ult:=ptSal;
    colaSal.long:=colEnt.long
  fsi
fin

función iguales(cola1,cola2:cola) devuelve booleano
{Devuelve verdad si y sólo si cola1 y cola2 tienen los mismos elementos y en
las mismas posiciones}
variables pt1,pt2:ptDato; iguales:booleano:=verdad
principio
  si cola1.long≠cola2.long entonces
    devuelve falso;
  sino
    pt1:=cola1.pri;
    pt2:=cola2.pri;
    mientrasQue pt1≠nil and iguales hacer
      iguales:=pt1↑.dato=pt2↑.dato;
      pt1:=pt1↑.sig;
      pt2:=pt2↑.sig
    fmq;
    devuelve iguales
  fsi
fin

procedimiento liberar(e/s c:cola)
{Devuelve en c la cola vacía y además libera la memoria utilizada previamente por c}
variable aux:ptDato
principio
  aux:=c.pri;
  mientrasQue aux≠nil hacer
    c.pri:=c.pri↑.sig;
    disponer(aux);
    aux:=c.pri
  fmq;
  c.ult:=nil;
  c.long:=0
fin

procedimiento iniciarIterador(e/s c:cola)
{Prepara el iterador para que el siguiente elemento a visitar sea un primer
 elemento de c, si existe (situación de no haber visitado ningún elemento)}
principio
  c.iter:=c.pri
fin

función existeSiguiente(c:cola) devuelve booleano
{Devuelve falso si ya se han visitado todos los elementos de c; devuelve cierto en
 caso contrario}
principio
  devuelve c.iter≠nil
fin

procedimiento siguiente(e/s c:cola; sal e:elemento; sal error:booleano)
{Si existe algún elemento de c pendiente de visitar, devuelve en e el siguiente
 elemento a visitar y error=falso, y además avanza el iterador para que a
 continuación se pueda visitar otro elemento de c. Si no quedan elementos pendientes
 de visitar devuelve error=verdad, e queda indefinido y c queda como estaba}
principio
  si existeSiguiente(c) entonces
    error:=falso;
```

```
      e:=c.iter↑.dato;
      c.iter:=c.iter↑.sig
    sino
      error:=verdad
    fsi
  fin

fin
```

3. Representación estática circular

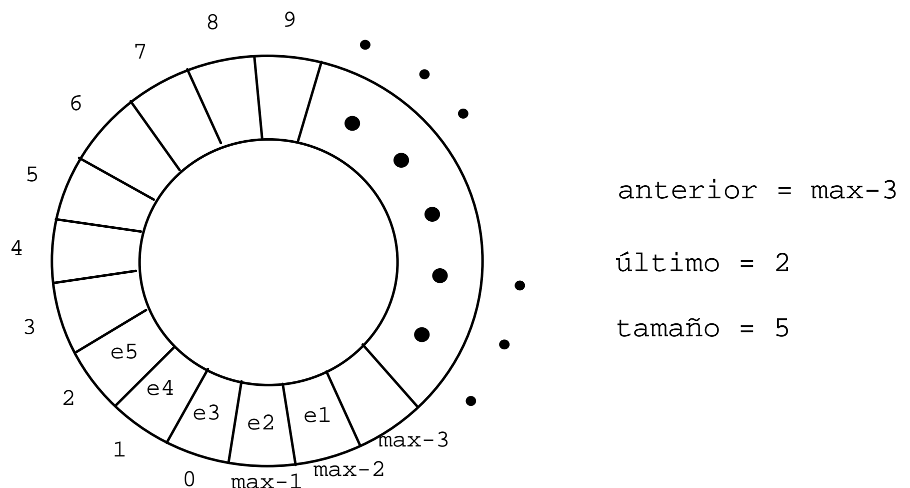
La representación estática o contigua de una cola se realiza mediante un vector con un espacio para un máximo `max` de elementos y dos contadores, `anterior` y `último`, que memoricen el índice de la componente anterior a aquélla en la que se encuentra el primer elemento de la cola y el índice donde se encuentra el último elemento de la cola, respectivamente. Al igual que sucede con la representación estática del TAD pila, esta representación hace que la operación añadir haya que implementarla como una operación parcial (sólo pueden representarse colas de longitud menor o igual que `max`).

El vector soporte debe considerarse como si fuese una estructura circular, es decir, las operaciones con los índices del vector deben realizarse con aritmética modular (módulo `max`).

En la figura se representa un ejemplo de cola `(e1,e2,e3,e4,e5)` almacenada en un vector (considerado como circular).

En este ejemplo, el índice anterior (en el sentido contrario a las agujas del reloj) al del primer elemento de la cola (`e1`) es `max-3`; el índice del último elemento de la cola (`e5`) es `2`.

En esta representación, `tamaño = 0` si y sólo si la secuencia es vacía. La capacidad máxima es de `max` elementos.




```
módulo genérico colasGenéricasEstáticas
parámetro
  tipo elemento

exporta
  constante max = 100
  tipo cola   {Los valores del TAD cola representan secuencias de 0 o más elementos,
              con longitud máxima max; llamamos primer elemento de la cola al primero
              que fue añadido y último al último que fue añadido}

  procedimiento crearVacía(sal c:cola)
  {Devuelve una cola vacía, sin elementos}

  procedimiento encolar(ent e:elemento; e/s c:cola; sal error:booleano)
  {Si c no está llena, añade e a c como último elemento; si está llena, devuelve error}

  función esVacía(c:cola) devuelve booleano
  {Devuelve verdad si y sólo si c no tiene elementos}

  procedimiento desencolar(e/s c:cola)
  {Si c es no vacía, devuelve en c la cola resultante de eliminar de c el primer
   elemento que fue añadido. Si c es vacía, la deja igual}
```

```
  procedimiento primero(ent c:cola; sal e:elemento; sal error:booleano)
  {Si c es no vacía, devuelve en e su primer elemento; si es vacía, devuelve error}

  función longitud(c:cola) devuelve natural
  {Devuelve el número de elementos de c}

  procedimiento duplicar(sal cSal:cola; ent cEnt:cola)
  {Duplica la representación de la cola cEnt en la cola cSal}

  función iguales(c1,c2:cola) devuelve booleano
  {Devuelve verdad si y sólo si las colas c1 y c2 tienen la misma longitud y los
   mismos elementos en idénticas posiciones}

  procedimiento iniciarIterador(e/s c:cola)
  {Prepara el iterador para que el siguiente elemento a visitar sea el
   primero de la cola (situación de no haber visitado ningún elemento)}

  función existeSiguiente(c:cola) devuelve booleano
  {Devuelve falso si ya se ha visitado el último elemento, cierto en caso contrario}

  procedimiento siguiente(e/s c:cola; sal e:elemento: sal error:booleano)
  {Si existe algún elemento de c pendiente de visitar, devuelve en e el siguiente
   elemento a visitar y error=falso, y además avanza el iterador para que a
   continuación se pueda visitar otro elemento de c. Si no quedan elementos
   pendientes de visitar devuelve error=verdad, e queda indefinido y c queda
   como estaba}

implementación
  tipos vectorDatos = vector[0..max-1] de elemento;
        cola = registro
                 datos:vectorDatos;
                 anterior,último,iter:0..max-1; {iter se usa para implementar el iterador}
                 tamaño:0..max;
                 iterEnInicio:booleano {se usa también para implementar el iterador}
               freg

  procedimiento crearVacía(sal c:cola)
  {Devuelve una cola vacía, sin elementos}
  principio
    c.anterior:=0;  {por ejemplo}
    c.último:=0;
    c.tamaño:=0
  fin

  procedimiento encolar(e/s c:cola; ent e:elemento; sal error:booleano)
  {Si c no está llena, añade e a c como último elemento; si está llena, devuelve error}
  principio
    si c.tamaño<max entonces
      error:=falso;
      c.último:=(c.último+1) mod max;
      c.datos[c.último]:=e;
      c.tamaño:=c.tamaño+1
    sino
      error:=verdad
    fsi
  fin

  función esVacía(c:cola) devuelve booleano
  {Devuelve verdad si y sólo si c no tiene elementos}
  principio
    devuelve c.tamaño=0
  fin

  procedimiento desencolar(e/s c:cola)
  {Si c es no vacía, devuelve en c la cola resultante de eliminar de c el primer
   elemento que fue añadido. Si c es vacía, la deja igual}
```

```
principio
  si not esVacía(c) entonces
    c.anterior:=(c.anterior+1) mod max;
    c.tamaño:=c.tamaño-1
  fsi
fin

procedimiento primero(ent c:cola; sal e:elemento; sal error:booleano)
{Si c es no vacía, devuelve en e su primer elemento; si es vacía, devuelve error}
principio
  si esVacía(c) entonces
    error:=verdad
  sino
    error:=falso;
    e:=c.datos[(c.anterior+1) mod max]
  fsi
fin

función longitud(c:cola) devuelve natural
{Devuelve el número de elementos de c}
principio
  devuelve c.tamaño
fin

procedimiento duplicar(sal cSal:cola; ent cEnt:cola)
{Duplica la representación de la cola cEnt en la cola cSal}
variable i:natural
principio
  si esVacía(cEnt) entonces
    crear(cSal)
  sino
    cSal.tamaño:=cEnt.tamaño;
    cSal.anterior:=cEnt.anterior;
    cSal.último:=cEnt.último;
    para i:=1 hasta cEnt.tamaño hacer
      cSal.datos[(cSal.anterior+i) mod max]:=cEnt.datos[(cEnt.anterior+i) mod max]
    fpara
  fsi
fin

función iguales(c1,c2:cola) devuelve booleano
{Devuelve verdad si y sólo si las colas c1 y c2 tienen la misma longitud y los
 mismos elementos en idénticas posiciones}
variable i:natural; igual:booleano
principio
  si esVacía(c1) and esVacía(c2) entonces
    devuelve verdad
  sino_si longitud(c1)≠longitud(c2) entonces
    devuelve falso
  sino
    i:=1;
    igual:=verdad;
    mientrasQue igual and i≤longitud(c1) hacer
      igual:=c1.datos[(c1.anterior+i) mod max]=c2.datos[(c2.anterior+i) mod max];
      i:=i+1
    fmq;
    devuelve igual
  fsi
fin

procedimiento iniciarIterador(e/s c:cola)
{Prepara el iterador para que el siguiente elemento a visitar sea el
 primero de la cola (situación de no haber visitado ningún elemento)}
principio
  c.iterEnInicio:=(c.tamaño>0);
  c.actual:=(c.anterior+1) mod max
fin
```

```
  función existeSiguiente(c:cola) devuelve booleano
  {Devuelve falso si ya se ha visitado el último elemento, cierto en caso contrario}
  principio
    devuelve (c.actual≠(c.último+1) mod max) OR
             ((c.actual=(c.último+1) mod max) AND c.iterEnInicio)
  fin

  procedimiento siguiente(e/s c:cola; sal e:elemento: sal error:booleano)
  {Si existe algún elemento de c pendiente de visitar, devuelve en e el siguiente
   elemento a visitar y error=falso, y además avanza el iterador para que a
   continuación se pueda visitar otro elemento de c. Si no quedan elementos
   pendientes de visitar devuelve error=verdad, e queda indefinido y c queda
   como estaba}
  principio
    si existeSiguiente(c) entonces
      error:=falso;
      e:=c.datos[c.actual];
      c.actual:=(c.actual+1) mod max;
      si c.iterEnInicio entonces
        c.iterEnInicio:=falso
      fsi
    sino
      error:=verdad
    fsi
  fin

fin
```

Nótese que el coste temporal de todas las operaciones es $O(1)$, excepto para las de duplicar y comparación de igualdad.

4. Ejemplo de aplicación: simulación de una cola de espera

Cada día tenemos que **hacer cola** en numerosas ocasiones para obtener un cierto "servicio" por parte de algún agente o "servidor". No debe sorprendernos, por tanto, que el uso de colas sea importante en muchas aplicaciones informáticas.

Ejemplos de colas se dan en las siguientes situaciones: personas esperando ante ventanillas de bancos o aeropuertos, coches esperando en una calle ante un semáforo en rojo o en una autopista ante un puesto de peaje, procesos generados por los usuario de un sistema informático multiusuario esperando a ser ejecutados por el procesador, llamadas telefónicas recibidas en una centralita esperando hasta obtener línea libre hacia una determinada extensión, piezas de un determinado tipo esperando en un almacén para ser tratadas por una máquina en un sistema de fabricación, etcétera.

Todas las situaciones anteriores pueden ser **simuladas** con un computador utilizando una variable del TAD cola que sirva para **modelar** o imitar la cola que se produce en la realidad y poder responder a preguntas del tipo de: ¿cuánto tiempo tiene que esperar, en media, un cliente para obtener un servicio?, ¿cuál es la longitud media de la cola?, ¿cuál es la varianza de las medidas anteriores?

Vamos a estudiar el caso más sencillo. Hay un solo servidor al que llegan clientes de forma **aleatoria** y el servicio de cada uno de ellos le toma un tiempo fijo al servidor (tiempo de servicio). Los parámetros de entrada para ejecutar una simulación con un computador son: la probabilidad de que durante un intervalo de tiempo de un minuto se produzca la llegada de un cliente, el tiempo (fijo) de servicio a un cliente y la longitud total del intervalo de tiempo que se quiere simular.

Para llevar a cabo la simulación utilizaremos una cola (entendida aquí como una variable del TAD cola) en la que almacenar los clientes esperando para ser servidos. Cada elemento de la cola será, en realidad, el instante (entero no negativo) en el que llegó el cliente.

```
tipo instanteDeLlegada = 0..maxEntero
```

Utilizaremos, además, una variable `tiempo` (entera no negativa) que modela el **reloj**; su valor inicial será cero y se incrementará de uno en uno, en cada paso de la simulación, contando los minutos transcurridos.

Cuando el cliente llega al principio de la cola, es decir, ante el servidor, la diferencia entre el tiempo en ese momento y el instante de su llegada a la cola es el número de minutos que ese cliente ha esperado en la cola (puede ser cero si cuando

llega el cliente la cola está vacía). Podemos sumar todos esos tiempos de espera hasta el final del tiempo total de simulación y dividir la suma por el número de clientes que han llegado, y obtener, así, el tiempo medio de espera de un cliente en la cola.

Para modelar las llegadas aleatorias, utilizaremos una función `random` que nos devuelve un número (pseudo-)aleatorio uniformemente distribuido en el intervalo (0,1). Para decidir si un cliente llega o no durante un intervalo de tiempo de un minuto, preguntamos si el valor obtenido con la función `random` es menor o no que el valor de la probabilidad de llegada de un pasajero en un minuto cualquiera (valor, este último, solicitado como parámetro de entrada antes de comenzar la simulación).

El algoritmo esbozado previamente es como sigue:

```
procedimiento simulador
importa colasDeInstantesDeLlegada
variables cola:colaDeInstantesDeLlegada;
          probabilidadLlegada,esperaMedia:real;
          tiempoServicio,tiempoSimulación,tiempo,
          tiempoQuedaServicio,númeroClientes,
          sumaDeEsperas,instanteLlegada:0..maxEntero

función random devuelve real
{Devuelve un número (pseudo-)aleatorio en el intervalo (0,1). No la implementamos.}

principio
  escribir('DATOS');
  escribir('Probabilidad de que llegue un cliente durante un minuto:');
  leer(probabilidadLlegada);
  escribir('Tiempo requerido por cada servicio (en minutos):');
  leer(tiempoServicio);
  escribir('Longitud de la simulación (tiempo total, en minutos):');
  leer(tiempoSimulación);

  creaVacía(cola);
  tiempo:=0;
  tiempoQuedaServicio:=0; { el servidor está libre, en principio }
  númeroClientes:=0;
  sumaDeEsperas:=0;

  mientrasQue tiempo ≤ tiempoSimulación hacer
    si random < probabilidadLlegada entonces
      añadir(cola,tiempo)
    fsi;

    si tiempoQuedaServicio = 0 entonces
      si not esVacía(cola) entonces
        instanteLlegada:=primero(cola);
        eliminar(cola);
        sumaDeEsperas:=sumaDeEsperas+(tiempo-instanteLlegada);
        númeroClientes:=númeroClientes+1;
        tiempoQuedaServicio:=tiempoServicio
      fsi
    fsi;

    tiempo:=tiempo+1;

    si tiempoQuedaServicio > 0 entonces
      tiempoQuedaServicio:=tiempoQuedaServicio-1
    fsi
  fmq;

  si númeroClientes = 0 entonces
    esperaMedia:=0.0
  sino
    esperaMedia:=sumaDeEsperas/númeroClientes
  fsi;
  escribir('RESULTADOS');
```

```
  escribir('Número de clientes servidos: ',númeroClientes);
  escribir('Tiempo medio de espera (en minutos): ', esperaMedia)
fin
```

A continuación, se muestra el resultado obtenido al ejecutar tres veces el algoritmo anterior para datos idénticos:

```
DATOS
Probabilidad de que llegue un cliente durante un minuto: 0.10
Tiempo requerido por cada servicio (en minutos): 5
Longitud de la simulación (tiempo total, en minutos): 200
RESULTADOS
Número de clientes servidos: 16
Tiempo medio de espera (en minutos): 0.25

DATOS
Probabilidad de que llegue un cliente durante un minuto: 0.10
Tiempo requerido por cada servicio (en minutos): 5
Longitud de la simulación (tiempo total, en minutos): 200
RESULTADOS
Número de clientes servidos: 29
Tiempo medio de espera (en minutos): 3.93

DATOS
Probabilidad de que llegue un cliente durante un minuto: 0.10
Tiempo requerido por cada servicio (en minutos): 5
Longitud de la simulación (tiempo total, en minutos): 200
RESULTADOS
Número de clientes servidos: 19
Tiempo medio de espera (en minutos): 0.68
```

En los tres casos, un cliente llega, en media, cada diez minutos. El tiempo de servicio de cada cliente es de cinco minutos. La observación más interesante sobre los resultados anteriores es la variación de los resultados. Dada la frecuencia de llegada de clientes (uno cada diez minutos) y la longitud de la simulación (200 minutos), el número esperado de clientes servidos debe ser 20, con una cierta varianza alrededor de la media, y en efecto así ocurre. Sin embargo, la gran varianza del tiempo medio de espera en la cola es sorprendente.

El algoritmo anterior puede modificarse de forma que se ejecute la simulación de 200 minutos un total de 200 veces, para el mismo valor de la velocidad de llegadas (1 cliente cada 10 minutos) y para un valor dado del tiempo de servicio. Los resultados que pueden obtenerse de este nuevo algoritmo son, por ejemplo, el máximo tiempo de espera de un cliente (en las 200 simulaciones), la espera media en las 200 simulaciones, la espera media mínima y la espera media máxima de entre las 200 simulaciones.

En la tabla siguiente se muestran los resultados obtenidos ejecutando el algoritmo modificado para distintos valores del tiempo de servicio. Por ejemplo, la primera línea refleja que para un tiempo de servicio de 3 minutos, la espera máxima de un cliente en las 200 simulaciones fue de 6 minutos. El tiempo medio de espera fue 0.40 minutos; sin embargo, en (al menos) una simulación de las 200 el tiempo medio de espera fue 0.0 mientras que en otra fue 1.29 minutos.

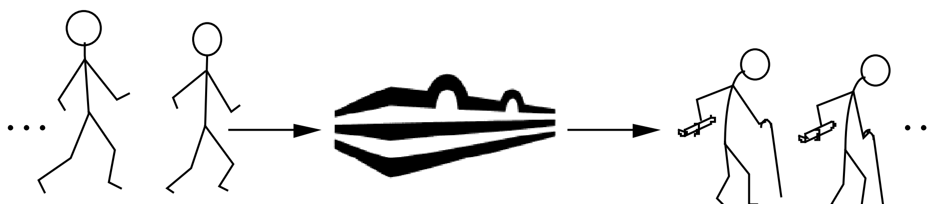
Tiempo de servicio	Máxima espera[4]	Espera media	Mínima espera media	Máxima espera media
3	6	0.40	0.00	1.29
4	11	0.90	0.00	3.35
5	23	1.69	0.00	5.96
6	38	2.81	0.27	16.82
7	48	5.55	0.20	24.00
8	70	7.80	0.28	34.48

Si el tiempo de servicio se aproxima al tiempo medio entre dos llegadas consecutivas de clientes, el sistema se aproxima a un estado de **saturación**. Si la velocidad de llegadas fuese mayor que la de servicio, obviamente el número de clientes en la cola aumentaría *ad infinitum*. Lo más sorprendente vuelve a ser que para un sistema lejos del estado de saturación, la variación de los resultados puede ser grande.

Por ejemplo, para un tiempo de servicio de 5 minutos y los clientes llegando en media cada 10 minutos, al menos un cliente tuvo que esperar en cola durante 23 minutos. En una de las 200 simulaciones, para esos mismos valores de entrada, el tiempo medio de espera en cola fue de 5.96 minutos (superior incluso que el tiempo de servicio), mientras que en otra de las simulaciones el tiempo medio de espera en cola fue de 0.0 minutos (es decir, ninguno de los clientes servidos en los 200 minutos de esa simulación tuvo que hacer cola).

La simulación propuesta en esta lección es la más sencilla que puede plantearse y, de hecho, existen **fórmulas analíticas** que permiten obtener los resultados buscados sin realizar ninguna simulación (por ejemplo, si el tiempo entre dos llegadas consecutivas es una variable aleatoria exponencial de parámetro λ y el tiempo de servicio es otra variable aleatoria exponencial de parámetro μ, entonces el tiempo medio de espera de un cliente en la cola es $W_q=(1/\mu^2)/(1-\lambda/\mu)$. El estudio de modelos de colas como el propuesto aquí y de otros más realistas y complejos, y su análisis, es el objeto de la denominada **Teoría de Colas**.

4



Lección 10

El TAD diccionario. Implementación con listas enlazadas ordenadas

Indice

1. Concepto de diccionario y su especificación
2. Implementaciones estáticas sencillas
3. Implementación con listas enlazadas ordenadas

1. Concepto de diccionario y su especificación

Un **diccionario o tabla** es un conjunto o **colección de pares** (*c*,*v*). El primer miembro de cada par suele llamarse **clave** y el segundo, **valor asociado a la clave**. En la colección se supondrá que **no puede haber dos pares que tengan la misma clave**.

Los diccionarios también se llaman tipos de datos **asociativos** o **funcionales** porque representan una función cuyo dominio es el género de las claves y cuyo rango es el género de los valores:

f : Dominio de las claves → Dominio de los valores

La función representada suele ser parcial (es decir, existen claves del dominio sin valor asociado), aunque podrían representarse como totales asociando a algunas claves un valor especial llamado "indefinido".

Ejemplos típicos de utilización de diccionarios son:

- Las *tablas de símbolos de los compiladores* (las claves son los identificadores y los valores los atributos de dichos identificadores).
- Los *directorios de ficheros* en un sistema operativo (las claves son los nombres de los ficheros y los valores sus atributos).
- Los *conjuntos* de elementos: los elementos son las claves y no existen valores asociados. En este caso la función representada por el diccionario es la función característica del conjunto (para cada clave, ¿está o no está en el conjunto?):
  f : Dominio de las claves → {verdad,falso}
  Las técnicas para representar diccionarios y conjuntos tienen mucho en común, aunque el conjunto de operaciones de interés es diferente (las operaciones de unión, intersección y diferencia de conjuntos no tienen sentido, por lo general, para diccionarios).
- Los *tipos vector*, `vector[tpIndice] de tpDato`, son un caso particular de diccionarios en el que las claves son un tipo de datos escalar numerable (el tipo de los índices) y los valores asociados a las claves son los de las correspondientes componentes del vector.
  f : Dominio del tipo `tpIndice` → Dominio del tipo `tpDato`

Las **operaciones** básicas para los diccionarios son: la **inserción** de un nuevo par (*c*,*v*) y la obtención del **valor** *v* asociado a una clave *c*. Si se intenta añadir un par (*c*,*v*) y ya existe en el diccionario un par (*c*,*v'*), se entiende que se está realizando una modificación (actualización) del valor asociado a esa clave *c*.

La operación de obtener el valor asociado a una clave es parcial, pues sólo tiene sentido si existe un par en el diccionario can tal clave.

En muchas ocasiones se precisa una operación de **borrado** que dada una clave *c* elimine el par (*c*,*v*), supuesto que este par existiese en la tabla.

Se presenta a continuación una posible **especificación** del TAD diccionario. Se va a añadir el requisito de que las **claves** sean **comparables** con una operación "<" (es decir, que exista en su dominio una relación de orden total).

espec diccionarios
usa boleanos, naturales
parámetros formales
 géneros clave, valor
 operaciones *{suponemos que en el género de las claves hay definida una función de orden y otra de igualdad}*
  _<_ : clave c1, clave c2 -> booleano
  _=_ : clave c1, clave c2 -> booleano
fpf
género diccionario *{Los valores del TAD representan conjuntos de pares (clave,valor) sin claves repetidas}*
operaciones
 crear: -> diccionario
 {Devuelve un diccionario vacío, sin elementos}

 añadir: diccionario d, clave c, valor v -> diccionario
 {Si en d no hay ningún par con clave c, devuelve el diccionario resultante de añadir el par (c,v) a d;
 si en d hay un par (c,v'), entonces devuelve el resultado de sustituirlo por el par (c,v)}

 está?: clave c, diccionario d -> booleano
 {Devuelve verdad si y sólo si en d hay algún par (c,v)}

 parcial obtenerValor: clave c, diccionario d -> valor
 {Devuelve el valor asociado a la clave c en d.
 Parcial: la operación no está definida si c no está en d}

 quitar: clave c, diccionario d -> diccionario
 {Si c está en d, devuelve el diccionario resultante de borrar c y su valor de d;
 si c no está en d, devuelve un diccionario igual a d}

 cardinal: diccionario d -> natural
 {Devuelve el nº de elementos en el diccionario d}

 esVacío?: diccionario d -> booleano
 {Devuelve verdad si y sólo si d no tiene elementos}

 {Las cinco operaciones siguientes son un Iterador definido para los diccionarios}
 iniciarIterador: diccionario d -> diccionario
 {Inicializa el iterador para recorrer los pares del diccionario d, de forma que el siguiente par a visitar sea el primero que visitamos (situación de no haber visitado ningún par).}

 existeSiguiente?: diccionario d -> booleano
 {Devuelve falso si ya se han visitado todos los pares de d, devuelve verdad en caso contrario}

 parcial siguienteClave: diccionario d -> clave
 {Devuelve la clave del siguiente par a visitar de d.
 Parcial: la operación no está definida si no existeSiguiente?(d)}

 parcial siguienteValor: diccionario d -> valor
 {Devuelve el valor del siguiente par a visitar de d.
 Parcial: la operación no está definida si no existeSiguiente?(d)}

 parcial avanza: diccionario d -> diccionario
 {Prepara el iterador para visitar el siguiente par del diccionario d.
 Parcial: la operación no está definida si ya se ha visitado el último par.}
fespec

Por razones de eficiencia, en las implementaciones del TAD diccionario las operaciones "está?" y "obtenerValor" se combinan en una sola operación que devuelve tanto un booleano como un valor. El booleano es verdad si y sólo si la clave está y en ese caso el valor es el asociado a la clave, y en caso contrario el booleano es falso y el valor queda indefinido.

2. Implementaciones estáticas sencillas

Se han propuesto varias representaciones estáticas (basadas en un vector de registros) sencillas para datos de tipo diccionario. Las más representativas son:

- **Representación desordenada:** consiste en almacenar los pares de datos del diccionario en las primeras componentes contiguas de un vector: `vector[1..max] de par(c,v)`, sin ordenar con ningún criterio (ese `par(c,v)` representa una tupla o registro con la clave y el valor).
  Esta representación no requiere la existencia de una relación de orden total definida sobre el dominio de las claves. Con esta representación, el coste en tiempo en el caso peor de las operaciones de búsqueda, inserción, y borrado esté en $O(n)$, siendo n el número de pares almacenados.
  El coste en espacio es $O(\texttt{max})$, independientemente del cardinal del diccionario almacenado.
- **Representación ordenada:** consiste en almacenar el diccionario en las primeras componentes contiguas de un vector: `vector[1..max] de par(c,v)`, ordenando los pares (c,v) por valores crecientes de la clave.
  Esta representación sí que requiere la existencia de una relación de orden total ("_<_") definida sobre el dominio de las claves.
  El coste en espacio es, como antes, $O(\texttt{max})$. El coste en tiempo en el caso peor de las operaciones de inserción y borrado vuelve a estar en $O(n)$, siendo n el cardinal del diccionario. Sin embargo, el coste en tiempo en el caso peor de la búsqueda de una clave (y su valor asociado) puede reducirse a $O(\log n)$, sin más que implementar una búsqueda dicotómica de la clave. Esta representación es útil, por tanto, para tablas "poco dinámicas" (en las que hay pocas inserciones y borrados y muchas consultas) pero es muy ineficiente para tablas muy dinámicas.

En general, la utilización de un vector conlleva que el número de pares del diccionario está acotado en todo momento por el valor `max` (tamaño del vector), y ese valor queda fijado en tiempo de compilación.

Se plantea, como ejercicio sencillo de programación de primer curso, el realizar ambas implementaciones del TAD diccionario.

3. Implementación con listas enlazadas ordenadas

Para evitar la **limitación en tiempo de compilación del cardinal máximo** (`max`) del diccionario que imponen las implementaciones estáticas, basadas en vectores, y para evitar que el **coste espacial** para almacenar un diccionario sea $O(\texttt{max})$, independientemente del cardinal del diccionario, vamos a detallar una implementación dinámica (i.e., basada en punteros) consistente en almacenar cada par del diccionario en un registro en memoria dinámica y enlazar esos registros en una lista con punteros, de forma similar a como hicimos con pilas y colas.

Caben dos opciones para crear esa lista encadenada con punteros: mantener los pares del diccionario en algún orden específico o dejarlos en cualquier orden. Desde el punto de vista del tiempo de ejecución de las operaciones en el caso peor, esa decisión no afecta al orden de magnitud del coste. Sin embargo, desde el punto de vista más práctico, puede resultar útil mantener los pares de la lista ordenados por valores crecientes de la clave. Evidentemente, para que esto sea posible, debe exigirse que el TAD de las claves disponga de una relación de orden total, es decir, de una operación "_<_" de comparación de claves como la que se ha exigido al parámetro de tipo "clave" en la especificación del principio de esta lección.

La que sigue es una implementación del TAD diccionario utilizando una **lista enlazada con punteros, ordenada por valores crecientes de las claves**.

```
módulo genérico diccionarios
parámetros
  tipos clave,valor
  con función "<"(c1,c2:clave) devuelve booleano
  con función "="(c1,c2:clave) devuelve booleano
  {suponemos que el tipo clave tiene definidas una función de orden y otra de
   igualdad}

exporta
  tipo diccionario  {Los valores del TAD diccionario representan conjuntos de pares
                     (clave,valor) en los que no se permiten claves repetidas}

  procedimiento crear(sal d:diccionario)
  {Devuelve en d un diccionario vacío, sin elementos}

  procedimiento añadir(e/s d:diccionario; ent c:clave; ent v:valor)
  {Si en d no hay ningún par con clave c, añade a d el par (c,v);
   si en d hay un par (c,v'), entonces lo sustituye por el par (c,v)}

  procedimiento buscar(ent d:diccionario; ent c:clave;
                       sal éxito:booleano; sal v:valor)
  {Devuelve éxito=verdad si en d hay algún par con clave c, falso en caso contrario.
   En caso de éxito, además, devuelve en v el valor asociado a la clave c en d}

  procedimiento quitar(ent c:clave; e/s d:diccionario)
  {Si en d hay un par con clave c, lo borra. En caso contrario, d no se modifica}

  función cardinal(d:diccionario) devuelve natural
  {Devuelve el nº de pares del diccionario d}

  función esVacío(d:diccionario) devuelve booleano
  {Devuelve verdad si y sólo si d no tiene pares}

  procedimiento duplicar(sal dSal:diccionario; ent dEnt diccionario)
  {Duplica la representación del diccionario dEnt en el diccionario dSal}

  función iguales(d1,d2:diccionario) devuelve booleano
  {Devuelve verdad si y sólo si los diccionarios d1 y d2 tienen los mismos pares de
   claves y valores}

  procedimiento liberar(e/s d:diccionario)
  {Devuelve en d el diccionario vacío y además libera la memoria utilizada
   previamente por d}

  procedimiento iniciarIterador(e/s d:diccionario)
  {Inicializa el iterador para recorrer los pares del diccionario d, de forma que el
   siguiente par a visitar sea el primero que visitamos (situación de no haber
   visitado ningún par).}

  función existeSiguiente(d:diccionario) devuelve booleano
  {Devuelve falso si ya se han visitado todos los pares de d; verdad en otro caso}

  procedimiento siguiente(e/s d:diccionario;
                          sal c:clave; sal v:valor; sal error:booleano)
  {Si existe algún par de d pendiente de visitar, devuelve en c y v la clave
   y el valor, respectivamente, del siguiente par a visitar y error=falso, y
   además avanza el iterador para que a continuación se pueda visitar otro par
   de d. Si no quedan pares pendientes de visitar devuelve error=verdad, c y v
   quedan indefinidos y d queda como estaba}
```

```
implementación
  tipos ptCelda = ↑unaCelda;
        unaCelda = registro
                     laClave:clave;
                     elValor:valor;
                     sig:ptCelda
                   freg
        diccionario = registro
                        primerPar:ptCelda; {lista enlazada, ordenada por clave ("<")}
                        tamaño:natural;
                        iter:ptCelda {para implementar el iterador}
                      freg

  procedimiento crear(sal d:diccionario)
  {Devuelve en d un diccionario vacío, sin elementos}
  principio
    d.primerPar:=nil;
    d.tamaño:=0
  fin

  procedimiento añadir(e/s d:diccionario; ent c:clave; ent v:valor)
  {Si en d no hay ningún par con clave c, añade a d el par (c,v);
   si en d hay un par (c,v'), entonces lo sustituye por el par (c,v)}
  variables pAux,nuevo:ptCelda
  principio
    si d.primerPar=nil entonces {lista vacía}
      nuevoDato(d.primerPar);
      d.primerPar↑.laClave:=c;
      d.primerPar↑.elValor:=v;
      d.primerPar↑.sig:=nil;
      d.tamaño:=1
    sino
      si c<d.primerPar↑.laClave entonces {inserción al principio}
        pAux:=d.primerPar;
        nuevoDato(d.primerPar);
        d.primerPar↑.laClave:=c;
        d.primerPar↑.elValor:=v;
        d.primerPar↑.sig:=pAux;
        d.tamaño:=d.tamaño+1
      sino
        si c=d.primerPar↑.laClave entonces {ya existe, cambiar valor}
          d.primerPar↑.elValor:=v
        sino {c>d.primerPar↑.laClave => buscar punto de inserción}
          pAux:= d.primerPar;
          mq pAux↑.sig≠nil andthen pAux↑.sig↑.laClave<c hacer {andthen: evaluación
          perezosa (o cortocircuitada) del producto lógico, como el operador && de C++}
            pAux:=pAux↑.sig
          fmq;
          si pAux↑.sig≠nil andthen c=pAux↑.sig↑.laClave entonces {clave ya existe}
            pAux↑.sig↑.elValor:=v
          sino {inserción entre dos registros o al final}
            nuevoDato(nuevo);
            nuevo↑.laClave:=c;
            nuevo↑.elValor:=v;
            nuevo↑.sig:=pAux↑.sig;
            pAux↑.sig:=nuevo;
            d.tamaño:=d.tamaño+1
          fsi
        fsi
      fsi
    fsi
  fin
```

```
procedimiento buscar(ent d:diccionario; ent c:clave;
                     sal éxito:booleano; sal v:valor)
{Devuelve éxito=verdad si en d hay algún par con clave c, falso en caso contrario.
 En caso de éxito, además, devuelve en v el valor asociado a la clave c en d}
variable pAux:ptCelda
principio
  pAux:=d.primerPar;
  mientrasQue pAux≠nil andthen pAux↑.laClave<c hacer
    pAux:=pAux↑.sig
  fmq;
  si pAux=nil entonces
    éxito:=falso
  sino
    si pAux↑.laClave=c entonces
      v:=pAux↑.elValor;
      éxito:=verdad
    sino
      éxito:=falso
    fsi
  fsi
fin

procedimiento quitar(ent c:clave; e/s d:diccionario)
{Si en d hay un par con clave c, lo borra. En caso contrario, d no se modifica}
variables pAux1,pAux2:ptCelda;
          parar:booleano
principio
  si d.primerPar≠nil entonces {caso contrario, no hacer nada}
    si d.primerPar↑.laClave<=c entonces {caso contrario, no hacer nada}
      si d.primerPar↑.laClave=c entonces {borrar el primer elemento}
        pAux1:=d.primerPar;
        d.primerPar:=d.primerPar↑.sig;
        disponer(pAux1);
        d.tamaño:=d.tamaño-1
      sino {d.primera↑.laClave<c => buscar la clave c a partir del 2º elemento}
        parar:=falso;
        pAux1:=d.primerPar↑.sig;
        pAux2:=d.primerPar;
        mientrasQue pAux1≠nil and not parar hacer
          si c<pAux1↑.laClave entonces {la clave no está, no hacer nada}
            parar:=verdad
          sino_si c=pAux1↑.laClave entonces {borrar el registro}
            pAux2↑.sig:=pAux1↑.sig;
            disponer(pAux1);
            parar:=verdad;
            d.tamaño:=d.tamaño-1
          sino {pAux1↑.laClave<c => avanzar}
            pAux2:=pAux1;
            pAux1:=pAux1↑.sig
          fsi
        fmq
      fsi
    fsi
  fsi
fin

función cardinal(d:diccionario) devuelve natural
{Devuelve el nº de pares del diccionario d}
principio
  devuelve d.tamaño
fin
```

```
función esVacío(d:diccionario) devuelve booleano
{Devuelve verdad si y sólo si d no tiene pares}
principio
  devuelve d.tamaño=0
fin

procedimiento duplicar(sal dSal:diccionario; ent dEnt:diccionario)
{Duplica la representación del diccionario dEnt en el diccionario dSal}
variables pAuxEnt,pAuxSal:ptCelda
principio
  si esVacío?(dEnt) entonces
    crear(dSal)
  sino {dEnt no vacío => tiene una primera celda}
    nuevoDato(dSal.primerPar);
    dSal.primerPar↑.laClave:=dEnt.primerPar↑.laClave;
    dSal.primerPar↑.elValor:=dEnt.primerPar↑.elValor;
    pAuxEnt:=dEnt.primerPar↑.sig;
    pAuxSal:=dSal.primerPar;
    mientrasQue pAuxEnt≠nil hacer
      nuevoDato(pAuxSal↑.sig);
      pAuxSal:=pAuxSal↑.sig;
      pAuxSal↑.laClave:=pAuxEnt↑.laClave;
      pAuxSal↑.elValor:=pAuxEnt↑.elValor;
      pAuxEnt:=pAuxEnt↑.sig
    fmq;
    pAuxSal↑.sig:=nil;
    dSal.tamaño:=dEnt.tamaño
  fsi
fin

función iguales(d1,d2:diccionario) devuelve booleano
{Devuelve verdad si y sólo si los diccionarios d1 y d2 tienen los mismos pares de
  claves y valores}
variables pAux1,pAux2:ptCelda;
          igual:booleano
principio
  si esVacío(d1) and esVacío(d2) entonces
    devuelve verdad
  sino_si cardinal(d1)≠cardinal(d2) entonces
    devuelve falso
  sino {ambos tienen el mismo número (no nulo) de pares}
    igual:=verdad;
    pAux1:=d1.primerPar;
    pAux2:=d2.primerPar;
    mientrasQue igual and pAux1≠nil hacer
      igual:=(pAux1↑.laClave=pAux2↑.laClave) and (pAux1↑.elValor=pAux2↑.elValor);
      pAux1:=pAux1↑.sig;
      pAux2:=pAux2↑.sig
    fmq;
    devuelve igual
  fsi
fin

procedimiento liberar(e/s d:diccionario)
{Devuelve en d el diccionario vacío y además libera la memoria utilizada
 previamente por d}
variable pAux:ptCelda
principio
  pAux:=d.primerPar;
  mientrasQue pAux≠nil hacer
    d.primerPar:=d.primerPar↑.sig;
```

```
      disponer(pAux);
      pAux:=d.primerPar
    fmq;
    crear(d)
  fin

  procedimiento iniciarIterador(e/s d:diccionario)
  {Inicializa el iterador para recorrer los pares del diccionario d, de forma que el
   siguiente par a visitar sea el primero que visitamos (situación de no haber
   visitado ningún par).}
  principio
    d.iter:=d.primerPar
  fin

  función existeSiguiente(d:diccionario) devuelve booleano
  {Devuelve falso si ya se han visitado todos los pares de d; verdad en otro caso}
  principio
    devuelve d.iter≠nil
  fin

  procedimiento siguiente(e/s d:diccionario;
                          sal c:clave; sal v:valor; sal error:booleano)
 {Si existe algún par de d pendiente de visitar, devuelve en c y v la clave y
  el valor, respectivamente, del siguiente par a visitar y error=falso, y
  además avanza el iterador para que a continuación se pueda visitar otro par
  de d. Si no quedan pares pendientes de visitar devuelve error=verdad, c y v
  quedan indefinidos y d queda como estaba}
  principio
    si existeSiguiente(d) entonces
      error:=falso;
      c:=d.iter↑.laClave; v:=d.iter↑.elValor;
      d.iter:=d.iter↑.sig
    sino
      error:=verdad
    fsi
  fin

fin {del módulo diccionarios}
```

A continuación, para finalizar, se presenta un código alternativo del procedimiento de añadir que puede sugerir nuevas ideas para insertar en una lista enlazada. Se recomienda una lectura reposada, comparándolo con la implementación escrita más arriba.

```
procedimiento añadir(e/s d:diccionario; ent c:clave; ent v:valor)
{Si en d no hay ningún par con clave c, añade a d el par (c,v);
 si en d hay un par (c,v'), entonces lo sustituye por el par (c,v)}
variables pAux,nuevo:ptCelda; celdaAux:unaCelda;
principio
  si d.primerPar=nil entonces {lista vacía}
    nuevoDato(d.primerPar);
    d.primerPar↑.laClave:=c;
    d.primerPar↑.elValor:=v;
    d.primerPar↑.sig:=nil;
    d.tamaño:=1;
  sino
    si c<d.primerPar↑.laClave entonces {inserción al principio}
      pAux:= d.primerPar;
      nuevoDato(d.primerPar);
      d.primerPar↑.laClave:=c;
      d.primerPar↑.elValor:=v;
      d.primerPar↑.sig:=pAux;
      d.tamaño:=d.tamaño+1
```

```
      sino {buscar punto de inserción}
        pAux:= d.primerPar;
        mientrasQue pAux↑.laClave<c and pAux↑.sig≠nil hacer
          pAux:=pAux↑.sig
        fmq;
        si c<pAux↑.laClave entonces {inserción entre dos registros}
          celdaAux.laClave:=c;
          celdaAux.elValor:=v;
          nuevoDato(nuevo);
          nuevo↑:=pAux↑;
          pAux↑:=celdaAux;
          pAux↑.sig:=nuevo;
          d.tamaño:=d.tamaño+1
        sino
          si c=pAux↑.laClave entonces {clave ya existe, cambiar valor}
            pAux↑.elValor:=v
          sino {inserción al final}
            nuevoDato(pAux↑.sig);
            pAux:=pAux↑.sig;
            pAux↑.laClave:=c;
            pAux↑.elValor:=v;
            pAux↑.sig:=nil;
            d.tamaño:=d.tamaño+1
          fsi
        fsi
      fsi
    fsi
  fin
```

En cuanto al coste asintótico en tiempo para el caso peor, puede deducirse con facilidad que las operaciones “crear”, “cardinal”, “esVacío”, “iniciarIterador”, “existeSiguiente” y “siguiente” tienen un coste en $O(1)$, es decir, constante o independiente del cardinal del diccionario, mientras que todas las demás operaciones tienen un coste $O(n)$ en el caso peor, siendo n el cardinal del diccionario.

TEMA III

Tipos de datos arborescentes

Lección 11

Introducción a los árboles

Indice

1. Conceptos, definiciones y terminología básica
2. Los árboles como estructura de datos para representar TAD contenedores

1. Conceptos, definiciones y terminología básica

Un **árbol** con raíz es una colección de elementos de un mismo tipo, llamados nodos o vértices, que pueden representarse en un **grafo no orientado, conexo y acíclico** en el que existe un **vértice destacado llamado raíz**. Por tanto, en general un árbol no define una estructura lineal sino jerárquica (define una relación paterno-filial entre los nodos). Damos a continuación una definición recursiva.

Un **árbol *n*–ario** (con $n \geq 1$) es un conjunto no vacío de elementos o **nodos** del **mismo tipo** tal que:

- existe un elemento destacado llamado **raíz** del árbol,
- el resto de los elementos se distribuyen en m (con $0 \leq m \leq n$) subconjuntos disjuntos, llamados **subárboles** del árbol original, cada uno de los cuales es a su vez un árbol n–ario.

Si en el conjunto de los subárboles de un árbol n–ario se supone definida una relación de orden total, el árbol se llama **ordenado**.

Gráficamente, un árbol ordenado con raíz x y subárboles $A_1, \ldots, A_m$ puede representarse como indica la figura 1. En la figura 2 se muestra un ejemplo de árbol 3–ario de números enteros.

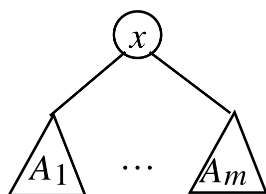


Figura 1

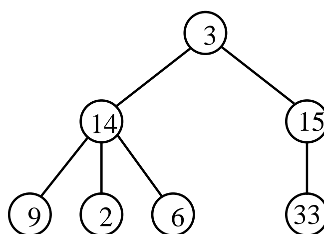
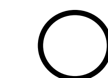


Figura 2

Leyenda:

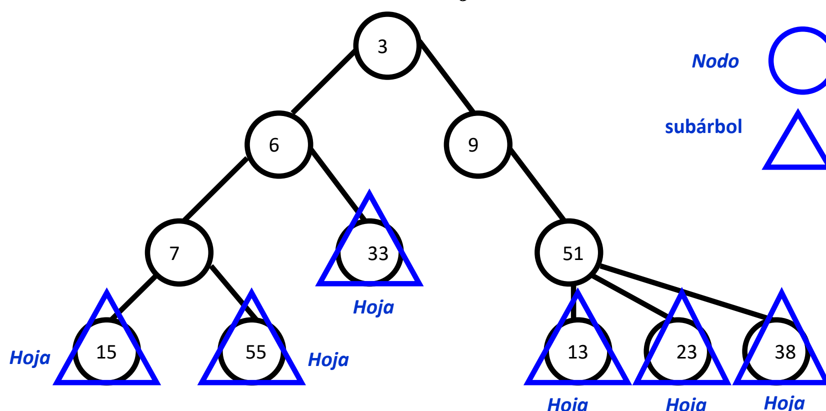
nodo 

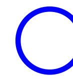
subárbol

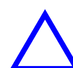
Otro tipo de árboles diferentes son los **árboles binarios**. Un árbol binario es un conjunto de elementos o nodos del mismo tipo tal que:

- o bien es el conjunto vacío, y entonces se llama **árbol vacío**,
- o bien es no vacío, en cuyo caso existe un elemento destacado llamado **raíz** y el resto de elementos se distribuyen en dos subconjuntos disjuntos, llamados **subárbol izquierdo** y **subárbol derecho**, cada uno de los cuales es un árbol binario.

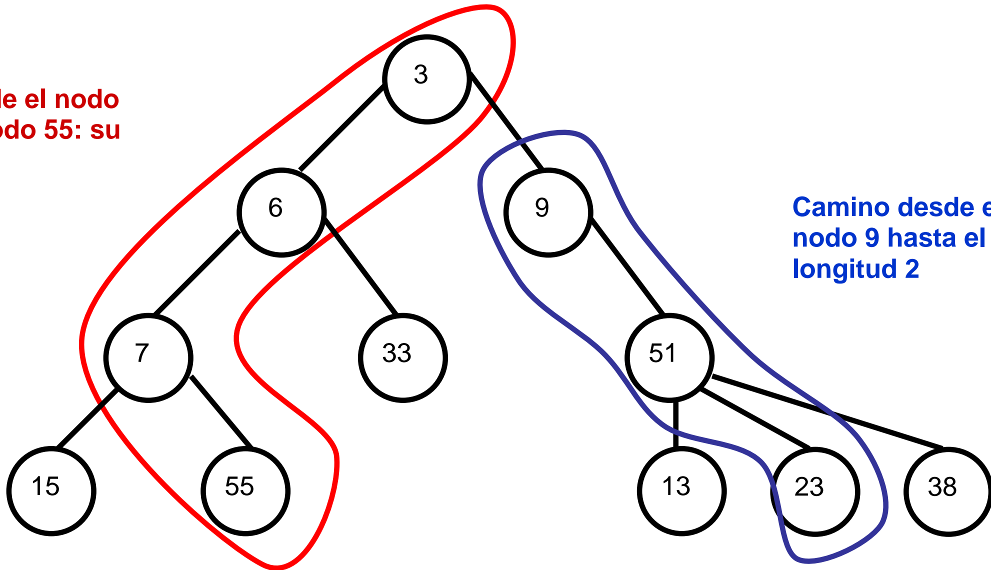
Un árbol compuesto de un solo elemento se denomina **hoja**.



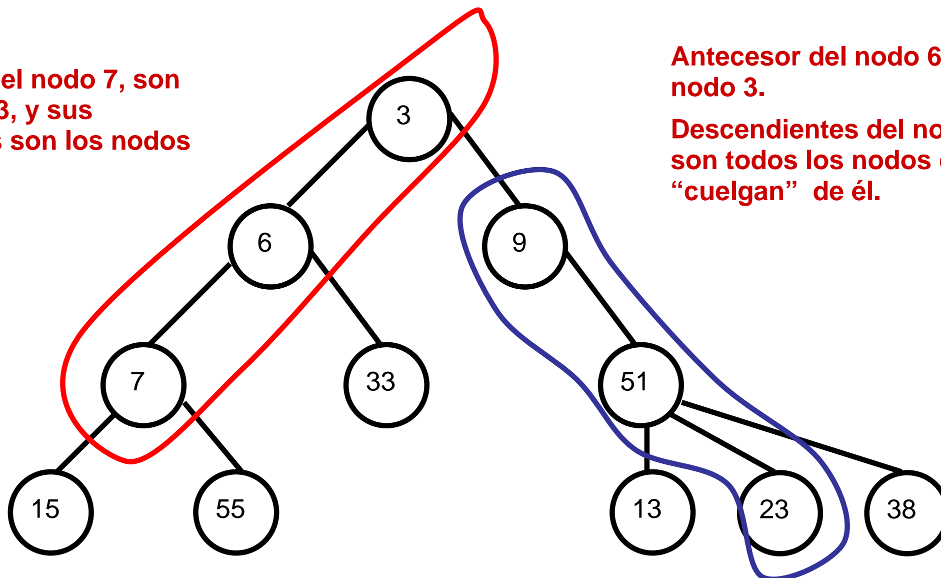
Nodo 

subárbol 

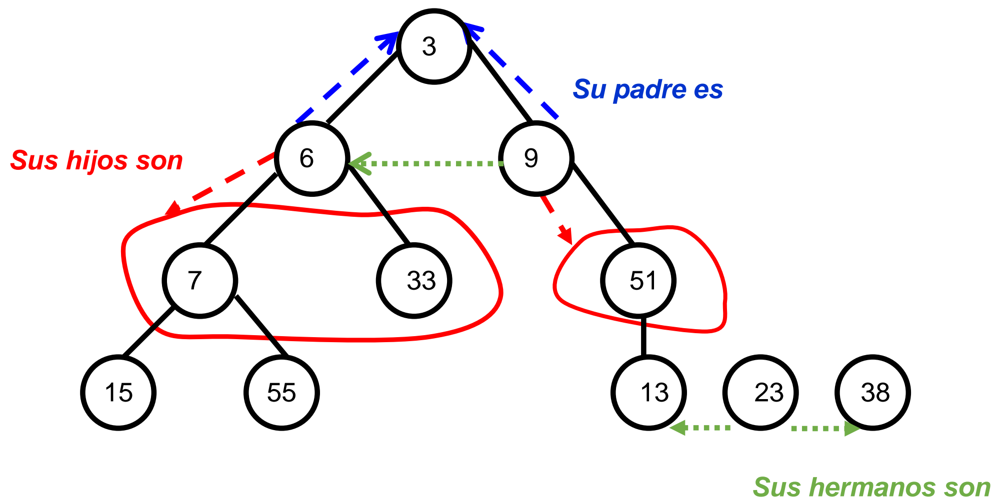
Un **camino** es una secuencia de nodos $n_1, \ldots, n_s$, $s \geq 1$, tal que n_{i+1} es hijo de n_i, para todo $1 \leq i \leq s$-1. El número de nodos de la secuencia menos uno se llama **longitud** del camino (es decir, es el número de *aristas* que conectan los nodos de ese camino). Por convenio, diremos que existe un camino de longitud cero de todo nodo a sí mismo.



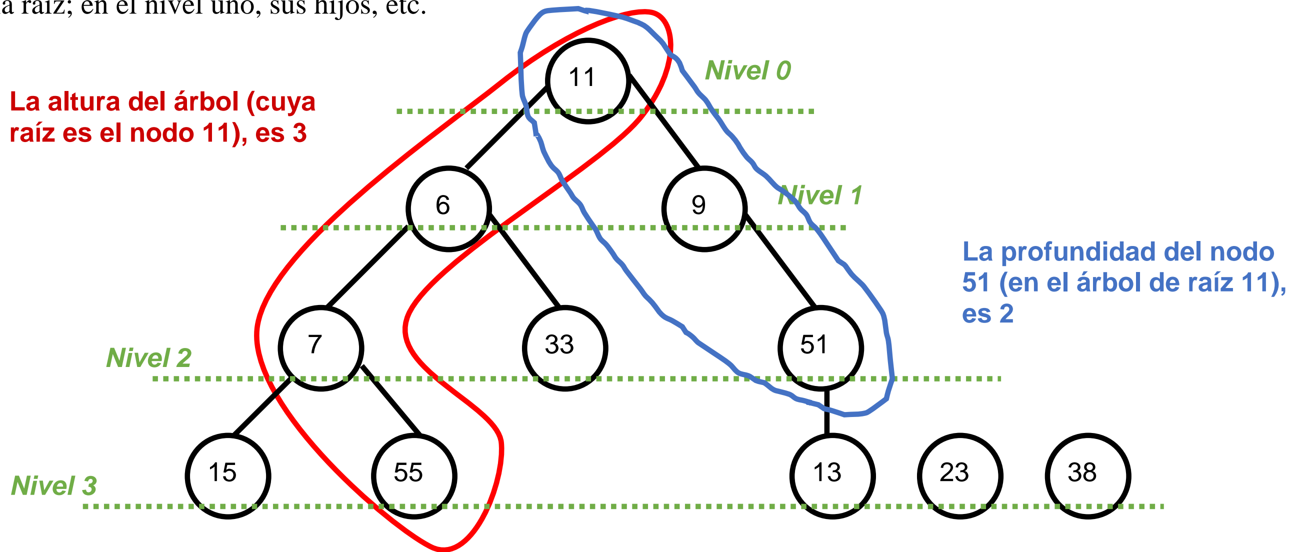
Si en un árbol *A* existe un camino desde el nodo n_1 hasta el nodo n_2, se dice que n_1 es **antecesor** de n_2 y que n_2 es **descendiente** de n_1. Los antecesores o descendientes de un nodo distintos del mismo nodo se denominan **propios**.



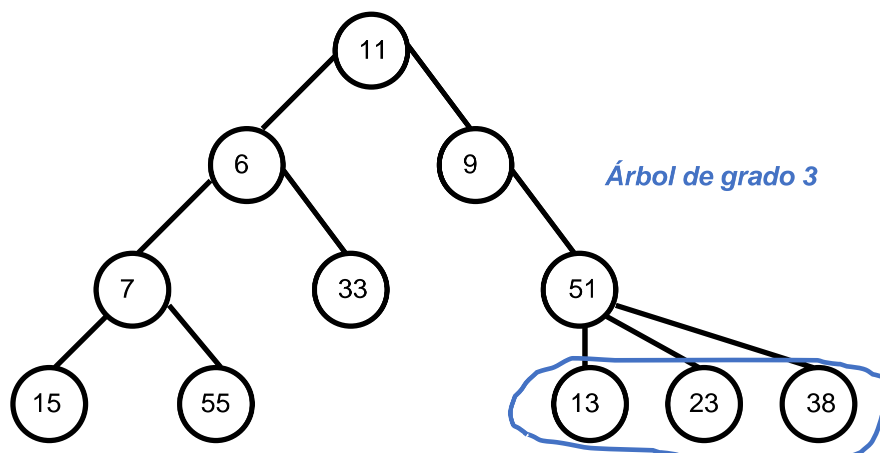
El **padre** de un nodo es su primer antecesor propio, si existe. Los **hijos** de un nodo son sus primeros descendientes propios, si existen. Dos nodos son **hermanos** si tienen el mismo padre.



La **altura** de un árbol es la longitud del camino más largo que puede encontrarse en el árbol desde la raíz a un nodo. No está definida para árboles vacíos. La **profundidad** de un nodo en un árbol es la longitud del único camino existente desde la raíz del árbol hasta ese nodo. Un **nivel** es un conjunto de nodos de un árbol con igual profundidad. En el nivel cero sólo está la raíz; en el nivel uno, sus hijos, etc.



El **grado** de un árbol es el número máximo de hijos que pueden tener sus nodos (si el árbol es *n*–ario su grado es *n*; si es binario, su grado es dos).



2. Los árboles como estructura de datos para representar TAD contenedores

En programación, utilizaremos estructuras arborescentes (árboles) para almacenar colecciones de datos de un mismo tipo (parámetro formal *elemento*). Podrán darse dos casos:

a) **Colecciones de elementos entre los que existe alguna relación jerárquica** que queremos quede patente en la estructura. Por ejemplo, árboles genealógicos de personas (relación paterno-filial), organigramas de empresas (relación de jefe y subalternos), árboles de ficheros y directorios en un sistema operativo (relación de directorio y subdirectorio), expresiones aritméticas (operando y operadores a los que debe aplicarse), clasificaciones (biológicas, geológicas, bibliográficas, etcétera).

b) **Colecciones de elementos entre los que NO existe una relación jerárquica** (o no interesa que quede patente). Por ejemplo, conjuntos o multiconjuntos de elementos, diccionarios o tablas, u otros, que decidamos representar en una estructura de árbol con objeto de **mejorar la eficiencia en tiempo de las operaciones de manipulación** con respecto a la eficiencia conseguida con una representación lineal (estática o dinámica).

Lección 12

Árboles binarios

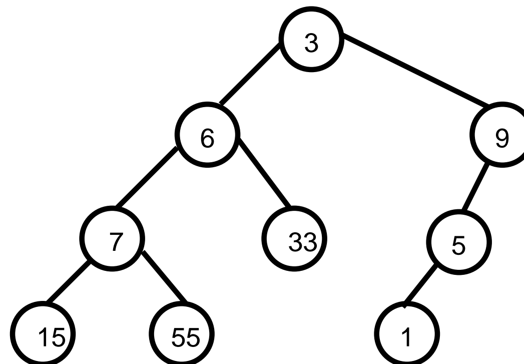
Indice

1. Concepto y especificación
2. Implementación estática
3. Implementación dinámica
4. Recorridos

1. Concepto y especificación

Como ya se adelantó en la lección previa, un árbol binario es un conjunto de elementos o nodos del mismo tipo tal que:

- o bien es el conjunto vacío, y entonces se llama **árbol vacío**,
- o bien es no vacío, en cuyo caso existe un elemento destacado llamado **raíz** y el resto de elementos se distribuyen en dos subconjuntos disjuntos, llamados **subárbol izquierdo** y **subárbol derecho**, cada uno de los cuales es un árbol binario.



En la siguiente especificación sencilla −**genérica**− de un primer TAD árbol binario, vamos a incorporar las operaciones mínimas para poder **generar** todos los árboles binarios y un conjunto muy pequeño de operaciones para poder extraer/observar algo de información de un árbol binario.

espec árbolesBinarios
usa booleanos, naturales
parámetro formal
  género elemento
fpf
género arbin *{Su dominio de valores son los árboles binarios de elementos}*
operaciones
    vacío: -> arbin
    {Devuelve el árbol vacío}

    plantar: elemento e, arbin ai, arbin ad -> arbin
    {Devuelve un árbol binario cuyo elemento raíz es e, subárbol izquierdo es ai y subárbol derecho es ad}

    esVacío?: arbin a -> booleano
    {Devuelve verdad si y sólo si a es el árbol vacío}

    parcial raíz: arbin a -> elemento

{Devuelve el elemento raíz de a.
Parcial: la operación no está definida si a es vacío}

parcial subIzq: arbin a -> arbin
{Devuelve el subárbol izquierdo de a.
Parcial: la operación no está definida si a es vacío}

parcial subDer: arbin a → arbin
{Devuelve el subárbol derecho de a.
Parcial: la operación no está definida si a es vacío}

parcial altura: arbin a → natural
{Devuelve la altura de a.
Parcial: la operación no está definida si a es vacío}

{Las cuatro siguientes operaciones son un Iterador definido sobre los árboles binarios}
iniciarIterador: arbin a -> arbin
{Prepara el iterador para que el siguiente elemento a visitar sea un primer elemento del árbol a, si existe (situación de no haber visitado ningún elemento)}

existeSiguiente?: arbin a -> booleano
{Devuelve falso si ya se han visitado todos los elementos de a, devuelve verdad en caso contrario}

parcial siguiente: arbin a -> elemento
{Devuelve el siguiente elemento de a.
Parcial: la operación no está definida si no existeSiguiente?(a)}

parcial avanza: arbin a -> pila
{Devuelve el árbol binario resultante de avanzar el iterador a otro elemento no visitado todavía de a.
Parcial: la operación no está definida si no existeSiguiente?(a)}

fespec

2. Implementación estática

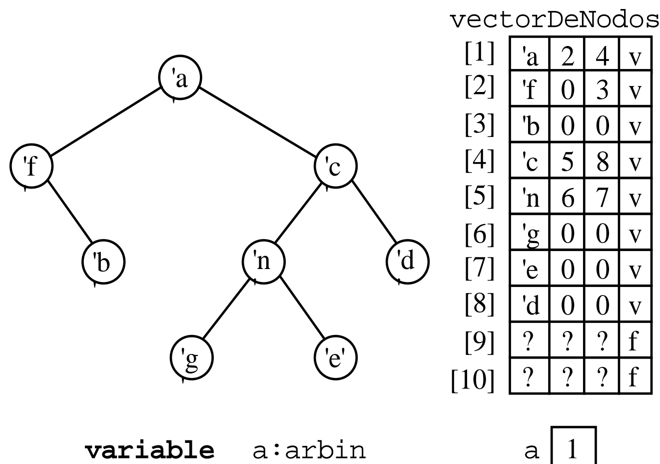
En lenguajes que incluyen tipos puntero la representación generalmente utilizada para los valores de tipo árbol es dinámica, encadenando mediante punteros los elementos del árbol. Pero en aquellos lenguajes que carecen de tipos puntero, es preciso representar árboles en base a vectores. Veremos aquí la representación estática de árboles binarios que consiste en el almacenamiento, para cada elemento, de los índices de las componentes en que se guardan sus hijos, y la llamaremos **representación basada en cursores a los hijos**.

En la representación de árboles binarios basada en cursores a los hijos, cada componente del vector debe almacenar, además del campo con la información del elemento, dos **cursores** (campos de tipo entero) que guardarán los índices de almacenamiento de los hijos izquierdo y derecho, y un campo booleano que guardará si esa componente del vector está siendo utilizada (es decir, almacena un nodo) o no.

```
constante max = 1000 {valor arbitrario, máximo número de elementos almacenables}

tipo arbin = 0..max  {el 0 significa árbol vacío; en otro caso, es la posición de
                     la raíz del árbol en el vector de nodos}
     nodo = registro
              dato:elemento;
              izq,der:arbin;
              ocupado:booleano
            freg;
     tpVectorDeNodos = vector[1..max] de nodo;
    {es un vector en el que se pueden almacenar uno o más árboles binarios}
```

```
variables vectorDeNodos:tpVectorDeNodos; a1,a2:arbin
```

El árbol vacío se representa dando el valor `0` a la variable de tipo `arbin`. Un árbol no vacío se representa de la siguiente forma: la variable de tipo `arbin` guarda el índice de la componente del vector `vectorDeNodos` en el que está el valor del elemento raíz. En esa componente se guardan además (en los campos `izq` y `der`) los índices de las componentes donde están almacenados los valores de los nodos hijos, etc. Si un nodo no tiene hijo izquierdo o derecho el valor del correspondiente campo `izq` ó `der` es `0`.



[1]	'a	2	4	v
[2]	'f	0	3	v
[3]	'b	0	0	v
[4]	'c	5	8	v
[5]	'n	6	7	v
[6]	'g	0	0	v
[7]	'e	0	0	v
[8]	'd	0	0	v
[9]	?	?	?	f
[10]	?	?	?	f

Nótese que con la representación propuesta anteriormente pueden almacenarse varios árboles binarios en un mismo vector de nodos. El campo ocupado debe consultarse si se implementan operaciones de modificación consistentes en añadir nodos, para localizar componentes libres del vector, y debe modificarse en operaciones de borrado de nodos.

La implementación de las operaciones de árboles binarios representados en un vector almacenando cursores a los hijos se plantea como ejercicio (añádanse además las operaciones de duplicar árbol y de comparación de igualdad, y estúdiese el coste de todas las operaciones).

3. Implementación dinámica

Veamos ahora la implementación dinámica de árboles binarios obtenida enlazando mediante punteros los registros que guardan la información de cada nodo. Será una **implementación basada en punteros a los hijos**, es decir, cada registro incluirá sendos punteros al subárbol izquierdo y al subárbol derecho. Incluimos las operaciones de asignación y liberación de memoria dinámica, y una operación de comparación de igualdad. Dejamos el iterador para la siguiente sección.

```
módulo genérico árbolesBinarios
parámetros
  tipo elemento
exporta
  tipo arbin {Su dominio de valores son los árboles binarios de elementos.}

  procedimiento vacío(sal a:arbin)
  {Devuelve en a el árbol vacío.}

  procedimiento plantar(sal a:arbin; ent e:elemento; ent ai,ad:arbin)
  {Devuelve en a un árbol binario cuyo elemento raíz es e, subárbol izquierdo es ai
   y subárbol derecho es ad, sin duplicar la representación de éstos en memoria,
   es decir, simplemente apuntando a éstos.}

  función esVacío(a:arbin) devuelve booleano
  {Devuelve verdad si y sólo si a es el árbol vacío.}

  procedimiento raíz(ent a:arbin; sal error:booleano; sal e:elemento)
  {Si esVacío(a) devuelve error=verdad.
   Si no, devuelve error=falso y en e el elemento raíz de a.}

  procedimiento subIzq(ent a:arbin; sal error:booleano; sal ai:arbin)
  {Si esVacío(a) devuelve error=verdad.
```

```
 Si no, devuelve error=falso y en ai el subárbol izquierdo de a, sin duplicar la
 representación en memori.a}

procedimiento subDer(ent a:arbin; sal error:booleano; sal ad:arbin)
{Si esVacío(a) devuelve error=verdad.
 Si no, devuelve error=falso y en ad el subárbol derecho de a, sin duplicar la
 representación en memoria.}

procedimiento altura(ent a:arbin; sal error:booleano; sal h:natural)
{Si esVacío(a) devuelve error=verdad.
 Si no, devuelve error=falso y en h la altura de a}

procedimiento duplicar(sal nuevo:arbin; ent viejo:arbin)
{Duplica la representación en memoria del árbol viejo guardándolo en nuevo.}

procedimiento liberar(e/s a:arbin)
{Libera la representación en memoria de a, quedando a vacío.}

función iguales(a1,a2:arbin) devuelve booleano
{Devuelve verdad si y sólo si a1 y a2 tienen los mismos elementos y en las
 mismas posiciones del árbol.}

{Las operaciones del iterador: pendientes de realizar.}

implementación
  tipos arbin = ↑nodo; {almacenamiento en memoria dinámica con punteros a los hijos}
        nodo = registro
                 dato:elemento;
                 izq,der:arbin
               freg

  procedimiento vacío(sal a:arbin)
  {Devuelve en a el árbol vacío.}
  principio
    a:=nil
  fin

  procedimiento plantar(sal a:arbin; ent e:elemento; ent ai,ad:arbin)
  {Devuelve en a un árbol binario cuyo elemento raíz es e, subárbol izquierdo es ai
   y subárbol derecho es ad, sin duplicar la representación de éstos en memoria,
   es decir, simplemente apuntando a éstos.}
  principio
    nuevoDato(a);
    a↑.dato:=e;
    a↑.izq:=ai;
    a↑.der:=ad
  fin

  función esVacío(a:arbin) devuelve booleano
  {Devuelve verdad si y sólo si a es el árbol vacío.}
  principio
    devuelve a=nil
  fin

  procedimiento raíz(ent a:arbin; sal error:booleano; sal e:elemento)
  {Si esVacío(a) devuelve error=verdad.
   Si no, devuelve error=falso y en e el elemento raíz de a.}
  principio
    si esVacío(a) entonces
      error:=verdad
    sino
      error:=falso;
      e:=a↑.dato
    fsi
  fin

  procedimiento subIzq(ent a:arbin; sal error:booleano; sal ai:arbin)
```

```
{Si esVacío(a) devuelve error=verdad.
 Si no, devuelve error=falso y en ai el subárbol izquierdo de a, sin duplicar la
 representación en memoria.}
principio
  si esVacío(a) entonces
    error:=verdad
  sino
    error:=falso;
    ai:=a↑.izq
  fin
fin

procedimiento subDer(ent a:arbin; sal error:booleano; sal ad:arbin)
{Si esVacío(a) devuelve error=verdad.
 Si no, devuelve error=falso y en ad el subárbol derecho de a, sin duplicar la
 representación en memoria.}
principio
  si esVacío(a) entonces
    error:=verdad
  sino
    error:=falso;
    ad:=a↑.der
  fin
fin

función max(a,b:entero) devuelve entero
{Función auxiliar para calcular el máximo de dos enteros}
principio
  si a≥b entonces
    devuelve a
  sino
    devuelve b
  fsi
fin

función altRec(a:arbin) devuelve natural
{PRECONDICIÓN: a es no vacío.
 Función recursiva auxiliar para calcular la altura de un árbol no vacío.}
principio
  selección
    (a↑.izq=nil) and (a↑.der=nil): devuelve 0;
    (a↑.izq=nil) and (a↑.der≠nil): devuelve 1+altRec(a↑.der);
    (a↑.izq≠nil) and (a↑.der=nil): devuelve 1+altRec(a↑.izq);
    (a↑.izq≠nil) and (a↑.der≠nil): devuelve 1+max(altRec(a↑.izq),altRec(a↑.der))
  fselección
fin

procedimiento altura(ent a:arbin; sal error:booleano; sal h:natural)
{Si esVacío(a) devuelve error=verdad.
 Si no, devuelve error=falso y en h la altura de a}
principio
  si esVacío(a) entonces
    error:=verdad
  sino
    error:=falso;
    h:=altRec(a)
  fsi
fin

procedimiento duplicar(sal nuevo:arbin; ent viejo:arbin)
{Duplica la representación en memoria del árbol viejo guardándolo en nuevo.}
variables ai,ad:arbin
principio
  si viejo=nil entonces
    nuevo:=nil
  sino
    nuevoDato(nuevo);
```

```
      nuevo↑.dato:=viejo↑.dato;
      duplicar(ai,viejo↑.izq);
      duplicar(ad,viejo↑.der);
      nuevo↑.izq:=ai;
      nuevo↑.der:=ad
    fsi
  fin

  procedimiento liberar(e/s a:arbin)
  {Libera la representación en memoria de a, quedando a vacío.}
  principio
    si a≠nil entonces
      liberar(a↑.izq);
      liberar(a↑.der);
      disponer(a);
      a:=nil
    fsi
  fin

  función iguales(a1,a2:arbin) devuelve booleano
  {Devuelve verdad si y sólo si a1 y a2 tienen los mismos elementos y en las
   mismas posiciones del árbol.}
  principio
    si a1=nil entonces
      devuelve a2=nil
    sino_si a2=nil entonces
      devuelve falso
    sino {a1 y a2 son no nulos}
      devuelve a1↑.dato=a2↑.dato and iguales(a1↑.izq,a2↑.izq)
                                 and iguales(a1↑.der,a2↑.der)
    fsi
  fin

fin {del módulo árbolesBinarios}
```

El **coste de las operaciones** anteriores está en $O(1)$, excepto para las que tienen que recorrer todo el árbol, que son: `altura`, `duplicar`, `liberar` e `iguales`, cuyo coste es lineal en el número de elementos del árbol, en el caso peor.

4. Recorridos

Para poder implementar las operaciones del iterador especificado para árboles binarios, deberemos decidir un **orden en el cual visitar todos los elementos del árbol, pasando una vez y sólo una por cada uno de los elementos**. Ese orden puede fijarse de muchas formas, existiendo al menos cuatro diferentes perfectamente identificables en la literatura:

- Recorridos en **profundidad**:
    1. Recorrido en **pre-orden**:
        1. se visita la raíz,
        2. se recorre en pre-orden el subárbol izquierdo, y
        3. se recorre en pre-orden el subárbol derecho.
    2. Recorrido en **post-orden**:
        1. se recorre en post-orden el subárbol izquierdo,
        2. se recorre en post-orden el subárbol derecho, y
        3. se visita la raíz.
    3. Recorrido en **in-orden** u orden central:
        1. se recorre en in-orden el subárbol izquierdo,
        2. se visita la raíz, y
        3. se recorre en in-orden el subárbol derecho.
- Recorrido en **anchura**:
    1. primero se visita el elemento del nivel 0 (la raíz),

2. luego los del nivel 1 (de izquierda a derecha),
3. luego los del nivel 2 (de izquierda a derecha),

… y así sucesivamente.

Se presentan a continuación las **implementaciones recursivas** de recorridos de árboles binarios. Se hace de forma que el resultado del recorrido es la devolución de una lista genérica de elementos que contiene una secuencia con todos los elementos del árbol. Para ello, se supone que previamente se ha especificado e implementado el TAD listasGenéricas con, como mínimo las siguientes operaciones:

espec listasGenéricas
usa boleanos, naturales
parámetro formal
  género elemento
fpf
género lista *{Los valores del TAD lista representan secuencias de 0 o más elementos, con al menos una operación de añadir un elemento al final de la secuencia.}*
operaciones
 crear: -> lista
 {Devuelve una lista vacía, sin elementos}
 añadir: lista l, elemento e -> lista
 {Devuelve la lista resultante de añadir el elemento e al final de la lista l}
 …
fespec

Así, una implementación de los tres recorridos en profundidad es la siguiente:

```
módulo recorridosArbin
importa árbolesBinarios, listasGenéricas
exporta
  procedimiento preOrden(ent a:arbin; e/s L:lista)
   {añade a la lista L la secuencia de elementos resultante de recorrer en pre-orden el
    árbol a, es decir, 1º la raíz, luego el subárbol izquierdo y después el derecho,
    ambos en pre-orden}

  procedimiento inOrden(ent a:arbin; e/s L:lista)
   {añade a la lista L la secuencia de elementos resultante de recorrer en in-orden el
    árbol a, es decir, 1º el subárbol izquierdo en in-orden, luego la raíz y después el
    subárbol derecho en in-orden}

  procedimiento postOrden(ent a:arbin; e/s L:lista)
   {añade a la lista L la secuencia de elementos resultante de recorrer en post-orden
    el árbol a, es decir, 1º el subárbol izquierdo en post-orden, luego el derecho en
    post-orden y finalmente la raíz}

implementación

  procedimiento preOrden(ent a:arbin; e/s L:lista)
   {añade a la lista L la secuencia de elementos resultante de recorrer en pre-orden el
    árbol a, es decir, 1º la raíz, luego el subárbol izquierdo y después el derecho,
    ambos en pre-orden}
  variables ai,ad:arbin;
            e:elemento;
            error:booleano
  principio
    si not esVacío(a) entonces
      raíz(a,error,e); {devuelve en el parámetro e el elemento raíz de a}
      añadir(L,e);     {añade el elemento e al final de la lista L}
      subIzq(a,error,ai); {devuelve en ai el subárbol izquierdo de a}
      preOrden(ai,L);     {recorre en pre-orden ai, añadiendo sus elementos a L}
```

```
      subDer(a,error,ad); {devuelve en ad el subárbol derecho de a}
      preOrden(ad,L)      {recorre en pre-orden ad, añadiendo sus elementos a L}
    fsi
  fin

  procedimiento inOrden(ent a:arbin; e/s L:lista)
  {añade a la lista L la secuencia de elementos resultante de recorrer en in-orden el
   árbol a, es decir, 1º el subárbol izquierdo en in-orden, luego la raíz y después el
   subárbol derecho en in-orden}
  variables ai,ad:arbin;
            e:elemento;
            error:booleano
  principio
    si not esVacío(a) entonces
      subIzq(a,error,ai); {devuelve en ai el subárbol izquierdo de a}
      inOrden(ai,L);      {recorre en in-orden ai, añadiendo sus elementos a L}
      raíz(a,error,e);   {devuelve en el parámetro e el elemento raíz de a}
      añadir(L,e);       {añade el elemento e al final de la lista L}
      subDer(a,error,ad); {devuelve en ad el subárbol derecho de a}
      inOrden(ad,L)       {recorre en in-orden ad, añadiendo sus elementos a L}
    fsi
  fin

  procedimiento postOrden(ent a:arbin; e/s L:lista)
  {añade a la lista L la secuencia de elementos resultante de recorrer en post-orden
   el árbol a, es decir, 1º el subárbol izquierdo en post-orden, luego el derecho en
   post-orden y finalmente la raíz}
  variables ai,ad:arbin;
            e:elemento;
            error:booleano
  principio
    si not esVacío(a) entonces
      subIzq(a,error,ai); {devuelve en ai el subárbol izquierdo de a}
      postOrden(ai,L);    {recorre en post-orden ai, añadiendo sus elementos a L}
      subDer(a,error,ad); {devuelve en ad el subárbol derecho de a}
      postOrden(ad,L);    {recorre en post-orden ad, añadiendo sus elementos a L}
      raíz(a,error,e);    {devuelve en el parámetro e el elemento raíz de a}
      añadir(L,e)         {añade el elemento e al final de la lista L}
    fsi
  fin

fin
```

El **coste en tiempo** de los tres recorridos anteriores es lineal en el número de elementos del árbol (siempre que la operación de añadir un elemento al final de la lista se pueda implementar con un coste en tiempo de orden constante, lo cual no plantea ningún problema, tal como se hizo con las pilas o las colas).

Si los recorridos no se implementan en un módulo distinto, sino que se implementan dentro del **módulo genérico** `árbolesBinarios` visto en la sección anterior, el código queda de la siguiente forma:

```
procedimiento preOrden(ent a:arbin; e/s L:lista)
{añade a la lista L la secuencia de elementos resultante de recorrer en pre-orden el
 árbol a, es decir, 1º la raíz, luego el subárbol izquierdo y después el derecho,
 ambos en pre-orden}
principio
  si not esVacío(a) entonces
    añadir(L,a↑.dato);
    preOrden(a↑.izq,L);
    preOrden(a↑.der,L)
  fsi
fin

procedimiento inOrden(ent a:arbin; e/s L:lista)
{añade a la lista L la secuencia de elementos resultante de recorrer en in-orden el
 árbol a, es decir, 1º el subárbol izquierdo en in-orden, luego la raíz y después el
```

```
 subárbol derecho en in-orden}
principio
  si not esVacío(a) entonces
      inOrden(a↑.izq,L);
      añadir(L,a↑.dato);
      inOrden(a↑.der,L)
  fsi
fin

procedimiento postOrden(ent a:arbin; e/s L:lista)
{añade a la lista L la secuencia de elementos resultante de recorrer en post-orden
 el árbol a, es decir, 1º el subárbol izquierdo en post-orden, luego el derecho en
 post-orden y finalmente la raíz}
principio
  si not esVacío(a) entonces
    postOrden(a↑.izq,L);
    postOrden(a↑.der,L);
    añadir(L,a↑.dato)
  fsi
fin
```

La implementación de un recorrido en anchura es sensiblemente más difícil y, desde luego, más ineficiente. Implementamos previamente un procedimiento que añade, al final de una lista, todos los elementos de un cierto nivel del árbol.

```
procedimiento nivel(ent a:arbin; ent i:0..maxEntero; e/s L:lista)
{Pre: a es no vacío}
{Post: añade a la lista L los elementos del nivel i de a (si existen) de izquierda a
 derecha}
variables e:elemento;
          error:boolean;
          ai,ad:arbin;
          hi,hd:natural
principio
  si i=0 entonces    {es el nivel de la raíz}
    raíz(a,error,e); {devuelve en el parámetro e el elemento raíz de a}
    añadir(L,e)      {añade el elemento e al final de la lista L}
  sino {i>0}
    subIzq(a,error,ai);
    subDer(a,error,ad); {se guarda en ai el subárbol izquierdo y en ad el derecho}
    selección
      esVacío(ai) and esVacío(ad): {no hacer nada};
      esVacío(ai) and not esVacío(ad): nivel(ad,i-1,L); {sólo hay que añadir los
                                       elementos del nivel i-1 del subárbol derecho}
      not esVacío(ai) and esVacío(ad): nivel(ai,i-1,L); {caso simétrico}
      not esVacío(ai) and not esVacío(ad): nivel(ai,i-1,L); nivel(ad,i-1,L)
     fselección
  fsi
fin
```

Y el procedimiento del recorrido en anchura queda como sigue.

```
procedimiento anchura(ent a:arbin; e/s L:lista)
{añade a la lista L los elementos de a recorridos en anchura, es decir, por niveles
 desde el 0 hasta el último y, para cada nivel, de izquierda a derecha}
variables h,i:natural
principio
   si not esVacío(a) entonces
    altura(a,error,h);
    para i:=0 hasta h hacer
      nivel(a,i,L)
    fpara
  fsi
fin
```

También es posible **implementar los recorridos con algoritmos iterativos** (más eficiente en el caso del recorrido en anchura), pero utilizando estructuras de datos auxiliares: una pila o una cola de elementos particularizadas para el tipo de dato `arbin`.

```
procedimiento preOrdenIterativo(ent a:arbin; e/s L:lista)
{añade a la lista L la secuencia de elementos resultante de recorrer en pre-orden el
 árbol a, es decir, 1º la raíz, luego el subárbol izquierdo y después el derecho,
 ambos en pre-orden}
variables p:pila; {pila de elementos de tipo arbin}
          aux:arbin
principio
  crearVacía(p);
  apilar(p,a);
  mientrasQue not esVacía(p) hacer
    aux:=cima(p);
    desapilar(p);
    si aux≠nil entonces
      añadir(L,aux↑.dato);
      apilar(p,aux↑.der);
      apilar(p,aux↑.izq)
    fsi
  fmq
fin
```

Para entender el algoritmo anterior y convencerse de que realiza un recorrido en pre-orden, se recomienda realizar trazas del mismo usando varios árboles binarios como entrada.

A continuación, de manera similar, pero utilizando una cola de elementos particularizada para el tipo de dato `arbin`, se presenta un algoritmo iterativo para el recorrido en anchura.

```
procedimiento anchuraIterativo(ent a:arbin; e/s L:lista)
{añade a la lista L los elementos de a recorridos en anchura, es decir, por niveles
 desde el 0 hasta el último y, para cada nivel, de izquierda a derecha}
variables c:cola; {cola de elementos de tipo arbin}
          aux:arbin;
          error:booleano
principio
  crearVacía(c);
  encolar(c,a);
  mientrasQue not esVacía(c) hacer
    primero(c,aux,error);
    desencolar(c);
    si aux≠nil entonces
      añadir(L,aux↑.dato);
      encolar(c,aux↑.izq);
      encolar(c,aux↑.der)
    fsi
  fmq
fin
```

Por último, es también posible implementar de forma iterativa el recorrido en in-orden, usando una pila de datos de tipo `arbin` auxiliar.

```
procedimiento inorden(ent a:arbin; e/s L:lista)
{añade a la lista L la secuencia de elementos resultante de recorrer en in-orden el
 árbol a, es decir, 1º el subárbol izquierdo en in-orden, luego la raíz y después el
 subárbol derecho en in-orden}
variables p:pila; {pila de elementos de tipo arbin}
          aux:arbin
principio
 {equivale al inicio de un hipotético iterador}
  crearVacía(p);
  aux:=a;
```

```
  mientrasQue aux≠nil hacer
    apilar(p,aux);
    aux:=aux↑.izq
  fmq;

  {not(esVacía(p)) es equivalente a 'existeSiguiente' de un iterador}
  mientrasQue not esVacía(p) hacer

    {equivale al código de 'siguiente' de un iterador}
    aux:=cima(p);
    desapilar(p);
    añadir(L,aux↑.dato);
    aux:=aux↑.der;
    mientrasQue aux≠nil hacer
      apilar(p,aux);
      aux:=aux↑.izq
    fmq

  fmq
fin
```

La idea del algoritmo anterior se puede utilizar para implementar un iterador para los elementos de un árbol binario, recorriéndolos en in-orden. Para eso, es necesario modificar la representación del tipo `arbin`, añadiendo una pila auxiliar que se usa exclusivamente para las operaciones del iterador. Llamamos `árbol` al nuevo tipo.

```
tipo árbol = registro {aumentamos la representación de arbin con una pila auxiliar
                        para implementar las operaciones del iterador}
               raíz:arbin; {puntero a la raíz de un árbol binario}
               iter:pila {pila de elementos de tipo arbin, es decir, de punteros a nodos
                          del árbol, para el iterador}
             freg



procedimiento iniciarIterador(e/s a:árbol)
{Prepara el iterador para que el siguiente elemento a visitar sea un primer
 elemento de a, si existe (situación de no haber visitado ningún elemento)}
variable aux:arbin
principio
  crearVacía(a.iter); {crea pila vacía de punteros a nodos del árbol}
  aux:=a.raíz; {raíz del árbol}
  mientrasQue aux≠nil hacer
    apilar(a.iter,aux); {apila el puntero aux (a un nodo del árbol) en la pila
                         del iterador}
    aux:=aux↑.izq
  fmq
fin

función existeSiguiente(a:árbol) devuelve booleano
{Devuelve falso si ya se han visitado todos los elementos de a; devuelve cierto en
 caso contrario}
principio
  devuelve not esVacía(a.iter) {hay siguiente si la pila del iterador es no vacía}
fin

procedimiento siguiente(e/s a:árbol; sal e:elemento; sal error:booleano)
{Si existe algún elemento de a pendiente de visitar, devuelve en e el siguiente
 elemento a visitar y error=falso, y además avanza el iterador para que a
 continuación se pueda visitar otro elemento de a. Si no quedan elementos pendientes
 de visitar devuelve error=verdad y e queda indefinido}
variable aux:arbin
principio
  si existeSiguiente(a) entonces
    error:=falso;
    aux:=cima(a.iter);
    desapilar(a.iter);
```

```
    e:=aux↑.dato; {este es el siguiente elemento visitado}
    aux:=aux↑.der;
    mientrasQue aux≠nil hacer
      apilar(a.iter,aux);
      aux:=aux↑.izq
    fmq
  sino
    error:=verdad
  fsi
fin
```

Lección 13

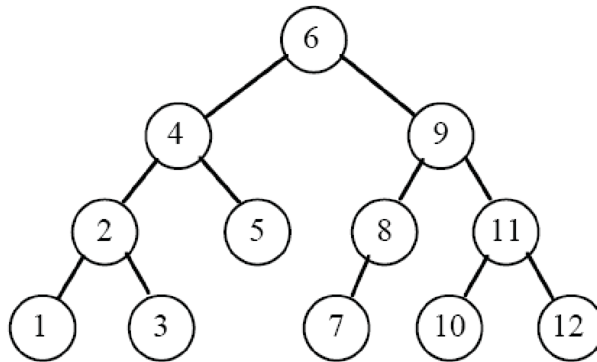
Árboles binarios de búsqueda

Indice

1. Concepto y especificación
2. Implementación dinámica
3. Implementación del TAD diccionario con ABB

1. Concepto y especificación

Los **árboles binarios de búsqueda** (ABB) pueden definirse cuando el tipo de los elementos almacenados en los nodos del árbol posee una relación de orden total ("≤"). Tienen la propiedad de que el elemento raíz es mayor o igual que todos los elementos del subárbol izquierdo y menor que todos los elementos del subárbol derecho. Además, los subárboles izquierdo y derecho son a su vez árboles binarios de búsqueda. Si no se permite que el árbol tenga elementos repetidos (la relación de orden es estricta, "<"), entonces el elemento raíz es estrictamente mayor que todos los elementos del subárbol izquierdo.



Ejemplo de árbol binario de búsqueda de naturales (sin elementos repetidos)

Las **operaciones básicas** con árboles de búsqueda son la **inserción** y la **búsqueda** de elementos. En la mayoría de ocasiones se precisa además la operación de **borrado** de un elemento. Por tanto, **si no se permiten elementos repetidos**, los ABB **sirven para representar el TAD diccionario de elementos**, visto en un tema anterior (sin más que almacenar en cada nodo del árbol una clave y un valor en lugar de un elemento, y considerar el orden de las claves "<" como el criterio de ordenación de los nodos del árbol) . Además, si el iterador se construye para obtener un recorrido en in-orden del árbol (como se vio en la lección anterior), el iterador recorre los elementos en el orden que tienen definido (<).

Veamos la especificación de las operaciones básicas, suponiendo que puede haber elementos repetidos (comparables mediante "≤").

espec abbs
parámetro formal
  género elemento
  operación _≤_: elemento e1, elemento e2 -> booleano (… ídem con =,≠,≥,<,>)
fpf
género abb *{Su dominio de valores son los árboles binarios de búsqueda de elementos, con la posibilidad de contener elementos repetidos; sirven, por tanto, para almacenar multiconjuntos de elementos}*

operaciones

vacío: -> abb
{Devuelve el árbol binario de búsqueda vacío, sin elementos}

esVacío?: abb a -> booleano
{Devuelve verdad si y sólo si a es vacío}

añadir: abb a, elemento e -> abb
{Devuelve el abb resultante de añadir un ejemplar del elemento e al árbol a, manteniendo la propiedad de abb, es decir, todo elemento almacenado en el árbol es mayor o igual que los elementos de su subárbol izquierdo y menor que los elementos de su subárbol derecho}

está?: abb a, elemento e -> booleano
{Devuelve verdad si y sólo si hay algún ejemplar de e está en a}

borrar: abb a, elemento e -> abb
{Si e está en a, devuelve un árbol igual al resultante de borrar una de las apariciones de e en a. Si e no está en a, devuelve un árbol igual que a}

fespec

2. Implementación dinámica

Veremos ahora una implementación dinámica, es decir, basada en punteros, que para cada elemento del árbol guardará los punteros a sus hijos.

```
módulo genérico árbolesBinariosBúsqueda {implementa la especificación abbs}
parámetros
  tipo elemento
  con función "≤"(e1,e2:elemento) devuelve booleano
          {… ídem con el resto funciones de comparación: <=,=,>,>= }
exporta
  tipo abb {Su dominio de valores son los árboles binarios de búsqueda de elementos,
          con la posibilidad de contener elementos repetidos;
          sirven, por tanto, para almacenar multiconjuntos de elementos}

  procedimiento vacío(sal a:abb)
  {Devuelve en a el árbol binario de búsqueda vacío, sin elementos}

  función esVacío(a:abb) devuelve booleano
  {Devuelve verdad si y sólo si a es vacío}

  procedimiento añadir(e/s a:abb; ent e:elemento)
  {Devuelve en a el abb resultante de añadir un ejemplar del elemento e, manteniendo
   la propiedad de abb (todo elemento del árbol es mayor o igual que los elementos de
   su subárbol izquierdo y menor que los elementos de su subárbol derecho}

  función está(a:abb; e:elemento) devuelve booleano
  {Devuelve verdad si y sólo si hay algún ejemplar de e está en a}

  procedimiento borrar(e/s a:abb; ent e:elemento)
  {Si e está en a, devuelve un árbol igual al resultante de borrar de a una de las
   apariciones de e en a. Si e no está en a, devuelve un árbol igual que a}

implementación
  tipos abb = ↑nodo
        nodo = registro
                 dato:elemento;
                 izq,der:abb
```

```
            freg

procedimiento vacío(sal a:abb)
{Devuelve en a el árbol binario de búsqueda vacío, sin elementos}
principio
  a:=nil
fin

función esVacío(a:abb) devuelve booleano
{Devuelve verdad si y sólo si a es vacío}
principio
  devuelve a=nil
fin

procedimiento añadir(e/s a:abb; ent e:elemento)
{Devuelve en a el abb resultante de añadir un ejemplar del elemento e, manteniendo
 la propiedad de abb (todo elemento del árbol es mayor o igual que los elementos de
 su subárbol izquierdo y menor que los elementos de su subárbol derecho}
principio
  si a=nil entonces
    nuevoDato(a);
    a↑.dato:=e;
    a↑.izq:=nil;
    a↑.der:=nil
  sino
    si e≤a↑.dato entonces
      añadir(a↑.izq,e)
    sino
      añadir(a↑.der,e)
    fsi
  fsi
fin

función está(a:abb; e:elemento) devuelve booleano
{Devuelve verdad si y sólo si hay algún ejemplar de e está en a}
principio
  si a=nil entonces
    devuelve falso
  sino
    selección
      e<a↑.dato: devuelve está(a↑.izq,e);
      e=a↑.dato: devuelve verdad;
      e>a↑.dato: devuelve está(a↑.der,e)
    fselección
  fsi
fin

{Este procedimiento es auxiliar para el procedimiento de borrar, ver abajo}
procedimiento borrarMáximo(e/s a:abb; sal e:elemento)
{Precondición: a es no vacío. Devuelve en e un elemento máximo de a y lo borra de a}
variable aux:abb
principio
  si a↑.der=nil entonces {el máximo del árbol está en la raíz}
    e:=a↑.dato;
    aux:=a;
    a:=a↑.izq;
    disponer(aux)
  sino {el máximo del árbol está en el subárbol derecho}
    borrarMáximo(a↑.der,e)
  fsi
fin

procedimiento borrar(e/s a:abb; ent e:elemento)
variable aux:abb
principio
  si a≠nil entonces
    selección
```

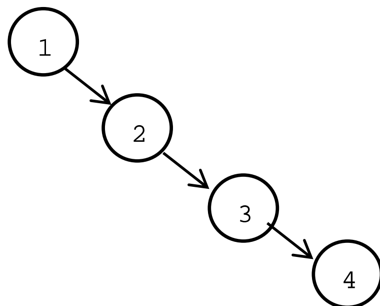
```
        e<a↑.dato: borrar(a↑.izq,e);
        e>a↑.dato: borrar(a↑.der,e);
        e=a↑.dato: si a↑.izq=nil entonces
                     aux:=a;
                     a:=a↑.der;
                     disponer(aux)
                   sino
                     borrarMáximo(a↑.izq,a↑.dato)
                   fsi
      fselección
    fsi
  fin

fin {del modulo}
```

Al módulo anterior se le podrían añadir las operaciones de: **altura, duplicar, liberar y comparación de igualdad** con idéntica implementación que la vista en la lección anterior para árboles binarios cualesquiera.

De forma análoga, las **operaciones del iterador** vistas para árboles binarios son válidas también (aumentando la representación del tipo con el campo necesario para implementar el iterador, como se hizo en la lección anterior). Con el iterador de la lección anterior, basado en el recorrido en in-orden, el recorrido de visita de los elementos del árbol se produciría en el orden definido para el tipo elemento por el operador "≤".

Si en la especificación e implementación del TAD `abb` vistas anteriormente, modificamos la semántica de la operación `añadir` para que **no se añadan ejemplares repetidos de un mismo elemento**, se consigue tener **árboles binarios de búsqueda sin elementos repetidos**. En ese caso, tenemos una **nueva forma de implementar el TAD `conjunto`.**

En cuanto al **coste en tiempo de las operaciones en el caso peor**, las operaciones `vacío` y `esVacío` tienen coste en $O(1)$. En cambio, las operaciones `añadir`, `está` y `borrar` tienen un coste en el caso peor en $O(n)$, es decir, lineal en n, siendo n el número de elementos almacenados en el árbol. Esto es así porque **el árbol podría ser degenerado**, como el de la figura siguiente (obtenido, por ejemplo, si se han añadido los elementos en este orden: 1, 2, 3, 4), y cualquiera de esas cuatro operaciones, en el caso peor, tendrían que llegar hasta la hoja del árbol.



Desde un punto de vista práctico, **si no se precisa garantizar un coste en tiempo bajo en el caso peor y es suficiente con obtener un coste bajo en el caso promedio**, los árboles binarios de búsqueda son una buena solución pues puede demostrarse que **la altura promedio de un ABB generado aleatoriamente tiene orden logarítmico**.

3. Implementación del TAD diccionario con ABB

Si **en lugar de almacenar un dato de tipo `elemento`** en cada nodo del ABB, **almacenamos una `clave` y un `valor`** asociado a la clave y **no permitimos almacenar claves repetidas**, disponemos de una estructura de datos alternativa a la que vimos en el tema de TAD lineales (que estaba basada en una lista enlazada mediante punteros) para almacenar un valor del TAD **diccionario**. Cada nodo del árbol contendrá una clave estrictamente mayor que todas las claves almacenadas en su subárbol izquierdo y estrictamente menor que todas las claves almacenadas en su subárbol derecho. De esta forma, se podrá realizar una búsqueda de clave utilizando el orden "<" definido en el dominio del tipo `clave`.

Nótese que la interfaz del módulo es idéntica a la vista en la lección 10. Sólo cambiaremos la parte privada (implementación) del módulo.

```
módulo genérico diccionariosEnABB {implementa un diccionario usando un ABB}
parámetros
  tipos clave,valor
  con función "<"(c1,c2:clave) devuelve booleano
  con función "="(c1,c2:clave) devuelve booleano
  {suponemos que el tipo clave tiene definidas una función de orden y otra de
   igualdad}
exporta
  tipo diccionario {Los valores del TAD diccionario representan conjuntos de pares
                    (clave,valor) en los que no se permiten claves repetidas}

  procedimiento crear(sal d:diccionario)
  {Devuelve en d un diccionario vacío, sin elementos}

  procedimiento añadir(e/s d:diccionario; ent c:clave; ent v:valor)
  {Si en d no hay ningún par con clave c, añade a d el par (c,v);
   si en d hay un par (c,v'), entonces lo sustituye por el par (c,v)}

  procedimiento buscar(ent d:diccionario; ent c:clave;
                       sal éxito:booleano; sal v:valor)
  {Devuelve éxito=verdad si en d hay algún par con clave c, falso en caso contrario.
   En caso de éxito, además, devuelve en v el valor asociado a la clave c en d}

  procedimiento quitar(ent c:clave; e/s d:diccionario)
  {Si en d hay un par con clave c, lo borra. En caso contrario, d no se modifica}

  función cardinal(d:diccionario) devuelve natural
  {Devuelve el nº de pares del diccionario d}

  función esVacío(d:diccionario) devuelve booleano
  {Devuelve verdad si y sólo si d no tiene pares}

  procedimiento duplicar(sal dSal:diccionario; ent dEnt diccionario)
  {Duplica la representación del diccionario dEnt en el diccionario dSal}

  función iguales(d1,d2:diccionario) devuelve booleano
  {Devuelve verdad si y sólo si los diccionarios d1 y d2 tienen los mismos pares de
   claves y valores}

  procedimiento liberar(e/s d:diccionario)
  {Devuelve en d el diccionario vacío y además libera la memoria utilizada
   previamente por d}

  procedimiento iniciarIterador(e/s d:diccionario)
  {Inicializa el iterador para recorrer los pares del diccionario d, de forma que el
   siguiente par a visitar sea el primero que visitamos (situación de no haber
   visitado ningún par).}

  función existeSiguiente(d:diccionario) devuelve booleano
  {Devuelve falso si ya se han visitado todos los pares de d; verdad en otro caso}

  procedimiento siguiente(e/s d:diccionario;
                          sal c:clave; sal v:valor; sal error:booleano)
  {Si existe algún par de d pendiente de visitar, devuelve en c y v la clave
   y el valor, respectivamente, del siguiente par a visitar y error=falso, y
   además avanza el iterador para que a continuación se pueda visitar otro par
   de d. Si no quedan pares pendientes de visitar devuelve error=verdad, c y v
   quedan indefinidos y d queda como estaba}
```

```
implementación

  tipos ptNodo = ↑nodo {almacenamos el diccionario en un ABB ordenado por claves}
        nodo = registro
                 laClave:clave; {no podrá haber 2 nodos en el árbol con la misma clave}
                 elValor:valor;
                 izq,der:ptNodo
               freg
        diccionario = registro
                        raíz:ptNodo;    {puntero a la raíz del árbol}
                        tamaño:natural; {número de pares (clave,valor) almacenados}
                        iter:pila       {pila de datos de tipo ptNodo, para implementar
                      freg               el iterador}

  procedimiento crear(sal d:diccionario)
  {Devuelve en d un diccionario vacío, sin elementos}
  principio
    d.raíz:=nil;
    d.tamaño:=0
  fin

  {Este procedimiento es auxiliar para el de añadir}
  procedimiento añadirRec(e/s p:ptNodo; ent c:clave; ent v:valor; sal nuevo:booleano}
  {Si en el árbol de raíz apuntada por p no hay ningún par con clave c, añade a ese
   árbol el par (c,v) y nuevo=verdad.
   Si en el árbol de raíz apuntada por p hay un par (c,v'), entonces lo sustituye
   por el par (c,v) y nuevo=falso}
  variables p:ptNodo
  principio
    si p=nil entonces
      nuevoDato(p);
      p↑.laClave:=c;
      p↑.elValor:=v;
      p↑.izq:=nil;
      p↑.der:=nil;
      nuevo:=verdad
    sino
      si c<p↑.laClave entonces
        añadirRec(p↑.izq,c,v,nuevo)
      sino_si c>p↑.laClave entonces
        añadirRec(p↑.der,c,v,nuevo)
      sino {c=p↑.laClave, es decir, ya existía un par con esa clave}
        p↑.elValor:=v;
        nuevo:=falso
      fsi
    fsi
  fin

  procedimiento añadir(e/s d:diccionario; ent c:clave; ent v:valor)
  {Si en d no hay ningún par con clave c, añade a d el par (c,v);
   si en d hay un par (c,v'), entonces lo sustituye por el par (c,v)}
  variable nuevo:booleano
  principio
    añadirRec(d.raíz,c,v,nuevo);
    si nuevo entonces
      d.tamaño:=d.tamaño+1
    fsi
  fin

  {Este procedimiento es auxiliar para el de buscar}
  procedimiento buscarRec(ent p:ptNodo; ent c:clave; sal éxito:booleano; sal v:valor)
  {Devuelve éxito=verdad si en el árbol de raíz apuntada por p hay algún par con
   clave c, falso en caso contrario.
   En caso de éxito, además, devuelve en v el valor asociado a la clave c}
  principio
```

```
  si p=nil entonces
    éxito:=falso
  sino
    selección
      c<p↑.laClave: buscarRec(p↑.izq,c,éxito,v);
      c=p↑.laClave: éxito:=verdad;
                    v:=p↑.elValor;
      c>p↑.laClave: buscarRec(a↑.der,c,éxito,v)
    fselección
  fsi
fin

procedimiento buscar(ent d:diccionario; ent c:clave;
                     sal éxito:booleano; sal v:valor)
{Devuelve éxito=verdad si en d hay algún par con clave c, falso en caso contrario.
 En caso de éxito, además, devuelve en v el valor asociado a la clave c en d}
principio
  buscarRec(d.raíz,c,éxito,v)
fin

{Este procedimiento es auxiliar para el procedimiento de quitarRec, ver abajo}
procedimiento borrarMáximo(e/s p:ptNodo; sal c:clave; sal v:valor)
{Precondición: p es no vacío. Devuelve el par (c,v) cuya clave c es la máxima del
 subárbol de raíz p y borra ese par del subárbol de raíz p}
variable aux:ptNodo
principio
  si p↑.der=nil entonces
    c:=p↑.laClave;
    v:=p↑.elValor;
    aux:=p;
    p:=p↑.izq;
    disponer(aux)
  sino
    borrarMáximo(p↑.der,c,v)
  fsi
fin

{Este procedimiento es auxiliar para el de quitar}
procedimiento quitarRec(ent c:clave; e/s p:ptNodo; sal borrado:booleano)
{Si en el árbol de raíz apuntada por p hay un par con clave c, lo borra y
 borrado=verdad. En caso contrario, p no se modifica y borrado=falso}
variable aux:ptNodo
principio
  si p=nil entonces
    borrado:=falso
  sino
    selección
      c<p↑.laClave: quitarRec(c,p↑.izq,borrado);
      c>p↑.laClave: quitarRec(c,p↑.der,borrado);
      c=p↑.laClave: si p↑.izq=nil entonces
                      aux:=p;
                      p:=p↑.der;
                      disponer(aux);
                      borrado:=verdad
                    sino
                      borrarMáximo(p↑.izq,p↑.laClave,p↑.elValor)
                    fsi
    fselección
  fsi
fin

procedimiento quitar(ent c:clave; e/s d:diccionario)
{Si en d hay un par con clave c, lo borra. En caso contrario, d no se modifica}
variable borrado:booleano
principio
  quitarRec(c,d.raíz,borrado)
```

```
  si borrado entonces
    d.tamaño:=d.tamaño-1
  fsi
fin

función cardinal(d:diccionario) devuelve natural
{Devuelve el nº de pares del diccionario d}
principio
  devuelve d.tamaño
fin

función esVacío(d:diccionario) devuelve booleano
{Devuelve verdad si y sólo si d no tiene pares}
principio
  devuelve d.raíz=nil
fin

{Este procedimiento es auxiliar para el de duplicar}
procedimiento duplicarRec(sal pSal:ptNodo; ent pEnt:ptNodo)
{Duplica el árbol apuntado por pEnt en el árbol apuntado por pSal}
principio
  si pEnt=nil entonces
    pSal:=nil
  sino
    nuevoDato(pSal);
    pSal↑.laClave:=pEnt↑.laClave;
    pSal↑.elValor:=pEnt↑.elValor;
    duplicarRec(pSal↑.izq,pEnt↑.izq);
    duplicarRec(pSal↑.der,pEnt↑.der)
  fsi
fin

procedimiento duplicar(sal dSal:diccionario; ent dEnt diccionario)
{Duplica la representación del diccionario dEnt en el diccionario dSal}
principio
  duplicarRec(dSal.raíz,dEnt.raíz);
  dSal.tamaño:=dEnt.tamaño
fin

{Este procedimiento es auxiliar para el de liberar}
procedimiento liberarRec(e/s p:ptNodo)
{Libera toda la memoria utilizada por el árbol apuntado por p}
principio
  si p≠nil entonces
    liberarRec(p↑.izq);
    liberarRec(p↑.der);
    disponer(p)
  fsi
fin

procedimiento liberar(e/s d:diccionario)
{Devuelve en d el diccionario vacío y además libera la memoria utilizada
 previamente por d}
principio
  liberarRec(d.raíz);
  d.tamaño:=0;
  d.raíz:=nil
fin

procedimiento iniciarIterador(e/s d:diccionario)
{Inicializa el iterador para recorrer los pares del diccionario d, de forma que el
 siguiente par a visitar sea el primero que visitamos (situación de no haber
 visitado ningún par).}
variable aux:ptNodo
principio
```

```
  crearVacía(d.iter); {crea pila vacía de punteros a nodos del árbol}
  aux:=d.raíz; {raíz del árbol}
  mientrasQue aux≠nil hacer
    apilar(d.iter,aux); {apila el puntero aux (a un nodo del árbol) en la pila
                         del iterador}
    aux:=aux↑.izq
  fmq
fin

función existeSiguiente(d:diccionario) devuelve booleano
{Devuelve falso si ya se han visitado todos los pares de d; verdad en otro caso}
principio
  devuelve not esVacía(d.iter) {existe siguiente si la pila del iterador es no vacía}
fin

procedimiento siguiente(e/s d:diccionario;
                        sal c:clave; sal v:valor; sal error:booleano)
{Si existe algún par de d pendiente de visitar, devuelve en c y v la clave
 y el valor, respectivamente, del siguiente par a visitar y error=falso, y
 además avanza el iterador para que a continuación se pueda visitar otro par
 de d. Si no quedan pares pendientes de visitar devuelve error=verdad, c y v
 quedan indefinidos y d queda como estaba}
variable aux:ptNodo
principio
  si existeSiguiente(d) entonces
    error:=falso;
    aux:=cima(d.iter);
    desapilar(d.iter);
    c:=aux↑.laClave; {esta es la clave del siguiente elemento visitado}
    v:=aux↑.elValor; {y el valor asociado a la clave}
    aux:=aux↑.der;
    mientrasQue aux≠nil hacer
      apilar(d.iter,aux);
      aux:=aux↑.izq
    fmq
  sino
    error:=verdad
  fsi
fin

{Para la función iguales veamos dos implementaciones alternativas}

{La primera, basada en recorrer el árbol del primer diccionario en pre-orden y para
 Cada uno de sus pares <clave,valor> comprobar si está en el otro diccionar.}

función iguales(d1,d2:diccionario) devuelve booleano
{Devuelve verdad si y sólo si los diccionarios d1 y d2 tienen los mismos pares de
 claves y valores}
principio
  si esVacío(d1) and esVacío(d2) entonces
    devuelve verdad
  sino_si cardinal(d1)≠cardinal(d2) entonces
    devuelve falso
  sino {ambos tienen el mismo número (no nulo) de pares}
    devuelve igualesRec(d1.raíz,d2.raíz)
  fsi
fin

{Esta función es auxiliar para la función iguales anterior}
función igualesRec(p1,p2:ptNodo) devuelve booleano
{Devuelve verdad si y sólo si todas las parejas <clave,valor> de p1 están en p2.}
variable éxito:booleano;
         v:valor
principio
  si p1=nil entonces
```

```
      devuelve verdad {trivialmente}
    sino
      buscarRec(p2,p1↑.laClave,éxito,v); {busca en p2 la raíz de p1}
      si éxito andthen v=p1↑.elValor entonces {la raíz de p1 está en p2}
        devuelve igualesRec(p1↑.izq,p2) andthen igualesRec(p1↑.der,p2)
      sino
        devuelve falso
      fsi
    fsi
  fin

  {La segunda alternativa para la función iguales se basa en usar recorridos in-orden
   (como el usado en el iterador) simultáneos de ambos árboles.}

  función iguales(d1,d2:diccionario) devuelve booleano
  {Devuelve verdad si y sólo si los diccionarios d1 y d2 tienen los mismos pares de
   claves y valores}
  variables igual:booleano;
            aux1,aux2:ptNodo;
            pila1,pila2:pila {pilas de datos de tipo ptNodo}
  principio
    si esVacío(d1) and esVacío(d2) entonces
      devuelve verdad
    sino_si cardinal(d1)≠cardinal(d2) entonces
      devuelve falso
    sino {ambos tienen el mismo número (no nulo) de claves:
          los recorremos simultáneamente en orden por clave (in-orden)}
      igual:=verdad;
      {inicializar recorrido en diccionario d1: llegar hasta el primero en in-orden
       dejando la pila preparada con los nodos del camino desde la raíz}
      crear(pila1);
      aux1:=d1.raíz;
      mientrasQue aux1≠nil hacer
        apilar(pila1,aux1);
        aux1:=aux1↑.izq
      fmq;
      {lo mismo con diccionario d2}
      crear(pila2);
      aux2:=d2.raíz;
      mientrasQue aux2≠nil hacer
        apilar(pila2,aux2);
        aux2:=aux2↑.izq
      fmq;
      {mientras sean iguales y no se hayan tratado todos}
      mientrasQue igual and not esVacía(pila1) hacer {pila2 se vacía al mismo tiempo}
        {obtener el siguiente par <clave,valor> de cada diccionario}
        aux1:=cima(pila1);
        aux2:=cima(pila2);
        {compararlos}
        igual:=(aux1↑.laClave=aux2↑.laClave) and (aux1↑.elValor=aux2↑.elValor);
        {avanzar, para preparar el siguiente dato a tratar en diccionario d1}
        desapilar(pila1);
        aux1:=aux1↑.der;
        mientrasQue pAux1≠nil hacer
          apilar(pila1,aux1);
          aux1:=aux1↑.izq
        fmq
        {lo mismo con diccionario d2}
        desapilar(pila2);
        aux2:=aux2↑.der;
        mientrasQue aux2≠nil hacer
          apilar(pila2,aux2);
          aux2:=aux2↑.izq
```

```
          fmq
        fmq;
        devuelve igual
      fsi
    fin

  fin {del módulo}
```

En cuanto al **coste en tiempo de las operaciones en el caso peor**, las operaciones fundamentales del diccionario, `buscar`, `añadir` y `quitar`, tienen un **coste en el caso peor en $O(n)$, es decir, lineal en n**, siendo n el número de claves del diccionario. Esto es así porque **el árbol podría ser degenerado**, como ya se explicó en la sección anterior.

Tal y como se dijo en la sección anterior, si no es necesario garantizar un coste bajo en el caso peor y nos conformamos con el **caso promedio**, un ABB es una buena solución para almacenar un diccionario porque puede demostrarse que, **en media, la altura de un ABB generado aleatoriamente es logarítmica en el número de claves**.

Lección 14

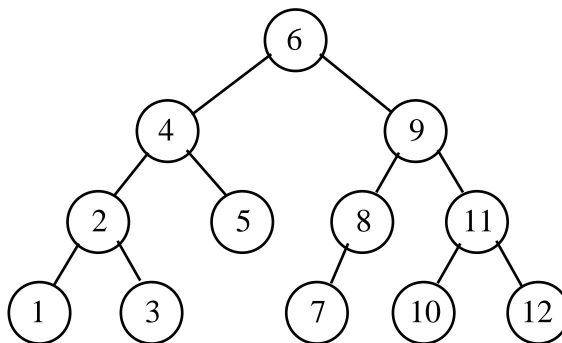
Árboles AVL

Indice

1. Definición y teorema de AVL
2. Representación de los árboles AVL
3. Inserción en AVL
4. Borrado en AVL

1. Definición y teorema AVL

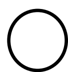
Un **árbol binario se dice equilibrado** si, y sólo si, para cada uno de sus nodos ocurre que las alturas de sus dos subárboles (izquierdo y derecho) difieren como mucho en una unidad (definición dada en 1962 por **G. M. Adelson-Velskii y Y. M. Landis**). Los **árboles binarios de búsqueda equilibrados** reciben el nombre abreviado de **árboles AVL** (ver el ejemplo de la figura).



El interés fundamental de los árboles AVL es que las **operaciones habituales sobre árboles binarios de búsqueda** (inserción, pertenencia y borrado) se pueden realizar en el peor de los casos con un **coste en $O(\log n)$**, si n es el número de nodos del árbol. Se mejora, por tanto, el coste en el caso peor que vimos para árboles binarios de búsqueda cualesquiera, que estaba en $O(n)$.

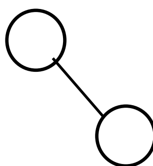
Para demostrar esa afirmación, veamos cuál es la **máxima altura h que puede tener un AVL de n nodos**. O lo que es lo mismo, veamos cómo calcular el **mínimo número de nodos n que puede tener un AVL de altura h**. Los árboles con esa propiedad se denominan **árboles de Fibonacci**. Se definen de la siguiente forma: un árbol de Fibonacci de altura h, denotado por F_h, es un AVL con esa altura y que tiene un número mínimo de nodos (es decir, no existe otro árbol binario de altura h y con menos nodos que F_h).

El árbol de Fibonacci de altura 0, F_0, es el siguiente, y tiene 1 nodo:



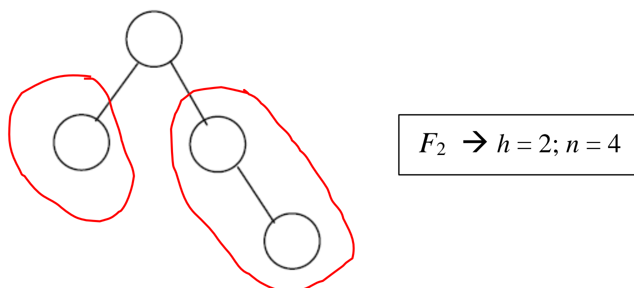
F_0 → $h = 0$; $n = 1$

Un árbol de Fibonacci de altura 1, F_1, es el siguiente, y tiene 2 nodos (evidentemente, también su simétrico lo es):

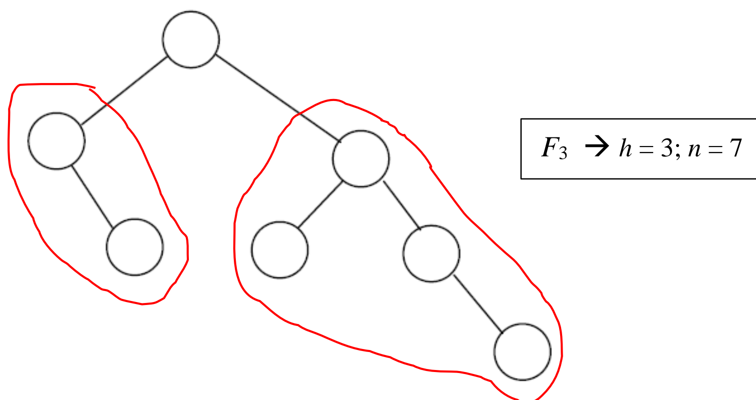


F_1 → $h = 1$; $n = 2$

Un árbol de Fibonacci de altura 2, F_2, se conseguirá poniendo una raíz de la que cuelguen como subárboles un Fibonacci de altura 0 y otro de altura 1. Cualquier otro AVL de altura 2 tendrá los mismos o más nodos:

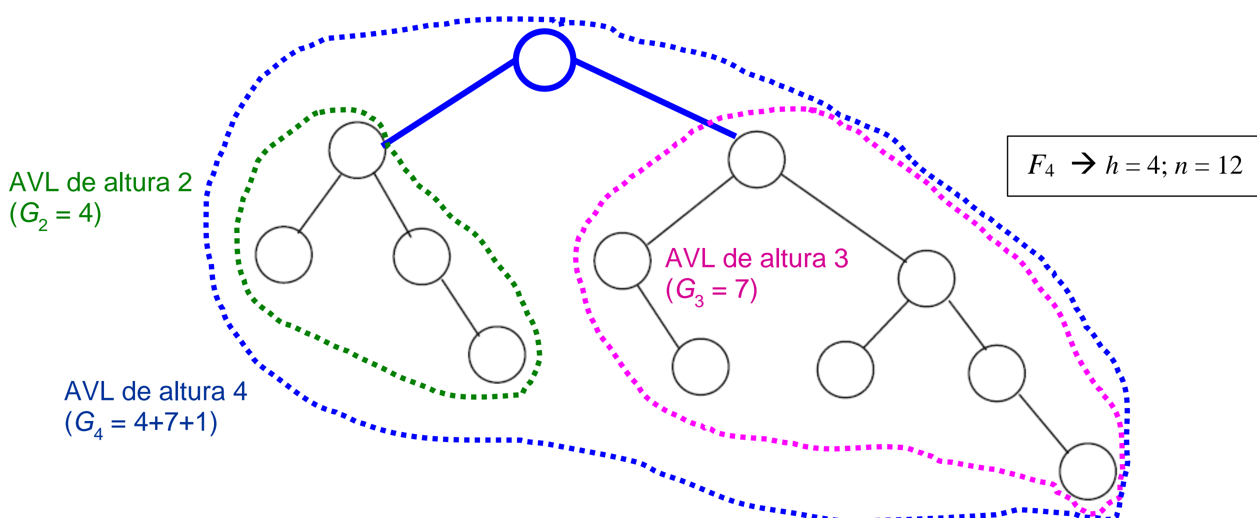


De igual forma, para crear un Fibonacci de altura 3, F_3, se coloca una raíz de la que cuelgan un F_1 y un F_2. Cualquier otro AVL de altura 3 tendrá los mismos o más nodos:



En general, para obtener un árbol de Fibonacci de altura h, F_h, se coloca un F_{h-2} y un F_{h-1} colgando como subárboles de un nodo raíz. Por tanto, si **llamamos G_h al número de nodos de un AVL de altura h que tenga el mínimo número de nodos posibles**, se tiene que:

$$G_h = G_{h-1} + G_{h-2} + 1$$



Los árboles de Fibonacci se llaman así por la semejanza de la recurrencia anterior de la función G_h con la sucesión de Fibonacci:

$$F_0 = 1;\ F_1 = 1;\ F_h = F_{h-1} + F_{h-2},\ h > 1$$

n	G_n	F_n
0	1	1
1	2	1
2	4	2
3	7	3
4	12	5
5	20	8
6	33	13
…	…	…

Como puede observarse en la tabla de la izquierda, se tiene que:

$$G_h = F_{h+2} - 1$$

Y para la sucesión de Fibonacci se sabe que:

$$F_h > (\phi^h / \sqrt{5}) - 1, \text{ con } \phi = (1 + \sqrt{5})/2 \text{ la razón áurea.}$$

Por tanto:

$$G_h > (\phi^h + 2 / \sqrt{5}) - 2$$

Es decir, el número de nodos n de cualquier árbol AVL de altura h verifica (recordar que G_h es número de nodos de un AVL de altura h que tenga el mínimo número de nodos posibles, por tanto $n \geq G_h$):

$$n \geq G_h > (\phi^{h+2} / \sqrt{5}) - 2$$

Y de ahí, despejando la h, se deduce el **Teorema de Adelson-Velskii y Landis** (1962), que asegura que la altura h de un AVL cualquiera está acotada por la siguiente función logarítmica del número de nodos n del árbol:

$$h < 1'4404 \log_2(n + 2) - 0'328$$

Por tanto, como la **altura de un AVL está acotada por el logaritmo del número de nodos del árbol**, se puede garantizar que la **búsqueda de un nodo será, en el peor de los casos, de coste en tiempo en $O(\log n)$**, siendo n el número de nodos del árbol. Si además se consigue una implementación de la inserción y del borrado que mantengan el árbol dentro de la clase de los AVL, esas dos operaciones también tendrán garantizado un coste en $O(\log n)$ en el caso peor.

2. Representación de los árboles AVL

Para implementar las operaciones de AVL se precisa almacenar en cada nodo del árbol la información sobre el factor de equilibrio del nodo. **Llamamos factor de equilibrio, F_e, de un nodo a la diferencia de las alturas de su subárbol derecho e izquierdo** (interpretando que, si uno de esos subárboles es vacío, ese subárbol tiene altura "– 1").

Decimos que un **nodo es perfectamente equilibrado si su F_e es 0**, es decir, sus subárboles tienen la misma altura. Un nodo se dice **pesado a derechas si su F_e es +1**, es decir, la altura de su subárbol derecho es una unidad mayor que la del izquierdo. Por último, un nodo es **pesado a izquierdas si su F_e es –1**, es decir, la altura de su subárbol izquierdo es una unidad mayor que la del derecho.

La implementación del algoritmo de búsqueda de una clave o elemento en un AVL es exactamente la misma que hemos visto en la lección anterior para árboles de búsqueda binaria cualesquiera. Dado que el árbol no se modifica en la búsqueda, los factores de equilibrio de sus nodos permanecerán inalterados.

Sin embargo, en la implementación de los algoritmos de inserción y borrado de claves, será necesario recalcular el factor de equilibrio de los nodos (debido a la ganancia o pérdida de altura de sus subárboles) y, por tanto, en ocasiones, será necesario **reequilibrar el árbol para mantenerlo dentro de la clase de los AVL**, es decir, para que el factor de equilibrio de sus nodos se mantenga siempre entre –1 y +1.

La representación de los AVL puede hacerse, por tanto, así:

```
tipos ptNodo = ↑nodo;
      factorEquil = (pesadoIzq,equilibrado,pesadoDer); {o lo que es lo mismo:-1,0,1}
      nodo = registro
               laClave:clave; {o bien dato:elemento si no se trata de un diccionario}
               elValor:valor;
               equilibrio:factorEquil;
               izq,der:ptNodo
             freg
```

```
diccionario = registro
                raíz:ptNodo;
                tamaño:natural;
                iter:pila   {pila de datos de tipo ptNodo, para el iterador}
              freg
```

El algoritmo de búsqueda queda exactamente igual que en el caso de los ABB:

```
{Este procedimiento es auxiliar para el de buscar}
procedimiento buscarRec(ent p:ptNodo; ent c:clave; sal éxito:booleano; sal v:valor)
{Devuelve éxito=verdad si en el árbol de raíz apuntada por p hay algún par con
 clave c, falso en caso contrario.
 En caso de éxito, además, devuelve en v el valor asociado a la clave c}
principio
  si p=nil entonces
    éxito:=falso
  sino
    selección
      c<p↑.laClave: buscarRec(p↑.izq,c,éxito,v);
      c=p↑.laClave: éxito:=verdad;
                    v:=p↑.elValor;
      c>p↑.laClave: buscarRec(p↑.der,c,éxito,v)
    fselección
  fsi
fin

procedimiento buscar(ent d:diccionario; ent c:clave; sal éxito:booleano; sal v:valor)
{Devuelve éxito=verdad si en d hay algún par con clave c, falso en caso contrario.
 En caso de éxito, además, devuelve en v el valor asociado a la clave c en d}
principio
  buscarRec(d.raíz,c,éxito,v)
fin
```

Dado que en un AVL la altura está acotada por el logaritmo del número de nodos, el coste en el caso peor del algoritmo anterior para AVL está en $O(\log n)$.

3. Inserción en AVL

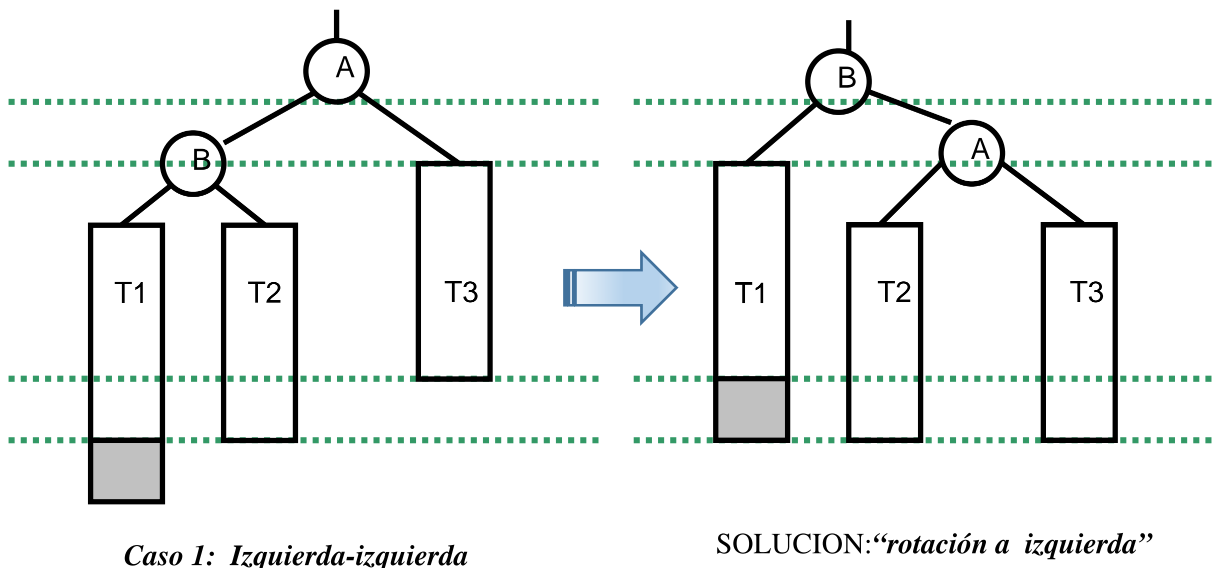
El proceso de inserción en un AVL consta de tres pasos:

1. Buscar la clave hasta encontrar la posición de inserción o modificación del valor asociado. Es un proceso idéntico al de la inserción en árbol binario de búsqueda. Si debe insertarse un nuevo nodo, se inserta siempre en una hoja.
2. Insertar el nuevo nodo hoja, con factor de equilibrio `equilibrado`.
3. Desandar el camino de búsqueda, verificando el equilibrio de los nodos del camino, y reequilibrándolos si es necesario.

Evidentemente, la implementación más fácil de escribir es recursiva. Se añade un parámetro de salida de tipo booleano que indica si el subárbol en que se ha insertado la nueva clave ha aumentado de altura (dato necesario para recalcular el factor de equilibrio). Si no ha aumentado la altura del subárbol, no cambia el factor de equilibrio del nodo padre de ese subárbol. Si, por el contrario, ha aumentado la altura del subárbol, aparecen varias posibilidades que resumimos en la siguiente tabla:

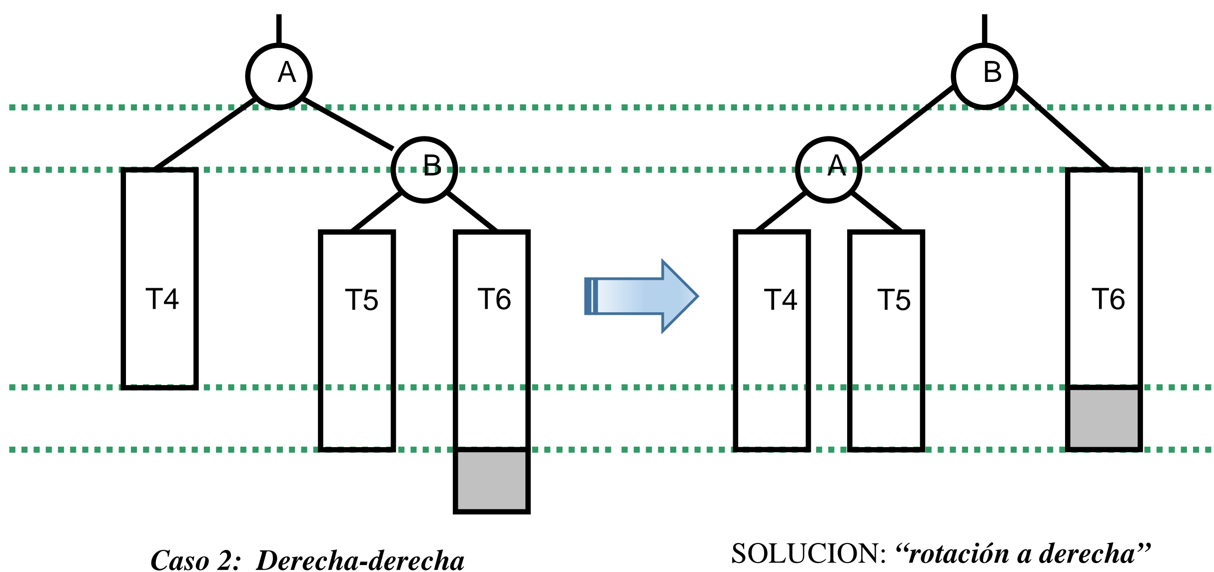
Caso	**Se insertó en el izquierdo**	**Se insertó en el derecho**
Era `pesadoDer`	Ahora es `equilibrado`	***Hay que reequilibrar***
Era `equilibrado`	Ahora es `pesadoIzq`	Ahora es `pesadoDer`
Era `pesadoIzq`	***Hay que reequilibrar***	Ahora es `equilibrado`

Pueden darse cuatro casos distintos de **desequilibrio** al realizar la inserción. Quedan reflejados en las figuras siguientes, junto con su **solución para re-equilibrar**. Las soluciones etiquetadas como **“rotación a izquierda”**, **“rotación a derecha”**, **“rotación doble derecha-izquierda”** y **“rotación doble izquierda-derecha”**, pueden implementarse muy fácilmente y con coste en $O(1)$. Se presentará alguna de ellas posteriormente como ejemplo.



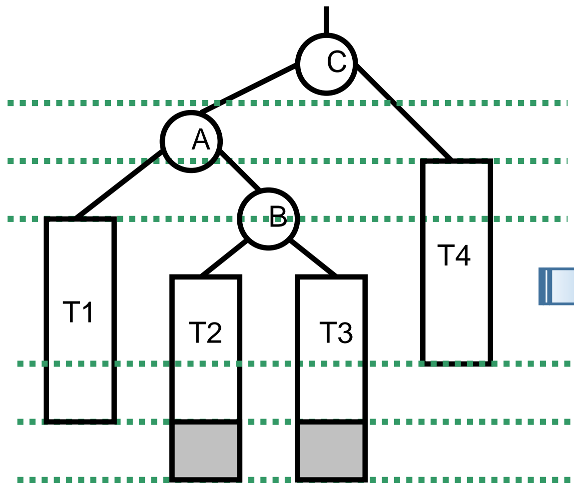
Caso 1: Izquierda-izquierda

SOLUCION:***“rotación a izquierda”***

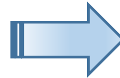


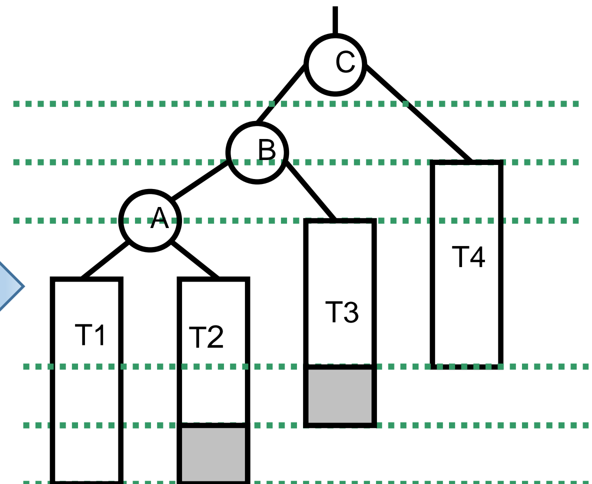
Caso 2: Derecha-derecha

SOLUCION: ***“rotación a derecha”***

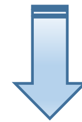


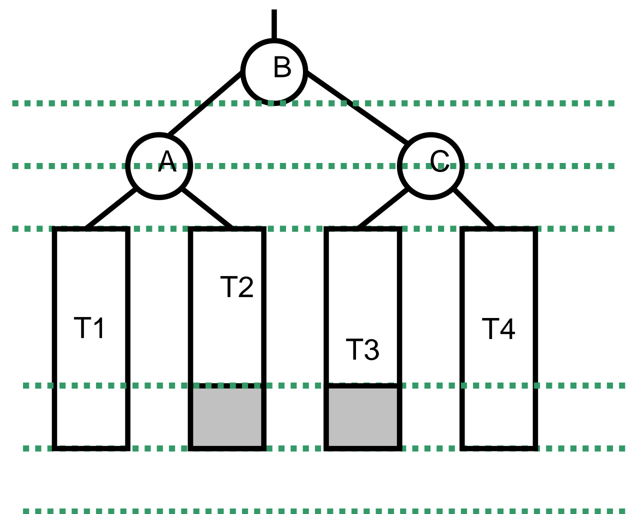
Caso 3: Izquierda-derecha



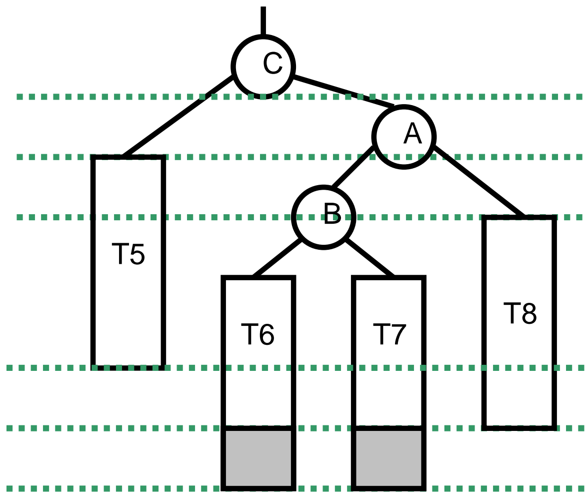


SOLUCION: ***"rotación doble derecha-izquierda"***
PASO 1: ***"rotación a derecha"***

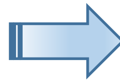


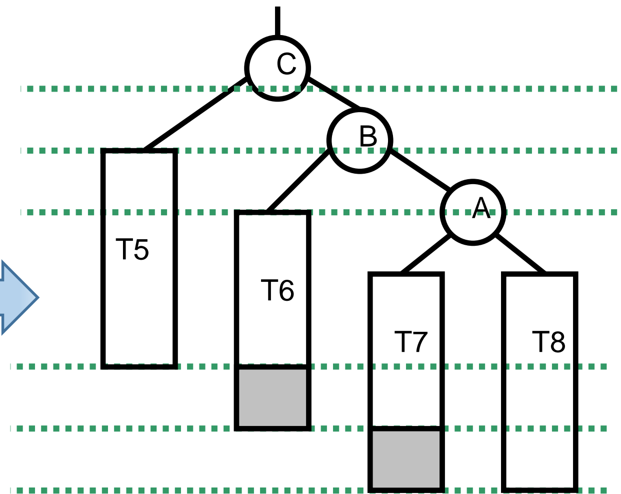


PASO 2: ***"rotación a izquierda"***

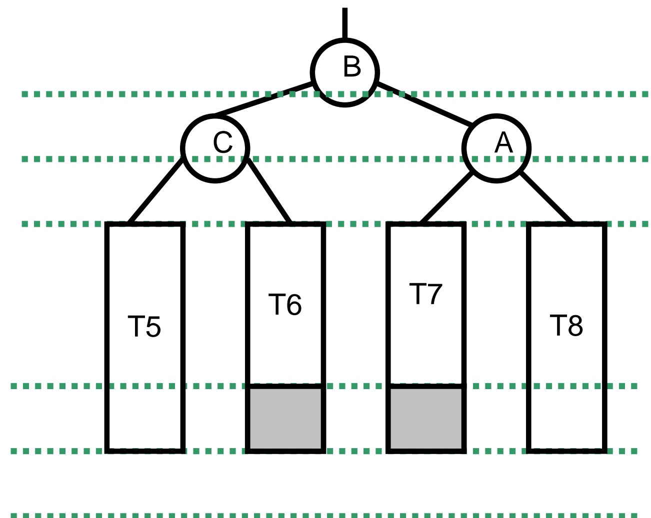


Caso 4: Derecha-izquierda





SOLUCION: ***"rotación doble izquierda-derecha"***
PASO 1: ***"rotación a izquierda"***



PASO 2: ***"rotación a derecha"***

Como ya se indicó, el nodo hoja insertado se establece como `equilibrado`. Una vez realizada la inserción, se regresa por el camino de búsqueda recalculando el factor de equilibrio de los nodos, hasta alcanzar la raíz o hasta encontrar un nodo que no verifique la condición de AVL (desequilibrado), y que requiera una re-estructuración de las cuatro descritas en las figuras anteriores para re-equilibrarlo.

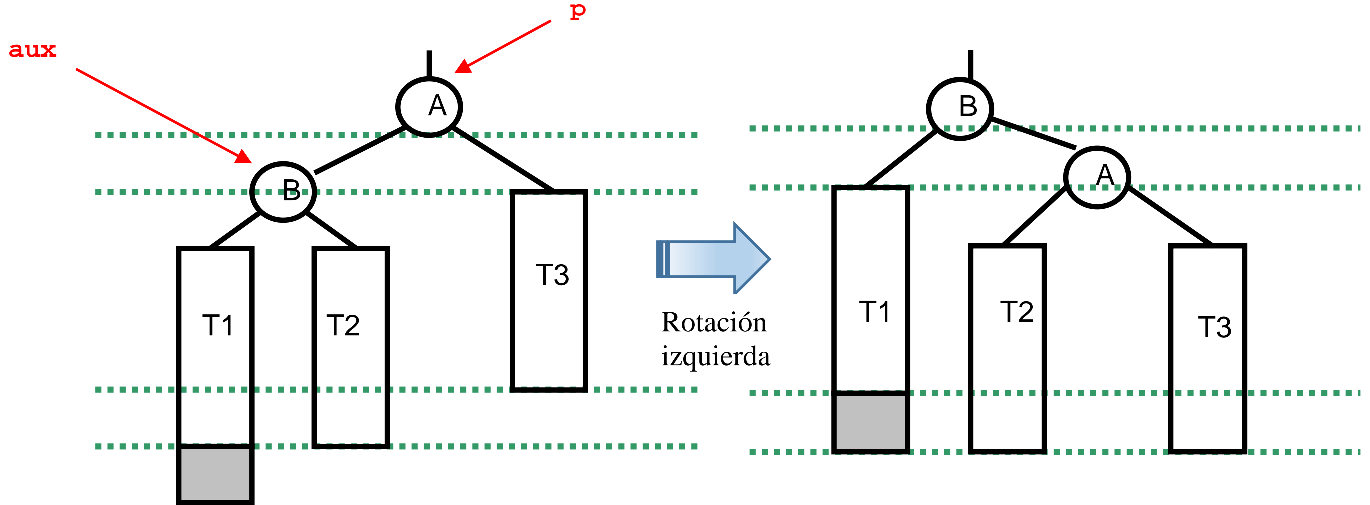
Una vez re-equilibrado el nodo, el árbol resultante queda con la misma altura que tenía el árbol antes de realizar la inserción (ver las figuras de los cuatro casos). Por lo tanto, si inicialmente el árbol estaba equilibrado, el resultante tras la inserción también estará equilibrado. Es decir, basta con un único reajuste u operación para re-equilibrar (una de las cuatro descritas en las figuras anteriores), para dejar todo el árbol resultante equilibrado, y no hará falta tras ella seguir regresando por el camino de búsqueda hasta la raíz.

Un código posible para la operación de inserción, aplicando las ideas anteriores, es el siguiente:

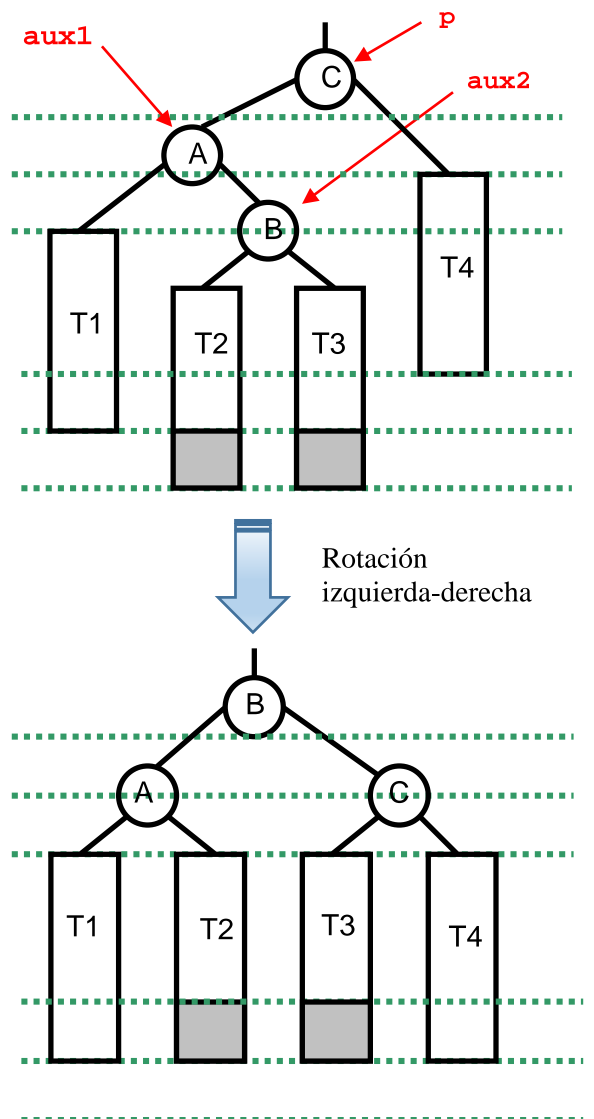
```
procedimiento añadirRec(e/s p:ptNodo; ent c:clave; ent v:valor;
                        e/s alturaModificada:booleano; sal nuevo:booleano)
principio
  si p=nil entonces
    nuevodato(p);
    p↑.laClave:=c;
    p↑.elValor:=v;
    p↑.equilibrio:=equilibrado;
    p↑.izq:=nil;
    p↑.der:=nil;
    alturaModificada:=verdad; {se ha modificado su altura}
    nuevo:=verdad             {se ha añadido una nueva clave y valor}
  sino_si c<p↑.laClave entonces
    añadirRec(p↑.izq,c,v,alturaModificada,nuevo);
    si alturaModificada entonces
      selección
        p↑.equilibrio=pesadoIzq:   si p↑.izq↑.equilibrio=pesadoIzq entonces
                                     rotaciónIzq(p)
                                   sino
                                     rotaciónIzqDer(p)
                                   fsi;
                                   alturaModificada:=falso;
        p↑.equilibrio=equilibrado: p↑.equilibrio:=pesadoIzq;
        p↑.equilibrio=pesadoDer:   p↑.equilibrio:=equilibrado;
                                   alturaModificada:=falso
      fselección
    fsi
  sino_si p↑.laClave<c entonces
    ...
    {caso simétrico al anterior; ejercicio}
    ...
  sino {la clave ya estaba, se actualiza el valor}
    p↑.elValor:=v;
    nuevo:=falso
  fsi
fin

procedimiento añadir(e/s d:diccionario; ent c:clave; ent v:valor)
variables alturaModificada,nuevo:booleano
principio
  alturaModificada:=falso;
  añadirRec(d.raíz,c,v,alturaModificada,nuevo);
  si nuevo entonces
    d.tamaño:=d.tamaño+1
  fsi
fin
```

El **coste en tiempo en el caso peor está en $O(\log n)$**, siendo n el número de claves del diccionario. En cuanto a las operaciones de "rotación", presentamos dos de ellas (las otras dos se implementan de forma simétrica).

```
procedimiento rotaciónIzq(e/s p:ptNodo)
variable aux:ptNodo
principio
  aux:=p↑.izq;
  p↑.izq:=aux↑.der;
  p↑.equilibrio:=equilibrado;
  aux↑.der:=p;
  p:=aux;
  p↑.equilibrio:=equilibrado
fin
```




```
procedimiento rotacionIzqDer(e/s p:ptNodo)
variables aux1,aux2:ptNodo
principio
  aux1:=p↑.izq;
  aux2:=p↑.izq↑.der;
  aux1↑.der:=aux2↑.izq;
  aux2↑.izq:=aux1;
  p↑.izq:=aux2;
  si aux2↑.equilibrio=pesadoIzq entonces
    aux1↑.equilibrio:=equilibrado;
    p↑.equilibrio:=pesadoDer
  sino_si aux2↑.equilibrio=equilibrado
                                    entonces
    aux1↑.equilibrio:=equilibrado;
    p↑.equilibrio:=equilibrado
  sino
    aux1↑.equilibrio:=pesadoIzq;
    p↑.equilibrio:=equilibrado
  fsi;
  p↑.izq:=aux2↑.der;
  aux2↑.der:=p;
  aux2↑.equilibrio:=equilibrado;
  p:=aux2
fin
```

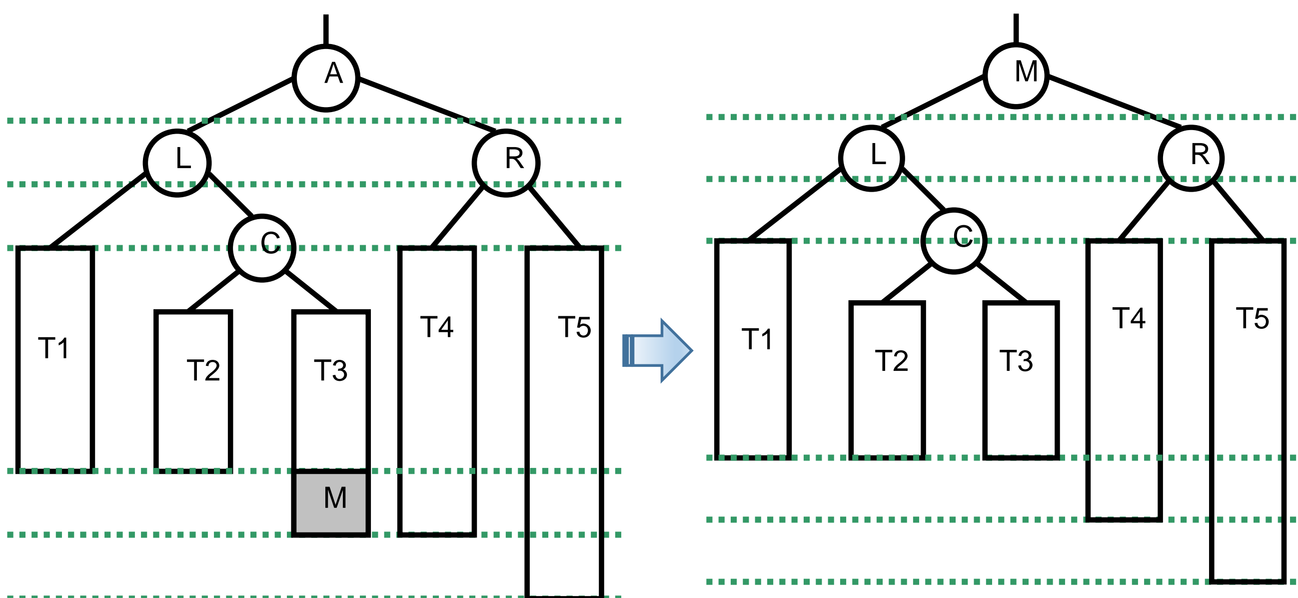


4. Borrado en AVL

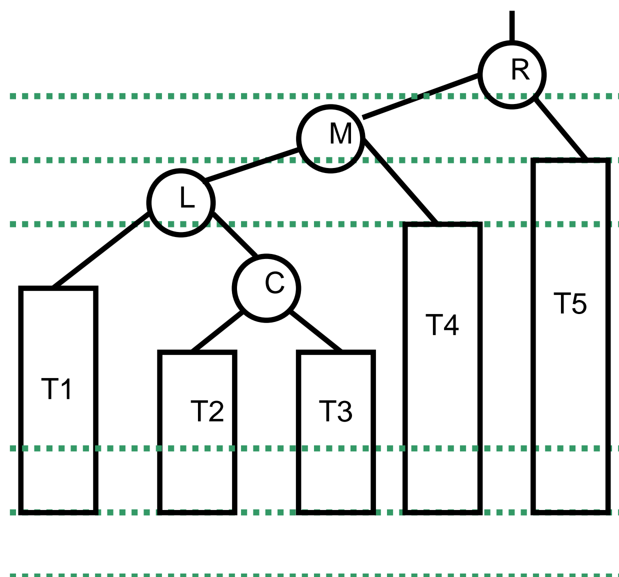
En cuanto al proceso de borrado, tiene una dificultad similar. Se compone de dos partes:

1. Proceso de borrado como en un árbol binario de búsqueda:
   a. Buscar el nodo a borrar.
   b. Si el nodo es hoja, se borra.
   c. Si no es hoja:
      - se sustituye por el máximo del subárbol izquierdo, y
      - se borra dicho máximo del subárbol izquierdo.
2. Se regresa por el camino de búsqueda calculando los nuevos factores de equilibrio:
   - Si en alguno de los nodos se pierde la condición de equilibrio, debe ser restaurada (re-equilibrado).
   - Debe continuarse hasta la raíz, porque pueden ser necesarios más re-equilibrados.

Ejemplo: Borrar el nodo *A* del siguiente AVL. La nueva raíz será *M* (máximo de su subárbol izquierdo).



El árbol resultante es pesado a la derecha. Con una solución similar a la de la inserción, se re-equilibra con una rotación a la derecha, quedando así:



En este caso, el árbol resultante ha perdido altura.

En un caso general, en el borrado, pueden ser necesarias varias operaciones de restauración del equilibrio, y hay que seguir comprobando hasta llegar a la raíz.

El algoritmo puede escribirse de la forma siguiente (incompleto):

```
procedimiento quitarRec(e/s p:ptNodo; ent c:clave;
                        e/s alturaModificada:booleano; sal borrado:booleano)
variable aux:ptNodo
principio
  si p=nil entonces
      borrado:=falso
  sino
    si c<p↑.laClave entonces
      quitarRec(p↑.izq,c,alturaModificada,borrado);
      si alturaModificada entonces
        equilIzq(p,alturaModificada)
      fsi
    sino_si p↑.laClave<c entonces
      borrarRec(p↑.der,c,alturaModificada,borrado);
      si alturaModificada entonces
        equilDer(p,alturaModificada);
      fsi
    sino {clave encontrada}
      si p↑.izq=nil entonces
        aux:=p;
        p:=p↑.der;
        disponer(aux);
        alturaModificada:=verdad
      sino_si p↑.der=nil entonces
        aux:=p;
        p:=p↑.izq;
        disponer(aux);
        alturaModificada:=verdad
      sino
        borrarMaxClave(p↑.izq,p↑.laClave,p↑.elValor,alturaModificada);
        si alturaModificada entonces
          equilIzq(p,alturaModificada);
        fsi
      fsi;
      borrado:=verdad
    fsi
  fsi
fin

procedimiento equilIzq(e/s p:ptNodo; e/s alturaModificada:booleano)
procedimiento equilDer(e/s p:ptNodo; e/s alturaModificada:booleano)
{contienen un estudio de casos similar al de la inserción}

procedimiento borrarMaxClave(e/s p:ptNodo; sal c:clave; sal v:valor;
                             e/s alturaModificada:boolean)
variable aux:ptNodo
principio
  si p↑.der=nil entonces
    c:=p↑.laClave;
    v:=p↑.elValor;
    aux:=p;
    p:=p↑.izq;
    disponer(aux);
    alturaModificada:=verdad
  sino
    borrarMaxClave(p↑.der,c,v,alturaModificada);
    si alturaModificada entonces
      equilDer(p,alturaModificada)
    fsi
  fsi
fin
```

```
procedimiento quitar(e/s d:diccionario; ent c:clave)
variable alturaModificada,borrado:booleano
principio
  alturaModificada:=falso;
  quitarRec(d.raíz,clave,alturaModificada,borrado);
  si borrado entonces
    d.tamaño:=d.tamaño-1
  fsi
fin
```

El **coste en tiempo en el caso peor está en $O(\log n)$**, siendo n el número de claves del diccionario.

Resumimos en la siguiente tabla los costes en espacio y tiempo, en el caso peor, de las implementaciones de diccionarios vistas hasta el momento.

Diccionario (con n claves comparables)	Vector tamaño MAX (ordenado) ($n \leq$ MAX)	Lista enlazada (ordenada)	Árbol binario de búsqueda sin equilibrar	Árbol binario de búsqueda equilibrado (AVL)
Coste en espacio	$O(\text{MAX})$	$O(n)$	$O(n)$	$O(n)$
Costes en tiempo (caso peor)				
añadir	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$
buscar	$O(\log n)$	$O(n)$	$O(n)$	$O(\log n)$
borrar	$O(n)$	$O(n)$	$O(n)$	$O(\log n)$

Lección 15

Árboles *n*–arios

Indice

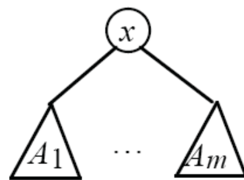
1. Conceptos y especificación
2. Recorridos
3. Implementación dinámica
4. Ideas sobre implementación estática

1. Conceptos y especificación

Como ya adelantamos en una lección anterior, un **árbol *n*–ario** (con $n \geq 1$) es un conjunto **no vacío** de elementos o **nodos** del **mismo tipo** tal que:

- existe un elemento destacado llamado **raíz** del árbol,
- el resto de los elementos se distribuyen en m (con $0 \leq m \leq n$) subconjuntos disjuntos, llamados **subárboles** del árbol original, cada uno de los cuales es a su vez un árbol *n*–ario.

Si en el conjunto de los subárboles de un árbol *n*–ario se supone definida una relación de orden total, el árbol se llama **ordenado**.



Árbol *n*–ario ordenado,
cada nodo tiene un número de
subárboles no superior a *n*,
numerados desde el 1 en adelante:
$A_1, \ldots, A_m$

Para poder facilitar la especificación de árboles *n*–arios, se define además el concepto de **bosque**. Un **bosque ordenado de grado *n*** (con $n \geq 1$) es una secuencia $A_1, \ldots, A_m$, con $0 \leq m \leq n$, de árboles *n*–arios ordenados. Si $m = 0$, el bosque se llama vacío.

Un árbol *n*–ario ordenado se **genera** a partir de un elemento y un bosque ordenado de grado *n*, sin más que considerar el elemento como raíz del árbol y el bosque como sus subárboles.

A continuación, vemos la especificación conjunta de los bosques ordenados y de los árboles ordenados. Los bosques se generan como secuencias de árboles a partir de un bosque vacío y una operación de añadir por la derecha un árbol a un bosque.

espec árbolesOrdenados
usa booleanos, naturales
parámetro formal
    género elemento
fpf

géneros bosque, árbol *{Los valores del género bosque representan secuencias de elementos del género árbol.
Los valores del género árbol se componen de una raíz de género elemento y un bosque de árboles hijos}*

operaciones

crearVacío: -> bosque

{Devuelve el bosque vacío, es decir, la secuencia vacía de árboles}

añadirÚltimo: bosque b, árbol a -> bosque

{Devuelve un bosque igual al resultante de añadir el árbol a como último elemento de b}

long: bosque b -> natural

{Devuelve el número de árboles del bosque b, es decir, la longitud de la secuencia de árboles}

parcial observar: bosque b, natural n -> árbol

{Devuelve el n-ésimo árbol de la secuencia de árboles b.
Parcial: sólo está definida si $1 \leq n \leq long(b)$}

parcial resto: bosque b -> bosque

{Devuelve un bosque igual al resultante de borrar el primer árbol de b.
Parcial: sólo está definida si $long(b) > 0$}

plantar: elemento e, bosque b -> árbol

{Devuelve un árbol cuya raíz es e y sus subárboles son un bosque igual que b}

raíz: árbol a -> elemento

{Devuelve la raíz del árbol a}

elBosque: árbol a -> bosque

{Devuelve un bosque igual al bosque de los subárboles del árbol a}

numHijos: árbol a -> natural

{Devuelve el número de subárboles de a, es decir long(elBosque(a))}

parcial subÁrbol: árbol a , natural n → árbol

{Devuelve un árbol igual al n-ésimo subárbol de a.
Parcial: sólo está definida si $1 \leq n \leq numHijos(a)$}

esHoja?: árbol a -> booleano

{Devuelve verdad si y sólo si a no tiene subárboles, es decir, $numHijos(a) = 0$}

alturaBosque: bosque b -> natural

{Si $long(b) > 0$, devuelve la altura del árbol más alto de b. Si $long(b) = 0$, devuelve 0}

alturaÁrbol: árbol a -> natural

{Devuelve la altura de a}

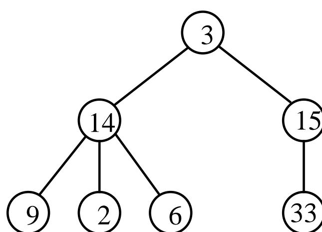
fespec

2. Recorridos

De forma análoga a lo visto para árboles binarios, se pueden especificar varias operaciones de **recorrido** de árboles ordenados. Todas ellas consisten en visitar todos los elementos del árbol una sola vez. En el recorrido **pre-orden** se visita en primer lugar la raíz del árbol y después se recorren en pre-orden todos los subárboles de izquierda a derecha. En el recorrido **post-orden** se recorren en post-orden los subárboles de izquierda a derecha y por último se visita la raíz. Por

último, el recorrido **en anchura** de un árbol consiste en visitar todos los elementos del árbol una sola vez, de la siguiente forma: primero se visita el elemento del nivel 0 (la raíz), luego los elementos del nivel 1, y así sucesivamente; en cada nivel, sus elementos se visitan de izquierda a derecha.

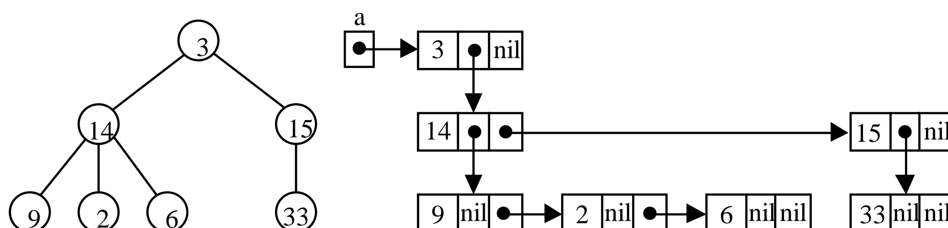
Por ejemplo, para el árbol 3–ario de la figura siguiente,



el recorrido en pre-orden es la lista 3-14-9-2-6-15-33, el recorrido es post-orden es 9-2-6-14-33-15-3, y el recorrido en anchura es 3-14-15-9-2-6-33.

3. Implementación dinámica

Para implementar dinámicamente árboles *n*–arios ordenados, utilizaremos la representación llamada **primogénito–siguiente hermano** (*child–brother*, en inglés). El tipo árbol es un puntero a un registro que guarda, además de un campo con la información correspondiente al elemento raíz, un puntero al registro correspondiente al primer hijo (si la raíz no es una hoja) y otro puntero que apunta al registro correspondiente al siguiente hermano (si existe) (ver la figura).




```
tipos árbol = ↑nodo;
      nodo = registro
                 dato:elemento;
                 primogénito,sigHermano:árbol
             freg;
      bosque = árbol
```

En esta representación, la lista enlazada que se forma considerando los punteros al siguiente hermano (`sigHermano`) representa un bosque ordenado (secuencia de árboles).

A continuación, se desarrolla la implementación de todas las operaciones básicas definidas en la especificación genérica árbolesOrdenados, además de los recorridos en pre-orden y post-orden.

```
modulo genérico árbolesOrdenados
importa listasGenéricas {como las utilizadas en la Lección 12}
parámetro
  tipo elemento

exporta
  tipos árbol, bosque {Los valores del tipo bosque representan secuencias de elementos
                       del tipo árbol. Los valores del tipo árbol se componen de una
                       raíz de tipo elmto y un bosque de árboles hijos}

  procedimiento crearVacío(sal b:bosque)
  {devuelve el bosque vacío, es decir, la secuencia vacía de árboles}

  procedimiento añadirÚltimo(e/s b:bosque; ent a:árbol)
  {devuelve el bosque resultante de añadir el árbol a como último elemento de b}
```

```
función long(b:bosque) devuelve natural
{devuelve el número de árboles del bosque b, es decir, la longitud de la secuencia}

procedimiento observar(ent b:bosque; ent n:natural; sal error:booleano; sal a:árbol)
{si 1≤n≤long(b): devuelve el n-ésimo árbol de la secuencia de árboles b y error=falso;
 en caso contrario devuelve error=verdad}

procedimiento resto(ent b:bosque; sal error:booleano; sal rb:bosque)
{si b no es vacío devuelve en rb el bosque resultante de borrar el primer árbol de b
 y error=falso; en caso contrario devuelve error=verdad}

procedimiento plantar(sal a:árbol; ent e:elemento; ent b:bosque)
{devuelve un árbol a cuya raíz es e y sus subárboles son el bosque b}

función raíz(a:árbol) devuelve elemento
{devuelve la raíz del árbol a}

procedimiento elBosque(ent a:árbol; sal b:bosque)
{devuelve el bosque de los subárboles del árbol a}

función numHijos(a:árbol) devuelve natural
{devuelve el número de subárboles de a, es decir long(elBosque(a))}

procedimiento subÁrbol(ent a:árbol; ent n:natural; sal error:booleano; sal sa:árbol)
{si 1≤n≤numHijos(a): devuelve el n-ésimo subárbol de a y error=falso;
 en caso contrario error=verdad}

función esHoja(a:árbol) devuelve booleano
{devuelve verdad si y sólo si a no tiene subárboles, es decir, numHijos(a)=0}

función alturaBosque(b:bosque) devuelve natural
{devuelve la altura del árbol más alto de b}

función alturaÁrbol(a:árbol) devuelve natural
{devuelve la altura de a}

procedimiento duplicarBosque(sal nuevo:bosque; ent viejo:bosque)
{Duplica la representación del bosque viejo guardándolo en nuevo}

procedimiento liberarBosque(e/s b:bosque)
{Libera la memoria dinámica accesible desde b, quedando b vacío}

procedimiento duplicarÁrbol(sal nuevo:árbol; ent viejo:árbol)
{Duplica la representación del árbol viejo guardándolo en nuevo}

procedimiento liberarÁrbol(e/s a:árbol)
{Libera la memoria dinámica accesible desde a}

procedimiento preOrden(ent a:árbol; e/s L:lista)
{añade a la lista L los elementos de a recorridos en pre-orden}

procedimiento preBosque(ent b:bosque; e/s L:lista)
{añade a la lista L los elementos de todos los árboles del bosque b recorridos en
 pre-orden}

procedimiento postOrden(ent a:árbol; e/s L:lista)
{añade a la lista L los elementos de a recorridos en post-orden}

procedimiento postBosque(ent b:bosque; e/s L:lista)
{añade a la lista l los elementos de todos los árboles del bosque b recorridos
 en post-orden}
```

```
implementación
  tipos
    árbol = ↑nodo;
    nodo = registro
             dato:elemento;
             primogénito,sigHermano:árbol
           freg;
    bosque = árbol  {Suponemos coerción de tipos automática}

  procedimiento crearVacío(sal b:bosque)
  {devuelve el bosque vacío, es decir, la secuencia vacía de árboles}
  principio
    b:=nil
  fin

  procedimiento añadirÚltimo(e/s b:bosque; ent a:árbol)
  {devuelve el bosque resultante de añadir el árbol a como último elemento de b}
  variable aux:bosque
  principio
    si b=nil entonces
      b:=a
    sino
      aux:=b;
      mientrasQue aux↑.sigHermano≠nil hacer
        aux:=aux↑.sigHermano
      fmq;
      aux↑.sigHermano:=a
    fsi
  fin

  función long(b:bosque) devuelve natural
  {devuelve el número de árboles del bosque b, es decir, la longitud de la secuencia}
  principio
    si b=nil entonces
      devuelve 0
    sino
      devuelve 1+long(b↑.sigHermano)
    fsi
  fin

  procedimiento observar(ent b:bosque; ent n:natural; sal error:booleano; sal a:árbol)
  {si 1≤n≤long(b): devuelve el n-ésimo árbol de la secuencia de árboles b y error=falso;
   en caso contrario devuelve error=verdad}
  principio
    si n=0 or b=nil entonces
      error:=verdad
    sino
      si n=1 entonces
        a:=b;
        error:=falso
      sino
        observar(b↑.sigHermano,n-1,error,a)
      fsi
    fsi
  fin

  procedimiento resto(ent b:bosque; sal error:booleano; sal rb:bosque)
  {si b no es vacío devuelve en rb el bosque resultante de borrar el primer árbol de b
   y error=falso; en caso contrario devuelve error=verdad}
  principio
    si b=nil entonces
      error:=verdad
    sino
      error:=falso;
      rb:=b↑.sigHermano
    fsi
  fin
```

```
procedimiento plantar(sal a:árbol; ent e:elemento; ent b:bosque)
{devuelve un árbol a cuya raíz es e y sus subárboles son el bosque b}
principio
  nuevoDato(a);
  a↑.dato:=e;
  a↑.primogénito:=b;
  a↑.sigHermano:=nil
fin

función raíz(a:árbol) devuelve elemento
{devuelve la raíz del árbol a}
principio
  devuelve a↑.dato
fin

procedimiento elBosque(ent a:árbol; sal b:bosque)
{devuelve el bosque de los subárboles del árbol a}
principio
  b:=a↑.primogénito
fin

procedimiento subárbol(ent a:árbol; ent n:natural; sal error:booleano; sal sa:árbol)
{si 1≤n≤numHijos(a): devuelve el n-ésimo subárbol de a y error=falso;
 en caso contrario error=verdad}
principio
  observar(a↑.primogénito,n,error,sa)
fin

función numHijos(a:árbol) devuelve natural
{devuelve el número de subárboles de a, es decir long(elBosque(a))}
principio
  devuelve long(a↑.primogénito)
fin

función esHoja(a:árbol) devuelve booleano
{devuelve verdad si y sólo si a no tiene subárboles, es decir, numHijos(a)=0}
principio
  devuelve a↑.primogénito=nil
fin

función alturaBosque(b:bosque) devuelve natural
{devuelve la altura del árbol más alto de b}
principio
  si b=nil entonces
    devuelve 0
  sino
    devuelve max(alturaÁrbol(b),alturaBosque(b↑.sigHermano))
  fsi
fin

función alturaÁrbol(a:árbol) devuelve natural
{devuelve la altura de a}
principio
  si esHoja(a) entonces
    devuelve 0
  sino
    devuelve 1+alturaBosque(a↑.primogénito)
  fsi
fin

procedimiento duplicarBosque(sal nuevo:bosque; ent viejo:bosque)
{Duplica la representación del bosque viejo guardándolo en nuevo}
variables error:booleano
principio
  si viejo=nil entonces
    nuevo:=nil
  sino
```

```
      duplicarÁrbol(nuevo,viejo);
      duplicarBosque(nuevo↑.sigHermano,viejo↑.sigHermano)
    fsi
  fin

  procedimiento duplicarÁrbol(sal nuevo:árbol; ent viejo:árbol)
  {Duplica la representación del árbol viejo guardándolo en nuevo}
  variables nuevoBosque:bosque
  principio
    duplicarBosque(nuevoBosque,viejo↑.primogénito);
    plantar(viejo↑.dato,nuevoBosque,nuevo)
  fin

  procedimiento liberarBosque(e/s b:bosque)
  {Libera la memoria dinámica accesible desde b, quedando b vacío}
  principio
    si b≠nil entonces
      liberarBosque(b↑.sigHermano);
      liberarÁrbol(b)
    fsi
  fin

  procedimiento liberarÁrbol(e/s a:árbol)
  {Libera la memoria dinámica accesible desde a}
  principio
    liberarBosque(a↑.primogénito);
    disponer(a)
  fin

  procedimiento preOrden(ent a:árbol; e/s L:lista)
  {añade a la lista L los elementos del árbol a recorridos en pre-orden}
  principio
    añadirÚltimo(L,a↑.dato);     {añadir la raíz del árbol}
    preBosque(a↑.primogénito,L) {recorrer el bosque de hijos de la raíz}
  fin

  procedimiento preBosque(ent b:bosque; e/s L:lista)
  {añade a la lista L los elementos de todos los árboles del bosque b recorridos en
   pre-orden}
  principio
    si b≠nil entonces
      preOrden(b,L);              {recorrer en pre-orden el primer árbol del bosque}
      preBosque(b↑.sigHermano,L) {recorrer el resto del bosque en pre-orden}
    fsi
  fin

  procedimiento postOrden(ent a:árbol; e/s L:lista)
  {añade a la lista L los elementos de a recorridos en post-orden}
  principio
    postBosque(a↑.primogénito,L); {recorrer el bosque de hijos de la raíz }
    añadirÚltimo(L,a↑.dato)       {añadir la raíz del árbol}
  fin

  procedimiento postBosque(ent b:bosque; e/s L:lista)
  {añade a la lista L los elementos de todos los árboles del bosque b recorridos
   en post-orden}
  principio
    si b≠nil entonces
      postOrden(b,L);              {recorrer en post-orden el primer árbol del bosque}
      postBosque(b↑.sigHermano,L) {recorrer el resto del bosque en post-orden}
    fsi
  fin

fin  {del módulo árbolesOrdenados}
```

La implementación anterior diferencia explícitamente los tipos de datos “árbol” y “bosque” aunque realmente ambos son punteros a datos del mismo tipo “nodo”. Se ha escrito así para remarcar la diferencia conceptual entre un árbol y una secuencia de árboles. No obstante, puede verse que en la implementación de las operaciones se utiliza **la coerción automática de tipos**, invocando por ejemplo desde el procedimiento “preBosque” al procedimiento “preOrden” con un argumento de tipo “bosque” cuando el parámetro formal en el procedimiento “preOrden” es de tipo “árbol”.

Caben implementaciones alternativas a los procedimientos de recorridos de árboles y bosques como las siguientes.

```
procedimiento preOrden(ent b:bosque; e/s L:lista)
{añade a la lista L los elementos del bosque n-ario b recorridos en pre-orden}
principio
  si b≠nil entonces
    añadirÚltimo(L,b↑.dato);
    preOrden(b↑.primogénito,L);
    preOrden(b↑.sigHermano,L)
  fsi
fin

procedimiento postOrden(ent b:bosque; e/s L:lista)
{añade a la lista L los elementos del bosque n-ario b recorridos en post-orden}
principio
  si b≠nil entonces
    postOrden(b↑.primogénito,L);
    añadirÚltimo(L,b↑.dato);
    postOrden(b↑.sigHermano,L)
  fsi
fin
```

En ambos casos el parámetro de tipo “bosque” podría ser igualmente de tipo “árbol”, pues sirven para recorrer ambas estructuras.

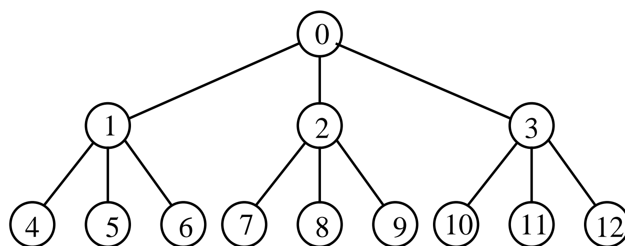
4. Ideas sobre implementación estática

En la **representación calculada** o estática de árboles n–arios ordenados existe una correspondencia biunívoca entre cada elemento potencial del árbol y cada componente del vector soporte.

```
constante max = ... {máximo número de elementos almacenables}
tipo árbol = vector[0..max-1] de elemento
```

Si se considera un árbol de grado n, se puede asociar un índice a los elementos potenciales del árbol de la siguiente forma:

- índice(a) = 0, si a es la raíz del árbol,
- índice(a)= n*índice(padre(a))+k, si a es el hijo k–ésimo de padre(a).



Como el número máximo de elementos en el nivel k es n^k, el número de componentes necesarias para almacenar un árbol n–ario arbitrario de altura h es $\sum_{k=0}^{h} n^k = \frac{n^{h+1}-1}{n-1}$.

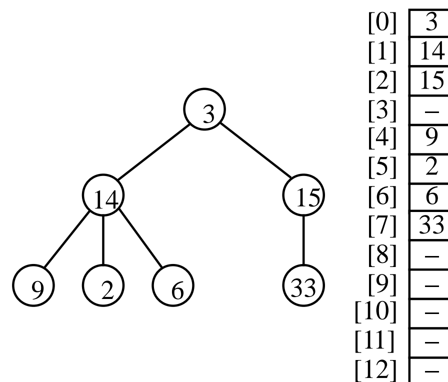
Conocido el índice que le corresponde a un elemento *e* en el vector, es fácil calcular el índice que le corresponde a su padre, hermanos o hijos:

elemento	índice	condición
e	i	
hijo k–ésimo de e	$n*i + k$	si $n*i+k <$ max
padre de e	$(i\text{-}1)$ div n	si $i \neq 0$
hermano siguiente a e	$i+1$	si i mod $n \neq 0$
nº de orden entre sus hermanos	$((i\text{-}1)$ mod $n) + 1$	si $i \neq 0$

Un árbol n–ario se dice **homogéneo** si todos sus subárboles excepto las hojas tienen n hijos. Un árbol homogéneo es **completo** cuando todas sus hojas tienen la misma profundidad. Dicho de otra forma, un árbol n–ario de altura h es completo si y sólo si tiene $\sum_{k=0}^{h} n^k = \frac{n^{h+1}-1}{n-1}$ elementos. Un árbol se dice **casi-completo** cuando se puede obtener a partir de un árbol completo eliminando hojas consecutivas del último nivel, comenzando por la que está más a la derecha.

En el caso de un árbol completo, la representación calculada introducida anteriormente es elegante y sencilla, y no se malgasta nada de memoria. Si el árbol es casi-completo, se desaprovechan algunas componentes del vector, pero todas ellas situadas al final; por tanto, basta con almacenar en una variable entera auxiliar el índice de la última componente que contiene un elemento del árbol.

En el caso general (para árboles que no sean completos ni casi-completos), la representación calculada puede desaprovechar gran parte de las componentes del vector. Además, sería preciso añadir en cada componente un campo booleano que almacene si esa componente guarda un elemento del vector o no.



La implementación de las operaciones con árboles ordenados, supuesta la representación calculada, se plantea como ejercicio.

Lección 16

Árboles *n*–arios de búsqueda

Indice

1. Definición
2. Árboles *n*–arios de búsqueda equilibrados
3. Ejemplo de implementación (árboles 2–3)

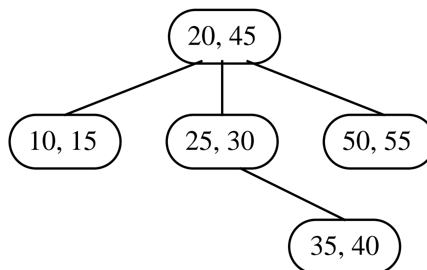
1. Definición

La definición de árbol de búsqueda dada para árboles binarios en una lección anterior, puede generalizarse al caso de árboles *n*–arios ordenados, dando lugar a los **árboles *n*–arios de búsqueda**.

Los árboles *n*–arios de búsqueda son árboles de grado *n* definidos en la forma siguiente:

- si A es vacío entonces A es un árbol *n*–ario de búsqueda;
- si A es no vacío entonces A es un árbol *n*–ario de búsqueda si verifica:
  - A tiene m subárboles *n*–arios de búsqueda $A_1, \ldots, A_m$, con $1 \leq m \leq n$;
  - la raíz de A contiene la siguiente información: $(m, e_1, e_2, \ldots, e_{m-1})$, de forma que $e_i < e_{i+1}$, para $1 \leq i < m - 1$;
  - todo elemento e perteneciente al subárbol A_i es tal que $e < e_i$, para $1 \leq i \leq m - 1$;
  - todo elemento e perteneciente al subárbol A_m es tal que $e > e_{m-1}$.

Como ejemplo de árbol 3–ario de búsqueda, puede verse el de la siguiente figura.



Si se trata de buscar un elemento e en este árbol, debe mirarse en primer lugar en el interior de la raíz. En el caso $e = e_i$, para algún i entre 1 y $m - 1$, la búsqueda termina con éxito. Si no, determinar si $e < e_1$, o $e > e_{m-1}$, o hallar el valor de i (entre 1 y $m - 2$) para el que $e_i < e < e_{i+1}$. En el caso $e < e_1$, debe buscarse en el subárbol A_1. En el caso $e_i < e < e_{i+1}$, para algún i entre 1 y $m - 2$, debe buscarse en el subárbol A_{i+1}. Finalmente, en el caso en que $e > e_{m-1}$, debe buscarse en A_m.

Si el valor de m es grande, la búsqueda entre los elementos $e_1, e_2, \ldots, e_{m-1}$ debe realizarse como una búsqueda dicotómica. En cambio, para valores pequeños de m una búsqueda secuencial puede ser más apropiada.

En el ejemplo, si el dato buscado es $e = 35$, una búsqueda en el nodo raíz indica que el subárbol apropiado es A_2 (el 2º subárbol del nodo raíz). Una búsqueda en la raíz de A_2 indica que el siguiente árbol a explorar es su tercer (y único) subárbol. El elemento e se encuentra entonces y la búsqueda termina con éxito.

Los árboles *n*-arios de búsqueda se llaman también **árboles de búsqueda múltiple o multicamino** (aquéllos en los que cada nodo puede tener más de dos hijos) cuyas ramas están ordenadas "a modo de" un árbol binario de búsqueda.

En un árbol multicamino de grado n, cada nodo interno puede tener como máximo n nodos descendientes, y puede almacenar como máximo $n - 1$ claves ordenadas.

Los árboles *n*-arios de búsqueda son especialmente útiles cuando se utilizan en búsquedas externas (los datos almacenados en disco, etc., no en memoria), permitiendo reducir así el número de accesos a disco (1 acceso por nodo consultado o nivel del árbol).

2. Árboles *n*–arios de búsqueda equilibrados

Al igual que ocurre con los árboles binarios de búsqueda, el **coste de la operación de búsqueda es del orden de la altura del árbol**. Por tanto, es conveniente que el árbol *n*–ario de búsqueda sea "lo más **equilibrado** posible" (como los árboles binarios AVL).

Un tipo particular de árboles *n*–arios de búsqueda equilibrados son los árboles B. Un **árbol B de orden *n*** tiene las siguientes propiedades[1]**:**

- es un árbol *n*–ario de búsqueda;
- la raíz es una hoja o tiene al menos 2 hijos;
- cada nodo, excepto la raíz y las hojas, tiene al menos $\lceil n/2 \rceil$ hijos;
- todas las hojas están en un mismo nivel.

En la siguiente figura puede verse un árbol B de orden 5.

Si un árbol B tiene *M* hojas y las hojas están en el nivel *L*, se tiene que el número de nodos en los niveles 1, 2, 3... es por lo menos 2, $2\lceil n/2 \rceil$, $2\lceil n/2 \rceil^2$, ..., y por tanto: $M \geq 2\lceil n/2 \rceil^{L-1}$, es decir: $L \leq 1 + \log_{\lceil n/2 \rceil}(M / 2)$. **Es decir, la altura está acotada por el logaritmo del número de claves**.

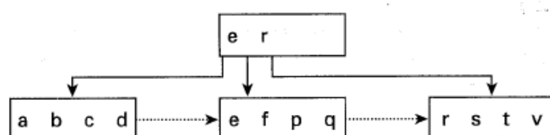
La implementación de las operaciones de búsqueda, inserción y borrado en este tipo de árboles puede encontrarse, por ejemplo, en el capítulo 10 libro *Estructura de datos. Libro de problemas* de L. Joyanes, I. Zahonero, M. Fernández y L. Sánchez (editorial McGraw Hill, 1999), o en el libro *Data Structures and Algorithm Analysis in C++, 4th Edition* de M.A. Weiss (editorial Pearson/Addison Wesley, 2014).

Los árboles B de orden 3 se denominan también **árboles 2–3**. En un árbol 2–3, cada nodo, excepto las hojas, tiene dos o tres hijos. Un estudio detallado sobre este tipo de árboles puede encontrarse, por ejemplo, en el libro *Estructuras de Datos y Algoritmos* de A. V. Aho, J. E. Hopcroft y J. D. Ullman (páginas 171 a 182). Se presenta una implementación en la siguiente sección.

Otra clase de árboles *n*–arios de búsqueda estudiada en la literatura es la de los **árboles B***. La raíz de un árbol B* de orden *n* tiene como mínimo 2 y como máximo $2\lfloor (2n\text{-}2)/3 \rfloor + 1$ hijos. Cada nodo de un árbol B* de orden *n*, excepto la raíz y las hojas, tiene como mínimo $\lceil (2n\text{-}1)/3 \rceil$ hijos. Todas las hojas de un árbol B* de orden *n* están en un mismo nivel.

Los **árboles B*** constituyen una mejora de los árboles B para aumentar el promedio de utilización de los nodos. La inserción de claves en el árbol B* supone que si el nodo que le corresponde está lleno, se mueven las claves a uno de sus hermanos, y así se pospone la división del nodo hasta que ambos hermanos están completamente llenos. Cuando finalmente se dividan, esos dos hermanos pasarán a ser 3, y cada uno estará lleno en 2/3 partes de su capacidad.

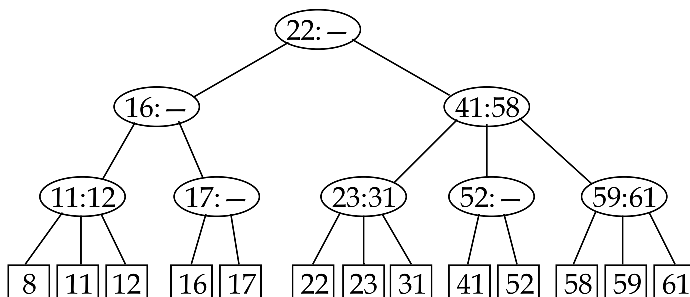
Los **árboles B+** son otra mejora de los árboles B. Mantienen la propiedad de acceso rápido en búsquedas para claves cualquiera y, además, permiten un recorrido secuencial rápido. En un árbol B+ todas las claves se encuentran en las hojas, duplicándose en la raíz y en los nodos interiores aquellas que definen los caminos de búsqueda. Las claves en el nodo raíz e interiores se utilizan únicamente como índices. Para facilitar el recorrido secuencial rápido, las hojas están encadenadas (enhebradas). Véase la figura.



[1] Introducidos por R. Bayer y E. M. Mc Creight en "Organization and maintenance of large ordered indices", *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

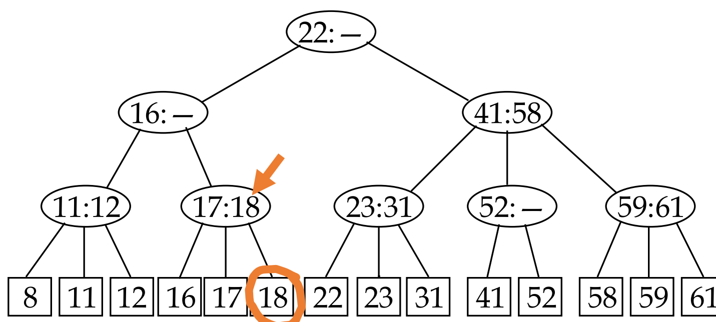
3. Ejemplo de implementación (árboles 2–3)

Presentamos, a modo de ejemplo, los detalles de implementación de una de las múltiples clases de árboles *n*–arios de búsqueda equilibrados definidos en la literatura. Se trata de árboles B de grado 3, pero "al estilo de" los árboles B+. Es decir, suponiendo que las claves se guardan únicamente en las hojas. En los nodos internos se guardan las copias de las claves necesarias para hacer la búsqueda.

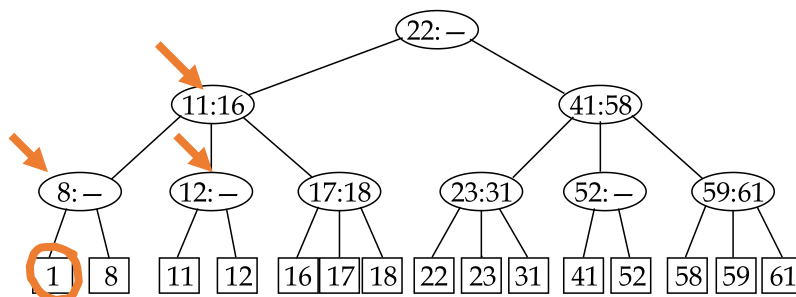


Ejemplo de árbol 2–3

Supongamos que en el árbol anterior se quiere insertar la clave 18. El resultado sería el siguiente.

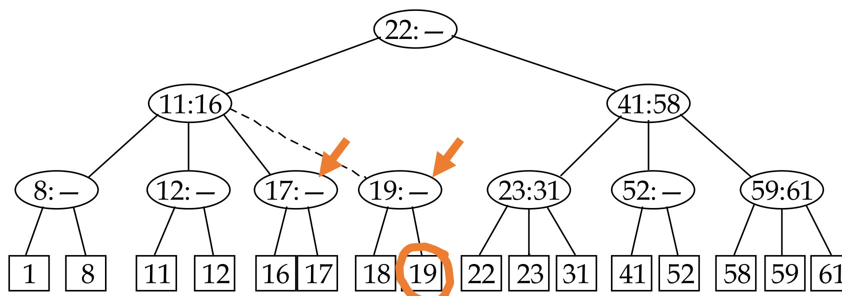


Si a continuación se desea insertar la clave 1, resultaría lo siguiente.

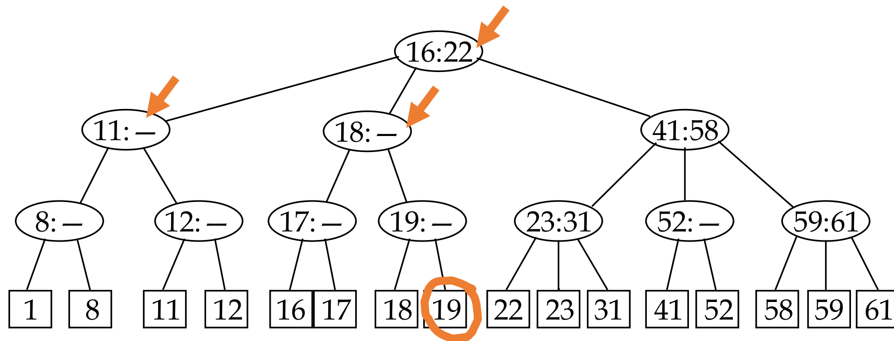


Como puede verse, ha sido necesario partir el nodo "11:12" en dos ("8:–" y "12:–"), y repartir entre los dos nodos resultantes los hijos del nodo anterior tras insertar la nueva clave. Y además modificar el nodo padre del "11:12".

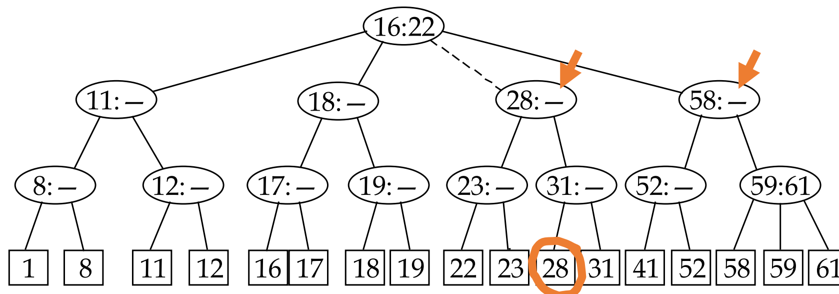
Si en el árbol resultante insertamos la clave 19, se necesita partir en dos el nodo "17:18", pero en ese momento su nodo padre resulta tener 4 hijos, lo que no está permitido.



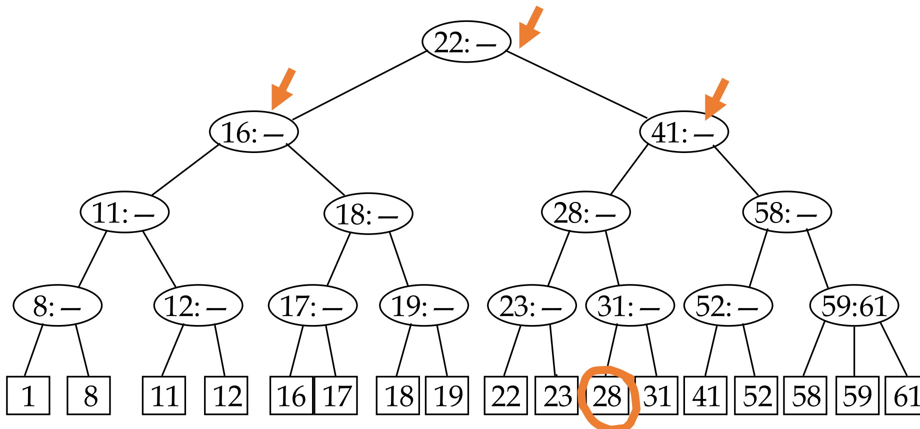
Por tanto, se hace necesario también partir el nodo “11:16” en dos (afectando a la raíz), resultando el árbol siguiente.



Si, para terminar, se precisa insertar la clave 28, se hace necesario partir en dos el nodo “23:31”, y también su padre, el “41:58”, porque pasaría a tener 4 hijos.



Pero entonces el nodo raíz del árbol, “16:22”, quedaría con 4 hijos, y como no es posible, se hace necesario partir en dos el nodo raíz y colocar como padre de ambos un nuevo nodo raíz, con el siguiente resultado.



Para implementarlo, declaramos la estructura de datos necesaria:

```
tipos
  tipoElmto = registro
                 clave:tipoClave;
                 ... {el resto de campos necesarios (valor asociado a la clave)}
               freg;
  tipoNodo = (hoja,interior);
  diccionario = ↑nodoDosTres;
  nodoDosTres =
    registro
      clase:tipoNodo;
      {el siguiente campo sólo se usa si clase=hoja}
      elmto:tipoElmto;
      {los siguientes campos sólo se usan si clase=interior}
      primerHijo,segundoHijo,tercerHijo:diccionario;
      menorDeSegundo,menorDeTercero:tipoClave
    freg
```

El algoritmo de inserción (o modificación, si la clave ya estaba en el diccionario) sería el siguiente.

```
procedimiento insertar(ent x:tipoElmto; e/s d:diccionario)
variables
  ptAtrás:diccionario; {puntero al nuevo nodo devuelto por insertaRec}
  menorAtrás:tipoClave; {valor mínimo en el subárbol de ptAtrás}
  guardaD:diccionario; {para almacenar una copia temporal de d}
  ptNuevo:diccionario
principio
  {mirar si d está vacío o si tiene un solo nodo y tratar esos casos especiales}
  si d=nil entonces {d era vacío; se crea con una sola clave}
    nuevoDato(d);
    d↑.clase:=hoja;
    d↑.elmto:=x;
  sino_si d↑.clase=hoja entonces {d no era vacío pero era sólo una hoja}
    si d↑elmto.clave=x.clave entonces {la clave es igual que la que había}
      d↑elmto:=x  {se actualiza el resto de la información asociada a la clave}
    sino {hay que crear una nueva hoja para la nueva clave, usamos ptNuevo}
      nuevoDato(ptNuevo);
      ptNuevo↑.clase:=hoja;
      ptNuevo↑.elmto:=x;
      guardaD:=d; {copia de la antigua raíz, que era una hoja}
      nuevoDato(d); {será la nueva raíz}
      d↑.clase:=interior;
      d↑.tercerHijo:=nil; {d sólo tendrá 2 hijos, ptNuevo y guardaD}
      si x.clave<guardaD↑.elmto.clave entonces {la nueva es menor que la que había}
        d↑.primerHijo:=ptNuevo;
        d↑.segundoHijo:=guardaD;
        d↑.menorDeSegundo:=guardaD↑elmto.clave
      sino {la nueva es mayor que la que había}
        d↑.primerHijo:=guardaD;
        d↑.segundoHijo:=ptNuevo;
        d↑.menorDeSegundo:=x.clave {es igual a ptNuevo↑.elmto.clave}
      fsi
    fsi
  sino {d tenía al menos dos claves (inserción con un algoritmo recursivo)}
    insertarRec(d,x,ptAtrás,menorAtrás);
    si ptAtrás≠nil entonces
      {crea la raíz nueva; sus hijos están ahora apuntados por d y ptAtrás}
      guardaD:=d;
      nuevoDato(d);
      d↑.clase:=interior;
      d↑.primerHijo:=guardaD;
      d↑.segundoHijo:=ptAtrás;
      d↑.menorDeSegundo:=menorAtrás;
      d↑.tercerHijo:=nil
    fsi
  fsi
fin
```

Una primera descripción del algoritmo de inserción recursivo utilizado por el anterior es la siguiente.

```
procedimiento insertarRec(e/s nodo:diccionario;
                          ent x:tipoElmto;  {x se insertará en el subárbol de nodo}
                          sal ptNuevo:diccionario;
                              {puntero al nodo recién creado a la derecha de nodo}
                          sal menor:tipoClave)
                              {elemento más pequeño del subárbol al que apunta ptNuevo}
principio
  ptNuevo:=nil;
  si nodo es una hoja entonces
    si x no es el elemento que está en nodo entonces
      crea un nodo nuevo apuntado por ptNuevo;
      pone x en el nodo nuevo;
      menor:=x.clave
    fsi
  sino {nodo es un nodo interno}
```

```
    sea w el hijo de nodo a cuyo subárbol pertenece x;
    insertarRec(w,x,ptAtrás,menorAtrás);
    si ptAtrás≠nil entonces
      insertar el puntero ptAtrás entre los hijos de nodo justo a la derecha de w;
      si nodo tiene cuatro hijos entonces
        crear un nodo nuevo apuntado por ptNuevo;
        dar al nuevo nodo los hijos 3º y 4º de nodo;
        ajustar menorDeSegundo y menorDeTercero en nodo y el nodo nuevo;
        colocar menor como la menor clave entre los hijos del nodo nuevo
      fsi
    fsi
  fsi
fin
```

Y ya con todos los detalles, queda como sigue.

```
procedimiento insertarRec(e/s nodo:diccionario;
                          ent x:tipoElmto; {x se insertará en el subárbol de nodo}
                          sal ptNuevo:diccionario;
                              {puntero al nodo recién creado a la derecha de nodo}
                          sal menor:tipoClave)
                              {elemento más pequeño del subárbol al que apunta ptNuevo}
variables
  ptAtrás:diccionario;
  menorAtrás:tipoClave;
  hijo:1..3; {indica qué hijo de nodo se sigue en la llamada recursiva (relacionado
             con la variable w de la descripción anterior del algoritmo)}
  w:diccionario {puntero al hijo}
principio
  ptNuevo:=nil;
  si nodo↑.clase=hoja entonces
    si nodo↑.elmto.clave=x.clave entonces {la clave es igual que una que había}
      nodo↑.elmto:=x  {se actualiza el resto de la información asociada a la clave}
    sino {crea una hoja nueva que contiene x.clave y "devuelve" este nodo}
      nuevoDato(ptNuevo);
      ptNuevo↑.clase:=hoja;
      si nodo↑.elmto.clave<x.clave entonces
        {pone x en el nuevo nodo a la derecha del nodo actual}
        ptNuevo↑.elmto:=x;
        menor:=x.clave
      sino {x está a la izquierda del elemento en el nodo actual}
        ptNuevo↑.elmto:=nodo↑.elmto;
        nodo↑.elmto:=x;
        menor:=ptNuevo↑.elmto.clave
      fsi
    fsi
  sino {nodo es un nodo interno}
    {selecciona el hijo de nodo que se debe seguir}
    si x.clave<nodo↑.menorDeSegundo entonces
      hijo:=1;
      w:=nodo↑.primerHijo
    sino
      si (nodo↑.tercerHijo=nil) or (x.clave<nodo↑.menorDeTercero) entonces
        {x está en el segundo subárbol}
        hijo:=2;
        w:=nodo↑.segundoHijo
      sino {x está en el tercer subárbol}
        hijo:=3;
        w:=nodo↑.tercerHijo
      fsi
    fsi;
    insertarRec(w,x,ptAtrás,menorAtrás);
    si ptAtrás≠nil entonces  {debe insertarse un nuevo hijo de nodo}
      si nodo↑.tercerHijo=nil entonces {nodo tiene 2 hijos, así que se inserta el
                                        nuevo en el lugar adecuado}
        si hijo=2 entonces
          nodo↑.tercerHijo:=ptAtrás;
```

```
          nodo↑.menorDeTercero:=menorAtrás
        sino {hijo=1}
          nodo↑.tercerHijo:=nodo↑.segundoHijo;
          nodo↑.menorDeTercero:=nodo↑.menorDeSegundo;
          nodo↑.segundoHijo:=ptAtrás;
          nodo↑.menorDeSegundo:=menorAtrás
        fsi
      sino {nodo ya tiene tres hijos}
        nuevoDato(ptNuevo);
        ptNuevo↑.clase:=interior;
        si hijo=3 entonces {ptAtrás y el 3er hijo se convierten en hijos del nuevo nodo}
          ptNuevo↑.primerHijo:=nodo↑.tercerHijo;
          ptNuevo↑.segundoHijo:=ptAtrás;
          ptNuevo↑.tercerHijo:=nil;
          ptNuevo↑.menorDeSegundo:=menorAtrás;
          {menorDeTercero está indefinido para ptNuevo}
          menor:=nodo↑.menorDeTercero;
          nodo↑.tercerHijo:=nil
        sino {hijo≤2; pasa el tercer hijo de nodo a ptNuevo}
          ptNuevo↑.segundoHijo:=nodo↑.tercerHijo;
          ptNuevo↑.menorDeSegundo:=nodo↑.menorDeTercer;
          ptNuevo↑.tercerHijo:=nil;
          nodo↑.tercerHijo:=nil
          si hijo=2 entonces  {ptAtrás se convierte en el primer hijo de ptNuevo}
            ptNuevo↑.primerHijo:=ptAtrás;
            menor:=menorAtrás
          else {hijo=1; el 2º hijo de nodo pasa a ptNuevo y
                ptAtrás se convierte en el 2º hijo de nodo}
            ptNuevo↑.primerHijo:=nodo↑.segundoHijo;
            menor:=nodo↑.menorDeSegundo;
            nodo↑.segundoHijo:=ptAtrás;
            nodo↑.menorDeSegundo:=menorAtrás
          fsi
        fsi
      fsi
    fsi
  fsi
fin
```

En cuanto al algoritmo de borrado, queda como sigue.

```
procedimiento borrar(e/s d:diccionario; ent c:tipoClave)
variables unHijo:booleano;
          ptAux:diccionario;
          menor:tipoClave
principio
  si d≠nil entonces {caso contrario no hay nada que borrar}
    si d↑.clase=hoja entonces {d sólo tenía un elemento}
      si d↑.elmto.clave=c entonces {hay que borrarlo; en otro caso, no hacer nada}
        disponer(d);
        d:=nil
      fsi
    sino {el diccionario tiene más de un elemento}
      borrarRec(d,c,unHijo,menor);
      si unHijo entonces {la raíz (d) sólo tiene un hijo}
        {hay que eliminar la raíz y hacer que d sea el que era su único hijo}
        ptAux:=d;
        d:=d↑.primerHijo;
        disponer(ptAux)
      fsi
    fsi
  fsi
fin
```

Una implementación del algoritmo recursivo de borrado es la siguiente. Resulta algo más largo por el elevado número de casos.

```
procedimiento borrarRec(e/s p:diccionario; ent c:tipoClave;
                        sal unHijo:booleano; sal menor:tipoClave)
{borra el elmto con clave c de p; si p se queda con un solo hijo devuelve verdad
 en unHijo; si c es la menor de p entonces 'menor' devuelve
 la siguiente clave más pequeña de p}
variables soloUno:booleano;
          w:diccionario;
          hijo:1..3
principio
  unHijo:=falso;
  si p↑.primerHijo↑.clase=hoja entonces {los hijos de p son hojas}
    si p↑.primerHijo↑.elmnto.clave=c entonces {la clave a borrar es el 1er hijo de p}
      disponer(p↑.primerHijo); {se borra}
      p↑.primerHijo:=p↑.segundoHijo; {se desplaza el segundo hacia la izquierda}
      menor:=p↑.primerHijo↑.elmto.clave; {se ha borrado la menor, se actualiza}
      si p↑.tercerHijo=nil entonces {ahora p tiene un solo hijo}
        unHijo:=verdad;
        p↑.segundoHijo=nil
      sino {se desplaza el tercero hacia su izquierda}
        p↑.segundoHijo:=p↑.tercerHijo;
        p↑.tercerHijo:=nil;
        p↑.menorDeSegundo:=p↑.menorDeTercero
      fsi
    sino_si p↑.segundoHijo↑.elmnto.clave=c entonces {hay que borrar el 2º hijo de p}
      disponer(p↑.segundoHijo); {se borra}
      si p↑.tercerHijo=nil entonces {ahora p tiene un solo hijo}
        unHijo:=verdad;
        p↑.segundoHijo:=nil
      sino {se desplaza el tercero hacia su izquierda}
        p↑.segundoHijo:=p↑.tercerHijo;
        p↑.tercerHijo:=nil;
        p↑.menorDeSegundo:=p↑.menorDeTercero
      fsi
    sino_si p↑.tercerHijo≠nil andthen p↑.tercerHijo↑.elmnto.clave=c entonces
      {la clave a borrar es el tercer hijo de p}
      disponer(p↑.tercerHijo); {se borra}
      p↑.tercerHijo:=nil
    fsi
  sino {los hijos de p no son hojas}
    {se selecciona el hijo de p en que hay que borrar}
    si c<p↑.menorDeSegundo entonces {si está, está en el 1er subárbol}
      hijo:=1;
      w:=p↑.primerHijo
    sino_si p↑.tercerHijo=nil orelse c<p↑.menorDeTercero entonces
      {si está, está en el 2º subárbol}
      hijo:=2;
      w:=p↑.segundoHijo
    sino  {si está, está en el tercer subárbol}
      hijo:=3;
      w:=p↑.tercerHijo
    fsi;
    borrarRec(w,c,soloUno,menor);
    si soloUno entonces {arreglar hijos de p para que ninguno tenga menos de 2 hijos}
      si hijo=1 entonces {el primer hijo de p sólo tiene un hijo}
        si p↑.segundoHijo↑.tercerHijo≠nil entonces {el 2º hijo de p tiene 3 hijos}
          {pasar el 1º hijo del 2º de p a 2º hijo del 1º de p}
          p↑.primerHijo↑.segundoHijo:=p↑.segundoHijo↑.primerHijo;
          p↑.primerHijo↑.menorDeSegundo:=p↑.menorDeSegundo;
          p↑.menorDeSegundo:=p↑.segundoHijo↑.menorDeSegundo;
          p↑.segundoHijo↑.primerHijo:=p↑.segundoHijo↑.segundoHijo;
          p↑.segundoHijo↑.segundoHijo:=p↑.segundoHijo↑.tercerHijo;
          p↑.segundoHijo↑.menorDeSegundo:=p↑.segundoHijo↑.menorDeTercero;
          p↑.segundoHijo↑.tercerHijo:=nil
        sino {el segundo hijo de p tiene dos hijos}
          {pasar el hijo del primer hijo de p como primer hijo del segundo de p
           y eliminar el primer hijo de p}
```

```
      w:=p↑.primerHijo {se hace una copia para eliminarlo luego}
      p↑.primerHijo:=p↑.segundoHijo;
      p↑.segundoHijo:=p↑.tercerHijo;
      p↑.tercerHijo:=nil;
      p↑.primerHijo↑.tercerHijo:=p↑.primerHijo↑.segundoHijo;
      p↑.primerHijo↑.menorDeTercero:=p↑.primerHijo↑.menorDeSegundo;
      p↑.primerHijo↑.segundoHijo:=p↑.primerHijo↑.primerHijo;
      p↑.primerHijo↑.menorDeSegundo:=p↑.menorDeSegundo;
      p↑.primerHijo↑.primerHijo:=w↑.primerHijo;
      disponer(w);
      p↑.menorDeSegundo:=p↑.menorDeTercero;
      si p↑.segundoHijo=nil entonces {ahora p sólo tiene un hijo}
        unHijo:=verdad
      fsi
    fsi
  sino_si hijo=2 entonces {el 2º hijo de p sólo tiene un hijo}
    si p↑.primerHijo↑.tercerHijo≠nil entonces {el primer hijo de p tiene tres hijos}
      {pasar el tercer hijo del primer hijo de p como primer hijo del 2º de p}
      p↑.segundoHijo↑.segundoHijo:=p↑.segundoHijo↑.primerHijo;
      si p↑.menorDeSegundo=c entonces {se borró la clave menor del 2º hijo de p}
        p↑.segundoHijo↑.menorDeSegundo:=menor
      sino {no ha cambiado la clave menor de 2º hijo de p}
        p↑.segundoHijo↑.menorDeSegundo:=p↑.menorDeSegundo
      fsi;
      p↑.segundoHijo↑.primerHijo:=p↑.primerHijo↑.tercerHijo;
      p↑.menorDeSegundo:=p↑.primerHijo↑.menorDeTercero;
      p↑.primerHijo↑.tercerHijo:=nil
    sino {el primer hijo de p tiene dos hijos}
      si p↑.tercerHijo≠nil andthen p↑.tercerHijo↑.tercerHijo≠nil entonces
        {el 3º hijo de p existe y tiene tres hijos}
        {se pasa el 1º hijo del 3º de p como 2º del 2º de p}
        p↑.segundoHijo↑.segundoHijo:=p↑.tercerHijo↑.primerHijo;
        p↑.segundoHijo↑.menorDeSegundo:=p↑.menorDeTercero;
        p↑.menorDeTercero:=p↑.tercerHijo↑.menorDeSegundo;
        si p↑.menorDeSegundo=c entonces
        {se ha borrado la menor clave del 2º hijo de p}
          p↑.menorDeSegundo:=menor
        fsi;
        p↑.tercerHijo↑.primerHijo:=p↑.tercerHijo↑.segundoHijo;
        p↑.tercerHijo↑.segundoHijo:= p↑.tercerHijo↑.tercerHijo;
        p↑.tercerHijo↑.tercerHijo:=nil;
        p↑.tercerHijo↑.menorDeSegundo:=p↑.tercerHijo↑.menorDeTercero
      sino {ningún otro hijo de p tiene tres hijos}
        {el único hijo del 2º hijo de p pasa a ser el 3er hijo del 1er hijo de p}
        p↑.primerHijo↑.tercerHijo:=p↑.segundoHijo↑.primerHijo;

        si p↑.menorDeSegundo=c entonces {se ha borrado la menor clave del
                                         2º hijo de p}
          p↑.primerHijo↑.menorDeTercero:=menor
        sino
          p↑.primerHijo↑.menorDeTercero:=p↑.menorDeSegundo
        fsi;
        disponer(p↑.segundoHijo);
        p↑.segundoHijo:=p↑.tercerHijo;
        p↑.tercerHijo:=nil;
        p↑.menorDeSegundo:=p↑.menorDeTercero
        si p↑.segundoHijo=nil entonces {p se ha quedado con un solo hijo}
          unHijo:=verdad
        fsi
      fsi
    fsi
  sino {hijo=3; el tercer hijo de p sólo tiene un hijo}
    si p↑.segundoHijo↑.tercerHijo≠nil entonces {el 2º hijo de p tiene 3 hijos}
      {pasar el 3er hijo del 2º hijo de p como 1er hijo del 3er hijo de p}
      p↑.tercerHijo↑.segundoHijo:=p↑.tercerHijo↑.primerHijo;
      si p↑.menorDeTercero=c entonces {borrada la menor clave del 3er hijo de p}
        p↑.tercerHijo↑.menorDeSegundo:=menor
```

```
          sino
            p↑.tercerHijo↑.menorDeSegundo:=p↑.menorDeTercero
          fsi;
          p↑.tercerHijo↑.primerHijo:=p↑.segundoHijo↑.tecerHijo;
          p↑.menorDeTercero:=p↑.segundoHijo↑.menorDeTercero;
          p↑.segundoHijo↑.tecerHijo:=nil
        sino {el segundo hijo de p tiene dos hijos}
          {el único hijo del 3er hijo de p pasa como 3er hijo del 2º hijo de p}
          p↑.segundoHijo↑.tercerHijo:=p↑.tercerHijo↑.primerHijo;
          si p↑.menorDeTercero=c entonces {borrada la menor clave del 3er hijo de p}
            p↑.segundoHijo↑.menorDeTercero:=menor
          sino
            p↑.segundoHijo↑.menorDeTercero:=p↑.menorDeTercero
          fsi;
          disponer(p↑.tercerHijo);
          p↑.tercerHijo:=nil
        fsi
      fsi
    sino {soloUno=falso; todos los hijos de p tienen 2 ó 3 hijos}
      {falta ver si hay que cambiar p↑.menorDeSegundo o p↑.menorDeTercero}
      si hijo=2 entonces
        si p↑.menorDeSegundo=c entonces
          p↑.menorDeSegundo:=menor
        fsi
      sino_si hijo=3 entonces
        si p↑.menorDeTercero=c entonces
          p↑.menorDeTercero:=menor
        fsi
      fsi {si hijo=1 no cambian p↑.menorDeSegundo ni p↑.menorDeTercero}
    fsi
  fsi
fin
```

Para terminar, veamos el algoritmo de búsqueda.

```
procedimiento buscarRec(ent p:diccionario; ent c:tpClave;
                        sal exito:booleano; sal x:tipoElmto)
principio
  si p↑.clase=hoja entonces
    si c=p↑.elemto.clave entonces {clave encontrada}
      exito:=verdad;
      x:= p↑.elemto
    sino {clave no encontrada}
      exito:=falso
    fsi
  sino {p↑.clase=interior}
    si c<p↑.menorDeSegundo entonces {buscar en el primer subárbol}
      buscarRec(p↑.primerHijo,c,exito,x)
    sino {buscar en el segundo o en el tercero}
      si (p↑.tercerHijo=nil) or (p↑.tercerHijo≠nil andthen c<p↑.menorDeTercero) entonces
        buscarRec(p↑.segundoHijo,c,exito,x)
      sino {buscar en el tercer subárbol}
        buscarRec(p↑.tercerHijo,c,exito,x)
      fsi
    fsi
  fsi
fin

procedimiento buscar(ent d:diccionario; ent c:tpClave;
                     sal exito:booleano; sal x:tipoElmto)
principio
  si d=nil entonces  {diccionario vacío}
    exito:=falso
  sino  {diccionario no vacío}
    buscarRec(d,c,exito,x)
  fsi
fin
```

Lección 17

Árboles lexicográficos (o *tries*)

Indice

1. Definición y características principales
2. Implementaciones
3. Detalles de la implementación nodo–lista

1. Definición y características principales

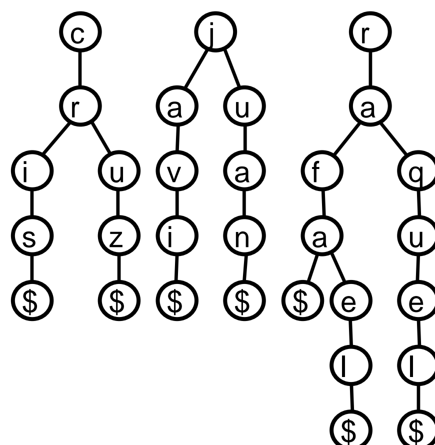
En lecciones anteriores hemos visto que un diccionario puede representarse con un árbol de búsqueda binaria (equilibrado o no), o con un árbol *n*–ario de búsqueda (equilibrado o no). En todos ellos, cada clave se almacena completa en un nodo del árbol, y la organización del árbol, y por tanto la búsqueda, se implementa basándose en comparaciones de esa clave, completa, con otras claves, también completas, almacenadas en los otros nodos.

En cambio, un ***trie*** es un árbol que sirve para almacenar claves, pero se almacenan fragmentadas en los nodos internos, para guiar la búsqueda. En un *trie*, cada clave (**cadena o secuencia de símbolos**) se reconstruye concatenando las partes de la misma (**símbolos de un alfabeto finito**) que se encuentran en todos los nodos del camino que va desde la raíz a un nodo hoja que se corresponde con la información asociada a esa clave.

Por ejemplo, si se considera que el dominio de símbolos son los caracteres, y por tanto el de las claves son las cadenas de caracteres, en cada nodo intermedio se almacena (o representa) un carácter, y la concatenación de los caracteres de cada camino desde la raíz de un árbol hasta una hoja, conforma una clave. Y en esa hoja se puede guardar el resto de la información (si la hay) asociada a la clave.

En general, si el **alfabeto de símbolos utilizado tiene definida una relación de orden** (es decir, se puede decir si un símbolo es "anterior" a otro, o no), como es el caso de los caracteres (ordenación fijada por el código ASCII, por ejemplo), el *trie* se denomina **árbol lexicográfico**.

Véase el siguiente ejemplo de bosque, secuencia de árboles lexicográficos, que almacena una colección compuesta por las cadenas de caracteres: "cris", "cruz", "javi", "juan", "rafa", "rafael", "raquel".



Nótese que **para poder almacenar una cadena que es prefijo de otra** (en el ejemplo, "rafa" y "rafael"), se hace necesario añadir a cada secuencia de nodos del camino que conforman una cadena, un nodo final con un **símbolo**

terminador. En el ejemplo, se ha utilizado el carácter '$', de forma que se supone que ese símbolo no forma parte de ninguna de las cadenas almacenadas, y sólo se usa para "terminarlas".

Nótese además que si se desea almacenar un **diccionario** y se utiliza el dominio de las claves para organizar el *trie*, como por ejemplo las cadenas de caracteres en la figura anterior, el valor asociado a cada clave se almacenaría en el nodo terminador, representado en la figura con el carácter terminador, '$'. Por tanto, para implementar un diccionario con un árbol lexicográfico, se precisa que existan **dos tipos de nodos**, los **intermedios** que almacenan los símbolos que, concatenados, conforman cada clave y los nodos **hojas**, que hacen el papel del terminador de cada clave y guardan el valor asociado a la clave.

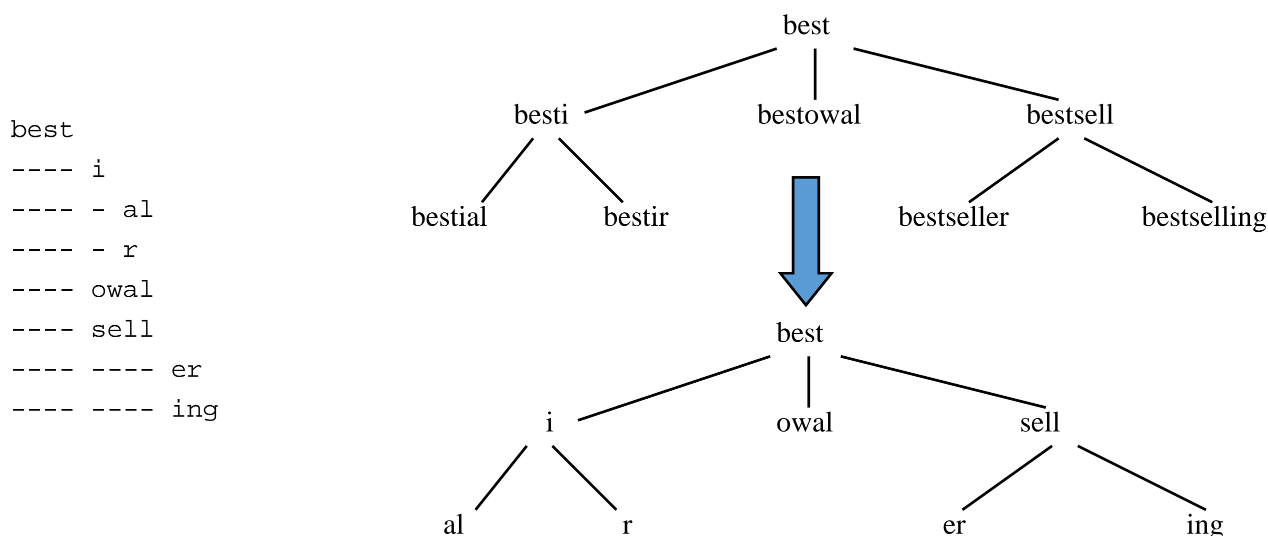
El nombre de *trie* proviene de usar las letras centrales de la palabra "*retrieval*", recuperación (de información), y se pronuncia como "*try*" para distinguirlo del sonido de "*tree*".

Un árbol lexicográfico o, en general, un *trie*, es un **árbol de prefijos**. Es decir, los prefijos comunes a un subconjunto de claves se almacenan únicamente una vez. Dicho de otra forma, **todas las hojas descendientes de un cierto nodo interno *n*, representan claves que tienen un prefijo común**, y ese prefijo sólo se almacena una vez: es la concatenación de los símbolos almacenados en el camino que va desde la raíz hasta el nodo *n*. En el ejemplo anterior, el prefijo común de "cris" y "cruz" es "cr", y sólo se ha almacenado una vez. Lo mismo con "j", el prefijo común de "javi" y "juan". De igual forma, "ra" es el prefijo común a "rafa", "rafael" y "raquel". Finalmente, "rafa" es el prefijo común a "rafa" y "rafael".

Una consecuencia de almacenar una única vez cada prefijo común a un subconjunto de claves es que **los *tries* requieren menos espacio para almacenar un conjunto de claves** que las otras soluciones vistas hasta ahora.

Por ejemplo, el diccionario de palabras inglesas de un sistema operativo (tipo *aspell* en *Linux*) puede tener unas 82.000 cadenas, entre palabras y sus inflexiones (incluyendo el inglés británico y el de EE.UU.). Y cada palabra tiene una media de unos 8 caracteres. Si lo almacenamos en un árbol lexicográfico, podemos ahorrar el espacio de tener que repetir todos los prefijos comunes en esas 82.000 palabras, que son muchos. Un ejemplo con 5 palabras (cuantas más palabras, el ahorro es mayor): "bestial", "bestir", "bestowal", "bestseller", "bestselling"; contienen muchos prefijos comunes y se pueden almacenar usando únicamente 21 caracteres, en lugar de la suma de los caracteres de las 5 (que es 42, el doble):

```
best
---- i
---- - al
---- - r
---- owal
---- sell
---- ---- er
---- ---- ing
```



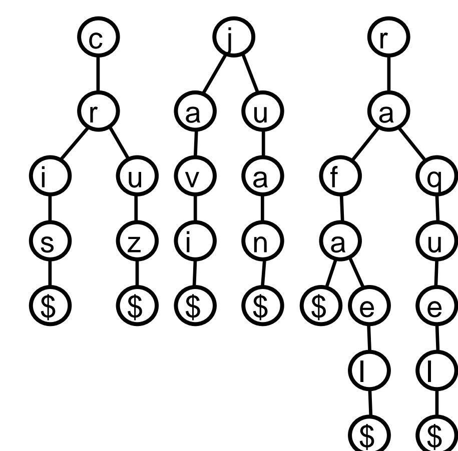
Además del ahorro en espacio, el **ahorro en tiempo** es también considerable: **la búsqueda en un *trie* de una clave de longitud *m* es de orden *O*(*m*) en el peor caso**, lineal en la longitud de la clave. A primera vista podría parecer que esto es mejor que logarítmico en el número de claves, pero para almacenar *N* claves con un alfabeto de *k* símbolos se precisan palabras de longitud log *N*, con base del logaritmo igual a *k*. Es decir, la longitud de las claves es logarítmica en el número de palabras.

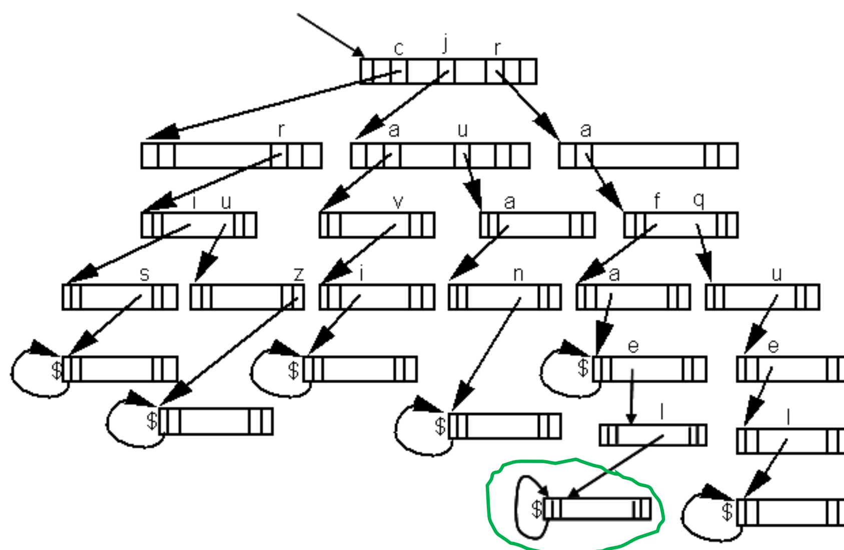
Aplicaciones habituales de árboles lexicográficos son la manipulación de diccionarios (búsqueda, inserción, borrado, etc., de claves y sus valores asociados), búsqueda de prefijos de palabras o, de forma más general, búsqueda y comparación de subcadenas: procesamiento de textos, completado automático de comandos, etc.

2. Implementaciones

Las variantes fundamentales en la implementación de los árboles lexicográficos vienen dadas por la forma elegida para almacenar un árbol *n*–ario, concretamente, en cómo almacenar el acceso a todos los subárboles (que pueden ser hasta *n*) de cada nodo interno.

Una posibilidad es que **cada nodo guarde un vector de punteros a todos los posibles hijos**, tantos como el cardinal del alfabeto de símbolos. Esta **representación se denomina nodo–vector**. Permite el acceso directo (coste constante) a la dirección de cada hijo de un nodo (acceso directo a cada componente del vector), con lo que es muy rápida. A cambio, el coste en espacio es alto porque exige almacenar un vector de punteros en cada nodo interno, y muchos de esos punteros no se utilizan (especialmente en los niveles inferiores del árbol).



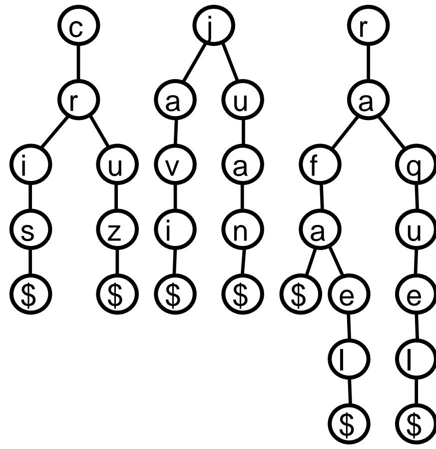


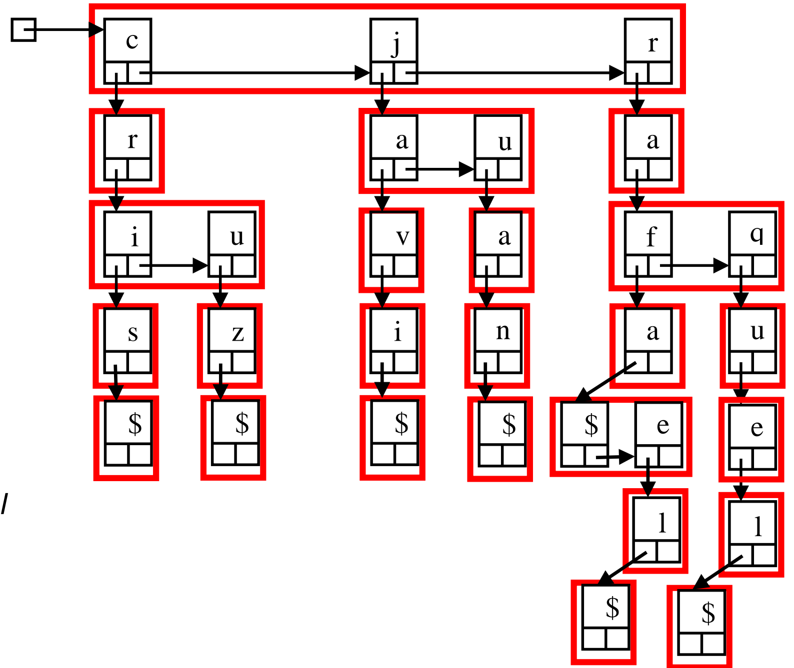
La estructura de datos necesaria es la siguiente:

```
tipos
  símbolo = predecesor('a')..'z'; {subrango de los caracteres,
                                   suponiendo que queremos cadenas de letras minúsculas;
                                   como símbolo de fin de palabra se usará el
                                   predecesor de 'a' en lugar del carácter '$'}

  trie = ↑nodo;
  nodo = vector[símbolo] de trie
```

Nótese que con esta representación no se almacenan realmente los caracteres de una cadena, sino que esa información se deduce de qué puntero del vector `nodo` es el que apunta al subárbol correspondiente. El terminador de cada cadena es un `nodo` final que tiene todos los punteros iguales a `nil` excepto el primero, el de la componente `predecesor('a')`, que es distinto de `nil` y apunta, por ejemplo, al mismo `nodo` (véase en la figura el nodo terminador, marcado en verde, de la cadena “rafael”, por ejemplo).

Otra posibilidad es que cada nodo interno incluya una lista enlazada por punteros que contenga las raíces de los subárboles. Se denomina **representación nodo–lista** y se corresponde con la representación **primogénito–siguiente hermano** de un árbol *n*–ario. Esta representación permite un menor coste en espacio, puesto que sólo se almacenan los caracteres o símbolos estrictamente necesarios (más un par de punteros por símbolo), pero un mayor tiempo para acceder a los hijos de un nodo interno, pues obliga a recorrer la lista de nodos hijos.





En la siguiente sección veremos una implementación completa utilizando esta representación de los datos.

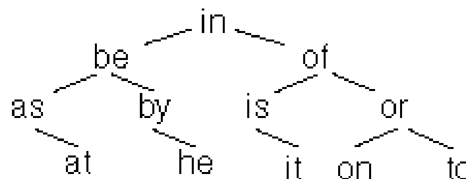
Una tercera alternativa, intermedia entre las dos anteriores, consiste en que para almacenar los nodos hijos de un nodo interno no se utiliza ni un vector ni una lista, sino un árbol de búsqueda binario. Se denomina, **representación nodo–abb** de un árbol lexicográfico. Para poder implementarla, cada nodo del árbol requiere tres punteros, por eso también se conoce esta representación de *tries* como **árbol ternario de búsqueda**:

- dos **punteros**, al hijo **izquierdo** y **derecho**, como en un árbol binario de búsqueda, y
- un **puntero**, denominado **central**, a la raíz del *trie* al que da acceso ese nodo.

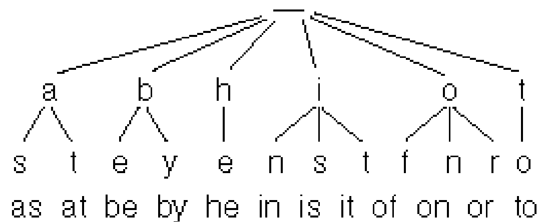
El objetivo de esta representación es combinar la eficiencia de los *tries* con la de los ABB, al usar éstos en cada nodo del *trie*. Una búsqueda de una cadena en un *trie* con representación nodo–abb:

- primero, compara el carácter actual de la cadena buscada (empezando por el primero) con el carácter del nodo;
- después, si el carácter buscado es menor que el almacenado en el nodo, la búsqueda de ese carácter sigue en el hijo izquierdo;
- si el carácter buscado es mayor que el del nodo, se sigue buscando el mismo carácter en el hijo derecho;
- si, por último, el carácter buscado coincide con el carácter de ese nodo, se prosigue la búsqueda en el hijo central, pero con el siguiente carácter de la cadena buscada.

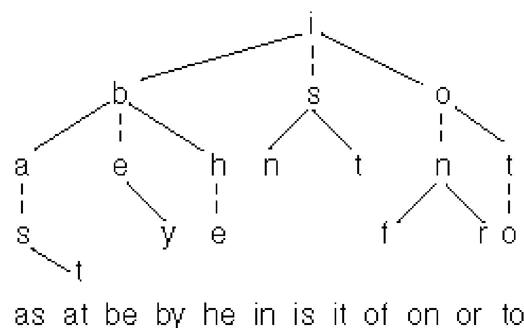
Por ejemplo, si se quiere almacenar en un árbol ternario de búsqueda las palabras: “as”, “at”, “be”, “by”, “he”, “in”, “is”, “it”, “of”, “on”, “or”, “to”, si se guardan en un ABB (árbol binario de búsqueda), quedarían, por ejemplo (hay muchas formas distintas de ordenarlas en un ABB), así (de hecho, éste sería un AVL):



En un *trie* quedarían así:



Pero si lo dibujamos teniendo en cuenta la representación en memoria, como árbol ternario de búsqueda (o *trie* con representación nodo–abb), quedaría de esta forma:



No se han dibujado los caracteres terminadores al final de las cadenas (serían 12 hojas con el símbolo '$'). Los trazos continuos representan a los punteros izquierdo y derecho. Los trazos discontinuos representan a los punteros centrales.

Además de las variantes anteriores de representación de los *tries*, existe una variante muy interesante, más compacta, denominada ***PATRICIA*** (*Practical Algorithm To Retrieve Information Coded In Alphanumeric*). Los *PATRICIA* vienen a resolver un problema de los *tries*, tal y como los hemos presentado, que consiste en que si el número de claves es mucho menor que el total de claves que podrían almacenarse, la mayoría de los (o muchos) nodos internos tienen un único hijo, lo cual aumenta inútilmente los costes en espacio y tiempo. En los *PATRICIA*, se evita que exista cualquier nodo interno que tenga un único hijo, por ello, se compacta el árbol haciendo que todos los nodos con un único hijo se fusiones con sus padres. Los *PATRICIA* se utilizan, por ejemplo, para implementar las tablas de encaminamiento en los *router*.

3. Detalles de la implementación nodo–lista

Como dijimos, la representación nodo–lista se corresponde con la representación **primogénito–siguiente hermano** de un árbol *n*–ario. Esta representación permite un bajo coste en espacio, puesto que sólo se almacenan los caracteres o símbolos estrictamente necesarios (además de un par de punteros por símbolo), pero requiere un mayor tiempo para acceder a los hijos de un nodo interno, pues obliga a recorrer la lista de nodos hijos.

Las declaraciones de tipos de datos necesarias para implementar un **árbol lexicográfico** que contenga un **conjunto de cadenas de caracteres** mediante la **representación nodo–lista** son las siguientes:

```
tipos trie = ↑nodo;
      nodo = registro
               dato:carácter; {usaremos '$' como carácter terminador de cada cadena}
               primogenito,sigHermano:trie
             freg
```

Evidentemente, si se pretendiese almacenar un diccionario genérico, sería preciso definir el tipo de los símbolos que componen las claves (por ejemplo, caracteres), a partir de ellos el tipo de las claves como secuencias de símbolos (por ejemplo, cadenas de caracteres), y el tipo de los valores asociados a las claves. Además, habría que incluir en el tipo nodo

un campo que indicase si se trata de nodo interno u hoja, en los nodos internos se guardaría un símbolo (por ejemplo, un carácter) y en las hojas se almacenarían los valores asociados a las correspondientes claves (secuencias de símbolos).

A continuación, presentamos una implementación de las operaciones básicas para *crear* un *trie* vacío, *insertar* una palabra en un *trie* (requiere un procedimiento auxiliar que denominamos *plantarPalabra*), saber si una palabra *pertenece* a un *trie* y *eliminar* una palabra del *trie*, para la estructura de datos definida anteriormente (válida para almacenar un conjunto de cadenas de caracteres).

```
procedimiento crearVacío(sal t:trie)
{Devuelve en t un conjunto vacío de palabras o cadenas de caracteres}
principio
  t:=nil
fin

procedimiento plantarPalabra(ent palabra:cadena; e/s t:trie)
{Algoritmo auxiliar para plantar un árbol vertical (lista) con los caracteres de
 la palabra recibida como parámetro. Esa palabra NO puede incluir el carácter '$'.}
variables taux:trie;
          resto:cadena
principio
  si long(palabra)=0 entonces {long devuelve la longitud de una cadena}
    nuevoDato(t);
    t↑.dato:='$'; {marca de fin de palabra}
    t↑.primogenito:=nil;
    t↑.sigHermano:=nil
  sino
    resto:=palabra[2..long(palabra)]; {subcadena desde la posición 2 a la última}
    plantarPalabra(resto,taux);
    nuevoDato(t);
    t↑.dato:=palabra[1]; {primer carácter de la palabra}
    t↑.primogenito:=taux;
    t↑.sigHermano:=nil
  fsi
fin

procedimiento insertar(ent palabra:cadena; e/s t:trie)
{Añade la palabra (que NO puede incluir el carácter '$') al conjunto t, si no estaba.
 Si ya estaba, t queda igual.}
variables taux:trie;
          resto:cadena
principio
  si t=nil entonces
    plantarPalabra(palabra,t);
  sino
    si long(palabra)=0 entonces
      si t↑.dato≠'$' entonces {en caso contrario, la palabra ya estaba en t}
        nuevoDato(taux);
        taux↑.dato:='$';
        taux↑.primogenito:=nil;
        taux↑.sigHermano:=t;
        t:=taux
      fsi
    sino {t≠nil and long(palabra)>0}
      si palabra[1]<t↑.dato entonces
        plantarPalabra(palabra,taux);
        taux↑.sigHermano:=t;
        t:=taux
      sino_si palabra[1]=t↑.dato entonces
        resto:=palabra[2..long(palabra)];
        insertar(resto,t↑.primogenito)
      sino {palabra[1]>t↑.dato}
        insertar(palabra,t↑.sigHermano)
      fsi
    fsi
  fsi
fin
```

```
función pertenece(palabra:cadena; t:trie) devuelve booleano
{Devuelve verdad si y sólo si la palabra está en t}
variable resto:cadena
principio
  si t=nil entonces
    devuelve falso
  sino
    si long(palabra)=0 entonces
      devuelve t↑.dato='$'
    sino
      si palabra[1]<t↑.dato entonces
        devuelve falso
      sino_si palabra[1]=t↑.dato entonces
        resto:=palabra[2..long(palabra)];
        devuelve pertenece(resto,t↑.primogenito)
      sino {palabra[1]>t↑.dato}
        devuelve pertenece(palabra,t↑.sigHermano)
      fsi
    fsi
  fsi
fin

procedimiento eliminar(ent palabra:cadena; e/s t:trie)
{Borra la palabra del conjunto t, si estaba. Si no, t queda igual.}
variables taux:trie;
          resto:cadena
principio
  si t≠nil entonces
    si long(palabra)=0 entonces
      si t↑.dato='$' entonces
        taux:=t;
        t:=t↑.sigHermano;
        disponer(taux)
      fsi
    sino
      si palabra[1]=t↑.dato entonces
        resto:=palabra[2..long(palabra)];
        eliminar(resto,t↑.primogenito);
        si t↑.primogenito=nil entonces
          taux:=t;
          t:=t↑.sigHermano;
          disponer(taux)
        fsi
      sino_si palabra[1]>t↑.dato entonces
        eliminar(palabra,t↑.sigHermano)
      fsi
    fsi
  fsi
fin
```

El coste en tiempo en el caso peor es de orden lineal en la longitud de la cadena (insertada, buscada, o borrada), si bien incluye una constante multiplicativa alta, del orden del cardinal de valores posibles del tipo carácter elevado a la longitud de la cadena, puesto que, en cada nivel del árbol, debido a la representación nodo–lista, puede ser necesario recorrer una lista de longitud máxima igual al cardinal del dominio de valores del tipo carácter.

Lección 18

Colas con prioridad, montículos y el *heapsort*

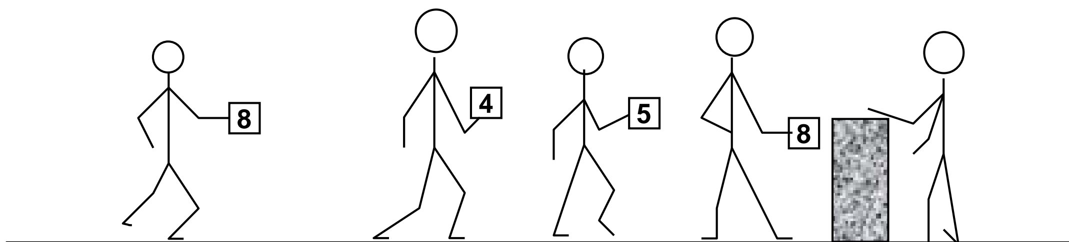
Indice

1. Concepto y especificación de cola con prioridad genérica
2. Implementación basada en un montículo o *heap*
3. Aplicación a la ordenación de elementos de un vector: el *heapsort*

1. Concepto y especificación de cola con prioridad genérica

Una **cola con prioridad** es un **TAD genérico** cuyo **dominio de valores** son las colecciones de elementos dotados de un valor comparable, denominado prioridad del elemento, y **operaciones** de *insertar* un nuevo elemento, *observar* un elemento de máxima prioridad y *borrar* un elemento de máxima prioridad. Es decir, los elementos se extraen o "salen de la cola" por orden, según su prioridad.

Una metáfora admisible para el TAD es la de una sala de espera a la que llegan clientes, cada uno con su prioridad, y son servidos por un agente según el orden marcado por su prioridad (de mayor a menor prioridad).



Nótese que varios elementos pueden tener la misma prioridad. En ese caso, el elemento denominado "de máxima prioridad" es uno cualquiera de los que tengan el valor máximo de prioridad.

Una especificación del TAD genérico cola con prioridad es la siguiente.

espec colasConPrioridadesDeMáximos
usa booleanos
parámetro formal
  género elemento
  operación _ ≤ _: elemento e1, elemento e2 -> booleano *{suponemos que en el género elemento hay definida una relación de orden "≤" y se interpreta que a ≤ b significa que b tiene más prioridad que a}*
fpf
género cp *{cola de elementos con prioridad (de máximos), su dominio son colecciones de elementos dotados de una relación de orden que define su prioridad relativa}*

operaciones
  crearVacía: -> cp
  {Devuelve una cola vacía, sin elementos}

  añadir: cp c, elemento e -> cp
  {Devuelve la cola resultante de añadir un ejemplar del elemento e a la cola c}

esVacía?: cp c -> booleano
{Devuelve verdad si y sólo si c no tiene elementos}

parcial max: cp c -> elemento
{Devuelve un elemento máximo de c. Parcial: la operación no está definida si c es vacía}

eliminarMax: cp c -> cp
{Si c no es vacía, devuelve una cola igual a la resultante de eliminar de c un elemento máximo según la relación de orden definida por la función "≤". Si c es vacía, queda igual.}

fespec

El TAD definido por la especificación anterior se denomina **cola con prioridad de máximos**, puesto que el elemento destacado (por la operación "max") y el eliminado (con "eliminarMax"), es un elemento máximo para la relación de orden definida sobre los elementos. De manera análoga se puede especificar una **cola con prioridad de mínimos**, en ese caso las operaciones definidas serían: crearVacía, añadir, esVacía?, min, y eliminarMin.

2. Implementación basada en un montículo o *heap*

Aunque son posibles otras opciones, la **representación más habitual y eficiente** para el dominio de valores de una cola con prioridad es la **estructura de datos denominada montículo** o, en inglés, *heap* (nótese que este *heap* no tiene ninguna relación con la memoria libre o "memoria montón", o zona de memoria dinámica, en la que se almacenan los datos creados dinámicamente, es decir, durante la ejecución de los programas, y apuntados por punteros, aunque en inglés también se llama *heap* a esa zona de memoria libre).

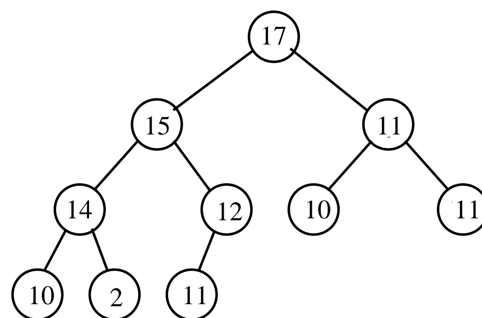
Para definir la estructura montículo, definimos antes los árboles binarios parcialmente ordenados y, luego, recordamos la definición de árboles casi-completos.

Un **árbol binario** se dice **parcialmente ordenado** (de máximos) si verifica las siguientes propiedades:

- el elemento raíz es mayor o igual que el resto de los elementos del árbol, y
- los subárboles izquierdo y derecho son árboles binarios parcialmente ordenados.

Debe observarse que, aunque el valor de un nodo nunca es menor que los valores de sus hijos, pueden encontrarse valores más grandes en niveles inferiores a los de valores más pequeños (en la figura, el 14 y el 12 se encuentran en el nivel 2 mientras que un 11 se encuentra en el nivel 1).

Sin embargo, en la raíz siempre debe encontrarse un nodo de valor máximo.



Representación de un árbol parcialmente ordenado de máximos, sólo se muestran las prioridades

Debe hacerse notar que un árbol de este tipo **no es útil para buscar elementos**, puesto que no está ordenado al estilo de un árbol binario de búsqueda.

Representando una cola con prioridades mediante un árbol binario parcialmente ordenado, se consigue que las operaciones `crearVacía`, `max` y `esVacía?` pueden ser implementadas fácilmente con coste en tiempo constante (independiente del número de elementos de la cola). La operación `max` consiste simplemente en acceder al elemento que está en la raíz, si el árbol es no vacío.

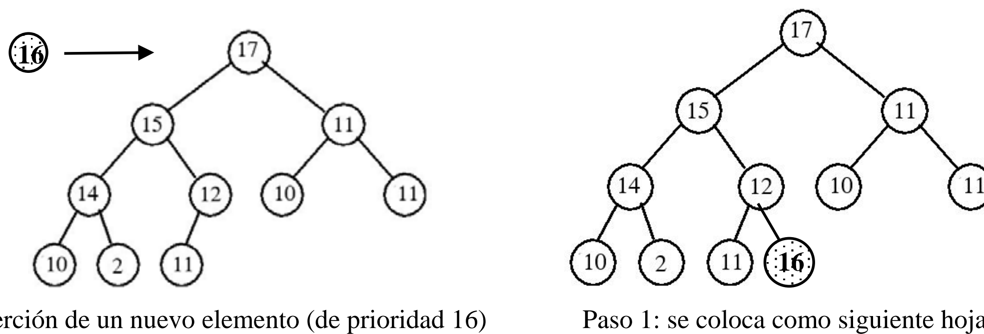
En el caso de las operaciones `añadir` y `eliminarMax` la implementación exige, como veremos a continuación, **recorrer un camino en el árbol desde la raíz hasta una hoja** (o al revés). En el peor caso, la longitud de tal camino es del orden del número de elementos del árbol (árbol degenerando, con forma de lista).

El coste puede reducirse en el peor caso al **orden del logaritmo del número de elementos del árbol** si se garantiza que el árbol sea **completo o casi-completo**. Como ya se definió en una lección anterior, un árbol se dice casi-completo cuando se puede obtener a partir de un árbol **completo** eliminando hojas consecutivas del último nivel, comenzando por la que está más a la derecha. Y un árbol se dice completo cuando su nivel más profundo está lleno (y por tanto, todos los anteriores también lo están).

Recibe el nombre de **montículo** (*heap*, en inglés) **un árbol binario parcialmente ordenado casi-completo o completo**.

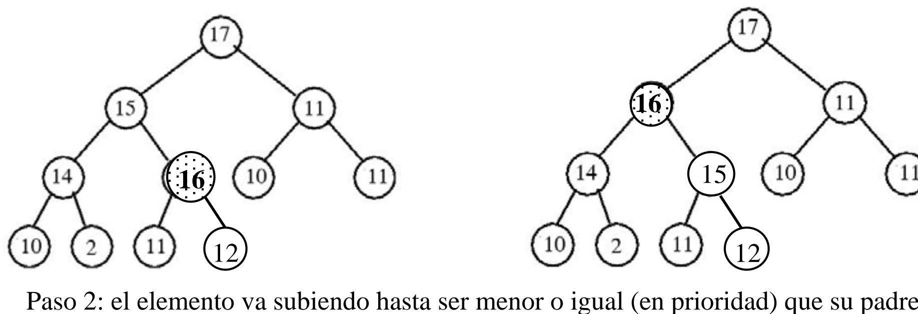
La operación de **añadir** un nuevo elemento en un montículo (manteniéndolo casi-completo) puede realizarse de la siguiente forma:

- en primer lugar, se coloca el nuevo elemento como siguiente hoja en el nivel más profundo; se inicia un nuevo nivel, con una única hoja lo más a la izquierda posible, si el nivel más profundo se encuentra ya completamente lleno (ver la figura);



Inserción de un nuevo elemento (de prioridad 16) Paso 1: se coloca como siguiente hoja

- si el nuevo elemento es mayor que su padre, se intercambian; así se repite el proceso (de comparación del nuevo elemento con su padre e intercambio) hasta que el nuevo elemento llegue hasta la raíz o alcance una posición en la que sea menor o igual que su padre (ver la figura).

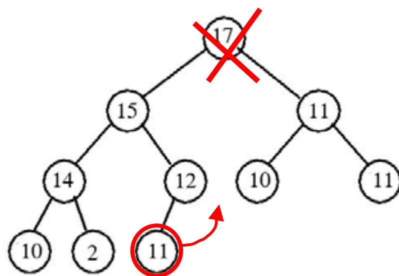


Paso 2: el elemento va subiendo hasta ser menor o igual (en prioridad) que su padre

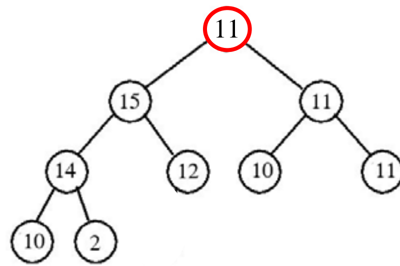
De esta forma, si es posible acceder desde cualquier elemento a su padre en tiempo constante, el tiempo de ejecución de la operación de inserción es proporcional a la distancia que el nuevo elemento asciende en el árbol. Esta distancia, por ser el árbol completo o casi-completo, nunca es mayor que el logaritmo del número de elementos del árbol (es decir, la altura del árbol).

En cuanto a la operación de **eliminación** del elemento máximo (que se encuentra en la raíz del árbol), no basta con borrar la raíz (porque lo que se obtiene es un bosque con dos árboles y no un árbol), sino que deben darse los pasos siguientes:

- se toma la hoja de más a la derecha del nivel más bajo y se coloca de forma provisional en la raíz, sustituyendo al elemento máximo (ver la figura);

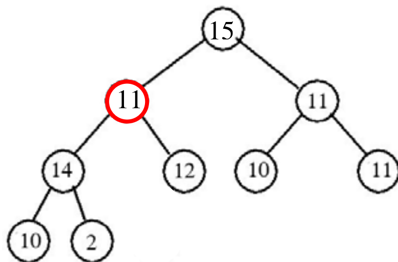


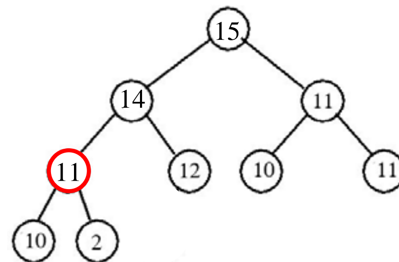
Borrado del elemento máximo



Paso 1: la hoja última pasa a ser raíz

- después, el elemento que se ha colocado en la raíz se compara con su hijo de mayor valor y, si es menor que dicho, hijo se intercambian; así se repite el proceso de comparación del elemento con su hijo de mayor valor (e intercambio en caso de que sea menor que dicho hijo), hasta que el elemento esté en una hoja o tenga un valor mayor o igual que sus hijos.





Paso 2: la raíz va bajando hasta alcanzar la posición adecuada

Nótese que el número máximo de operaciones que hay que realizar para suprimir el máximo elemento de un árbol con *n* nodos es del orden del logaritmo de *n*, que es el número de niveles del árbol y por tanto el máximo número de veces que la hoja que antes hemos colocado como raíz tiene que intercambiarse por alguno de sus hijos hasta llegar a su posición correcta. Evidentemente, para ello, debe ser posible acceder a la hoja más a la derecha con coste constante, y acceder desde cualquier elemento a sus hijos también con coste constante.

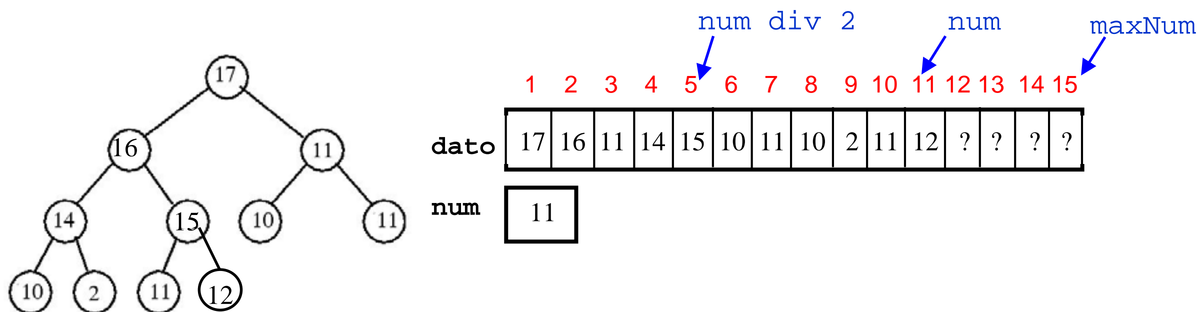
Por tratarse de un árbol casi-completo, una **representación estática del montículo (o cola con prioridad), basada en un vector** es eficiente tanto en espacio como en tiempo, con la típica limitación de toda representación estática: el tamaño máximo de la cola con prioridad estará acotado por el tamaño del vector (y por tanto la operación de añadir deberá ser implementada como parcial). Además, no es trivial dar con una buena representación dinámica debido a dos exigencias que hemos detectado en la inserción y el borrado: acceso con coste constante al lugar en el que debería estar la siguiente hoja más a la derecha, para insertar un nuevo nodo, y acceso con coste constante para borrar la hoja más a la derecha y volver a tener acceso con coste constante a la nueva hoja más a la derecha.

```
constante maxNum = ... {Máximo nº de elementos almacenables en la cola con prioridad}
tipo cp = registro
            dato: vector[1..maxNum] de elemento;
            num: 0..maxNum  {nº actual de elementos en la cola con prioridad}
          freg
```

Un **montículo vacío** se representa almacenando el valor `0` en el campo `num`. Para un **montículo no vacío**, el campo `num` guarda el número de elementos del árbol; los elementos se guardan en las componentes `1..num` del campo `dato` de la siguiente forma:

- la raíz se almacena en la primera componente, `dato[1]`;
- el hijo izquierdo de `dato[i]`, si existe, se guarda en `dato[2*i]` y el hijo derecho, si existe, en `dato[2*i+1]`.

Por tanto, el padre de un nodo (distinto de la raíz) almacenado en `dato[i]` está siempre en `dato[i div 2]`. Dicho de otra forma, **los nodos del árbol se almacenan por niveles, de arriba a abajo y de izquierda a derecha en cada nivel** (ver la figura).



Nótese que si el montículo tiene `num` elementos (11 en la figura de arriba), entonces los primeros `num div 2` (en el caso de la figura son 5) elementos almacenados en el vector representan nodos internos del montículo y el resto son hojas.

Al igual que en todas las representaciones estáticas de TAD con número de valores no finito, únicamente se puede **representar un subconjunto de los valores posibles** (en este caso, el de los montículos con número de elementos menor o igual que `maxNum`). Este hecho se traduce en que la operación añadir se implementa como **operación parcial**.

La implementación detallada es la siguiente.

```
módulo colaConPrioridadesDeMáximos
parámetro
  tipo elemento
  con función "<"(e1,e2:elemento) devuelve booleano
  {el tipo elemento tiene definida una función de orden que determina la prioridad}
exporta
  constante maxNum {número máximo de elementos de la cola}
  tipo cp  {Representación del TAD cola con prioridad de máximos}

  procedimiento crearVacía(sal c:cp)
  {devuelve una cola vacía, sin elementos}

  procedimiento añadir(e/s c:cp; ent e:elemento; sal error:booleano)
  {si nºelementos(c)<maxNum entonces error=falso y añade e a c; si no error=verdad}

  función esVacía(c:cp) devuelve booleano
  {devuelve verdad si y sólo si c es la cola vacía}

  función max(c:cp) devuelve elemento
  {Pre: esVacía(c)=falso} {Post: devuelve un elemento máximo de c con respecto a "<"}

  procedimiento eliminarMax(e/s c:cp)
  {Si c es vacía sigue igual. Si no elimina de c un elemento máximo con respecto a "<".}

implementación
  constante maxNum = ... {número máximo de elementos de la cola}
  tipo cp = registro
              dato: vector[1..maxNum] de elemento;
              num:0..maxNum
            freg

  procedimiento crearVacía(sal c:cp)
  {devuelve una cola vacía, sin elementos}
  principio
     c.num:=0
  fin

  procedimiento añadir(e/s c:cp; ent e:elem; sal error:booleano)
  {si nºelementos(c)<maxNum entonces error=falso y añade e a c; si no error=verdad}
  variables i:natural;
            debeSubir:booleano;
            aux:elemento
  principio
    si c.num=maxNum entonces
      error:=verdad
```

```
    sino
      error:=falso;
      c.num:=c.num+1;
      c.dato[c.num]:=e;
      i:=c.num; {índice de la posición actual de e}
      si i>1 entonces
        debeSubir:=c.dato[i]>c.dato[i div 2]  {¿tiene más prioridad que su padre?}
      sino {i=1}
        debeSubir:=falso
      fsi;
      mientrasQue debeSubir hacer
        {subir e en el árbol, intercambiándolo con su padre}
        aux:=c.dato[i];
        c.dato[i]:=c.dato[i div 2];
        c.dato[i div 2]:=aux;
        i:=i div 2;   {índice de la posición actual de e}
        si i>1 entonces
          debeSubir:=c.dato[i]>c.dato[i div 2]  {¿tiene más prioridad que su padre?}
        sino {i=1}
          debeSubir:=falso
        fsi
      fmq
    fsi
  fin

  procedimiento eliminarMax(e/s c:cp)
  {Si c es vacía sigue igual. Si no elimina de c un elemento máximo con respecto a "<".}
  variables i,j:natural;
            aux:elemento
  principio
    si c.num>0 entonces
      c.dato[1]:=c.dato[c.num];
      c.num:=c.num-1;
      i:=1;
      {i es el índice de la posición actual del que antes era el último elemento}
      mientrasQue i≤c.num div 2 hacer {el elemento de la posición i tiene hijo(s)}
        {bajar el anterior último elemento del árbol}
        si (2*i=c.num) orelse (c.dato[2*i]>c.dato[2*i+1]) entonces
          j:=2*i
        sino
          j:=2*i+1
        fsi; {j=hijo de i con mayor prioridad, o único hijo}
        si c.dato[i]<c.dato[j] entonces
          {intercambia el anterior último elemento y su hijo}
          aux:=c.dato[i];
          c.dato[i]:=c.dato[j];
          c.dato[j]:=aux;
          i:=j
        sino
          i:=c.num {para salir del bucle}
        fsi
      fmq
    fsi
  fin

  función max(c:cp) devuelve elemento
  {Pre: esVacía(c)=falso} {Post: devuelve un elemento máximo de c con respecto a "<"}
  principio
    devuelve c.dato[1]
  fin

  función esVacía(c:cp) devuelve booleano
  {devuelve verdad si y sólo si c es la cola vacía}
  principio
    devuelve c.num=0
  fin
fin   {de colaConPrioridadesDeMáximos}
```

3. Aplicación a la ordenación de elementos de un vector: el *heapsort*

Veamos ahora cómo una cola con prioridades puede servir para lograr un algoritmo eficiente de ordenación de los datos de un vector in situ. Para ello, basta recordar el sencillo método de ordenación por **selección directa**. El primer paso consiste en elegir el dato más pequeño de entre todos los que se desea ordenar. A continuación, se elige el más pequeño de los restantes. Así se continua hasta haber elegido todos los datos por orden.

Para realizarlo de forma eficiente, **almacenaremos los datos** que deben ser ordenados **en una cola con prioridad de máximos**. Lógicamente, la función de ordenación que define la prioridad es la misma con la que queremos ordenar los elementos. Después se irá extrayendo la raíz del árbol almacenándola por orden en el vector, hasta que el árbol quede vacío.

```
procedimiento ordena(e/s v:vector[1..maxNum] de elemento; ent n:1..maxNum)
{Precondición: n>0.
 Postcondición: Permuta los n primeros datos de v, dejándolos en orden creciente.}
variables c:cp;
          i:entero
principio
  crearVacía(c); {una cola con prioridad de máximos vacía}
  para i:=1 hasta n hacer  {copia los datos a ordenar de v en c}
    añadir(c,v[i])
  fpara;
  para i:=n descendiendo hasta 1 hacer  {extrae los datos en orden decreciente de c}
    v[i]:= max(c);
    eliminarMax(c)
  fpara
fin
```

Puesto que el número de operaciones necesarias para realizar las instrucciones `añadir` y `eliminarMax` es, en el peor caso, de orden log *n*, la complejidad total de este algoritmo de ordenación es de orden $O(n \log n)$ (cada uno de los dos bucles se ejecuta *n* veces y su interior tiene una complejidad de orden log *n*).

Podría parecer que la disminución en el número de operaciones necesarias para ordenar el vector se debe a que se ha duplicado el espacio de memoria utilizada, empleando una cola con prioridad auxiliar. Sin embargo, no es así. Es posible modificar el algoritmo de forma que sólo se trabaje con el vector original, manteniendo la misma complejidad del número de operaciones a realizar. Para ello, la cola con prioridad `c` del algoritmo anterior se guardará en las componentes `v[1]`…`v[i]` siempre que `c` tenga `i` nodos. El elemento mayor estará siempre en `v[1]`. Los elementos que se vayan extrayendo de `v[1]` se irán almacenando en `v[n]`…`v[i+1]`, con lo que el vector quedará ordenado de menor a mayor. Se incluye a continuación el algoritmo en detalle. Previamente, un par de procedimientos auxiliares.

```
procedimiento intercambiar(e/s v:vector[1..maxNum] de elemento; ent i,j:1..maxNum)
{Intercambia los valores de v[i] y v[j]}
variable aux:elemento
principio
  aux:=v[i];
  v[i]:=v[j];
  v[j]:=aux
fin

procedimiento empujar(e/s v:vector[1..maxNum] de elemento; ent pri,ult:1..maxNum)
{Pre: v[pri]...v[ult] satisfacen la propiedad de montículo excepto, quizá,
      que los hijos de v[pri] pueden ser mayores que él}
{Post: permuta los elementos v[pri]..v[últ] para que TODOS, v[pri] incluido,
 verifiquen la propiedad del montículo}
variables r:entero {r indicará la posición 'actual' de v[pri]}
principio
  r:=pri;
  mientrasQue r≤ult div 2 hacer
    si ult=2*r entonces  {r tiene sólo un hijo, en la posición 2*r=ult}
      si v[r]<v[2*r] entonces
        intercambiar(v,r,2*r)
      fsi;
      r:=ult  {para forzar la terminación del bucle}
    sino  {r tiene dos hijos, en las posiciones 2*r y 2*r+1}
      si (v[r]<v[2*r]) and (v[2*r]≥v[2*r+1]) entonces {intercambiar r con su hijo izq}
```

```
          intercambiar(v,r,2*r);
          r:=2*r
        sino
          si (v[r]<v[2*r+1]) and (v[2*r+1]>v[2*r]) entonces {intercambiarlo con hijo dch}
            intercambiar(v,r,2*r+1);
            r:=2*r+1
          sino  {r no viola la propiedad de montículo}
            r:=ult  {para forzar la terminación del bucle}
          fsi
        fsi
      fsi
    fmq
  fin

  procedimiento heapsort(e/s v:vector[1..maxNum] de elemento; ent n:1..maxNum)
  {Precondición: n>0.
   Postcondición: Permuta los n primeros datos de v, dejándolos en orden creciente.}
  variables i:entero
  principio
    para i:=n div 2 descendiendoHasta 1 hacer
      empujar(v,i,n)
    fpara; {ahora v es un montículo de máximos}
    para i:=n descendiendoHasta 2 hacer
      {eliminar el máximo de la raíz del montículo}
      intercambiar(v,1,i);
      empujar(v,1,i-1) {restablecer el montículo hasta la posición i-1}
    fpara
  fin
```

Para analizar el **coste** del algoritmo anterior, debe analizarse primero el coste del subalgoritmo `empujar`. El interior del bucle **`mientrasQue`** `r≤último div 2` **`hacer`**`...` **`fmq`** lleva un tiempo en $O(1)$. Para calcular cuántas veces se ejecuta ese bucle, basta ver que, después de cada iteración, `r` tiene por lo menos el doble del valor que tenía. Por tanto, puesto que `r` empieza con valor igual a `pri`, después de `i` iteraciones se tiene:

`r ≥ pri*2i`

El bucle termina si:

`r>ult/2`

y esto ocurre después de `i` iteraciones si:

`pri*2i>últ/2 (r≥pri*2i>últ/2 fi r>últ/2)`

es decir, si:

`i>log(últ/pri)-1`

En consecuencia, el número de iteraciones del bucle **`mientrasQue`** `r≤último div 2` **`hacer`**`...` **`fmq`** está, a lo sumo, en O(log `ult/pri`).

Como en cada llamada al subalgoritmo `empujar` que se hace en `heapsort` se verifica que `pri≥1` y `ult≤n`, entonces cada llamada a empujar está en O(log `n`).

Por tanto, el coste total del algoritmo de ordenación in situ de vectores, *heapsort*, está en $O(n \log n)$.

TEMA IV

Tipos de datos funcionales

Lección 19

El TAD tabla y las tablas dispersas (*hash*)

Indice

1. Recordatorio del concepto de tabla o diccionario y su especificación
2. Repaso de implementaciones ya vistas
3. Tablas dispersas (o *hash*)

1. Recordatorio del concepto de tabla o diccionario y su especificación

Como ya dijimos en una lección anterior, una **tabla o diccionario** es un conjunto o colección de pares. El primer miembro de cada par es la clave y el segundo, el valor asociado a la clave. No puede haber dos pares que tengan la misma clave.

Las tablas o diccionarios también se llaman **tipos de datos asociativos o funcionales** porque representan una función cuyo dominio es el género de las claves y cuyo rango es el género de los valores:

f : Dominio de las claves → Dominio de los valores

La función representada suele ser parcial (es decir, existen claves del dominio sin valor asociado), aunque podrían representarse como totales asociando a algunas claves un valor especial llamado "indefinido".

Las **operaciones básicas** para los diccionarios son: la **inserción** de un nuevo par (o la modificación del valor asociado, si la clave ya existe), la **búsqueda** u obtención del valor asociado a una clave y el **borrado** de un par dada su clave.

Recordamos la especificación:

espec diccionariosGenéricos
usa boleanos, naturales
parámetros formales
  géneros clave, valor
  operaciones *{suponemos que en el género de las claves hay definida una función de orden y otra de igualdad}*
    _<_ : clave c1, clave c2 -> booleano
    _=_ : clave c1, clave c2 -> booleano
fpf
género diccionario *{Los valores del TAD representan conjuntos de pares (clave,valor) sin claves repetidas}*

operaciones
  crearVacío: -> diccionario
  {Devuelve un diccionario vacío, sin elementos}

  añadir: diccionario d, clave c, valor v -> diccionario
  {Si en d no hay ningún par con clave c, devuelve el diccionario resultante de añadir el par (c,v) a d; si en d hay un par (c,v'), entonces devuelve el resultado de sustituirlo por el par (c,v)}

  está?: clave c, diccionario d -> booleano
  {Devuelve verdad si y sólo si en d hay algún par (c,v)}

parcial obtenerValor: clave c, diccionario d -> valor
{Devuelve el valor asociado a la clave c en d.
Parcial: la operación no está definida si c no está en d}

borrar: clave c, diccionario d -> diccionario
{Si c está en d, devuelve el diccionario resultante de borrar c y su valor de d;
si c no está en d, devuelve un diccionario igual a d}

cardinal: diccionario d -> natural
{Devuelve el nº de elementos en el diccionario d}

esVacío?: diccionario d -> booleano
{Devuelve verdad si y sólo si d no tiene elementos}

{Las cinco operaciones siguientes son un Iterador definido para los diccionarios}
iniciarIterador: diccionario d -> diccionario
{Inicializa el iterador para recorrer los pares del diccionario d, de forma que el siguiente par a visitar sea el primero que visitamos (situación de no haber visitado ningún par).}

existeSiguiente?: diccionario d -> booleano
{Devuelve falso si ya se han visitado todos los pares de d, devuelve verdad en caso contrario}

parcial siguienteClave: diccionario d -> clave
{Devuelve la clave del siguiente par a visitar de d.
Parcial: la operación no está definida si no existeSiguiente?(d)}

parcial siguienteValor: diccionario d -> valor
{Devuelve el valor del siguiente par a visitar de d.
Parcial: la operación no está definida si no existeSiguiente?(d)}

parcial avanza: diccionario d -> diccionario
{Prepara el iterador para visitar el siguiente par del diccionario d.
Parcial: la operación no está definida si ya se ha visitado el último par.}
fespec

En las implementaciones del TAD tabla, se suelen juntar en una única las operaciones "está?" y "obtenerValor".

2. Repaso de implementaciones ya vistas

En el tema sobre **tipos lineales** ya repasamos un par de implementaciones **estáticas** sencillas (basadas en un **vector** desordenado u ordenado) del TAD diccionario, y vimos en detalle una implementación **dinámica** basada en **listas enlazadas** con punteros, ordenadas por claves.

En el tema sobre **árboles** vimos en detalle una implementación con **árboles de búsqueda binarios** (no equilibrados), y también vimos las ideas sobre varias implementaciones con **árboles de búsqueda equilibrados** (binarios, como los **AVL**, o ***n*–arios**, como los **árboles B** y sus variantes).

También vimos en el tema de árboles que si el dominio de las claves es una secuencia de símbolos (por ejemplo, cadenas de caracteres), se puede implementar el TAD diccionario mediante un **árbol lexicográfico**.

Además, existe un **caso muy especial (por poco frecuente)** que es aquél en el que el **dominio de las claves** coincida con **valores consecutivos de un tipo de dato escalar y enumerable que pueda ser usado como tipo de los índices de un vector**, por ejemplo, los naturales desde 1 hasta un cierto valor *max*. En ese caso, el diccionario se puede almacenar en un vector `v[1..max]` de valores, o, de forma más precisa, de registros con dos campos: uno de tipo valor y otro booleano para

indicar si esa clave existe o no en el diccionario. En esa representación, las claves ni siquiera se almacenan. Se denomina **representación de acceso directo** y el coste de las operaciones de búsqueda, inserción y borrado está en $O(1)$.

A modo de resumen, la siguiente tabla incluye los **costes en el caso peor** de las operaciones fundamentales para las representaciones comentadas.

Representación	insertar	modificar valor	obtener el valor	borrar
Acceso directo (vector)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Vector ordenado	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Lista enlazada ordenada	$O(n)$	$O(n)$	$O(n)$	$O(n)$
árbol de búsqueda equilibrado	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Como puede verse, siempre que sea posible, es preferible la representación de acceso directo. Pero la mayor parte de las veces, no es posible pues el dominio de las claves no puede usarse directamente como índices de un vector. En esos casos, la representación de árbol de búsqueda equilibrado resulta la mejor en cuanto a costes temporales en el caso peor.

Si no se necesita garantizar un coste en el caso peor pequeño y basta con un coste en el caso promedio, existe una alternativa útil que es la de tabla dispersa (en inglés, *hash*), que vemos a continuación.

3. Tablas dispersas (o *hash*)

Con frecuencia, al intentar representar una tabla ocurre que no es posible encontrar una **función inyectiva de codificación de las claves en un subrango pequeño de enteros, `1..max`**, y por tanto no se puede utilizar la representación de acceso directo descrita en la sección anterior.

Sin embargo, casi siempre es posible encontrar una función:

$$h : \textit{Dominio de las claves} \rightarrow 1..max$$

si se permite que h pueda ser **no inyectiva**.

Además, podemos buscar tal función h que **distribuya** las claves esperadas **del modo más uniforme posible** en el rango `1..max`. De esta forma, la probabilidad de que, para dos claves distintas, $c_1 \neq c_2$, se cumpla $h(c_1) = h(c_2)$, será lo más baja posible.

No obstante, encontrar una función que distribuya bien no es fácil. Como ejemplo, la conocida **paradoja del cumpleaños** afirma que, si en una habitación están presentes veintitrés personas o más, lo más probable es que haya al menos dos de esas personas cuyo cumpleaños sea el mismo día (en otras palabras, si seleccionamos una función al azar que aplica 23 claves en una tabla de tamaño 365, la probabilidad de que no haya dos claves en la misma posición es tan sólo 0.4927, es decir, menos que 1/2).

Se denomina **tabla dispersa** (en inglés, *hash table* o *hash map*) a una representación del TAD diccionario (o mapa, o tabla) basada en la utilización de una función de codificación no inyectiva, como se ha expuesto previamente, para acceder a la posición de un vector en la que almacenar cada par.

Para definir una tabla dispersa, el programador debe tomar dos decisiones independientes:

1. **Elegir la función**

$$h : Dominio\ de\ las\ claves \rightarrow 1..max$$

no inyectiva, que se denomina **función de dispersión** (o función de localización, de desmenuzamiento, o de transformación, según otras traducciones del término original inglés *hashing function*).
Esta función se utiliza para calcular la **entrada primaria**, es decir, la posición en el vector `v[1..max]` de pares `<clave,valor>` que, en primera instancia, le corresponde a una clave dada `c`.
Como *h* no es inyectiva es posible que la entrada primaria de la clave de un par a insertar esté ocupada por otro par con clave diferente. Este hecho se conoce por el nombre de **colisión**.

2. **Seleccionar un método para resolver las colisiones**.
El método puede consistir en situar los pares que colisionan sobre una misma entrada primaria en una **zona de desbordamiento**, consistente en una estructura de datos dinámica (**una lista enlazada, por ejemplo**) guardando la dirección del primer par en la componente del vector soporte correspondiente a esa entrada primaria. Esta técnica recibe el nombre de **resolución de colisiones por encadenamiento** (en algunos libros denominada **dispersión abierta**).
Por el contrario, la **recolocación en el mismo vector soporte** consiste en almacenar los pares que colisionan en otras posiciones del vector diferentes a la entrada primaria. En este caso, se precisa determinar un **algoritmo de recolocación** que calcule sucesivas **entradas secundarias** para una clave, hasta encontrar un lugar libre en el vector. En algunos libros se denomina **dispersión cerrada** a esta solución, en otros, **direccionamiento abierto**.

3.1. Diseño de la función de dispersión

Veamos más en detalle el primero de los pasos. Las **características fundamentales** que debe tener una función de dispersión son:

- la capacidad de **distribuir** las claves de la forma más **uniforme** posible sobre el rango `1..max`, y
- la posibilidad de ser **evaluada eficientemente** (es decir, debe consistir en la aplicación de unas pocas operaciones aritméticas sencillas).

Para **claves de tipo natural**, z, una función de dispersión muy utilizada y sencilla, y que distribuye bastante bien las claves se denomina **método de la división** o método del módulo.

$$h(z) = z \bmod max$$

En este caso, los índices del vector en el que se almacena la tabla o diccionario son: `0..max-1`.
Es habitual elegir un número primo para el valor *max*.
En ocasiones, este método puede distribuir mal las claves. Por ejemplo, si $max = 10$ y casi todas las claves acaban en el mismo dígito.

Otra alternativa para definir una función de dispersión sencilla para números naturales, y de uso frecuente, se basa en el **método del centro del cuadrado**. Consiste en elevar el número al cuadrado y extraer unos cuantos dígitos de la parte central del número resultante.

$$h(z) = \text{“los } \log_2(max) \text{ bits centrales de } z^2\text{”}$$

Por ejemplo, si $max = 100$ y la clave natural, z, tiene 5 dígitos, su cuadrado tiene 10 (ó 9) dígitos y $(z^2 / 10^5) \bmod 100$ es el entero formado por los dos dígitos centrales del cuadrado de z, y por tanto está comprendido entre 0 y 99.

Si las **claves son de tipo cadena de caracteres**, la función de dispersión *h* debe **en primer lugar transformarlas en un número natural**, y después aplicar alguna de las soluciones conocidas para claves naturales (método de la división, método del centro del cuadrado…).

Un primer método, el más sencillo y eficiente, para transformar una cadena en un natural es el de la **suma de los ordinales**. Es decir, dada la cadena $c = c[1..n]$ de n caracteres, calcular ord(c[1]) + ord(c[2]) + … + ord(c[n]), donde la función "ord" devuelve el ordinal de un carácter en el código de caracteres utilizado (ASCII, por ejemplo).

En ocasiones, ese método sencillo combinado con el método de la división puede dar resultados de distribución de las claves muy pobres. Por ejemplo, si *max* es demasiado grande, *max* = 10007 y todas las cadenas tienen 8 caracteres, o menos, sólo se utilizarían un máximo de 2048 posiciones del vector (256 posibles*8 = 2048). O, por ejemplo, si *max* = 100 y las claves son las cadenas "A0", "A1", …, "A99", es fácil verificar que la función *h*, basada en la suma de los ordinales y en el método de la división, intentaría colocar las 100 claves en tan sólo 29 componentes del vector (por ejemplo, las 9 claves "A18", "A27", "A36", …, "A90" tendrían la misma entrada primaria).

Una alternativa muy utilizada al método de la suma de los ordinales es la de la **suma ponderada**. Consiste en tratar la cadena de caracteres como si fuera un entero en base 256 (si éste es el número de caracteres del código) al realizar la suma en la función *h* (ejemplo: a la cadena "abcd" le correspondería ord('a') + ord('b')*256 + ord('c')*256^2 + ord('d')*256^3).

Es decir, **el método de la suma ponderada, combinado con el de la división**, se implementaría con la siguiente función:

```
constantes cardinalCódigo = 256 {Suponemos el código ASCII, con 256 valores posibles}
           max = ...  {El tamaño del vector con el que implementar la tabla dispersa}

función h(c:cadena) devuelve 0..max-1
variables i,suma,factor:entero
principio
  suma:=0; {Para calcular la suma ponderada explicada anteriormente}
  factor:=1; {Valor inicial de factor = cardinalCódigo elevado a 0 = 1}
  para i:=1 hasta long(c) hacer
    suma:=suma + ord(c[i])*factor;
    factor:=factor*cardinalCódigo {factor = cardinalCódigo elevado a i}
  fpara;
  devuelve (suma mod max) {Método de la división}
fin
```

El problema de este método es que el valor de la variable suma puede hacerse muy grande. Si las claves sólo se componen de letras, puede tratarse la cadena de caracteres como un entero en base 26 (número de letras del alfabeto inglés). Otra posibilidad es descomponer la clave en subcadenas más pequeñas (por ejemplo, de 4 caracteres), calcular el entero correspondiente a cada subcadena, sumar los enteros resultantes y, finalmente, aplicar el método de la división.

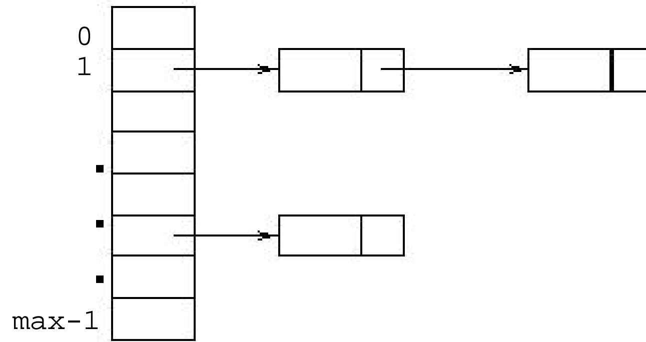
Una discusión detallada sobre estos y otros métodos para el diseño de una buena función de dispersión puede encontrarse en el volumen III de la enciclopédica obra de Donald E. Knuth, *El arte de programar ordenadores* (editorial Reverté, 1987). También puede consultarse el capítulo 9 del libro *Handbook of Data Structures and Applications*, de D.P. Mehta y S. Sahni (editorial *Chapman & Hall/CRC*, 2005).

3.2. Métodos de resolución de colisiones

La **resolución de colisiones por encadenamiento** (o **dispersión abierta**) consiste en que todos los pares a cuya clave es asignada por la función de dispersión una misma entrada primaria se almacenan en una lista enlazada mediante punteros. La dirección del primer elemento de la lista se guarda en la componente del vector correspondiente a la entrada primaria. La zona de memoria (dinámica) en la que se almacenan las listas enlazadas se llama **zona de desbordamientos**.

A continuación, se incluye un módulo de implementación de una tabla dispersa genérica (de pares <clave,valor>) con resolución de colisiones por encadenamiento.

h : *Dominio de las claves* → 0..*max*-1




```
módulo tablasDispersas
parámetros
  tipos clave,valor
exporta
  tipo diccionario {El TAD diccionario habitual, para parejas <clave,valor>}

  procedimiento crearVacío(sal t:diccionario)
  {Devuelve un diccionario (o tabla) vacío}

  procedimiento añadir(e/s t:diccionario; ent c:clave; ent v:valor)
  {Devuelve la tabla resultante de añadir el par (c,v) a t;
   si en t ya había un par (c,v'), entonces lo sustituye por el par (c,v)}

  procedimiento buscar(ent t:diccionario; ent c:clave;
                       sal está:booleano; sal v:valor)
  {está? y obtenerValor se implementan en una única operación}

  procedimiento borrar(e/s t:diccionario; ent c:clave)
  {Dada una clave c, la borra de la tabla junto con su valor;
   si c no estaba en t, t se queda igual}

implementación {tabla dispersa con resolución de colisiones por encadenamiento}
  constante max = 997 {por ejemplo (un primo cercano a 1000)}
  tipos ptNodo = ↑nodo;
        nodo = registro
                 laClave:clave;
                 elValor:valor;
                 sig:ptNodo
               freg;
        diccionario = vector[0..max-1] de ptNodo

  función h(valor c:clave) devuelve 0..max-1
  {Función de dispersión que distribuye uniformemente el
   dominio de las claves en el subrango 0..max-1.}
  ... {Implementación de alguna de las funciones de dispersión descritas antes}

  procedimiento crearVacío(sal t:diccionario)
  variable i:0..max-1
  principio
    para i:=0 hasta max-1 hacer
      t[i]:=nil
    fpara
  fin

  procedimiento añadir(e/s t:diccionario; ent c:clave; ent v:valor)
  variables aux:ptNodo;
            entrada:0..max-1
  principio
    entrada:=h(c);
    si t[entrada]=nil entonces
      nuevoDato(t[entrada]);
      aux:=t[entrada];
      aux↑.laClave:=c;
      aux↑.elValor:=v;
```

```
      aux↑.sig:=nil
    sino
      aux:=t[entrada];
      mientrasQue (aux↑.laClave≠c) and (aux↑.sig≠nil) hacer
        aux:=aux↑.sig
      fmq;
      si aux↑.laClave=c entonces
        aux↑.elValor:=v
      sino
        nuevoDato(aux↑.sig);
        aux:=aux↑.sig;
        aux↑.laClave:=c;
        aux↑.elValor:=v;
        aux↑.sig:=nil
      fsi
    fsi
  fin

  procedimiento buscar(ent t:diccionario; ent c:clave;
                       sal está:booleano; sal v:valor)
  variable aux:ptNodo
  principio
    aux:=t[h(c)];
    si aux=nil entonces
      está:=falso
    sino {buscar en la zona de desbordamiento (lista no ordenada)}
      mientrasQue (aux↑.laClave≠c) and (aux↑.sig≠nil) hacer
        aux:=aux↑.sig
      fmq;
      si aux↑.laClave=c entonces
        está:=verdad;
        v:=aux↑.elValor
      sino
        está:=falso
      fsi
    fsi
  fin

  procedimiento borrar(e/s t:diccionario; ent c:clave)
  variables entrada:0..max-1;
            aux,sigAux:ptNodo;
            borrado:booleano
  principio
    entrada:=h(c);
    aux:=t[entrada];
    si aux≠nil entonces {lista no vacía}
      si aux↑.laClave=c entonces {borrar el primero}
        t[entrada]:=aux↑.sig;
        disponer(aux)
      sino {borrar otro o ninguno}
        borrado:=falso;
        mientrasQue (aux↑.sig≠nil) and not borrado hacer
          si aux↑.sig↑.laClave=c entonces
            sigAux:=aux↑.sig;
            aux↑.sig:=sigAux↑.sig;
            disponer(sigAux);
            borrado:=verdad
          sino
            aux:=aux↑.sig
          fsi
        fmq
      fsi
    fsi
  fin

fin {del módulo}
```

El **factor de carga** de la tabla se define como $\alpha = N / max$, donde N es el número de pares almacenados en la tabla o diccionario y *max* es la capacidad del vector. Si el factor de carga es pequeño la resolución de colisiones por encadenamiento es bastante rápida puesto que las listas de desbordamientos son cortas.

Volviendo a la paradoja del cumpleaños, si hay 365 personas en una habitación es muy probable que haya muchas parejas con el mismo cumpleaños, pero el número medio de personas con un cierto cumpleaños es sólo 1 (asumiendo que los cumpleaños se distribuyen uniformemente en los 365 días del año).

En general, si la tabla tiene dimensión *max* y hay N pares <clave,valor> almacenados, y la función de dispersión es uniforme, la longitud media de cada lista es $\alpha = N / max$. En el libro de Donald Knuth mencionado en la sección anterior se presenta la siguiente fórmula del número medio de comparaciones de claves que se realizan en una operación de búsqueda:

$$C = \left(1 - \frac{1}{max}\right)^N + \frac{N}{max} \approx e^{-\alpha} + \alpha\,, \quad \text{si la búsqueda no tiene éxito}$$

$$C = 1 + \frac{N-1}{2 * max} \approx 1 + \frac{1}{2}\alpha\,, \quad \text{si la búsqueda tiene éxito}$$

Como puede verse, **el coste promedio en tiempo** de una búsqueda es del orden del factor de carga de la tabla (tenga o no tenga éxito la búsqueda), y por tanto, si el tamaño del vector, *max*, es del mismo orden de magnitud que el número de claves, N, entonces el factor de carga es de orden constante y también el coste promedio de la búsqueda.

Es decir, **si la función de dispersión distribuye uniformemente (caso "ideal") y el número de claves es similar al tamaño del vector, el coste promedio de una búsqueda de clave es de orden constante**. Lógicamente, en la práctica, la función de dispersión no es "perfecta" y no distribuye del todo uniformemente, y eso puede empeorar el coste promedio.

La **resolución de colisiones por recolocación en el mismo vector soporte** (también conocida en algunos libros como **dispersión cerrada**, y en otros como **direccionamiento abierto**) consiste en almacenar los pares <clave,valor> cuya entrada primaria ya está ocupada en otra posición secundaria (libre) del vector. Para eso, se precisa implementar una **función de recolocación** que llamaremos *hh*.

Evidentemente, con esta representación, el factor de carga debe mantenerse siempre por debajo de 1, $\alpha = N / max \leq 1$, puesto que sólo se dispone de las *max* componentes del vector para almacenar los datos.

Un inconveniente común a todos los métodos de recolocación de este tipo es que los pares recolocados ocupan posiciones de otras futuras claves, y las colisiones tienden a "arracimarse" o amontonarse. A pesar de este problema, como veremos, el funcionamiento suele ser bueno, en la práctica, si el factor de carga se mantiene por debajo de 0'9.

La representación de la tabla en este caso se limita a:

```
constante max = 997 {por ejemplo (un primo cercano a 1000)}
tipos componente = registro
                     libre:booleano;
                     laClave:clave; {el tipo tiene implementadas las funciones de
                                      dispersión, h, y recolocación, hh}
                     elValor:valor
                   freg;
      diccionario = vector[0..max-1] de componente
```

Como ejemplo, el algoritmo de búsqueda del valor de una clave es el siguiente, en una primera aproximación:

```
procedimiento buscar(ent t:diccionario; ent c:clave; sal está:booleano; sal v:valor)
variables primaria,secundaria:0..max-1;
          i:1..max
principio
  primaria:=h(c); {uso de la función de dispersión}
  si t[primaria].libre entonces
    está:=falso
  sino
    si t[primaria].laClave=c entonces
```

```
        está:=verdad;
        v:=t[primaria].elValor
      sino {t[primaria].laClave≠c}
        i:=1; {contador de intentos de recolocación}
        secundaria:=(primaria + hh(i)) mod max; {uso de la función de recolocación}
        mientrasQue (not t[secundaria].libre) and (t[secundaria].laClave≠c) hacer (*)
          i:=i+1;
          secundaria:=(primaria + hh(i)) mod max
        fmq;
        si t[secundaria].libre entonces
          está:=falso
        sino
          está:=verdad;
          v:=t[secundaria].elValor
        fsi
      fsi
    fsi
  fin
```

(*) Ojo, esta implementación no evita posibles ciclos infinitos. Una implementación detallada debería tenerlo en cuenta.

En el algoritmo anterior, **`hh` es la función de recolocación**. La solución más sencilla para esta función es tomar `hh(i)=i`, y recibe el nombre de **recolocación lineal**. Con esa función, se genera una secuencia de entradas secundarias para una determinada clave, hasta encontrar una posición libre del vector o encontrar la clave.

Si $\alpha = N / max$ es el factor de carga de la tabla (es decir, N = número de pares <clave,valor> almacenados), puede demostrarse que el número medio de intentos (o comparaciones) hasta encontrar, con una recolocación lineal, la clave buscada es:

$$C = \frac{1 - \alpha/2}{1 - \alpha}$$

En la tabla siguiente se presentan algunos valores de C en función de α. Puede observarse que los resultados son muy buenos, independientemente del valor de N, siempre que α sea inferior a 0.75.

α	0,1	0,25	0,5	0,75	0,9	0,95
C	1,06	1,17	1,50	2,50	5,50	10,50

El problema fundamental de la recolocación lineal es que los elementos almacenados tienden a "amontonarse" alrededor de las entradas primarias, y este hecho empeora la eficiencia promedio de las operaciones posteriores de búsqueda o modificación.

La solución ideal sería escoger una función de recolocación, *hh*, que, de nuevo, distribuyese uniformemente las claves en las posiciones libres restantes. En la práctica, construir esa función sería costoso y son preferibles otros métodos más sencillos. Un método comúnmente utilizado es **tomar *hh* como una función cuadrática**:

```
hh(i) = i*i
```

Nótese que hay un método eficiente del cálculo de los sucesivos valores de la función anterior basado en su definición recursiva:

```
hh(i+1) = hh(i) + 2*i + 1
hh(0) = 0
```

y en que la función

```
d(i) = 2*i +1
```

también admite una definición recursiva:

```
d(i+1) = d(i) + 2
d(0) = 1
```

El uso de la **función de recolocación cuadrática** evita el amontonamiento alrededor de las entradas principales. El principal problema es que no permite acceder a todos los índices del vector, pudiéndose dar el caso de que **existan huecos libres y sin embargo no se encuentre ninguno** para almacenar un nuevo par. No obstante, puede demostrarse que, si se elige *max* como un número primo, la función cuadrática de recolocación permite acceder al menos a la mitad de las posiciones del vector, desde cualquier entrada primaria. Esto resuelve el problema puesto que, si el factor de carga no es muy grande, es muy raro (poco frecuente) que se presenten *max*/2 colisiones consecutivas.

Puede demostrarse que, si se usara una **función de recolocación perfectamente uniforme en las componentes vacías (caso "ideal")** el número medio de comparaciones para realizar una búsqueda sería:

$$C \approx \frac{-1}{\alpha} \ln(1-\alpha)$$

La tabla siguiente recoge algunos valores de C según el valor de α:

α	0,1	0,25	0,5	0,75	0,9	0,95	0,99
C	1,05	1,15	1,39	2,85	2,56	3,15	4,66

Aunque la función cuadrática da unos resultados ligeramente peores a los de esta tabla, existen otros métodos que se aproximan bastante bien a la recolocación uniforme (consúltese la bibliografía ya citada sobre tablas dispersas).

Un **problema adicional** de las técnicas de recolocación en el mismo vector soporte es la implementación de la operación de **borrado** de un par. Para ello hay que sustituir el campo booleano `libre` por otro que pueda tomar tres valores distintos: `libre`, `ocupado` y `borrado`. Así, borrar un elemento supone dar el valor `borrado` al campo correspondiente. Si se está realizando una búsqueda, las componentes "borradas" han de ser tratadas como si estuviesen "ocupadas" y con una clave distinta a la buscada. En la operación de inserción de un nuevo elemento, se pueden aprovechar las componentes "borradas" para realizar el almacenamiento, tratándolas como si estuviesen vacías, pero teniendo en cuenta de no dejar claves duplicadas (apariciones previas de la clave localizadas en posiciones posteriores en la cadena de recolocaciones).

Y otro problema común a cualquier representación de tablas dispersas, es que se trata de una estructura de datos pensada para accesos individuales por clave, pero **no resulta fácil (implementable de forma eficiente) realizar un recorrido de datos que visite las claves por orden creciente de su valor** (iterador que recorra las claves por orden). El diseño de un iterador requerirá un **recorrido secuencial de todas las componentes del vector**, para visitar las posiciones ocupadas, obteniendo así un recorrido en el que **las claves no se obtienen ordenadas**, o bien duplicar la memoria en una estructura de datos auxiliar y ordenada por valores crecientes de las claves, solamente para realizar el recorrido (ineficiente en tiempo y en espacio).

Como **recomendaciones finales** sobre la elección de una implementación u otra para el TAD diccionario o tabla, puede concluirse lo siguiente:

- Si **no es posible estimar a priori un valor máximo de** ***N*** (número de elementos almacenados en la tabla) o este valor puede ser muy grande, la representación basada en **árboles de búsqueda equilibrados**, o árboles lexicográficos (si el dominio de las claves es una secuencia de símbolos), es la mejor pues, al menos, garantiza un coste de las operaciones en ***O*****(log *N*) en el caso peor**.
- Si **se tiene una estimación a priori del valor de** *N* y no es muy grande, una **tabla dispersa** supone la mejor solución pues el **coste de las operaciones es prácticamente, en promedio, constante**; en cualquier caso, conviene **sobredimensionar el vector soporte** con entre un 10% y un 25% más de componentes que *N*.
- Si se utilizan **tablas dispersas** y van a ser **frecuentes las operaciones de borrado**, la resolución de colisiones por **encadenamiento** será preferible a la recolocación en el mismo vector soporte.

- Si se ha optado por una **tabla dispersa**, el coste de las operaciones se mantiene bajo si el factor de carga, $\alpha = N / max$, es pequeño. Si esto no ocurre, es decir, **si se llena demasiado la tabla** (si $N \geq 2*max$ en el caso de resolución de colisiones por encadenamiento, o si $N \geq 0.9*max$ en el caso de recolocación en el mismo vector soporte) se puede optar por la técnica de **redispersión** (*rehashing*, en inglés), que consiste en mover todos los datos a una **nueva tabla (vector) de, por ejemplo, doble capacidad**, lógicamente re-implementando una nueva función de dispersión, *h*, para calcular las nuevas entradas primarias en el nuevo vector.
- Si lo más **frecuente es necesitar un recorrido ordenado** por clave de todos los datos, una **tabla dispersa no es una buena solución**.

Lección 20

Tablas multidimensionales

Indice

1. Concepto y ejemplo introductorio
2. Especificación de tablas bidimensionales
3. Implementación con estructuras de listas múltiples

1. Concepto y ejemplo introductorio

Los tipos funcionales que consideraremos en esta lección representan **funciones multidimensionales** cuyo dominio es el producto cartesiano de **varios géneros de claves** y cuyo rango es el **género de los valores**:

f : Dominio de claves A × Dominio de claves B × … × Dominio de claves Z → Dominio de los valores

Al igual que con las tablas unidimensionales, la función *f* suele ser parcial, es decir, existen tuplas de claves sin valor asociado, es decir, no almacenadas en el diccionario.

Veamos un **ejemplo** sencillo en el que aparece de forma natural una **tabla bidimensional**. Considérese la relación entre **estudiantes** y **asignaturas** en la Escuela de Ingeniería y Arquitectura, en la que hay unos 5000 estudiantes y unas 500 asignaturas[1]. Cada estudiante está o ha estado matriculado en varias asignaturas y para cada una de ellas es necesario guardar alguna información: una ficha que contiene, fundamentalmente, sus **notas** en esa asignatura. De igual forma, puede decirse que en cada asignatura hay (o ha habido) varios estudiantes matriculados y es necesario almacenar las notas obtenidas.

En este ejemplo hay dos géneros de claves: las que sirven para distinguir a cada estudiante del resto y las que diferencian a las asignaturas. El valor asociado a un par de claves `<claveEstudiante,claveAsignatura>` es, en este caso, la ficha que contiene las notas de ese alumno en esa asignatura.

Una tabla como la anterior puede considerarse una **generalización de un vector bidimensional**, en el que los tipos de los índices son las claves de estudiantes y las claves de asignaturas.

La tabla bidimensional del ejemplo puede verse también como un conjunto o **colección de ternas** `<claveEstudiante,claveAsignatura,notas>`. En esta colección no podrá haber dos ternas que tengan simultáneamente la misma clave de estudiante y de asignatura. Pero no es éste el único conjunto que puede definirse a partir de la tabla.

Si **seleccionamos una clave de estudiante**, `claveEstudiante`, puede ser interesante considerar la colección de todos los pares `<claveAsignatura,notas>` tales que la terna `<claveEstudiante,claveAsignatura,notas>` pertenece a la tabla bidimensional. Esta colección de pares es una tabla unidimensional que representa toda la información (asignaturas que cursa o ha cursado y notas) del estudiante con clave `claveEstudiante`.

Simétricamente, si se selecciona una asignatura, `claveAsignatura`, el conjunto de todos los pares `<claveEstudiante,notas>` tales que la terna `<claveEstudiante,claveAsignatura,notas>` pertenece a la tabla bidimensional constituye una nueva tabla unidimensional que representa toda la información de esa asignatura (estudiantes matriculados y notas).

En definitiva, las **proyecciones de la tabla bidimensional sobre cada una de sus claves definen sendas tablas unidimensionales**, que pueden ser de interés (de utilidad).

[1] Dato no actualizado.

2. Especificación de tablas bidimensionales

Vamos a presentar una especificación de las tablas bidimensionales (los casos de dimensión mayor se plantearían de forma análoga). Se utilizará el TAD genérico tabla (unidimensional) de la lección anterior, concretándolo con dos argumentos en un TAD concreto. Puesto que se usarán dos TAD concretos tabla (unidimensional), los nombres de operación del TAD genérico tabla quedarán sobrecargados (es decir, un mismo nombre de operación podrá representar operaciones diferentes). En cualquier caso, la observación del género o tipo de sus operandos eliminará la confusión.

espec tablasBidimensionales
usa booleanos, diccionariosGenéricos *{el TAD diccionario, o tabla unidimensional, de la lección anterior}*
parámetros formales
 géneros claveA,claveB,valor
fpf
géneros
 {concretamos los diccionariosGenéricos o tablas unidimensionales para las proyecciones de la tabla bidimensional:}
 tablaA = diccionario(claveA,valor)
 tablaB = diccionario(claveB,valor)
 {nuevo género de diccionario o tabla bidimensional:}
 tablaAB *{Los valores del género tablaAB representan conjuntos de ternas <claveA,claveB,valor> en los que no se permiten claves repetidas, estando las claves formadas por dos partes: <claveA,claveB>)}*

operaciones
 crear: -> tablaAB
 {Devuelve una tabla bidimensional vacía}

 añadir: tablaAB t, claveA ca, claveB cb, valor v -> tablaAB
 {Devuelve una tabla igual a la tabla resultante de añadir la terna <ca,cb,v> a t; si en t ya había una terna <ca,cb,v'> entonces devuelve una tabla igual a la resultante de sustituir dicha terna por <ca,cb,v> en t}

 está?: claveA ca, claveB cb, tablaAB t -> booleano
 {Devuelve verdad si y sólo si en t hay alguna terna <ca,cb,v> para algún v}

 parcial obtenerValor: claveA ca, claveB cb, tablaAB t -> valor
 {Si en t existe alguna terna con el par de claves <ca,cb> devuelve el valor asociado a ellas en dicha terna; Parcial: la operación no está definida si not(está?(ca,cb,t))}

 quitar: claveA ca, claveB cb, tablaAB t -> tablaAB
 {Si está?(ca,cb,t), devuelve una tablaAB igual a la resultante de borrar de t la terna <ca,cb,v> que tiene dichas claves; si not(está?(ca,cb,t)), devuelve una tabla igual a t}

 proyectarA: tablaAB t, claveA ca -> tablaB
 {Devuelve una tablaB unidimensional con todos los pares <cb,v> tales que en la tabla bidimensional t existe una terna <ca,cb,v>}
 proyectarB: tablaAB t, claveB cb -> tablaA
 {Devuelve una tablaA unidimensional con todos los pares <ca,v> tales que en la tabla bidimensional t existe una terna <ca,cb,v>}

fespec

Al igual que se hace con las tablas unidimensionales, la implementación de las operaciones `está?` y `obtenerValor` suele hacerse con un único procedimiento que devuelva un booleano y un valor.

3. Implementación con estructuras de listas múltiples

Una implementación estática basada en una matriz o vector bidimensional de este tipo:

	Estructuras de datos y algoritmos	Organización de computadores	Sistemas operativos I
Alberto …			nota
Belén …	nota	nota	
Carmen …			nota
Fernando …	nota		
Ignacio …		nota	nota
Marta …	nota		nota

sería una mala solución si muchas de las parejas `<claveEstudiante,claveAsignatura>` no tienen valor (notas) asignado, es decir, no pertenecen a la tabla, pues se malgastaría mucha memoria.

Una **estructura de listas múltiples** es una colección de datos dinámicos enlazados mediante punteros en la que cada dato dinámico tiene más de un campo de tipo puntero y, por tanto, puede pertenecer a más de una lista a la vez (recordar, por ejemplo, la representación de árboles ordenados en la forma "primogénito-siguiente hermano", en la que cada nodo puede pertenecer a una lista de primogénitos y a otra lista de hermanos).

Veamos cómo se puede representar una tabla bidimensional mediante una estructura de listas múltiples, utilizando el ejemplo de los estudiantes y asignaturas de la EINA.

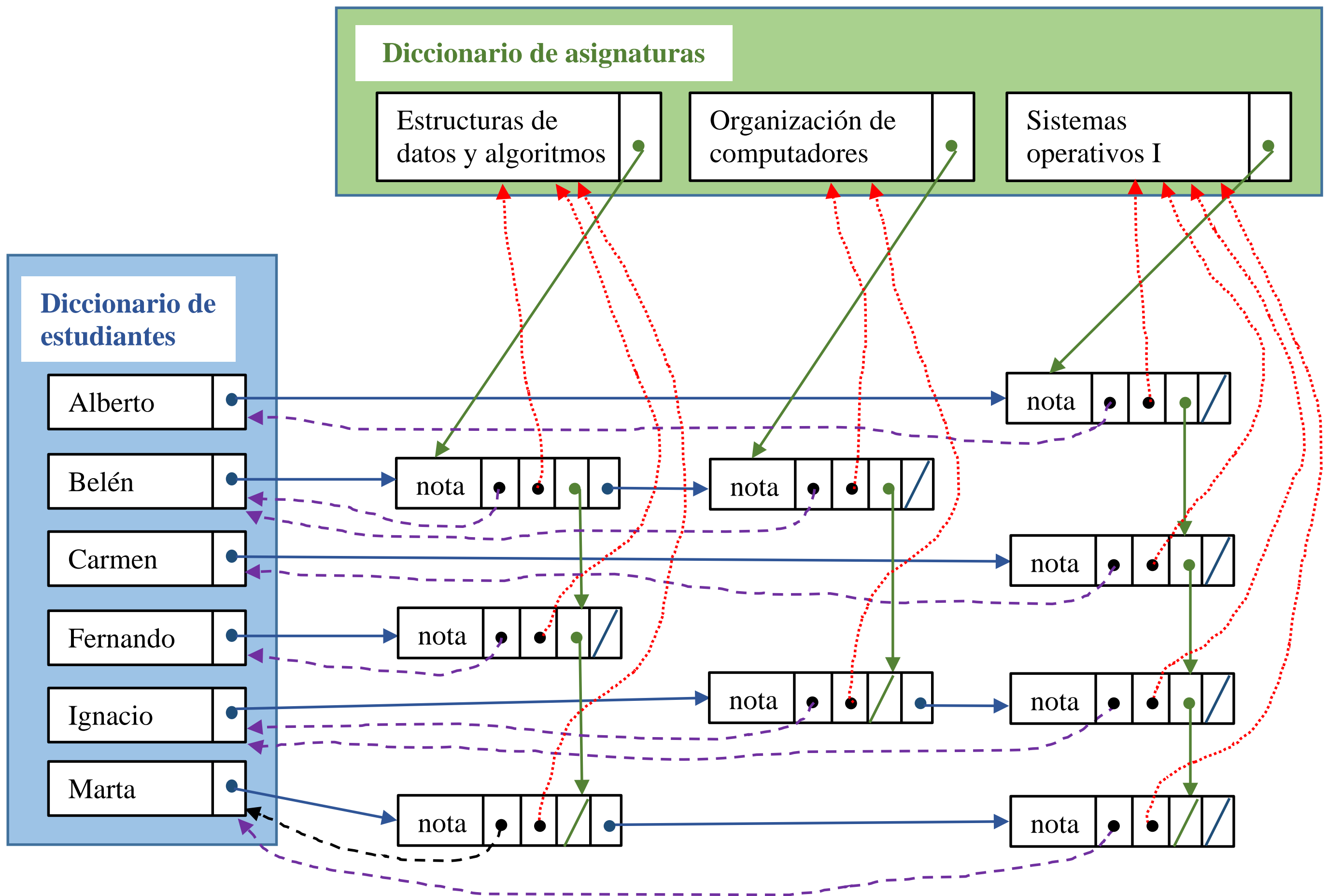
Cada registro de estudiante contendrá, además de sus datos personales (nombre, apellidos, número de matrícula, etc., a partir de los cuales se define la clave de estudiante) un **puntero al primer registro de matrícula** de ese estudiante en una asignatura. Este registro de matrícula guardará información sobre las notas de ese estudiante en esa asignatura.

De igual forma, **cada registro de asignatura** guardará, además de su información general (número de créditos, carácter optativo u obligatorio, etc.) un **puntero al primer registro de matrícula** de un estudiante en esa asignatura.

Cada registro de matrícula guarda, además de las notas de un estudiante en una asignatura, sendos **punteros al siguiente registro de matrícula de otro estudiante en la misma asignatura y al siguiente registro de matrícula del mismo estudiante en otra asignatura**.

De esta forma, un registro de matrícula no indica explícitamente el estudiante ni la asignatura que "empareja". Esa información está implícita en las dos listas "ortogonales" a las cuales pertenece. Esa limitación puede suponer un incremento considerable del coste en tiempo de las operaciones especificadas para el diccionario, por lo que es habitual **añadir, en cada registro de matrícula, la información mínima para poder averiguar eficientemente de qué estudiante y de qué asignatura se trata** (un cursor o un puntero al registro correspondiente de estudiante o asignatura, por ejemplo).

Por ejemplo, supónganse los conjuntos de estudiantes y asignaturas, con las matriculaciones señaladas con 'nota' que aparecen en la tabla de más arriba. Una implementación con estructuras de listas múltiples tendría la siguiente forma:



Por una parte, están sin detallar en la figura las estructuras de datos utilizadas para almacenar los **diccionarios o tablas unidimensionales de estudiantes y de asignaturas**. Dependiendo de sus características, se utilizaría alguna de las soluciones vistas con anterioridad para diccionarios unidimensionales (acceso directo, vector ordenado, lista ordenada, árbol de búsqueda, tabla dispersa…).

En cuanto a la **tabla bidimensional** en sí, puede observarse la disposición en una estructura de listas múltiples en la que, en **horizontal (punteros de color azul)** se representan listas para obtener la proyección **"notas obtenidas por un estudiante en todas las asignaturas en la que figura"**, mientras que en **vertical (punteros de color verde)** se representan listas para obtener la proyección **"notas de todos los estudiantes inscritos en una asignatura determinada"**. Por tanto, cada registro de esa tabla bidimensional pertenece a una lista horizontal y a otra vertical. Además, cada uno de esos registros guarda la forma de averiguar eficientemente a qué estudiante y a qué asignatura corresponde la nota que almacena; se ha optado por un **puntero al estudiante (color violeta)** y otro **puntero a la asignatura (color rojo)**. Esto presupone que los datos de los diccionarios unidimensionales de estudiantes y asignaturas están almacenados en memoria dinámica. Si por el contrario se almacenan en memoria estática (vector), en lugar de punteros a sus elementos se podrían usar cursores (variables de tipo natural indicando la componente del vector correspondiente).

Una declaración de tipos de datos (parcial, pues falta la decisión sobre los diccionarios de estudiantes y asignaturas) para esa representación sería la siguiente.

```
tipos estudiante = ... {información propia de un estudiante}
      asignatura = ... {información propia de una asignatura}
      nota   = ... {información propia de cada par <estudiante,asignatura>
                    (por ejemplo, incluye la nota)}

      ptNodoEstudiante = ↑nodoEstudiante;
      nodoEstudiante = registro
                         datoEstudiante:estudiante; {incluye una claveEstudiante}
                         primeraAsignatura:ptNodoNota
                       freg;

      ptNodoAsignatura = ↑nodoAsignatura;
```

```
nodoAsignatura = registro
                   datoAsignatura:asignatura; {incluye una claveAsignatura}
                   primerEstudiante:ptNodoNota
                 freg;

ptNodoNota = ↑nodoNota;
nodoNota = registro
             datoNota:nota;
             quéEstudiante:ptNodoEstudiante;
             quéAsignatura:ptNodoAsignatura;
             sigEstudiante,sigAsignatura:ptNodoNota
           freg

diccionarioEstudiantes = ... {representación de un diccionario unidimensional
                              de datos de tipo estudiante}
diccionarioAsignaturas = ... {representación de un diccionario unidimensional
                              de datos de tipo asignatura}
escuela = registro
            losEstudiantes:diccionarioEstudiantes;
            lasAsignaturas:diccionarioAsignaturas
          freg
```

Dada la estructura de datos anterior, para responder a una pregunta como "¿qué estudiantes están matriculados en la asignatura Estructuras de datos y algoritmos?, y escribir sus notas" (es decir, la proyección de notas fijada una asignatura), hay que hacer lo siguiente:

- buscar los datos de esa asignatura, dada su clave (en el campo `lasAsignaturas` de la variable de tipo `escuela` hasta llegar al registro de tipo `nodoAsignatura` que le corresponde); la implementación de esa búsqueda depende de cómo se haya representado el diccionario unidimensional de asignaturas (cualquiera de los métodos vistos hasta el momento sirve: vector de asignaturas, lista enlazada de asignaturas, árbol de búsqueda equilibrado de asignaturas, tabla dispersa de asignaturas…);
- luego, basta con seguir el puntero `primerEstudiante` para acceder al registro de la nota (de tipo `nodoNota`) del primer estudiante en esa asignatura; para saber de qué estudiante se trata, hay que seguir el puntero `quéEstudiante` hasta llegar a un registro de tipo `nodoEstudiante`;
- a continuación, pasar al siguiente registro de nota en la asignatura, siguiendo el puntero `sigEstudiante`, etcétera; así hasta llegar al final de la lista vertical correspondiente a notas de esa asignatura.

El siguiente algoritmo describe de forma un poco más precisa la solución:

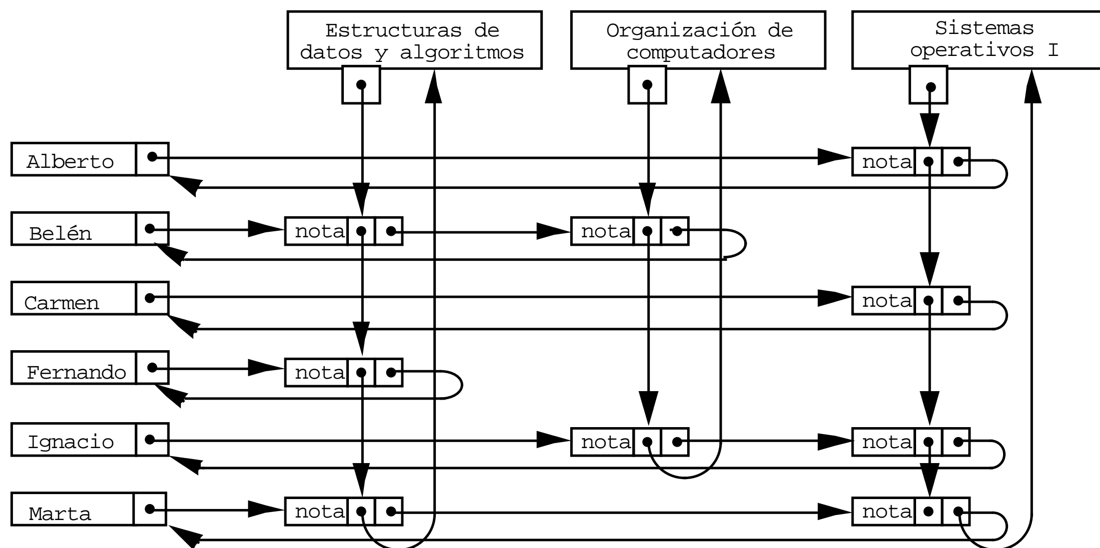
```
procedimiento escribeEstudiantes(ent EINA:escuela; ent c:claveAsignatura)
{Escribe en pantalla la información (notas) de los estudiantes matriculados en la
 asignatura de clave c.}
variables unaAsignatura:ptNodoAsignatura;
          unEstudiante:ptNodoNota
principio
  buscar(EINA,c,unaAsignatura); {devuelve en unaAsignatura un puntero al registro
                                 de asignatura correspondiente a la asignatura
                                 de clave c; su implementación depende de la
                                 representación del diccionario de asignaturas}
  unEstudiante:=unaAsignatura↑.primerEstudiante;
  mientrasQue unEstudiante≠nil hacer
    {unEstudiante apunta a un registro de nota de un estudiante en la asignatura
     de clave c}
    escribeEstudiante(unEstudiante↑.quéEstudiante); {escribe los datos personales
                                                     del estudiante almacenados en el
                                                     registro apuntado por ese campo}
    escribeNota(unEstudiante↑.datoNota); {escribe la nota}
    unEstudiante:=unEstudiante↑.sigEstudiante
  fmq
fin
```

Si en la estructura de datos de las listas múltiples **no se incluyen los punteros** `quéEstudiante` y `quéAsignatura` **que apuntan a los nodos correspondientes de los diccionarios unidimensionales** de estudiantes y asignaturas, **se ahorra memoria**, **pero no es posible saber con coste en *O*(1) de qué estudiante es esa nota** (primera instrucción del bucle en el algoritmo anterior) o, respectivamente, en qué asignatura, cuando se está implementando la proyección (vertical) `escribeEstudiantes` o, respectivamente, la (horizontal) análoga, `escribeAsignaturas`. Una forma de, al menos, poder hacerlo (con mayor coste en tiempo), consiste en convertir las listas horizontales y las verticales de la estructura de listas múltiples en **listas circulares** (horizontales y verticales), pero incluyendo en ellas los nodos de estudiante o de asignatura de los diccionarios unidimensionales. Así, si estamos accediendo a un nodo de tipo `nodoNota`, sería posible seguir su puntero en horizontal `sigAsignatura` hasta llegar al inicio de la lista circular, que sería el nodo de tipo `nodoEstudiante` del diccionario unidimensional de estudiantes. Evidentemente, esa implementación obliga a que todos los nodos (registros) utilizados para almacenar estudiantes, asignaturas y notas sean de un mismo tipo. Hay lenguajes que permiten hacer esto con **registros con campos variantes**, de forma que se definen tres variantes de registro (clase estudiante, clase asignatura, clase nota) y en cada una se almacenan exclusivamente los campos necesarios para cada variante. Si el lenguaje no permite la definición de registros con variantes, entonces todo nodo tiene que tener todos los campos de las tres variantes.

Si permitimos definir registros con variantes en nuestro pseudocódigo, la estructura de datos utilizada en esa solución alternativa sería:

```
tipos estudiante = ... {información propia de un estudiante}
      asignatura = ... {información propia de una asignatura}
      nota  = ... {información propia de cada par <estudiante,asignatura>
                      (por ejemplo, incluye la nota)}

      claseNodo = (estudiante,asignatura,nota)
      ptNodo = ↑nodo
      nodo = registro
               clase:claseNodo;
               selección
                 clase=estudiante: (datoEstudiante:estudiante;
                                    primeraAsignatura:ptNodo);
                 clase=asignatura: (datoAsignatura:asignatura;
                                    primerEstudiante:ptNodo);
                 clase=nota:       (datoNota:nota;
                                    sigEstudiante,sigAsignatura:ptNodo)
               fselección
             freg

      diccionarioEstudiantes = ... {representación de un diccionario unidimensional
                                     de datos de tipo estudiante, en el que cada
                                     estudiante se almacena en un registro de tipo nodo
                                     de clase estudiante}
      diccionarioAsignaturas = ... {representación de un diccionario unidimensional
                                     de datos de tipo asignatura, en el que cada
                                     asignatura se almacena en un registro de tipo nodo
                                     de clase asignatura}

      escuela = registro
                  losEstudiantes:diccionarioEstudiantes;
                  lasAsignaturas:diccionarioAsignaturas
                freg
```

Gráficamente, la representación de la estructura de listas múltiples para las definiciones anteriores de tipos resultaría como en la figura siguiente.



Con esta estructura de datos alternativa, el algoritmo de proyección vertical para responder a la pregunta: "¿qué estudiantes están matriculados en la asignatura Estructuras de datos y algoritmos?, y escribir sus notas", queda de la siguiente forma:

```
procedimiento escribeEstudiantes(ent EINA:escuela; ent c:claveAsignatura)
{Escribe en pantalla la información (notas) de los estudiantes matriculados en la
 asignatura de clave c.}
variables unaAsignatura,unEstudiante,aux:ptNodo
principio
  buscar(EINA,c,unaAsignatura); {devuelve en unaAsignatura un puntero al registro
                                 de asignatura correspondiente a la asignatura
                                 de clave c; su implementación depende de la
                                 representación del diccionario de asignaturas}
  unEstudiante:=unaAsignatura↑.primerEstudiante;
  mientrasQue unEstudiante↑.clase=nota hacer
    {unEstudiante apunta a un registro de nota de un estudiante en la asignatura
     de clave c}
    aux:=unEstudiante↑.sigAsignatura;
    mientrasQue aux↑.clase=nota hacer
      {recorrer, en horizontal, el resto de registros de nota de ese estudiante}
      aux:=aux↑.sigAsignatura
    fmq; {al terminar el bucle, aux apunta a un registro de clase estudiante}
    escribeEstudiante(aux); {escribe los datos personales del estudiante almacenados
                             en el registro apuntado por aux}
    escribeNota(unEstudiante↑.datoNota); {escribe la nota}
    unEstudiante:=unEstudiante↑.sigEstudiante
  fmq
fin
```

Como puede verse, el coste aumenta con respecto a la solución vista para la representación previa. Para cada nota de un estudiante en la asignatura dada, es preciso recorrer la lista horizontal de notas de ese estudiante (bucle interno del algoritmo; en el caso peor, lineal en el número total de asignaturas; en la práctica, lineal en el número de asignaturas en que está matriculado ese estudiante) hasta llegar al registro de clase estudiante correspondiente.

ANEXOS

Material adicional

Anexo 1: El TAD grafo y su representación en memoria

Indice

1. Conceptos básicos
2. Especificación
3. Representación con matriz de adyacencia
4. Representación con listas de adyacencia
5. Representación con listas múltiples de adyacencia

1. Conceptos básicos

Un **grafo** es una relación arbitraria entre objetos de un mismo tipo. Puede definirse formalmente como un par $G = (V,A)$, donde V es un conjunto de objetos llamados **vértices** (o nodos) y A es un conjunto de **aristas** (o arcos). Una arista es un par (u,v) de vértices de V.

El grafo se llama **dirigido** (o "digrafo") si sus aristas son pares ordenados. En este caso, dada una arista (u,v) (también representada como $u \rightarrow v$), su primer vértice u se llama **origen** (o cabeza) y su segundo vértice v, **destino** (o cola). Además, se dice que v es adyacente a u.

Si las aristas son pares no ordenados, el grafo se llama **no dirigido**. En este caso, si $(u,v) \in A$, entonces $(v,u) \in$ A y $(v,u) = (u,v)$. Se dice que u y v son **adyacentes** entre sí. En un grafo no dirigido no suelen permitirse aristas de la forma (u,u) con $u \in V$.

En muchas ocasiones es útil asociar información a cada arista de un grafo. En ese caso el grafo se dice **etiquetado** y se llama etiqueta de una arista a su información asociada. En ocasiones las etiquetas son valores numéricos que representan valores o costes asociados a las aristas, y se denominan **pesos**.

Por ejemplo, los vértices de un grafo no dirigido pueden representar ciudades con aeropuerto, las aristas servicios de vuelo de ida y vuelta entre ciudades, y el peso asociado a cada arista la duración del vuelo entre las dos ciudades correspondientes.

Un **camino** en un grafo $G = (V,A)$ es una secuencia de vértices $v_1,\ldots,v_n \in V$, $n \geq 1$, tal que $(v_i,v_{i+1}) \in A$ para $i = 1, \ldots, n - 1$. La **longitud** de un camino es su número de vértices menos uno.

Un camino es **simple** si todos sus vértices, excepto tal vez el primero y el último, son distintos. Un **ciclo** es un camino simple de longitud no nula que empieza y termina en el mismo vértice. Un grafo es **acíclico** si no tiene ningún ciclo. Un **subgrafo** de un grafo $G = (V,A)$ es otro grafo $G' = (V',A')$ tal que $V' \subseteq V$ y $A' \subseteq A$. Si A' consta de todas las aristas (u,v) de A tales que $u,v \in V'$, entonces el subgrafo se dice **inducido** (o generado por V').

Un grafo no dirigido se dice **conexo** si existe un camino entre cada par de vértices. Una **componente conexa** de un grafo no dirigido es un subgrafo conexo, inducido y maximal.

Sea $G = (V,A)$ un grafo dirigido. V se puede dividir en clases de equivalencia V_i, $1 \leq i \leq r$, tales que los vértices u y v son equivalentes si y sólo si existe un camino de u a v y otro de v a u. Sea A_i, $1 \leq i \leq r$, el conjunto de las aristas con origen y destino en V_i. Los grafos $G_i = (V_i,A_i)$ se llaman **componentes fuertemente conexas** de G. Un grafo dirigido con una sola componente fuertemente conexa se dice grafo **fuertemente conexo**.

Un **árbol libre** es un grafo no dirigido, conexo y acíclico. Si en un árbol libre se destaca un vértice, denominado **raíz**, el grafo se denomina **árbol** (ver la lección 11).

Un **árbol de recubrimiento** de un grafo no dirigido (o árbol abarcador o árbol de expansión, según distintas traducciones del término inglés *spanning tree*) es un árbol libre que es subgrafo del grafo original y que incluye todos sus vértices.

En un grafo no dirigido y con pesos, el cálculo del **árbol de recubrimiento de peso mínimo** (o árbol de expansión mínimo, que es el árbol de recubrimiento cuya suma de pesos es mínima) es un problema de especial interés. Si, por ejemplo,

los vértices representan ciudades; las aristas, las posibles líneas de comunicación entre ellas; y el peso de una arista, el coste de seleccionar esa línea de comunicación, un árbol de recubrimiento de peso mínimo representa una red de comunicaciones entre todas las ciudades que minimiza el coste total.

2. Especificación

Se ha definido un grafo como un par de conjuntos: de vértices y aristas (el segundo de ellos, de pares de elementos del primero). Por tanto, la signatura del TAD grafo debe contener como mínimo **operaciones para añadir vértices y aristas** a un grafo.

El resto de operaciones de la signatura depende de la aplicación que se vaya a dar al TAD. En la siguiente **especificación del TAD "grafo dirigido"** se incluyen operaciones de **pertenencia** y una operación que calcula los **vértices adyacentes** a uno dado.

espec grafosDirigidos
usa booleanos, conjuntosDeVértices *{es un TAD conjunto de elementos de tipo vértice}*
parámetro formal
  género vértice
  operación
    _ = _: vértice v1, vértice v2 **->** booleano *{una relación de igualdad entre vértices}*
fpf

género grafo *{sus valores son grafos genéricos no dirigidos cuyos vértices son de género vértice (parámetro)}*

operaciones
  crearVacío: -> grafo
  {Devuelve un grafo vacío, sin vértices (ni aristas)}

  añadirVértice: grafo g, vértice v -> grafo
  {Devuelve el grafo resultante de añadir el vértice v a g. Si en g ya estaba el vértice v, devuelve g}

  parcial añadirArista: grafo g, vértice v1, vértice v2 -> grafo
  {Si v1 y v2 son vértices de g, devuelve el grafo resultante de añadir la arista (v1,v2) al grafo g.
  Parcial: no está definida si v1 o v2 no son vértices de g}

  perteneceVértice: vértice v, grafo g -> booleano
  {Devuelve verdad si y sólo si v es un vértice de g}

  perteneceArista: vértice v1, vértice v2, grafo g -> booleano
  {Devuelve verdad si y sólo si (v1,v2) es una arista de g}

  adyacentes: grafo g, vértice v -> conjuntoDeVértices
  {Devuelve el conjunto de vértices adyacentes a v en g (es decir, los u tales que existe (v,u) en g)}

fespec

Para escribir la **especificación del TAD grafo no dirigido**, basta con modificar la semántica de la operación añadirArista, diciendo que la operación también añade el par (v2,v1).

3. Representación con matriz de adyacencia

Sea $G = (V,A)$ un **grafo dirigido** y supóngase $V = \{1,2,\ldots,n\}$. La **matriz de adyacencia** para G es una matriz A de dimensiones $n \times n$ de elementos booleanos en la que $A[i,j]$ es verdad si y sólo si existe una arista en G que va del vértice i al vértice j.

```
constante n = ... {cardinal de V}
tipo grafo = vector[1..n,1..n] de booleano
```

En el caso de un **grafo no dirigido**, la matriz de adyacencia tiene la particularidad de ser simétrica y los elementos de su diagonal son todos igual a falso.

La representación de la matriz de adyacencia es útil para aquellos algoritmos que precisan **saber si existe o no una arista entre dos vértices dados**.

En el caso de **grafos etiquetados** con pesos, los elementos de la matriz pueden ser enteros o reales (en lugar de booleanos) y representar el peso de la arista. Si no existe una arista de i a j, debe emplearse un valor que no pueda ser una etiqueta válida para almacenarse en $A[i,j]$.

Un **inconveniente** de la representación de la matriz de adyacencia es que requiere un **espacio** en $O(n^2)$ aunque el grafo tenga muy pocas aristas.

Dado un grafo $G = (V,A)$ representado mediante su matriz de adyacencia, la pregunta ¿$(u,v) \in$ A? se puede contestar en un tiempo en $O(1)$. Sin embargo, la operación *adyacentes*(g,v) necesita un tiempo en $O(n)$ (pues hay que recorrer una fila completa de la matriz). Más aún, para saber, por ejemplo, si un grafo no dirigido representado mediante su matriz de adyacencia es conexo, o simplemente para conocer el número de aristas, los algoritmos requieren un tiempo en $O(n^2)$, lo cual es más de lo que cabría esperar.

4. Representación con listas de adyacencia

Esta representación sirve para **mejorar la eficiencia** de los algoritmos sobre grafos que necesitan acceder a los **vértices adyacentes** a uno dado. Consiste en almacenar n listas enlazadas mediante punteros de forma que la lista i-ésima contiene los vértices adyacentes al vértice i.

```
constante n = ... {cardinal de V}
tipos ptNodo = ↑nodo
      nodo = registro
                 vértice:1..n;
                 sig:ptNodo
             freg
      grafo = vector[1..n] de ptNodo
```

La representación de un grafo con listas de adyacencia requiere un **espacio** del orden lineal del máximo entre n (el número de vértices) y a (el número de aristas, que para un grafo completo es $O(n^2)$ mientras que para un árbol es $O(n)$).

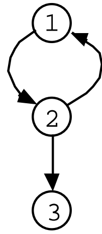
En el caso de un **grafo no dirigido**, si $(u,v) \in A$ entonces el vértice v estará en la lista correspondiente al vértice u, y el vértice u lo estará a su vez en la lista de adyacencia del vértice v.

Para representar un **grafo etiquetado** con pesos basta con añadir otro campo al tipo nodo que almacene el peso correspondiente.

Acceder a la lista de los **vértices adyacentes** a uno dado lleva un tiempo constante. Sin embargo, consultar si una determinada arista (u,v) está en el grafo precisa recorrer la lista asociada al vértice u, lo que en el caso peor se realiza en $O(n)$.

En el caso de un **grafo dirigido**, si se necesita saber a qué vértices es adyacente un vértice dado, la representación con listas de adyacencia no es adecuada. Ese problema puede superarse si se almacena otro vector de n listas de forma que la lista i-ésima contenga los vértices a los que es adyacente el vértice i. Estas listas se llaman **listas de adyacencia inversa**.

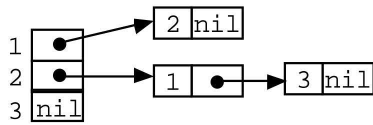
La siguiente figura ilustra la representación de un grafo dirigido con su matriz de adyacencia y con sus vectores de listas de adyacencia (directa e inversa):



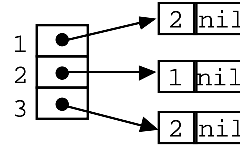
$$
A = \begin{bmatrix} \text{falso} & \text{verdad} & \text{falso} \\ \text{verdad} & \text{falso} & \text{verdad} \\ \text{falso} & \text{falso} & \text{falso} \end{bmatrix}
$$

Grafo dirigido

Matriz de adyacencia



Vector de listas de adyacencia

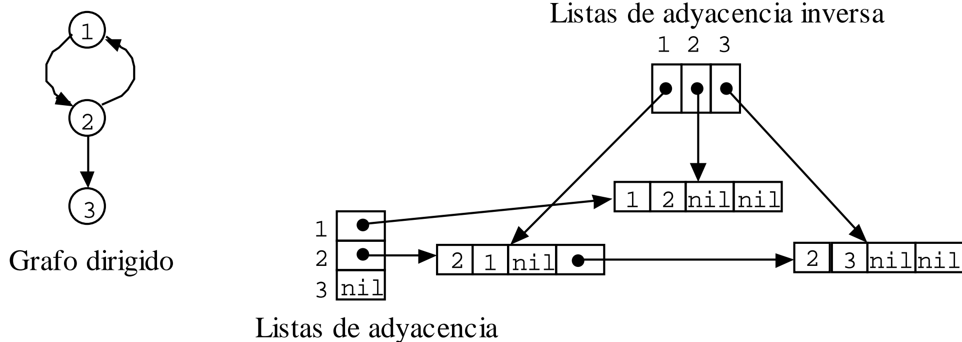


Vector de listas de adyacencia inversa

5. Representación con listas múltiples de adyacencia

En el caso de **grafos dirigidos**, las listas de **adyacencia directa e inversa** pueden representarse de forma más compacta.

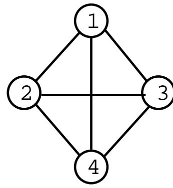
Como se definió en la lección 20, una **lista múltiple** es una estructura dinámica de datos enlazados mediante punteros en la que cada dato puede pertenecer a dos o más listas.



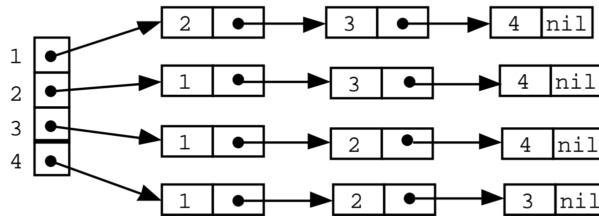
En esta representación, hay **un nodo por cada una de las aristas del grafo**. Cada nodo guarda la clave de dos vértices (origen y destino de la arista) y dos punteros: el primero al nodo que guarda la siguiente arista que tiene el mismo vértice destino y el segundo al nodo que guarda la siguiente arista que tiene el mismo vértice origen.

```
constante n = ... {cardinal de V}
tipos ptNodo = ↑nodo
       nodo = registro
                origen,destino: 1..n;
                sigInvAdy,sigAdy:ptNodo
              freg
       grafoDirigido = registro
                         adyacentes,invAdyacentes: vector[1..n] de ptNodo
                       freg
```

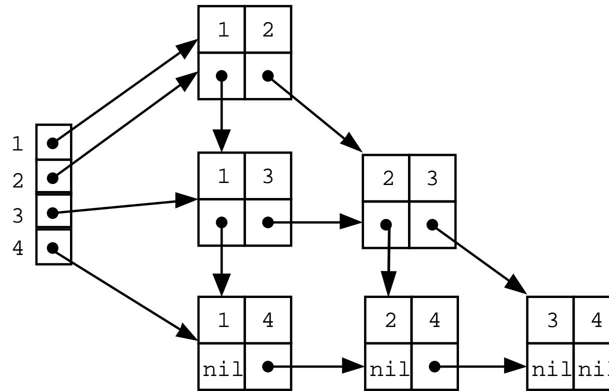
Las listas múltiples pueden utilizarse también para representar **grafos no dirigidos**. De esta forma, en lugar de que a cada arista le correspondan dos nodos en la estructura dinámica (como ocurre con la representación basada en listas "simples" de adyacencia), puede representarse cada arista con un solo nodo y hacer que éste pertenezca a dos listas de adyacencia diferentes. Véase el ejemplo siguiente:



Grafo no dirigido



Listas de adyacencia



Listas múltiples de adyacencia

La definición de la estructura de datos es la siguiente:

```
constante n = ... {cardinal de V}
tipos ptNodo = ↑nodo;
       nodo = registro
                v1,v2:1..n;
                sig1,sig2:ptNodo
              freg
       grafoNoDirigido = vector[1..n] de ptNodo
```

Una ventaja de esta representación es que es útil para implementar aquellos algoritmos que necesitan recorrer un grafo y al mismo tiempo “marcar” las aristas por las que se pasa (bastaría con añadir un campo booleano al tipo nodo).

Anexo 2: Algoritmos de vuelta atrás y árboles de juego

Indice

1. Introducción al esquema de vuelta atrás
2. Ejemplo: el problema de las ocho reinas
3. Ejemplo: recorrido de un laberinto
4. Árboles de juego: estrategia *minimax*

1. Introducción al esquema de vuelta atrás

Hay problemas que no admiten métodos directos y sencillos de solución como muchos de los tratados hasta el momento. En ocasiones es necesario aplicar la técnica o esquema general de **prueba y error**, que consiste en probar posibles soluciones al problema hasta encontrar la solución adecuada.

Un esquema muy útil de prueba y error es el de **vuelta atrás** o **retroceso** (en inglés, ***backtracking***). Consiste en **descomponer** la tarea de encontrar una solución en varias **subtareas**, de forma que cada una de ellas admita varias **candidatas a soluciones**. De esta forma se puede generar un **árbol** en el que las diferentes candidatas a soluciones de la primera subtarea son los elementos del primer nivel. Se elige el primer subárbol y se ensaya una solución a la segunda subtarea. De nuevo hay varias posibilidades que se encuentran en el siguiente nivel del árbol. De esta forma, la construcción de una candidata a solución completa del problema se obtiene siguiendo un camino en el árbol que empieza en la raíz y va descendiendo hasta llegar a una hoja. Es posible que en algún momento se descubra que la candidata tanteada no es la solución al problema. En ese momento se debe **retroceder en el árbol** hasta el nivel anterior y ensayar otra solución para la subtarea previamente planteada.

La exploración o **búsqueda exhaustiva** llevada a cabo en el árbol se denomina **implícita** porque el árbol nunca está íntegramente almacenado en memoria, simplemente se dispone de uno o varios nodos del árbol y de las reglas para acceder a nodos adyacentes (hijos o padre).

En resumen, en el esquema de vuelta atrás se intentan pasos hacia la solución completa del problema que son anotados, y en caso de que más tarde se descubra que un paso no conduce a una solución completa se borra la anotación correspondiente y se prueba otro.

De forma genérica, el **esquema de vuelta atrás** es el siguiente:

```
procedimiento ensaya
principio
  inicializar selección de candidatas;
  repetir
    seleccionar siguiente;
    si aceptable entonces
      registrarla;
      si solución incompleta entonces
        ensaya;
        si no exitoso entonces
          cancelar registro
        fsi
      fsi
    fsi
  hastaQue exitosa or no más candidatas
fin
```

En muchos casos, es necesario que el algoritmo de “ensayo” reciba como parámetro el índice del paso o subtarea que se va a resolver. Es frecuente también que el número de subtareas a realizar sea conocido (n) y que el número de candidatas a solución para cada subtarea sea fijo (m). Entonces, el esquema de algoritmo de vuelta atrás toma la siguiente forma:

```
procedimiento ensaya(ent i:entero)
variable k:entero
principio
  k:=0;
  repetir
    k:=k+1;
    seleccionar k-ésimo candidata;
    si aceptable entonces
      registrarla;
      si i<n entonces
        ensaya(i+1);
        si no exitoso entonces
          cancelar registro
        fsi
      fsi
    fsi
  hastaQue exitoso or (k=m)
fin
```

En el libro *Algoritmos + Estructuras de Datos = Programas*, de Niklaus Wirth, pueden encontrarse algunos **ejemplos clásicos** que admiten una solución con el esquema de vuelta atrás, como: la vuelta del caballo (encontrar un recorrido de un caballo por un tablero de ajedrez de forma que recorra todos los cuadros del tablero exactamente una vez), las ocho reinas (colocar ocho reinas en un tablero de ajedrez sin que ninguna de ellas pueda matar a otra), los matrimonios estables (dados dos conjuntos con igual número de elementos, encontrar una biyección entre ellos de forma que todas las parejas relacionadas por esa aplicación satisfagan ciertas propiedades), etc.

2. Ejemplo: el problema de las ocho reinas

El problema consiste en colocar **ocho reinas** sobre un **tablero de ajedrez** de forma que **no se amenacen**. Dos reinas se amenazan si comparten fila, columna o diagonal.

Puesto que no puede haber más de una reina por columna, podemos replantear el problema como: "**colocar una reina en cada columna** del tablero de forma que no se amenacen". En este caso, para ver si dos reinas se amenazan basta con ver si comparten fila o diagonal.

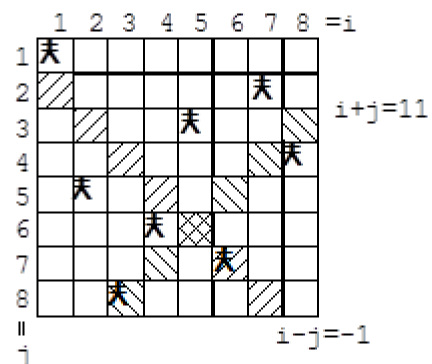
La siguiente decisión que debe tomarse es la **representación de los datos**. Esta debe permitir:

- interpretar la solución con facilidad, y para ello basta con tener un vector que almacene el índice de fila en que se coloca la reina en cada columna; y
- decidir con facilidad si una candidata a solución de una tarea (es decir, un índice de fila en el que colocar la reina de una columna) es aceptable o no (es decir, si existe otra reina ya colocada en la misma fila o diagonal).

Ambos objetivos se cumplen con la siguiente estructura de datos:

```
a:vector[1..8] de booleano;  {a[i]=niguna reina en fila i}
b:vector[2..16] de booleano; {b[i]=ninguna reina en diagonal-/ i}
c:vector[-7..7] de booleano; {c[i]=ninguna reina en diagonal-\ i}
x:vector[1..8] de entero     {x[i]=posición de la reina en la i-ésima columna}
```

Para entender los índices elegidos para los vectores `b` y `c`, nótese que todas las casillas (con índice de columna `i` e índice de fila `j`) de una misma diagonal de tipo '/' verifican que `i+j=constante`, y esta suma toma valores en el rango `2..16`. Por el contrario, todas las casillas (con índice de columna `i` e índice de fila `j`) de una misma diagonal de tipo '\' verifican que `i-j=constante`, y esta suma toma valores en el rango `-7..7`.



El siguiente algoritmo es la aplicación directa del esquema de vuelta atrás al problema de las ocho reinas en el tablero de ajedrez, considerando la representación de información detallada anteriormente.

```
procedimiento reinas
variables
  i:entero;
  q:booleano;
  a:vector[1..8] de booleano;  {a[i]=niguna reina en fila i}
  b:vector[2..16] de booleano; {b[i]=ninguna reina en diagonal-/ i}
  c:vector[-7..7] de booleano; {c[i]=ninguna reina en diagonal-\ i}
  x:vector[1..8] de entero     {x[i]=posición de la reina en la i-ésima columna}

  procedimiento ensaya(ent i:entero; sal q:booleano)
  variable j:entero
  principio
    j:=0;
    repetir
      j:=j+1;
      q:=falso;
      si a[j] and b[i+j] and c[i-j] entonces
        x[i]:=j;
        a[j]:=falso;
        b[i+j]:=falso;
        c[i-j]:=falso;
        si i<8 entonces
          ensaya(i+1,q);
          si not q entonces
            a[j]:=verdad;
            b[i+j]:=verdad;
            c[i-j]:=verdad
          fsi
        sino
          q:=verdad
        fsi
      fsi
    hastaQue q or (j=8)
  fin

principio
  para i:=1 hasta 8 hacer
    a[i]:=verdad
  fpara;
  para i:=2 hasta 16 hacer
    b[i]:=verdad
  fpara;
  para i:=-7 hasta 7 hacer
    c[i]:=verdad
  fpara;
  ensaya(1,q);
  para i:=1 hasta 8 hacer
    escribir(x[i])
  fpara
fin
```

La solución obtenida ejecutando este algoritmo es:

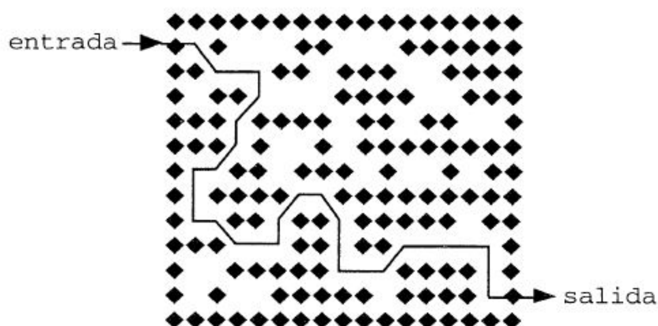
```
1 5 8 6 3 7 2 4
```

Esta solución es la descrita en la figura de la página anterior.

3. Ejemplo: recorrido de un laberinto

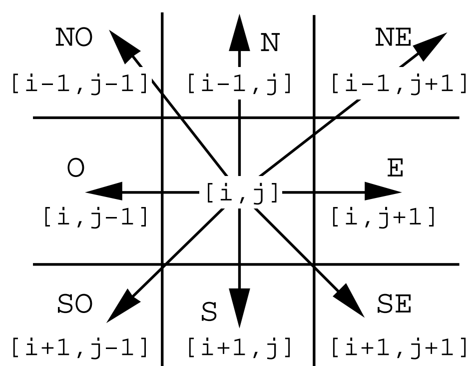
En este apartado va a utilizarse una pila como dato auxiliar para resolver el problema de encontrar la salida de un laberinto con un algoritmo de vuelta atrás. Existe una entrada al laberinto y, una vez dentro de él, el paseante se encuentra con paredes que le impiden el paso en muchas direcciones. Debe elegirse un camino, en caso de que no lleve a la salida hay

que volver atrás y continuar en una dirección diferente, y así hasta encontrar la única salida existente. El siguiente es un ejemplo de laberinto:



En primer lugar, para representar el laberinto podemos utilizar una estructura de datos consistente en una matriz de $m \times p$ componentes booleanas de forma que un valor falso en una componente indica que en la correspondiente posición del laberinto hay una pared, mientras que el valor verdad representa la existencia de un espacio libre. Supongamos que la entrada se realiza siempre por la componente 1,1 (como en la figura) y la salida por la componente *m*,*p*. La situación del paseante en el laberinto estará siempre descrita por la fila *i* y la columna *j* de su posición en la matriz.

Como puede verse, desde cada posición alcanzada en el interior del laberinto puede optarse por continuar en ocho direcciones diferentes (Norte, NorEste, Este, SurEste, Sur, SurOeste, Oeste y NorOeste):



No todas las posiciones *i*,*j* permiten moverse en ocho direcciones diferentes (si $i = 1$ ó m ó $j = 1$ ó p, entonces son menos los movimientos posibles). Para evitar ese problema, "orlaremos" la matriz que representa el laberinto con una pared (componentes con valor falso) en las filas 0 y $m + 1$ y en las columnas 0 y $p + 1$.

La solución al problema es la siguiente: al llegar a una nueva posición se examinan todas las posibles direcciones, desde la Este a la Noreste (en el sentido de las agujas de un reloj); cada vez que se realiza un movimiento, se guarda éste en una pila (posición y dirección del movimiento); si se llega a una posición desde la que no se puede seguir, hay que volver atrás, desapilando el último movimiento y realizando el siguiente posible. Para evitar pasar dos veces por el mismo camino se necesita almacenar en otra matriz de booleanos auxiliar (inicializada con valores falso) para marcar las posiciones por las que ya se ha pasado (con valor verdad).

```
constantes m = ...;
           p = ...
tipos laberinto = vector[0..m+1,0..p+1] de booleano;
      dirección = (E,SE,S,SO,O,NO,N,NE);
      movimiento = registro
                     fil:1..m;
                     col:1..p;
                     dir:dirección
                   freg

procedimiento inviertePila(ent p:pilaDeMov; e/s pI:pilaDeMov)
principio
  crearVacía(pI);
  mientrasQue not esVacía(p) hacer
    apilar(pI,cima(p));
```

```
    desapilar(p)
  fmq
fin

procedimiento camino(ent lab:laberinto)
{Escribe en pantalla, si existe, un camino del laberinto que va de la posición 1,1 a la
 posición m,p.}
variables hePasado:vector[1..m,1..p] de booleano;
          mov:movimiento;
          pila,pilaInv:pilaDeMov;
          éxito:booleano
principio
  {inicialización de la matriz de marcas a falso}
  para i:=1 hasta m hacer
    para j:=1 hasta p hacer
      hePasado[i,j]:=falso
    fpara
  fpara;
  {se parte de la casilla 1,1 hacia el Este}
  éxito:=falso;
  hePasado[1,1]:=verdad;
  mov.fil:=1;
  mov.col:=1;
  mov.dir:=E;
  crearVacía(pila);
  apilar(pila,mov);
  mientrasQue not esVacía(pila) and not éxito hacer
    {ver la posición actual y la dirección del movimiento}
    mov:=cima(pila);
    selección {cálculo de la nueva posición}
      mov.dir=N:  nuevaFil:=mov.fil-1;
                  nuevaCol:=mov.col;
      mov.dir=NE: nuevaFil:=mov.fil-1;
                  nuevaCol:=mov.col+1;
      mov.dir=E:  nuevaFil:=mov.fil;
                  nuevaCol:=mov.col+1;
      mov.dir=SE: nuevaFil:=mov.fil+1;
                  nuevaCol:=mov.col+1;
      mov.dir=S:  nuevaFil:=mov.fil+1;
                  nuevaCol:=mov.col;
      mov.dir=SO: nuevaFil:=mov.fil+1;
                  nuevaCol:=mov.col-1;
      mov.dir=O:  nuevaFil:=mov.fil;
                  nuevaCol:=mov.col-1;
      mov.dir=NO: nuevaFil:=mov.fil-1;
                  nuevaCol:=mov.col-1
    fselección;
    si (nuevaFil=m) and (nuevaCol=p) entonces  {se llega a la salida}
      mov.fil:=m;
      mov.col:=p;
      mov.dir:=E;
      apilar(mov);
      éxito:=verdad
    sino
      si lab[nuevaFil,nuevaCol] and not hePasado[nuevaFil,nuevaCol] entonces
        {nueva posición}
        hePasado[nuevaFil,nuevaCol]:=verdad;
        mov.fil:=nuevaFil;
        mov.col:=nuevaCol;
        mov.dir:=E;
        apilar(pila,mov)
      sino  {vuelta atrás}
        desapilar(pila);
        mientrasQue (mov.dir=NE) and not esVacía(pila) hacer
          mov:=cima(pila);
          desapilar(mov)
        fmq;
        si mov.dir<NE entonces
          mov.dir:=sucesor(mov.dir);
```

```
          apilar(p.mov)
        fsi
      fsi
    fsi
  fmq;
  si éxito entonces
    inviertePila(pila,pilaInv);
    mientrasQue not esVacía(pilaInv) hacer
      mov:=cima(pilaInv);
      desapilar(pilaInv);
      escribirLínea('Ir de ',mov.fil,',',mov.col,' hacia el ',mov.dir)
    fmq
  sino
    escribirLínea('¡No hay salida!')
  fsi
fin
```

4. Árboles de juego: estrategia *minimax*

Muchos **juegos de estrategia** se pueden representar en forma de árboles. Cada nodo se corresponde con una **configuración** posible del juego (por ejemplo, estado de un tablero) y cada arco es una transición legal, o **jugada**, desde una configuración posible a una de sus sucesoras.

Supongamos, por simplificar, que consideramos juegos en los que:

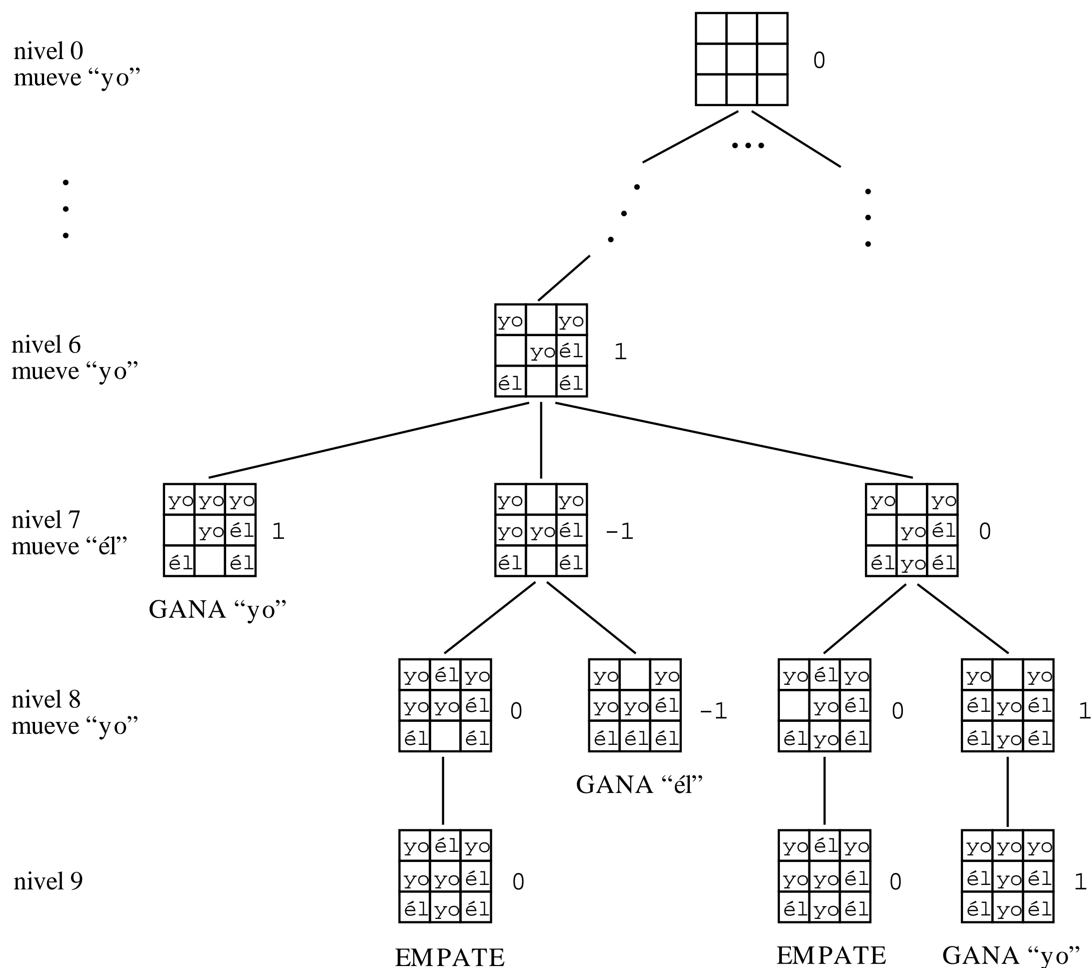
- los dos jugadores juegan alternadamente,
- los dos jugadores están sometidos a las mismas reglas (juegos simétricos),
- el azar no interviene (juegos deterministas),
- una partida no puede durar indefinidamente, y
- ninguna configuración tiene un número infinito de posibles sucesoras.

La **configuración inicial** del juego es la **raíz** del árbol correspondiente. Las **hojas** del árbol corresponden a las **configuraciones terminales** del juego, en las que no existe jugada siguiente, bien porque uno de los jugadores ha ganado o bien porque no ha ganado ninguno (situación de empate).

Los **niveles impares** del árbol están asociados con las configuraciones en las que **debe jugar uno de los dos jugadores**, mientras que los **niveles pares** se asocian a las configuraciones en las que **debe jugar el otro**.

A cada nodo del árbol se le asocia una etiqueta llamada **función de utilidad**. Por ejemplo, una función de utilidad habitual toma tres valores posibles: "configuración ganadora", "configuración perdedora" y "configuración empatadora o nula". La interpretación de la función de utilidad en cada configuración corresponde a la situación (o posibilidades) que tiene el jugador (tomamos partido por uno de ellos) en esa configuración, suponiendo que ninguno de los dos jugadores se equivocará y ambos realizarán en lo sucesivo la mejor jugada posible. La función de utilidad puede asignarse de forma sistemática.

Veamos, como ejemplo, el **juego del *tic-tac-toe*** (una especie de tres en raya con más de tres fichas por jugador). Llamaremos a los dos jugadores "él" y "yo". Supondremos, por ejemplo, que empieza "yo". El árbol tiene la forma siguiente:



Se ha anotado al lado derecho de cada nodo el valor correspondiente a la función de utilidad, representando con 1 la configuración ganadora, con 0 la empatadora y con -1 la perdedora. Para calcular la utilidad, se da valor, en primer lugar, a las hojas. La utilidad en una hoja vale 1, 0 ó -1 si la configuración del juego corresponde a una victoria, empate o derrota, respectivamente, del jugador por el que hemos tomado partido (“yo”). Los valores de la función de utilidad se propagan hacia arriba del árbol de acuerdo a la siguiente regla (**estrategia *minimax***):

- si un nodo corresponde a una configuración del juego en la que juega “yo” (nivel 0 ó par), se supone que ese jugador hará la mejor jugada de entre las posibles y, por tanto, el valor de la función de utilidad en la configuración actual coincide con el valor de esa función en la configuración de la mejor jugada posible (para “yo”) que se puede realizar desde la actual;
- si un nodo corresponde a una configuración del juego en la que juega “él” (nivel impar del árbol), se supone que ese jugador hará la mejor jugada de entre las posibles y, por tanto, el valor de la función de utilidad en la configuración actual coincide con el valor de esa función en la configuración de la peor jugada posible (para “yo”).

Si la raíz tuviera el valor 1, entonces “yo” tendría una estrategia que le permitiría ganar siempre. No es el caso del *tic-tac-toe*. En este juego la función de utilidad de la raíz vale 0, lo cual significa que ningún jugador tiene una estrategia ganadora, por lo que, si no se cometen errores, se puede garantizar al menos un empate. Si la raíz tuviera un valor -1, el otro jugador (“él”) tendría una estrategia ganadora.

La función de utilidad puede tener un rango más amplio (por ejemplo, los números enteros). Piénsese en el juego del **ajedrez**. El árbol del juego es tan grande (se ha estimado en más de 10^{100} nodos) que no se puede construir (si un computador pudiera generar 10^{11} nodos por segundo, necesitaría más de 10^{80} años) y dar valor a la función de utilidad de cada nodo de abajo hacia arriba (como en el caso del *tic-tac-toe*). Un programa de juego de ajedrez sencillo trabaja de la siguiente forma. Para cada configuración actual del juego, se toma dicha configuración como raíz del árbol. Se construyen varios niveles del árbol (el número depende de la velocidad del computador o puede ser seleccionado por el usuario). La mayor parte de las hojas del árbol construido son ambiguas (no indican triunfos, derrotas ni empates), por tanto, cada programa usa una función de utilidad de las posiciones del tablero que intenta estimar **la probabilidad de que el computador gane** desde esa posición (por ejemplo, la diferencia entre las piezas que quedan, el poder defensivo en el centro del tablero o alrededor de los reyes,

etc.). Así, el computador puede estimar la probabilidad de un triunfo después de hacer cada uno de sus siguientes movimientos posibles (en el supuesto de que el contrincante también hará su mejor jugada posible) y escoger el movimiento de mayor utilidad.

Para poder diseñar un algoritmo para jugar a un determinado juego contra el computador es necesario conocer los movimientos (o jugadas) válidos en el juego y las reglas de terminación.

Dada una configuración del juego en la que el computador deba jugar, basta con que el computador calcule la utilidad de las configuraciones accesibles (tras su jugada) y elija la jugada que conduzca a una configuración con mayor utilidad.

Es necesario resolver, fundamentalmente, lo siguiente:

a) la representación de una configuración del juego (definición del tipo de dato configuración),

b) la enumeración de las configuraciones accesibles desde una dada mediante la ejecución de una jugada, y

c) el cálculo de la función de utilidad para una configuración dada, sabiendo quién juega en ese momento.

El algoritmo que tiene mayor interés es el que ejecuta la jugada que realiza el computador:

```
procedimiento juegaElComputador(e/s c:configuración)
{c es una configuración no final; el algoritmo realiza la mejor jugada posible a partir
 de c, la comunica al usuario y actualiza la configuración c}
variables maxUtilidad,laUtilidad:entero;
          i:1..maxEntero;
          mejorJugada,unaJugada:configuración
principio
  i:=1;
  jugada(c,i,unaJugada); {calcula la 1ª configuración accesible desde c}
  mejorJugada:=unaJugada;
  maxUtilidad:=utilidad(mejorJugada,falso); {"falso" indica que cuando la configuración
                                             sea "mejorJugada", no juego yo}
  mientrasQue not esLaUltimaJugada(c,i) hacer
    i:=i+1;
    jugada(c,i,unaJugada);
    laUtilidad:=utilidad(unaJugada,falso);
    si laUtilidad > maxUtilidad entonces
      mejorJugada:=unaJugada;
      maxUtilidad:=laUtilidad
    fsi
  fmq;
  comunicaAlUsuario(mejorJugada);
  c:=mejorJugada
fin
```

El algoritmo principal, tras la inicialización de la configuración, contiene un bucle en el que mientras que la configuración no sea final se alternan una llamada al algoritmo anterior (`juegaElComputador`) y una petición de jugada al usuario.

La función utilidad se desarrolla a continuación:

```
procedimiento jugada(ent c:configuración; ent i:1..maxEntero;
                     sal unaJugada:configuración)
{Pre: c admite al menos i jugadas diferentes.}
{Post: unaJugada es la i-ésima jugada posible desde c.}
...
función esLaUltimaJugada(c:configuración; i:0..maxEntero) devuelve booleano
{Devuelve verdad si y sólo si c admite exactamente i jugadas diferentes.}
...
procedimiento comunicaAlUsuario(ent c:configuración)
{Muestra en pantalla la configuración c.}
...
función utilidadFinal(c:configuración) devuelve entero
{Pre: c es una configuración final del juego.}
{Post: devuelve la utilidad de c.}
...
función utilidad(c:configuración; juegoYo:booleano) devuelve entero
{Calcula la utilidad de c teniendo en cuenta quién juega.}
```

```
variables laUtilidad:entero;
          i:1..maxEntero;
          unaJugada:configuración
principio
  si esLaUltimaJugada(c,0) entonces
    devuelve utilidadFinal(c)
  sino
    i:=1;
    jugada(c,i,unaJugada);
    laUtilidad:=utilidad(unaJugada,not juegoYo);
    mientrasQue not esLaUltimaJugada(c,i) hacer
      i:=i+1;
      jugada(c,i,unaJugada);
      si juegoYo entonces
        laUtilidad:=max(laUtilidad,utilidad(unaJugada,falso))
      sino
        laUtilidad:=min(laUtilidad,utilidad(unaJugada,verdad))
      fsi
    fmq;
    devuelve laUtilidad
  fsi
fin
```

La función `utilidad` es la que implementa un recorrido exhaustivo del árbol de juego para encontrar la solución mediante la estrategia *minimax*. El algoritmo puede mejorarse fácilmente si se conocen los valores máximo y mínimo que puede tomar la función de utilidad: el bucle se termina en cuanto se accede a una configuración cuya utilidad es máxima (mínima) si el valor del parámetro `juegoYo` es verdad (falso).

Anexo 3: Introducción a los algoritmos voraces

Indice

1. Estrategias voraces
2. Aplicación al problema del recorrido del caballo de ajedrez
3. Aplicación al problema del viajante

1. Estrategias voraces

Un algoritmo **voraz** (o ávido; del inglés, *greedy*) se basa en construir la solución del problema dado en **sucesivos pasos** de forma que en cada paso la **solución parcial** adoptada sea **localmente óptima** en algún sentido.

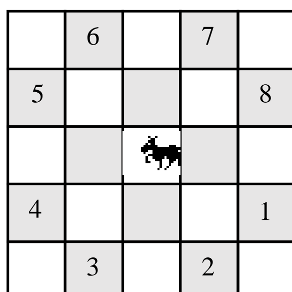
Un **ejemplo** cotidiano para el que aplicamos una solución voraz es el problema de dar el **cambio en monedas**, resultante tras el pago de una compra. Supongamos que disponemos en la caja de monedas de 100, 50, 25, 5 y 1 unidades monetarias, y tenemos que dar un cambio de 63 unidades, deseando hacerlo con el **menor número posible de monedas**. El procedimiento que seguimos consiste en seleccionar primero la moneda más grande cuyo valor no sea mayor que la cantidad a devolver, y para la cantidad restante aplicamos el mismo método (resultando una de 50, dos de 5 y tres de 1 unidad, es decir, un total de seis monedas). De esta forma, en cada paso intentamos dar el menor número posible de monedas (solución localmente óptima).

Una estrategia voraz, como la presentada en el ejemplo anterior, no necesariamente lleva a la solución del problema. En el caso del cambio en monedas, el algoritmo voraz lleva a la solución óptima (menor número de monedas) debido a las propiedades de los valores de las monedas (100, 50, 25, 5, 1). Si, por ejemplo, los valores de las monedas fueran de 11, 5 y 1 unidades y el cambio a devolver fuera de 15 unidades, el algoritmo voraz nos llevaría a seleccionar en primer lugar la moneda de 11 unidades, y el resto tendríamos que devolverlo con cuatro monedas de 1 unidad (es decir, con un total de cinco monedas). Sin embargo, la solución óptima consiste en devolver tres monedas de 5 unidades[1].

2. Aplicación al problema del recorrido del caballo de ajedrez

Consideremos el conocido problema del recorrido del caballo de ajedrez. Consiste en encontrar una sucesión de movimientos de forma que un caballo de ajedrez pueda visitar cada escaque (cuadro) del tablero una sola vez.

La solución más conocida se obtiene aplicando el esquema de **vuelta atrás**, introducido en el anexo anterior. Se utiliza una tablero de $n \times n$ enteros (con $n \leq 8$ leído interactivamente), inicialmente con todas las casillas iguales a cero (no visitadas). El caballo parte de una posición (i,j) solicitada interactivamente. Se almacena el entero 1 en la casilla (i,j), indicando el escaque inicial. El procedimiento ensaya realiza el siguiente movimiento posible (a una casilla no visitada). Para ello, ensaya sucesivamente con los ocho movimientos posibles:



Cada vez que se llega a un escaque no visitado, se almacena en la casilla correspondiente del tablero el número de escaques recorridos hasta el momento.

[1] ¿Qué propiedad deben tener los valores de las diferentes monedas para que la solución voraz sea siempre óptima?

```
procedimiento caballo
variables i,j,n,Nsqr:entero;
          q:booleano;
          dx:vector[1..8] de entero;
          dy:vector[1..8] de entero;
          h:vector[1..8,1..8] de entero

  procedimiento ensaya(ent i,x,y:entero; sal q:booleano)
  {Ensaya el movimiento al i-ésimo escaque desde el x,y. Si el movimiento es posible y
   tras él se puede seguir moviendo hasta encontrar la solución entonces q toma el valor
   verdad y falso en caso contrario.}
  variables k,u,v:entero
  principio
    k:=0;
    repetir {se ensaya con los 8 movimientos posibles}
      k:=k+1;
      q:=falso;
      u:=x+dx[k];
      v:=y+dy[k];
      si (1≤u) and (u≤n) and (1≤v) and (v≤n) entonces
        si h[u,v]=0 entonces
          h[u,v]:=i;
          si i<Nsqr entonces
            ensaya(i+1,u,v,q);
            si not q entonces
              h[u,v]:=0
            fsi
          sino
            q:=verdad
          fsi
        fsi
      fsi
    hastaQue q or (k=8)
  fin

principio
  dx[1]:=2; dx[2]:=1; dx[3]:=-1; dx[4]:=-2;
  dx[5]:=-2; dx[6]:=-1; dx[7]:=1; dx[8]:=2;
  dy[1]:=1; dy[2]:=2; dy[3]:=2; dy[4]:=1;
  dy[5]:=-1; dy[6]:=-2; dy[7]:=-2; dy[8]:=-1;
  para i:=1 hasta 8 hacer
    para j:=1 hasta 8 hacer
      h[i,j]:=0
    fpara
  fpara;
  escribir('Introduce n (1≤n≤8): '); leerLínea(n);
  escribir('Introduce i de la casilla inicial: '); leerLínea(i);
  escribir('Introduce j de la casilla inicial: '); leerLínea(j);
  Nsqr:=n*n; {número de escaques que hay que visitar}
  h[i,j]:=1;
  ensaya(2,i,j,q);
  si q entonces
    para i:=1 hasta n hacer
      para j:=1 hasta n hacer
        escribir(h[i,j])
      fpara;
      escribirLínea
    fpara
  sino
    escribir('No hay solución')
  fsi
fin
```

A continuación, se muestra un ejemplo de ejecución del algoritmo anterior:

```
Introduce n (1≤n≤8): 8
Introduce i de la casilla inicial: 1
Introduce j de la casilla inicial: 1
```

```
  1   60   39   34   31   18    9   64
 38   35   32   61   10   63   30   17
 59    2   37   40   33   28   19    8
 36   49   42   27   62   11   16   29
 43   58    3   50   41   24    7   20
 48   51   46   55   26   21   12   15
 57   44   53    4   23   14   25    6
 52   47   56   45   54    5   22   13
```

El problema de la solución de vuelta atrás para el problema del recorrido del caballo es su gran **ineficiencia** (la ejecución anterior tardó 7'24" en un computador con procesador 68LC040 50/25 MHz[2]). La razón del elevado tiempo de ejecución es el gran número de “vueltas atrás” que realiza el algoritmo (es decir, se recorren demasiados nodos del árbol de movimientos).

Una **heurística voraz** para el problema del recorrido del caballo consiste en seleccionar el siguiente escaque a visitar con la siguiente regla: **se elige aquel escaque no visitado desde el cual se puede acceder a un menor número de escaques no visitados**. La heurística se basa en la idea de que si ahora tenemos oportunidad de desplazarnos a un escaque “muy aislado” debemos hacerlo pues más adelante será más difícil llegar a él de nuevo. El siguiente algoritmo implementa la heurística anterior:

```
procedimiento caballo
variables x,y,i,j,n,Nsqr,k,kk,u,v,num,menor:entero;
          parar: booleano;
          dx:vector[1..8] de entero;
          dy:vector[1..8] de entero;
          h:vector[1..8,1..8] de entero

  función accesibles(x,y:entero) devuelve entero
  {Devuelve el número de escaques no visitados accesibles desde x,y}
  variables k,u,v,num:entero
  principio
    num:=0;
    para k:=1 hasta 8 hacer
      u:=x+dx[k];
      v:=y+dy[k];
      si (1≤u) and (u≤n) and (1≤v) and (v≤n) entonces
        si h[u,v]=0 entonces
          num:=num+1
        fsi
      fsi
    fpara;
    devuelve num
  fin

principio
  dx[1]:=2; dx[2]:=1; dx[3]:=-1; dx[4]:=-2;
  dx[5]:=-2; dx[6]:=-1; dx[7]:=1; dx[8]:=2;
  dy[1]:=1; dy[2]:=2; dy[3]:=2; dy[4]:=1;
  dy[5]:=-1; dy[6]:=-2; dy[7]:=-2; dy[8]:=-1;
  para i:=1 hasta 8 hacer
    para j:=1 hasta 8 hacer
      h[i,j]:=0
    fpara
  fpara;
  escribir('Introduce n (1≤n≤8): '); leerLínea(n);
  escribir('Introduce x de la casilla inicial: '); leerLínea(x);
  escribir('Introduce y de la casilla inicial: '); leerLínea(y);
  Nsqr:=n*n;
  h[x,y]:=1;
  i:=1;
  parar:=falso;
  mientrasQue (i<Nsqr) and not parar hacer
    i:=i+1;
    menor:=9;
```

[2] Esto debió ser en el año 1993 o 1994…

```
    para k:=1 hasta 8 hacer
      u:=x+dx[k];
      v:=y+dy[k];
      si (1≤u) and (u≤n) and (1≤v) and (v≤n) entonces
        si h[u,v]=0 entonces
          num:=accesibles(u,v);
          si num<menor entonces
            menor:=num;
            kk:=k
          fsi
        fsi
      fsi
    fpara;
    si menor=9 entonces
      parar:=verdad
    sino
      x:=x+dx[kk];
      y:=y+dy[kk];
      h[x,y]:=i
    fsi
  fmq;
  si not parar entonces
    para i:=1 hasta n hacer
      para j:=1 hasta n hacer
        escribir(h[i,j])
      fpara;
      escribirLínea
    fpara
  sino
    escribir('No encuentro solución');
  fsi
fin
```

La ejecución del algoritmo anterior para los mismos datos utilizados antes es la siguiente:

```
Introduce n (1≤n≤8): 8
Introduce x de la casilla inicial: 1
Introduce y de la casilla inicial: 1
   1   34    3   18   49   32   13   16
   4   19   56   33   14   17   50   31
  57    2   35   48   55   52   15   12
  20    5   60   53   36   47   30   51
  41   58   37   46   61   54   11   26
   6   21   42   59   38   27   64   29
  43   40   23    8   45   62   25   10
  22    7   44   39   24    9   28   63
```

Como puede verse, la solución encontrada es diferente de la anterior. Sin embargo, la gran diferencia se encuentra en el **tiempo de ejecución**. En este caso, el mismo computador necesita menos de 1 segundo (frente a los 7'24" de la solución de vuelta atrás). La razón es que **este algoritmo sólo recorre 64 nodos del árbol de movimientos** (los imprescindibles para llegar desde la raíz a una hoja, **sin realizar ninguna vuelta atrás**).

Si ensayamos otra ejecución del mismo algoritmo para los siguientes datos:

```
Introduce n (1≤n≤8): 5
Introduce x de la casilla inicial: 1
Introduce y de la casilla inicial: 3
No encuentro solucion
```

El algoritmo no encuentra solución. Sin embargo, existe solución. La siguiente es una ejecución del algoritmo de vuelta atrás para los mismos datos (18" de ejecución con el mismo computador):

```
Introduce n (1≤n≤8): 5
Introduce i de la casilla inicial: 1
Introduce j de la casilla inicial: 3
```

```
25   14    1    8   19
 4    9   18   13    2
15   24    3   20    7
10    5   22   17   12
23   16   11    6   21
```

Es decir, **la estrategia voraz no sirve en todos los casos**. Una solución razonable consiste en modificar el algoritmo de vuelta atrás, cambiando el orden de ensayo de las casillas accesibles desde una dada. En lugar de probar de forma consecutiva en el sentido de las agujas del reloj, se intentará en primer lugar a la casilla desde la que se pueda acceder al menor número de casillas no visitadas; si desde esa casilla no se llega a una solución, se intentará con la casilla con siguiente menor número de casillas accesibles, etcétera.

Para poder llevar a cabo la vuelta atrás, cada vez que se llegue a una casilla se almacenarán los movimientos posibles, es decir, sus casillas accesibles junto con el número de casillas no visitadas accesibles desde cada una de ellas en una cola de movimientos con prioridades (véase la lección 19). Supóngase que en el módulo colasDeMov están definidos los siguientes elementos:

```
tipo movimiento = registro
                    valor:1..8; {movimiento según figura al inicio de esta sección}
                    peso:0..8   {nº de casillas no visitadas accesibles si se realiza
                  freg           el movimiento}

función menor(m1,m2:movimiento) devuelve booleano
principio
  devuelve m1.peso<m2.peso;
fin

tipo colaDeMov {cola de movimientos con prioridades; un movimiento m1 tiene prioridad
                sobre otro m2 si menor(m1,m2)=verdad}
```

Entonces, el algoritmo de vuelta atrás modificado con una estrategia voraz para la elección del siguiente candidato a movimiento es el siguiente:

```
procedimiento caballo
importa colasDeMov
variables i,j,n,Nsqr:entero;
          q:booleano;
          dx:vector[1..8] de entero;
          dy:vector[1..8] de entero;
          h:vector[1..8,1..8] de entero

  procedimiento ensaya(ent i,x,y:entero; sal q:booleano)
  {Ensaya el movimiento al i-ésimo escaque desde el x,y. Si el movimiento es posible y
   tras él se puede seguir moviendo hasta encontrar la solución entonces q toma el valor
   verdad y falso en caso contrario.}
  variables k,u,v:entero;
            mov:movimiento;
            movimientos:colaDeMov

    función accesibles(x,y:entero) devuelve entero
    {Devuelve el número de escaques no visitados accesibles desde x,y}
    variables num,u,v,k:entero
    principio
      num:=0;
      para k:=1 hasta 8 hacer
        u:=x+dx[k];
        v:=y+dy[k];
        si (1≤u) and (u≤n) and (1≤v) and (v≤n) entonces
          si h[u,v]=0 entonces
            num:=num+1
          fsi
        fsi
      fpara;
      devuelve num
    fin
```

```
  principio {de ensaya}
    {se almacenan los movimientos posibles}
    crearVacía(movimientos);
    para k:=1 hasta 8 hacer
      u:=x+dx[k];
      v:=y+dy[k];
      si (1≤u) and (u≤n) and (1≤v) and (v≤n) entonces
        si h[u,v]=0 entonces
          mov.valor:=k;
          mov.peso:=accesibles(u,v);
          añadir(movimientos,mov)
        fsi
      fsi
    fpara;
    {se ensaya por orden de menor a mayor número de casillas dominadas no visitadas}
    q:=falso;
    mientrasQue not esVacía(movimientos) and not q hacer
      mov:=min(movimientos);
      eliminarMin(movimientos);
      k:=mov.valor;
      u:=x+dx[k];
      v:=y+dy[k];
      h[u,v]:=i;
      si i<Nsqr entonces
        ensaya(i+1,u,v,q);
        si not q entonces
          h[u,v]:=0
        fsi
      sino
        q:=verdad
      fsi
    fmq
  fin {de ensaya}

principio
  dx[1]:=2; dx[2]:=1; dx[3]:=-1; dx[4]:=-2;
  dx[5]:=-2; dx[6]:=-1; dx[7]:=1; dx[8]:=2;
  dy[1]:=1; dy[2]:=2; dy[3]:=2; dy[4]:=1;
  dy[5]:=-1; dy[6]:=-2; dy[7]:=-2; dy[8]:=-1;
  para i:=1 hasta 8 hacer
    para j:=1 hasta 8 hacer
      h[i,j]:=0
    fpara
  fpara;
  escribir('Introduce n (1≤n≤8): '); leerLínea(n);
  escribir('Introduce i de la casilla inicial: '); leerLínea(i);
  escribir('Introduce j de la casilla inicial: '); leerLínea(j);
  Nsqr:=n*n;
  h[i,j]:=1;
  ensaya(2,i,j,q);
  si q entonces
    para i:=1 hasta n hacer
      para j:=1 hasta n hacer
        escribir(h[i,j])
      fpara;
      escribirLínea
    fpara
  sino
    escribir('No hay solución');
  fsi
fin
```

El siguiente es el resultado de la ejecución de este algoritmo (en menos de un segundo en el mismo computador que las ejecuciones anteriores) para los datos para los que la heurística voraz sin vuelta atrás no encontró solución:

```
Introduce n (1≤n≤8): 5
Introduce i de la casilla inicial: 1
```

```
Introduce j de la casilla inicial: 3
   23    6    1   16   21
   12   17   22    7    2
    5   24   11   20   15
   10   13   18    3    8
   25    4    9   14   19
```

3. Aplicación al problema del viajante

El **problema del viajante** consiste en encontrar un **recorrido de longitud mínima** para un viajante que tiene que **visitar varias ciudades**, conocida la distancia existente entre cada dos ciudades.

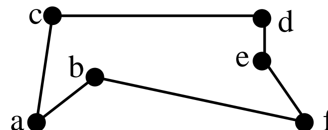
Este problema se lo plantean, por ejemplo, las compañías de distribución de paquetes para elegir la ruta que deben seguir los distribuidores. Considérese, por ejemplo, la siguiente distribución de cinco ciudades (b, c, d, e, f), para las que se conocen sus coordenadas relativas a (a), desde donde parte el distribuidor y a donde debe regresar al terminar, y se supone que la distancia entre cada dos viene dada por la línea recta:

distan.	a	b	c	d	e	f
a						
b	5					
c	7,07	5				
d	16,55	11,70	14			
e	15,52	11,05	14,32	3		
f	18	14,32	18,38	7,6	5	

c● (1,7) d● (15,7)
b● (4,3) e● (15,4)
a● (0,0) f● (18,0)

El único algoritmo conocido para resolver el problema de forma exacta y en el caso general (de calcular el recorrido de longitud mínima que visita una sola vez cada punto, partiendo y terminando en el origen) consiste en **intentar todas las posibilidades**, es decir, calcular las longitudes de todos los recorridos posibles, **y seleccionar la de longitud mínima**. Obviamente, el coste de tal algoritmo crece **exponencialmente** con el número de puntos a visitar.

En el ejemplo anterior, la solución viene dada por el siguiente recorrido (en realidad dos recorridos, pues ambos sentidos de marcha son posibles) de longitud 48,39:



Una **heurística voraz** que puede aplicarse para intentar resolver el problema de forma más eficiente (coste polinómico) consiste en ir **seleccionando parejas de puntos que serán visitados de forma consecutiva** y, lógicamente, **se seleccionará primero** aquella pareja de puntos entre los que la **distancia sea mínima**. A continuación, se selecciona la siguiente pareja separada con una distancia mínima siempre que esta nueva elección no haga que:

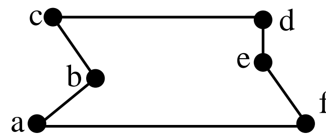
- se visite un punto dos veces o más (es decir, que el punto aparezca tres o más veces en las parejas de puntos seleccionadas), o
- se cierre un recorrido antes de haber visitado todos los puntos.

De esta forma, si hay que visitar *n* puntos (incluido el origen), se selecciona un conjunto de *n* parejas de puntos (que serán visitados de forma consecutiva) y la solución consiste en reordenar todas esas parejas de forma que constituyan un recorrido.

En el ejemplo anterior, las parejas ordenadas por distancia entre sus puntos son: (d,e), (a,b), (b,c), (e,f), (a,c), (d,f), (b,e), (b,d), (c,d), (b,f), (c,e), (a,e), (a,d), (a,f) y (c,f).

Se selecciona primero la pareja (d,e) pues la distancia entre ambos puntos es mínima (3 unidades). Después se seleccionan las parejas (a,b), (b,c) y (e,f). La distancia es para todas ellas igual (5 unidades). La siguiente pareja de distancia mínima es (a,c), con longitud 7,07. Sin embargo, esta pareja cierra un recorrido junto con otras dos ya seleccionadas, (b,c) y (a,b), por lo que se descarta. La siguiente pareja es (d,f) y se descarta también por motivos similares. La siguiente es la pareja (b,e), pero debe rechazarse también porque su inclusión haría visitar más de una vez los puntos b y e. La pareja (b,d) se rechaza por motivos similares (el punto b habría que visitarlo dos veces). La siguiente pareja que debe considerarse es

(c,d), y puede seleccionarse. Las parejas (b,f), (c,e), (a,e) y (a,d) no son aceptables. Finalmente, la pareja (a,f) cierra el recorrido:



Este recorrido no es el óptimo pues su longitud es de 50 unidades. No obstante, es el cuarto mejor recorrido de entre los sesenta (esencialmente distintos) posibles y es más costoso que el óptimo en sólo un 3,3%.

Anexo 4: Especificación algebraica de TAD

Indice

1. Especificación: sintaxis
2. Semántica de una especificación algebraica
3. Construcción de especificaciones
4. Verificación con especificaciones algebraicas

1. Especificación: sintaxis

La especificación algebraica es una técnica formal para especificar o definir TAD. El objetivo es definir sin ambigüedades un tipo de datos (conjunto de valores y efecto de cada operación permitida).

Las ventajas de especificar algebraicamente TAD son:

- permite definir tipos independientemente de cualquier posible representación y razonar sobre la corrección de una representación/implementación,
- unanimidad de la interpretación del tipo por los distintos programadores usuarios del mismo,
- deducir propiedades satisfechas por cualquier implementación correcta del tipo y, en consecuencia, posibilidad de verificar formalmente los módulos que usan el tipo.

En esta sección se introduce una sintaxis para realizar especificaciones algebraicas de tipos. Dichas especificaciones contarán de una signatura y de un conjunto de ecuaciones.

La **signatura** de una especificación algebraica define los **géneros** o nombres de los nuevos tipos especificados, los nombres de las **operaciones** y sus **perfiles** (es decir, su dominio o aridad y su rango o coaridad).

Se va a utilizar una **notación funcional** para definir las operaciones de los TAD, es decir, una operación será una función que toma **como parámetros cero o más valores** de diversos tipos y produce **como resultado un solo valor** de otro tipo.

Por ejemplo, la signatura del TAD tabla de frecuencias, definido, implementado y utilizado en el Tema I, es la siguiente:

```
espec tablas
  usa naturales,enteros
  género tabla
  operaciones
    inicializar: -> tabla
    añadir: tabla entero -> tabla
    total: tabla -> natural
    infoEnt: tabla natural -> entero
    infoFrec: tabla natural -> natural
fespec
```

La cláusula **`usa`** se utiliza para importar las definiciones hechas en otras especificaciones (definición de los géneros `natural` y `entero`). El género especificado en la signatura anterior es `tabla`. En general, una especificación puede contener la definición de varios géneros.

En el ejemplo, se han definido los nombre y perfiles de cinco operaciones. Nótese lo siguiente:

- Hay operaciones 0-arias, es decir, con cero parámetros (como `inicializar`). Estas operaciones se denominan **constantes** del tipo resultado (en el ejemplo, de tipo `tabla`). Su implementación puede consistir en una constante de un tipo predefinido en el lenguaje utilizado, en una función sin parámetros o en un procedimiento con un solo

parámetro de salida, al que da valor el procedimiento. En el ejemplo, la constante `inicializar` se implementó (en el Tema I) de esta última forma.

- La traducción de la notación algebraica (funcional) a la **imperativa** (la que utilizamos habitualmente, dotada de procedimientos y funciones) es normalmente inmediata. Por ejemplo, la operación `total` da lugar a una función con igual perfil. En el caso de la operación `añadir`, se optó en la implementación del Tema I por un procedimiento en el que el parámetro `t` de tipo `tabla`, de entrada y salida, hace un doble papel: el de uno de los parámetros de la aridad y el de resultado.
- La restricción a sólo un resultado por función no es importante. En la práctica, varias operaciones con la misma aridad y distinto resultado pueden combinarse en un solo procedimiento con varios parámetros de salida (correspondientes a los resultados). Por ejemplo, las operaciones `infoEnt` e `infoFrec` se implementaron en el Tema I con el procedimiento `info`.

Veamos, a continuación, una posible signatura de los tipos "booleano" y "naturales con el cero".

```
espec boolnat
  géneros booleano,natural
  operaciones
    verdad,falso: -> booleano
    ¬_: booleano -> booleano
    _∧_,_∨_: booleano booleano -> booleano
    0,1: -> natural
    suc: natural -> natural
    _+_,_*_: natural natural -> natural
    _≤_,_>_: natural natural -> booleano
fespec
```

En este ejemplo, `verdad`, `falso`, `0` y `1` son constantes. La operación `suc` es prefija, es decir, el nombre de la operación precede a los operandos y éstos van entre paréntesis y separados por comas. Para definir operaciones prefijas sin paréntesis o infijas, indicaremos mediante el símbolo '_' la posición de los argumentos con respecto al nombre de la operación (ejemplos: ¬_, _+_, …).

Para cada género existe un conjunto de términos bien formados (es decir, sintácticamente correctos) o, simplemente, **términos**. La signatura especifica cómo se construyen los términos. De manera informal, **cada constante es un término y la aplicación de un símbolo de operación a un número apropiado de términos de géneros adecuados es también un término**. En el caso de definir operaciones con notación infija se precisa además la utilización de paréntesis para construir los términos. Los siguientes son ejemplos de términos bien formados:

```
1
(suc(1+suc(0))*1)≤1
((verdad∨falso)∧(¬falso))∧(0>suc(suc(1)))
```

A la especificación algebraica de un TAD se le puede atribuir una **semántica o significado**. Dicho significado consiste en considerar que **cada término bien formado denota un valor del tipo** al que pertenece la expresión construida. Por ejemplo, `0`, `1`, `suc(0)`, `0+1`, son valores del tipo `natural`, mientras que `verdad`, `falso`, `¬falso`, `(suc(1+suc(0))*1)≤1`, son valores de tipo booleano.

En la denominada **semántica inicial** del TAD `natural` definido previamente, cada término bien formado de tipo `natural` denota un valor **diferente** de dicho tipo.

Es posible que, según la idea intuitiva del programador sobre el tipo que está definiendo, varios términos bien formados diferentes deban corresponder a un mismo valor. Por ejemplo, los términos `1`, `suc(0)`, `0+1`, `1+0` y `1*1`, corresponden a la idea abstracta "uno", que todos conocemos, y por tanto deberían tener un mismo significado. Sin embargo, la semántica inicial considera, por defecto, valores distintos aquéllos que se construyen con términos distintos.

Considerando la semántica inicial, la siguiente especificación construye exactamente el tipo de los números naturales (con el cero):

```
espec naturales_1
  género natural
  operaciones
    0: -> natural
    suc: natural -> natural
fespec
```

Los únicos valores que pueden construirse son `0`, `suc(0)`, `suc(suc(0))`, `suc(suc(suc(0)))`, etcétera. Cada término denota un valor diferente, que corresponde a la idea intuitiva de un natural diferente.

Si queremos añadir la operación “suma” a la especificación anterior, puede intentarse en la forma siguiente:

```
espec naturales_2
  género natural
  operaciones
    0: -> natural
    suc: natural -> natural
    _+_: natural natural -> natural
fespec
```

Sin embargo, esta especificación construye un tipo que no corresponde con lo que llamamos “naturales” puesto que, por ejemplo, los términos `suc(0)` y `0+suc(0)` denotan, en principio, valores diferentes (algo contrario a nuestro conocimiento sobre cómo los números naturales deberían comportarse).

La forma de expresar en una especificación que **varios términos corresponden a un mismo valor** y tienen, por tanto, un mismo significado es añadir **ecuaciones**:

```
término_1 = término_2
```

Donde `término_1` y `término_2` son términos bien formados de un mismo género.

Para poder expresar el hecho de que un número grande o infinito de términos bien formados tienen el mismo valor se pueden introducir **variables** en las ecuaciones. Se entiende que, en cada ecuación con variables, éstas están (implícitamente) cuantificadas universalmente ($\forall$).

```
espec naturales_3
  género natural
  operaciones
    0: -> natural
    suc: natural -> natural
    _+_: natural natural -> natural
  ecuaciones x,y:natural
    x+0 = x
    x+suc(y) = suc(x+y)
fespec
```

2. Semántica de una especificación algebraica

Definición (Signatura). Una signatura *SIG* es un par (*G*,*OP*), donde:

- *G* es un conjunto de símbolos que llamaremos géneros.
- *OP* es un conjunto de símbolos que llamaremos nombres de operaciones. Cada símbolo $\sigma \in OP$ tiene asociado un perfil $\sigma: g_1 \ldots g_n \rightarrow g$ que consta de una secuencia (quizás vacía) $g_1 \ldots g_n$ de símbolos de *G*, llamada aridad de *OP* y que especifica el número y género de sus argumentos, y de un símbolo *g* de *G*, llamado rango de *OP* que especifica el género del resultado. ♦

Si la aridad de una operación σ es vacía,es decir $\sigma: \rightarrow g$, se dice que σ es un símbolo de **constante**.

Nótese que, en una signatura, tanto G como OP sólo son conjuntos de símbolos sin significado. Los símbolos $g \in G$ y $\sigma \in OP$ son sólo nombres. A continuación, definimos un álgebra sobre dicha signatura que constituye una interpretación posible de sus símbolos.

Definición (*SIG*-álgebra). Dada una signatura $SIG = (G,OP)$, un álgebra heterogénea A sobre SIG, o simplemente SIG-álgebra es un par $(\{A_g\}_{g\in G},OP^A)$, donde:

- $\{A_g\}_{g\in G}$, es una familia de conjuntos no vacíos; los elementos de cada A_g se llaman soporte del género g.
- OP^A es un conjunto de aplicaciones σ^A tal que:
  $\forall\ \sigma \in OP$ con perfil $g_1\ldots g_n \to g$, $\exists\ \sigma^A \in OP^A$ de la forma $\sigma^A: A_{g1} \times \ldots \times A_{gn} \to A_g$.
  Cada aplicación σ^A se denomina **interpretación** en A del símbolo de operación σ. ♦

Ejemplo: Considérese la signatura definida en la siguiente especificación.

```
espec pilasDeEnteros
  géneros bool,ent,pila
  operaciones
    verdad,falso: -> bool
    ¬_: bool -> bool
    _∧_,_∨_: bool bool -> bool
    0: -> ent
    suc,pred: ent -> ent
    _+_,_-_: ent ent -> ent
    pilaVacía: -> pila
    apilar: pila ent -> pila
    cima: pila -> ent
    desapilar: pila -> pila
    vacía?: pila -> bool
fespec
```

Es decir, $SIG = (G,OP)$, con G = {`bool`,`ent`,`pila`} y OP = {`verdad`, `falso`, `¬`, `∧`, `∨`, `0`, `suc`, `+`, `-`, `pilaVacía`, `apilar`, `cima`, `desapilar`, `vacía?`}.

Presentamos a continuación tres posibles álgebras heterogéneas sobre *SIG*. Primera interpretación, A^1:

$A^1_{\texttt{bool}}$ = {true,false}; $A^1_{\texttt{ent}}$ = Z; $A^1_{\texttt{pila}}$ = Z*

Z* denota el conjunto de todas las secuencias de longitud finita de números enteros, incluida la secuencia vacía. Las operaciones internas sobre $A^1_{\texttt{bool}}$ y $A^1_{\texttt{ent}}$ son las habituales del álgebra booleana y de los números enteros. El resto son las siguientes ('i' denota un entero cualquiera, 'ε' denota la secuencia vacía y 's' denota un elemento cualquiera de Z*):

$\texttt{pilaVacía}^{A^1}$ = ε;	$\texttt{apilar}^{A^1}$(s,i) = si
$\texttt{desapilar}^{A^1}$(ε) = ε;	$\texttt{desapilar}^{A^1}$(si) = s
$\texttt{cima}^{A^1}$(ε) = 0;	$\texttt{cima}^{A^1}$(si) = i
$\texttt{vacía?}^{A^1}$(ε) = true;	$\texttt{vacía?}^{A^1}$(si) = false

Una segunda interpretación:

$A^2_{\texttt{bool}}$ = {T,F,KK}; $A^2_{\texttt{ent}}$ = Z; $A^2_{\texttt{pila}}$ = {¡y_a_mi_qué!}

Las operaciones internas sobre $A^2_{\texttt{ent}}$ son las habituales. El resto son las siguientes ('i' denota un entero cualquiera):

$\texttt{verdad}^{A^2}$ = T;	$\texttt{falso}^{A^2}$ = F	
¬(T) = F;	¬(F) = T;	¬(KK) = KK
T∧T = T;	F∧F = F∧T = T∧F = F∧KK = KK∧F = F	

T∧KK = KK∧T = KK; F∨F = F

T∨T = T∨F = F∨T = T∨KK = KK∨T = T; F∨KK = KK∨F = KK

`pilaVacía`$^{A^2}$ = ¡y_a_mi_qué!

`apilar`$^{A^2}$(¡y_a_mi_qué!,i) = ¡y_a_mi_qué!

`desapilar`$^{A^2}$(¡y_a_mi_qué!) = ¡y_a_mi_qué!

`cima`$^{A^2}$(¡y_a_mi_qué!) = 1111

`vacía?`$^{A^2}$(¡y_a_mi_qué!) = KK

Una interpretación más:

$A^3_{\texttt{bool}}$ = {true,false}; $A^3_{\texttt{ent}}$ = Z; $A^3_{\texttt{pila}}$ = $Z^\omega = Z^* \cup Z^\infty$

Z^∞ denota el conjunto de secuencias de longitud infinita (numerable) de enteros. Las operaciones internas sobre $A^3_{\texttt{bool}}$ y $A^3_{\texttt{ent}}$ son las habituales. El resto son las siguientes ('i' denota un entero cualquiera, 'ε' denota la secuencia vacía y 's' denota un elemento cualquiera de Z^ω):

`pilaVacía`$^{A^3}$ = ε; `apilar`$^{A^3}$(s,i) = is

`desapilar`$^{A^3}$(ε) = ε; `desapilar`$^{A^3}$(is) = s

`cima`$^{A^3}$(ε) = 0; `cima`$^{A^3}$(is) = i

`vacía?`$^{A^3}$(ε) = true; `vacía?`$^{A^3}$(is) = false

Se define a continuación el álgebra de términos cerrados o sin variables sobre una signatura.

Definición (Álgebra de términos cerrados). Dada una signatura $SIG = (G,OP)$, el álgebra de términos cerrados sobre SIG, denotada T_{SIG}, es una SIG-álgebra $(\{T_{SIGg}\}_{g\in G}, OP^{T_{SIG}})$, donde:

- $T_{SIGg} = \{\sigma \mid \sigma \in OP \wedge \sigma: \to g\} \cup \{\sigma(t_1,\ldots,t_n) \mid \sigma \in OP \wedge \sigma: g_1\ldots g_n \to g \wedge t_1 \in T_{SIGg1}, \ldots, t_n \in T_{SIGgn}\}$.
  Es decir, T_{SIGg} es el conjunto de términos de género g de SIG.
- Las aplicaciones de $OP^{T_{SIG}}$ se definen trivialmente:
  $\forall\ \sigma: \to g$, la interpretación $\sigma^{T_{SIG}}$ de σ en T_{SIG} es el propio término $\sigma \in T_{SIGg}$.
  $\forall\ \sigma: g_1\ldots g_n \to g$, $\forall\ t_1 \in T_{SIGg1}, \ldots, \forall\ tn \in T_{SIGgn}$, la interpretación $\sigma^{T_{SIG}}$ de σ en T_{SIG} es la siguiente:
  $\sigma^{T_{SIG}}(t_1,\ldots,t_n) \stackrel{\text{def}}{=} \sigma(t_1,\ldots,t_n)$. ♦

Es decir, cada término bien formado de la signatura es un valor distinto del tipo apropiado en el álgebra de términos cerrados. La imagen mediante una aplicación (del álgebra de términos cerrados) de un conjunto de términos de tipos apropiados es el término bien formado que puede construirse prefijando a dichos términos, separados por comas y entre paréntesis, con el símbolo de operación correspondiente (suponemos que cada símbolo de operación tiene una versión prefija).

Si una especificación algebraica no posee ecuaciones, el objeto formal que define es la SIG-álgebra de términos cerrados T_{SIG}. Como hemos visto, hay veces que interesa que distintos términos bien formados tengan un mismo significado. Para ello es necesario introducir ecuaciones. Definimos antes los conceptos de "conjunto de variables" y de "término abierto" o con variables.

Definición (Conjunto de variables). Dada una signatura $SIG = (G,OP)$, un conjunto de variables con respecto a SIG es una familia de conjuntos $X = \{X_g\}_{g\in G}$ tal que $X_g \cap X_{g'} = \varnothing$ si $g \neq g'$. Además, los identificadores de X no figuran ni en G ni en OP. ♦

Definición (Términos abiertos). Dada una signatura $SIG = (G,OP)$ y un conjunto de variables con respecto a SIG, $X = \{X_g\}_{g\in G}$, el conjunto de términos abiertos (o términos con variables) de género $g \in G$ es: $T_{SIG}(X)_g = X_g \cup \{\sigma \mid \sigma \in OP \wedge \sigma: \to g\} \cup \{\sigma(t_1,\ldots,t_n) \mid \sigma \in \text{OP} \wedge \sigma: g_1\ldots g_n \to g \wedge t_1 \in T_{SIG}(X)_{g1},\ldots, t_n \in T_{SIG}(X)_{gn}\}$. ♦

Definición (Álgebra de términos abiertos). Dada una signatura $SIG = (G,OP)$ y un conjunto X de variables con respecto a SIG, el álgebra de términos abiertos sobre SIG, denotada $T_{SIG}(X)$, se define de modo similar a T_{SIG}, con la única salvedad de que, además de los símbolos de constante, toda $x \in X_g$ es un término de $T_{SIG}(X)$. ♦

Definición (Ecuación). Dada una signatura $SIG = (G,OP)$ y un conjunto X de variables con respecto a SIG, una ecuación de género g con respecto a SIG es una terna $(X,t_1,t_2)_g$, con $t_1,t_2 \in T_{SIG}(X)_g$, y se denota $t_1 = t_2$. ♦

Definición (Especificación algebraica). Una especificación algebraica es un par $ESPEC = (SIG,E)$, donde SIG es una signatura y E un conjunto de ecuaciones con respecto a SIG. ♦

Ejemplo:

```
espec bolsasDeNaturales
  usa naturales_3[1]
  género bolsa                    {Bolsas de naturales}
  operaciones
    []: -> bolsa                  {Bolsa vacía}
    [_]: natural -> bolsa         {Bolsa unitaria}
    _∪_: bolsa bolsa -> bolsa     {Unir bolsas}
    _⊕_: natural bolsa -> bolsa   {Añadir natural a bolsa}
  ecuaciones x:natural; b,b1,b2,b3:bolsa
    b∪[] = b
    []∪b = b
    (b1∪b2)∪b3 = b1∪(b2∪b3)
    x⊕b = [x]∪b
fespec
```

De la misma forma que una signatura SIG genera el álgebra T_{SIG} (álgebra de términos cerrados sobre SIG), vamos a ver que una especificación también genera un álgebra: el álgebra cociente de T_{SIG} al introducir una relación de equivalencia definida por el conjunto de ecuaciones en el conjunto de los términos. Previamente, definimos la noción de sustitución y la relación de equivalencia en los términos de T_{SIG}.

Definición (Sustitución y extensión). Dada una signatura $SIG = (G,OP)$ y un conjunto X de variables con respecto a SIG, una sustitución $h = \{h_g\}_{g \in G}$ es una familia de aplicaciones h_g: $X_g \rightarrow T_{SIGg}$ de variables a términos cerrados. Llamaremos extensión de una sustitución h a la familia de aplicaciones, también denotada $h = \{h_g\}_{g \in G}$, h_g: $T_{SIG}(X)_g \rightarrow T_{SIGg}$ definida como sigue:

- si $t = x \in X_g$, $h_g(t) \stackrel{\text{def}}{=} h_g(x)$
- si $t = \sigma \in OP \wedge \sigma: \rightarrow g$, $h_g(t) \stackrel{\text{def}}{=} \sigma$
- si $t = \sigma(t_1,\ldots,t_n) \in OP \wedge \sigma: g_1,\ldots,g_n \rightarrow g$, $h_g(t) \stackrel{\text{def}}{=} \sigma(h(t_1),\ldots,h(t_n))$. ♦

Definición (Congruencia). Dada una especificación $ESPEC = (SIG,E)$, la congruencia engendrada en T_{SIG} por las ecuaciones E es la relación $\equiv_E$ que satisface:

i) $\forall g \in G, \forall t \in T_{SIGg}, t \equiv_E t$ (reflexiva)

ii) $\forall g \in G, \forall t_1,t_2 \in T_{SIGg}, t_1 \equiv_E t_2 \Rightarrow t_2 \equiv_E t_1$ (simétrica)

iii) $\forall g \in G, \forall t_1,t_2,t_3 \in T_{SIGg}, t_1 \equiv_E t_2 \wedge t_2 \equiv_E t_3 \Rightarrow t_1 \equiv_E t_3$ (transitiva)

iv) $\forall \sigma: g_1 \ldots g_n \rightarrow g, \forall t_1,t_1' \in T_{SIGg1},\ldots, \forall t_n,t_n' \in T_{SIGgn}, t_1 \equiv_E t_1' \wedge \ldots \wedge t_n \equiv_E t_n' \Rightarrow \sigma(t_1,\ldots,t_n) \equiv_E \sigma(t_1',\ldots,t_n')$ (congruente con $OP^{T_{SIG}}$)

v) $\forall (X,t_1,t_2)_g \in E, \forall h_g: X_g \rightarrow T_{SIGg}, h_g(t_1) \equiv_E h_g(t_2)$ (engendrada por E)

vi) si $\equiv'_E$ es otra relación que verifica (i,ii,iii,iv,v) entonces:
$\forall g \in G, \forall t_1,t_2 \in T_{SIGg}, t_1 \equiv_E t_2 \Rightarrow t_1 \equiv'_E t_2$ (mínima).

---

[1] Especificación del TAD `natural` introducida previamente.

Es decir, es la menor relación de equivalencia en T_{SIG}, congruente (o compatible) con las operaciones de T_{SIG}, engendrada por las ecuaciones E (dos términos son equivalentes si y sólo si dicha equivalencia se deduce de las ecuaciones E). ♦

Definición (Álgebra definida por una especificación). Dada una especificación $ESPEC = (SIG,E)$, el álgebra definida por $ESPEC$, denotada T_{ESPEC}, es la siguiente:

- soportes: $T_{ESPECg} \stackrel{def}{=} T_{SIGg} / \equiv_E$. Dado $t \in T_{SIG}$, denotaremos por $[t]$ a la clase de equivalencia del término t.
- operaciones: $\forall g \in G$,
  $\forall \sigma: \to g,\ \sigma^{T_{ESPEC}} \stackrel{def}{=} [\sigma]$
  $\forall \sigma: g_1 \ldots g_n \to g,\ \sigma^{T_{ESPEC}}([t_1],\ldots,[t_n]) \stackrel{def}{=} [\sigma(t_1,\ldots,t_n)]$. ♦

De la definición del álgebra T_{ESPEC} se deducen las siguientes importantes propiedades:

- Sólo pertenecen a $\{T_{ESPECg}\}_{g} \in G$ valores que puedan ser generados por algún término t.
- T_{ESPEC} satisface las ecuaciones E de la especificación: dos términos conducen al mismo valor (es decir, están en la misma clase o tienen el mismo soporte) si ello se deduce de E.
- Las únicas igualdades que pueden darse en T_{ESPEC} son las que se deducen de E (es decir, $\equiv_E$ es la menor relación de equivalencia que engloba las igualdades deducidas de E).

Al igual que ocurre con una signatura SIG, que el álgebra de términos T_{SIG} no es la única interpretación posible pues existen otras SIG-álgebras, dada una especificación $ESPEC$ existen muchas SIG-álgebras que satisfacen las ecuaciones de la especificación además de T_{ESPEC}. Las llamaremos ***ESPEC*-álgebras o modelos de *ESPEC***.

Definición (Valuación). Dada una SIG-álgebra $A = (\{A_g\}_{g \in G}, OP^A)$ y un conjunto de variables $X = \{X_g\}_{g \in G}$ con respecto a SIG, una valuación $V = \{V_g\}_{g \in G}$ es un conjunto de aplicaciones $V_g: X_g \to A_g$. ♦

Nótese que el concepto de sustitución ($h_g: X_g \to T_{SIGg}$), definido previamente, no es más que una valuación para el caso particular del álgebra de términos cerrados T_{SIG}. Al igual que en el caso particular de una sustitución, una valuación puede extenderse a los términos abiertos:

Definición (Evaluación). Dada una valuación V de un conjunto de variables X en una SIG-álgebra A, y un término $t \in T_{SIG}(X)_g$, la evaluación de t en A según V, denotada $t^{A,V}$, se define como:

- si $t = x \in X_g,\ t^{A,V} \stackrel{def}{=} V_g(x)$
- si $t = \sigma \in \mathrm{OP} \wedge \sigma: \to g,\ t^{A,V} \stackrel{def}{=} \sigma^A$
- si $t = \sigma(t_1,\ldots,t_n) \in \mathrm{OP} \wedge \sigma: g_1,\ldots,g_n \to \mathrm{g},\ t^{A,V} \stackrel{def}{=} \sigma^A(t_1^{A,V},\ldots,t_n^{A,V})$. ♦

En particular, si t $\in T_{SIGg}$, es decir t es un término sin variables, la evaluación $t^{A,V}$ es la misma para cualquier valuación V, y se denota por t^A.

Definición (*ESPEC*-álgebra). Dada una especificación $ESPEC = (SIG,E)$, una SIG-álgebra $A = (\{A_g\}_{g \in G}, OP^A)$ satisface $ESPEC$ o es un modelo de $ESPEC$ o es una $ESPEC$-álgebra, y se denota $A \models E$, si satisface todas las ecuaciones E, es decir: $\forall (X,t_1,t_2)_g \in E,\ \forall V_g: X_g \to A_g,\ t_1^{A,V} = t_2^{A,V}$. ♦

Hay que notar que, dada una especificación $ESPEC = (SIG,E)$, el álgebra T_{ESPEC} definida por $ESPEC$ es una $ESPEC$-álgebra.

Por ejemplo, la especificación `naturales_3` vista antes admite, entre otras, las siguientes $ESPEC$-álgebras:

1) los números naturales con la operación de suma;
2) los números enteros, interpretando `0` como el cero de los enteros, + como la suma de enteros y `suc(x)` como x+1;
3) los números naturales módulo 3, es decir, $A = \{0,1,2\}$, interpretando `0` como 0 de A, `suc` como el sucesor módulo 3 (`suc(0)=1`, `suc(1)=2`, `suc(2)=0`) y + como la suma módulo 3 (por ejemplo, `2+2=1`);

4) el cero, es decir, $A = \{0\}$, interpretando `0` como 0 de A, `suc(0)=0` y `0+0=0`.

Como veremos a continuación, los diferentes modelos de una especificación pueden compararse. Veremos que el modelo (1) de la especificación `naturales_3` es especial en cierto sentido puesto que es **isomorfo** al álgebra $T_{\texttt{naturales_3}}$ definida por la especificación, y se denomina **álgebra inicial**.

Definición (*SIG*-homomorfismo y *SIG*-isomorfismo). Dadas dos *SIG*-álgebras A y B, un *SIG*-homomorfismo es una familia $f = \{f_g\}_{g \in G}$ de aplicaciones $f_g: A_g \to B_g$ que conmuta con las operaciones, es decir, satisface:

- $\forall\, g \in G, \forall\, \sigma: \to g, f_g(\sigma^A) = \sigma^B$
- $\forall \sigma: g_1,\ldots,g_n \to g, \forall\, a_1 \in A_{g1}, \ldots, \forall\, a_n \in A_{gn}, f_g(\sigma^A(a_1,\ldots,a_n)) = \sigma^B(f_{g1}(a_1),\ldots,f_{gn}(a_n))$.

Un *SIG*-homomorfismo biyectivo se llama *SIG*-isomorfismo. Si existe un *SIG*-isomorfismo de A a B diremos que A y B son isomorfas, y se denota $A \approx B$. ♦

Si existe un *SIG*-homomorfismo f de una *SIG*-álgebra A a otra B, se puede definir sobre los valores de A la siguiente relación de equivalencia: dos valores son equivalentes si y sólo si su imagen en B coincide. El álgebra cociente de A con respecto a esa relación de equivalencia es, de nuevo, una *SIG*-álgebra, y además es isomorfa a la imagen mediante f de A, $f(A) \subseteq B$. Es decir, los homomorfismos pueden identificar en la imagen valores que son distintos en el origen. Por tanto, es de esperar que un álgebra con más valores distintos tenga homomorfismos hacia otras con menos valores, pero no a la inversa.

Dada una signatura *SIG*, la clase de todas las *SIG*-álgebras junto con todos los homomorfismos entre cada par de *SIG*-álgebras, denotada por *ALG*(*SIG*), es una estructura algebraica denominada **categoría** porque la composición de dos *SIG*-homomorfismos es otro *SIG*-homomorfismo y las aplicaciones identidad son *SIG*-homomorfismos.

De forma análoga, dada una especificación *ESPEC*, la clase de todas las *ESPEC*-álgebras junto con todos los homomorfismos entre cada par de *ESPEC*-álgebras, denotada por *ALG*(*ESPEC*), es también una categoría.

En una categoría, los isomorfismos entre objetos definen una relación de equivalencia en la clase de objetos. Por ejemplo, dos *SIG*-álgebras A y B de *ALG*(*SIG*) están relacionadas si entre ellas existe un *SIG*-isomorfismo (es decir, $A \approx B$). Las clases de equivalencia así definidas se llaman **clases de isomorfía**.

Definición (Álgebra inicial). Un álgebra I se dice inicial en una categoría si existe un único homomorfismo de I a cada una de las restantes álgebras de la categoría. ♦

Teorema. Si una categoría posee un álgebra inicial, ésta es única salvo isomorfismo. Es decir, en una categoría en la que existen álgebras iniciales, éstas forman una clase de isomorfía. ♦

Teorema. Dada una signatura *SIG* y una especificación *ESPEC* = (*SIG*,*E*):

1. El álgebra de términos cerrados sobre *SIG*, T_{SIG}, es inicial en la clase *ALG*(*SIG*). El único homomorfismo de T_{SIG} a cualquier *SIG*-álgebra A es la evaluación de términos t^A.
2. El álgebra definida por *ESPEC*, T_{ESPEC}, es inicial en la clase *ALG*(*ESPEC*). El único homomorfismo $f = \{f_g\}_{g \in G}$, con $f_g: T_{ESPECg} \to A_g$, de T_{ESPEC} a cualquier *ESPEC*-álgebra A se define, para cada término t de T_{ESPEC}, como: $f_g([t]) \overset{\text{def}}{=} t^A$. ♦

Dada una especificación *ESPEC*, un **tipo concreto de datos** es una *ESPEC*-álgebra cualquiera. Se denomina así porque tiene un conjunto de elementos determinado (soporte) y unas operaciones definidas por medio de aplicaciones explícitas (interpretaciones) entre los conjuntos soporte del álgebra.

Dada una especificación *ESPEC*, un **tipo abstracto de datos** (TAD) es la clase de isomorfía de un álgebra inicial en *ALG*(*ESPEC*). Es decir, es la clase de todas las *ESPEC*-álgebras o modelos de la especificación que son isomorfas al álgebra T_{ESPEC} definida por *ESPEC*.

Consideremos de nuevo la especificación `bolsasDeNaturales`. El tipo bolsa de naturales tiene una operación ∪ que es asociativa y tiene como elemento neutro a la bolsa vacía `[]`. Aparentemente, las bolsas se comportan como “conjuntos de naturales”. Sin embargo, es fácil ver que también los “multiconjuntos de naturales” (interpretando ∪ como la unión de

multiconjuntos), las "secuencias o listas de naturales" (interpretando ∪ como la concatenación de secuencias) y las "secuencias de enteros" (siendo igualmente ∪ la concatenación de secuencias) satisfacen las ecuaciones.

El modelo más adecuado de la especificación `bolsasDeNaturales` es el de las "secuencias de naturales". Los multiconjuntos y los conjuntos tienen más igualdades que las que se deducen de la especificación, ya que la operación ∪, además de asociativa, es conmutativa (la igualdad [3] ∪ [4] = [4] ∪ [3] es cierta en los multiconjuntos y en los conjuntos pero no en las secuencias). Los conjuntos satisfacen además la idempotencia, es decir, la ecuación b ∪ b = b.

Se puede definir un homomorfismo que a cada secuencia de naturales la haga corresponder el multiconjunto resultante de olvidar el orden de los elementos de la secuencia. Igualmente, existe un homomorfismo de multiconjuntos a conjuntos consistente en conservar solamente una copia de cada elemento del multiconjunto de partida. También existe un homomorfismo que a cada secuencia de naturales le hace corresponder una secuencia de enteros: el homomorfismo identidad (aunque hay secuencias de enteros que no tienen anti-imagen por ese homomorfismo).

Como vemos, el modelo de las "secuencias de naturales" satisface estrictamente las igualdades establecidas en la especificación y ninguna otra, es decir, sólo satisface las igualdades que son válidas en todos los demás modelos de la especificación (por ejemplo, los "conjuntos de naturales" o los "multiconjuntos de naturales"). En este sentido, el modelo de las "secuencias de naturales" se dice que es **típico** en la categoría *ALG*(`bolsasDeNaturales`). También se dice que en ese modelo no existe **confusión** de valores (como en el caso de los multiconjuntos o conjuntos, en el que dos secuencias diferentes de naturales pueden tener por imagen o "confundirse" en un mismo multiconjunto o conjunto).

Por otra parte, todo valor del modelo "secuencias de naturales" puede ser generado mediante la evaluación de algún término del álgebra de términos cerrados definida por la especificación. No ocurre así con el modelo de las "secuencias de enteros". Este último tiene valores que no pueden obtenerse a partir de la evaluación de ningún término del álgebra de términos cerrados (todas las secuencias que incluyan algún número negativo). Así, el modelo de las "secuencias de enteros" contiene **basura**.

El modelo de las "secuencias de naturales" es precisamente un **modelo inicial** de *ALG*(`bolsasDeNaturales`) porque es **típico y generable** (en otras palabras, **sin confusión y sin basura**). El TAD definido por la especificación `bolsasDeNaturales` es la clase de todos los modelos isomorfos a las "secuencias de naturales".

3. Construcción de especificaciones

Las especificaciones de tipos complejos pueden ser textos voluminosos. Por tanto, son necesarias las siguientes características en su diseño: modularidad, legibilidad, estructuración, documentación…

Las características de modularidad y estructuración se consiguen comenzando por los tipos básicos (booleanos, naturales, enteros…) y progresando hacia tipos más complejos (secuencias o listas de enteros, conjuntos, multiconjuntos, pilas, árboles, etc., etc.). Cada módulo (**`espec`** … **`fespec`**) incluye la definición de uno o varios (en número reducido) géneros, es decir tipos, o nuevas operaciones de un género previamente especificado ("enriquecimiento" del tipo).

El método de construcción de especificaciones de un TAD que se presenta consiste en introducir un cierto orden en la escritura de las ecuaciones relacionadas con el género correspondiente.

Sea *g* el símbolo o identificador del género correspondiente al tipo que se desea definir. Denotamos por *OP*(*g*) el conjunto de operaciones relacionadas con *g* (es decir, cuyo perfil incluye a *g*). Las operaciones de *OP*(*g*) se clasifican en:

- **constructoras**: operaciones cuyo resultado es de género *g*, denotadas por *Cons*(*g*);
- **observadoras**: operaciones que tienen uno o más argumentos de género *g* y cuyo resultado no es de género *g*, denotadas por *Obs*(*g*).

Ejemplo:

```
espec naturales
  usa booleanos
  género natural
  operaciones
    0: -> natural
    suc: natural -> natural
    _+_: natural natural -> natural
    _≤_: natural natural -> booleano
fespec
```

Las operaciones 0, suc y + son constructoras mientras que la operación ≤ es observadora.

Las operaciones constructoras se dividen, a su vez, de la siguiente forma:

- **generadoras**: son aquéllas que son necesarias para generar todos los valores del tipo y tales que excluyendo cualquiera de ellas ya no es posible generar todos esos valores, denotadas por *Gen*(*g*):
$$\forall\, t \in T_{SIGg},\ \exists\, t' \in T_{Gen(g)},\ t \equiv_E t'.$$
- **modificadoras**: son las operaciones constructoras que no son generadoras, se denotan por *Mod*(*g*).

En el ejemplo anterior, resulta obvio que la elección es *Gen*(natural) = {0,suc}, porque con cualquier otra elección no es posible generar todos los valores del tipo (por ejemplo, *Gen*(natural) = {0,+}). En otros casos hay más de una elección posible. Por ejemplo, en la especificación bolsasDeNaturales de la lección anterior, tanto *Gen*(bolsa) = {[ ],[_],_∪_} como *Gen*(bolsa) = {[ ],_⊕_} son generadoras.

En resumen, dado un género *g*, las operaciones relacionadas con *g* se clasifican en:

$$OP(g) = \begin{cases} Cons(g) = \begin{cases} Gen(g) & \text{generadoras} \\ Mod(g) & \text{modificadoras} \end{cases} \\ Obs(g) \ \ \text{observadoras} \end{cases}$$

En cuanto al conjunto *Gen*(*g*) de operaciones generadoras, pueden darse dos situaciones:

- **conjunto libre de generadoras**: todo término de $T_{Gen}(g)$ denota un valor diferente en el tipo de datos correspondiente a *g*;
- **conjunto no libre de generadoras**: dos o más términos distintos de $T_{Gen}(g)$ denotan un mismo valor del tipo.

Por ejemplo, *Gen*(natural) = {0,suc} es un conjunto libre de generadoras del tipo natural porque cada término de T_{Gen}(natural) (es decir: 0, suc(0), suc(suc(0)), …) denota un natural diferente.

Lo mismo sucede con *Gen*(bolsa) = {[ ],_⊕_} con respecto al tipo bolsa (por ejemplo, la sucesión [3,4,5] sólo puede generarse como 3⊕(4⊕(5⊕[ ]))).

En cambio, *Gen*(bolsa) = {[ ],[_],_∪_} es no libre puesto que, por ejemplo, la sucesión [3,4,5] puede generarse como ([3]∪[4])∪[5] ó como [3]∪([4]∪[5]).

Cuando el conjunto de operaciones generadoras es libre puede establecerse una biyección entre el conjunto $T_{Gen}(g)$ y el soporte T_{ESPECg}. Para cada clase [*t*] de T_{ESPECg}, es decir, para cada valor del tipo de interés, existe un único término $t_c \in$ [t] $\cap\, T_{Gen(g)}$, que se denomina **término canónico** de la clase. Es decir, para cada valor del tipo existe un único término formado sólo por operaciones generadoras.

Cuando el conjunto de generadoras no es libre el especificador debe **elegir** un término (el que, a su juicio, sea más sencillo) como representante canónico de cada clase.

La elección que se haga de la clase *Gen*(*g*) así como la elección de los términos canónicos en el caso en que *Gen*(*g*) sea no libre, influyen en la escritura de las ecuaciones de la especificación.

Una metodología adecuada para escribir ecuaciones es la siguiente:

1. Si *Gen*(g) es un conjunto libre, sólo hay que escribir ecuaciones con las operaciones modificadoras y observadoras (ver puntos 3 y 4). Si la signatura contiene sólo las operaciones generadoras, entonces no hay que escribir ecuaciones y el álgebra T_{SIG} de términos cerrados es el modelo de la especificación (ejemplo: la especificación `naturales_1` vista con anterioridad).
2. Si *Gen*(g) es un conjunto no libre, un primer conjunto de ecuaciones se utiliza para hacer congruentes entre sí ciertos términos de $T_{Gen(g)}$ (en los que aparecen sólo operaciones generadoras). Este conjunto se llama **ecuaciones entre generadoras**. Consisten en igualar (hacer congruente) cada término no canónico de $T_{Gen(g)}$ con el representante canónico de su clase. Por ejemplo, si en la especificación `bolsasDeNaturales` vista antes se considera *Gen*(`bolsa`) = {`[]`,`[_]`,`_∪_`} como conjunto de generadoras (no libres), las tres primeras ecuaciones de la especificación corresponden a esta clase.
3. Para cada operación modificadora se escriben tantas ecuaciones como sean necesarias para garantizar que todo término de $T_{Cons(g)}$ sea congruente a algún término de $T_{Gen(g)}$. Una estrategia habitual es poner la operación modificadora como la más externa de un término y descomponer cada parámetro de tipo correspondiente a g en todos sus posibles patrones formados con las operaciones generadoras. Por ejemplo, la cuarta ecuación de la especificación `bolsasDeNaturales`, considerando *Gen*(`bolsa`) = {`[]`,`[_]`,`_∪_`} como conjunto de generadoras, es de esta clase. También las tres ecuaciones de la especificación `naturales_3` vista antes pertenecen a esta clase.
4. Para cada operación observadora se escriben tantas operaciones como sean necesarias para garantizar que todo término de $T_{OP(g)}$ de género g' ($g' \neq g$) sea congruente a algún término de $T_{Cons(g')}$. La estrategia citada en el punto anterior, para operaciones modificadoras, es útil también aquí.

Para la siguiente especificación de “conjuntos de caracteres”:

```
espec conjuntosDeCaracteres
  usa booleanos,caracteres,naturales
  género conjcar
  operaciones
    vacío: -> conjcar
    poner: carácter conjcar -> conjcar
    quitar: carácter conjcar -> conjcar
    _∪_: conjcar conjcar -> conjcar
    _∩_: conjcar conjcar -> conjcar
    _∈_: carácter conjcar -> booleano
    esVacío: conjcar -> booleano
    cardinal: conjcar -> natural
  ecuaciones A,B:conjcar; c,c1,c2:carácter
    c1=c2 ⇒ poner(c1,poner(c2,A)) = poner(c2,A)
    c1≠c2 ⇒ poner(c1,poner(c2,A)) = poner(c2,poner(c1,A))

    quitar(c,vacío) = vacío
    c1=c2 ⇒ quitar(c1,poner(c2,A)) = quitar(c1,A)
    c1≠c2 ⇒ quitar(c1,poner(c2,A)) = poner(c2,quitar(c1,A))

    A∪vacío = A
    A∪poner(c,B) = poner(c,A∪B)

    A∩vacío = vacío
    c∈A ⇒ A∩poner(c,B) = poner(c,A∩B)
    c∉A ⇒ A∩poner(c,B) = A∩B

    c∈vacío = falso
    c1∈poner(c2,A) = (c1=c2)∨(c1∈A)

    esVacío(vacío) = verdad
    esVacío(poner(c,A)) = falso

    cardinal(vacío) = 0
    cardinal(poner(c,A)) = suc(cardinal(quitar(c,A)))
fespec
```

El conjunto obvio de generadoras es *Gen*(`conjcar`) = {`vacío`,`poner`}. Es un conjunto no libre. El hecho de añadir un carácter que ya está no modifica el conjunto (primera ecuación, caso c1=c2, propiedad idempotente). Términos obtenidos añadiendo en distinto orden los mismos caracteres son iguales (caso c1≠c2, propiedad conmutativa).

El conjunto de modificadoras es *Mod*(`conjcar`) = {`quitar`,∪,∩}. Para cada una de ellas se sigue la estrategia indicada en el paso 3. En el caso c1=c2, la operación modificadora quitar se expresa exclusivamente en términos de una operación modificadora (en este caso la misma), sin que aparezcan operaciones generadoras (se denomina ecuación derivada).

El conjunto de observadoras es *Obs*(`conjcar`) = {`∈`,`esVacío`,`cardinal`}. Las ecuaciones correspondientes siguen las pautas dictadas en el paso 4.

Consideremos la siguiente especificación del tipo “pila de elementos”:

```
espec pilas
  usa booleanos
  parámetro formal
    género elemento
  fpf
  género pila
  operaciones
    pilaVacía: -> pila
    apilar: pila elemento -> pila
    desapilar: pila -> pila
    cima: pila -> elemento
    vacía?: pila -> bool
fespec
```

Como puede verse, se trata de un tipo genérico (el género `elemento` es un parámetro formal).

Las operaciones generadoras son *Gen*(`pila`) = {`pilaVacía`,`apilar`} y son un conjunto libre (toda aplicación de apilar produce una pila distinta y todas ellas distintas de la pila vacía), por tanto las únicas ecuaciones a escribir corresponden a la operación modificadora *Mod*(`pila`) = {`desapilar`} y a las observadoras *Obs*(`pila`) = {`cima`,`vacía?`}.

Los términos canónicos del género `pila` son (`e1`, `e2`, …, son términos de tipo `elemento`):

```
pilaVacía
apilar(pilaVacía,e1)
apilar(apilar(pilaVacía,e1),e2)
...
```

Al tratar de escribir las ecuaciones relativas a la modificadora `desapilar`, probaríamos a encontrar términos canónicos congruentes a (`p` es una `pila` y `e` es un elemento):

```
desapilar(pilaVacía) = ...
desapilar(apilar(p,e)) = ...
```

En el segundo caso no hay ningún problema:

```
desapilar(apilar(p,e)) = p
```

Sin embargo, no hay ningún término canónico congruente a `desapilar(pilaVacía)`. Este término representa una situación indeseada o **situación de error**. Para salvar este problema, consideraremos la posible existencia de **operaciones parciales**, cuyo **dominio pueda ser restringido**. En el caso de la operación observadora `cima`, no tiene sentido el término `cima(pilaVacía)`. Su dominio de definición debe limitarse a los casos `cima(apilar(p,e))`, con `p` y `e` variables de géneros `pila` y `elemento`, respectivamente. Denotaremos las operaciones parciales de la siguiente forma:

```
espec pilas
  usa booleanos
  parámetro formal
    género elemento
```

```
  fpf
  género pila
  operaciones
    pilaVacía: -> pila
    apilar: pila elemento -> pila
    desapilar: pila -> pila
    parcial cima: pila -> elemento
    vacía?: pila -> bool
  dominio de definición p:pila; e:elemento
    cima(apilar(p,e))
  ecuaciones p:pila; e:elemento
    desapilar(pilaVacía) = pilaVacía
    desapilar(apilar(p,e)) = p
    cima(apilar(p,e)) = e
    vacía?(pilaVacía) = verdad
    vacía?(apilar(p,e)) = falso
fespec
```

Nótese que a las operaciones parciales se antepone la palabra clave **`parcial`**. Tras la cláusula **`dominios de definición`** se incluyen los términos para los que se extiende el dominio de las operaciones parciales. Finalmente, hay que notar que el **significado del símbolo = en las ecuaciones** cambia en el siguiente sentido: cuando el símbolo = actúe sobre dos términos en los que aparezca una operación parcial, hay que entender que expresa lo siguiente: "en caso de estar definidas las operaciones parciales, los dos términos son congruentes".

4. Verificación con especificaciones algebraicas

Un beneficio adicional que se obtiene al trabajar con TAD es la posibilidad de **modularizar las tareas de verificación formal**. Debe recordarse que los TAD se utilizan como base para la descomposición modular de programas grandes. Vemos ahora cómo esta metodología de diseño facilita la verificación de cada módulo por separado.

Consideremos un TAD especificado algebraicamente e implementado por medio de un módulo. La verificación del programa en lo concerniente a este TAD se realiza en dos fases:

- Verificación de los programas (más correctamente, módulos) usuarios del TAD. Esta tarea se realiza utilizando solamente la especificación algebraica del TAD. No debe tenerse en cuenta la implementación elegida para el tipo.
- Verificación de que la implementación del TAD es correcta, es decir, los valores del TAD son todos representados y la implementación de las operaciones verifica las propiedades expresadas mediante las ecuaciones. En esta fase no deben tenerse en cuenta el resto de módulos usuarios del TAD.

En resumen, la especificación algebraica de un tipo abstracto de datos actúa como una barrera permite descomponer la tarea de la verificación en dos niveles independientes.

La verificación de programas usuarios de TAD sigue las mismas reglas conocidas para la verificación de programas con tipos concretos sencillos, como booleanos, enteros o reales. Ahora bien, hay que tener en cuenta que:

- En un predicado pueden aparecer **variables del tipo abstracto** en cuestión, así como **operaciones del tipo** (especificadas algebraicamente).
- La regla de la asignación sigue siendo válida. Debe tenerse en cuenta que la asignación de un valor a una variable de un TAD se hará mediante la evaluación de una operación constructora del tipo correspondiente o bien de una operación observadora de otro TAD definido sobre el primero.

El principal **problema** en la verificación de programas con TAD está en la forma de simplificar predicados o, en general, **deducir** otros nuevos predicados a partir de los primeros. Cuando los tipos involucrados son sencillos (booleanos, enteros, reales), las reglas para deducir nuevos predicados son conocidas, por los conocimientos previos que de esos tipos tenemos (es decir, por nuestra experiencia con la Lógica o las Matemáticas). En el caso de TAD más complejos, las únicas reglas con que contamos para razonar son las ecuaciones de la especificación correspondiente y otras propiedades que puedan ser demostradas a partir de ellas.

Pueden deducirse dos tipos de propiedades atendiendo a la forma de demostrarlas:

- **Propiedades ecuacionales:** se deducen mediante cálculo ecuacional; éste consiste en aplicar las reglas de la congruencia $\equiv_E$, sustituyendo en cada paso una ecuación por otra nueva. Ejemplo:

```
desapilar(apilar(p,e1)) = q ⇔
p = q ⇔
apilar(p,e2) = apilar(q,e2)
```

- **Propiedades inductivas:** se deducen mediante inducción estructural; ésta consiste en un principio de inducción sobre los valores del TAD. La inducción estructural se aplica en dos fases:
  - **Base de la inducción:** se demuestra la propiedad para los valores básicos del tipo, es decir, para los generados mediante operaciones constantes o por generadoras en cuyos argumentos no aparezcan valores del tipo.
  - **Paso de inducción:** para cada valor no básico (es decir, aquél cuya operación más externa es una generadora con argumentos del tipo en cuestión), se demuestra que satisface la propiedad asumiendo la **hipótesis de inducción**; ésta consiste en asumir para cada valor no básico que sus argumentos del tipo satisfacen la propiedad.

Las propiedades ecuacionales son satisfechas por todos los modelos de la especificación. En cambio, las inductivas sólo las satisfacen los modelos sin basura.

Como ejemplo, consideremos de nuevo la especificación de las pilas:

```
espec pilas
  usa booleanos
  parámetro formal
    género elemento
  fpf
  género pila
  operaciones
    pilaVacía: -> pila
    apilar: pila elemento -> pila
    desapilar: pila -> pila
    parcial cima: pila -> elemento
    vacía?: pila -> bool
  dominio de definición p:pila; e:elemento
    cima(apilar(p,e))
  ecuaciones p:pila; e:elemento
    desapilar(pilaVacía) = pilaVacía
    desapilar(apilar(p,e)) = p
    cima(apilar(p,e)) = e
    vacía?(pilaVacía) = verdad
    vacía?(apilar(p,e)) = falso
fespec
```

Supóngase que el tipo `pila` ha sido enriquecido con la operación `inv: pila -> pila`, que invierte los elementos de una pila. Para especificar esta operación mediante ecuaciones vamos a utilizar una operación auxiliar, "apila pila", `_∇_: pila pila -> pila`, operación que apila una pila (su primer argumento) sobre otra (el segundo). Las ecuaciones que definen estas operaciones son las siguientes:

```
espec pilas
  ...
  operaciones
    ...
    _∇_: pila pila -> pila
    inv: pila -> pila
  ecuaciones ... q:pila
    ...
    pilaVacía ∇ p = p
    apilar(p,e) ∇ q = apilar(p ∇ q,e)

    inv(pilaVacía) = pilaVacía
    inv(apilar(p,e)) = inv(p) ∇ apilar(pilaVacía,e)
fespec
```

Se trata de verificar formalmente el siguiente algoritmo que implementa la operación de inversión utilizando las operaciones anteriores (`pilaVacía`, `apilar`, `desapilar`, `cima`, `vacía?`):

```
{Pre: p = p0}
q:=pilaVacía
mientrasQue not(vacía?(p)) hacer
  e:=cima(p);
  p:=desapilar(p);
  q:=apilar(q,e)
fmq
{Post: q = inv(p0)}
```

Conjeturamos que el invariante del bucle es el siguiente:

```
I: p0 = inv(q) ∇ p
```

El invariante se satisface al comienzo del bucle:

```
(p0 = p) ∧ (q = pilaVacía) ⇒[2]
(p0 = pilaVacía ∇ p) ∧ (q = pilaVacía) ⇒[3]
(p0 = inv(pilaVacía) ∇ p) ∧ (q = pilaVacía) ⇒
p0 = inv(q) ∇ p
```

El invariante conduce a la postcondición:

```
(p0 = inv(q) ∇ p) ∧ (vacía?(p)) ⇒[4]
(p0 = inv(q) ∇ p) ∧ (p = pilaVacía) ⇒
p0 = inv(q) ∇ pilaVacía ⇒[5]
p0 = inv(q) ⇒
inv(p0) = inv(inv(q)) ⇒[6]
inv(p0) = q
```

El invariante es tal invariante:

```
(p0 = inv(q) ∇ p) ∧ (¬vacía?(p)) ⇒[7]
(p0 = inv(q) ∇ p) ∧ (p = apilar(desapilar(p),cima(p))) ⇒
p0 = inv(q) ∇ apilar(desapilar(p),cima(p)) ⇒
p0 = inv(q) ∇ apilar(desapilar(p),cima(p)) ∇ pilaVacía ⇒
p0 = inv(q) ∇ apilar(desapilar(p) ∇ pilaVacía,cima(p)) ⇒
p0 = inv(q) ∇ apilar(pilaVacía ∇ desapilar(p),cima(p)) ⇒
p0 = inv(q) ∇ (apilar(pilaVacía,cima(p)) ∇ desapilar(p)) ⇒[8]
p0 = (inv(q) ∇ apilar(pilaVacía,cima(p))) ∇ desapilar(p) ⇒
p0 = inv(apilar(q,cima(p))) ∇ desapilar(p)
```

---

[2] Por la primera ecuación de definición de ∇.

[3] Por la primera ecuación de definición de `inv`.

[4] Es trivial que `vacía?(p) ⇒ (p = pilaVacía)` por las ecuaciones de definición de `vacía?`.

[5] Por inducción estructural: `p ∇ pilaVacía = p`. En efecto:

Base de la inducción (caso `p = pilaVacía`): `pilaVacía ∇ pilaVacía = pilaVacía`.

Paso de inducción (caso `p = apilar(q,e)`):

`apilar(q,e) ∇ pilaVacía =apilar(q ∇ pilaVacía,e) = apilar(q,e)`.

[6] Se demuestra igualmente por inducción estructural (se plantea como ejercicio).

[7] Se demuestra fácilmente que: `¬vacía?(p) ⇒ p = apilar(desapilar(p),cima(p))`.

[8] La propiedad asociativa de la operación ∇ se demuestra fácilmente (se plantea como ejercicio).

Para demostrar que el bucle termina hay que utilizar una nueva operación, `alt: pila -> natural` que nos da el número de elementos de una pila, como función limitadora.

```
espec pilas
  usa ...naturales
  ...
  operaciones
    ...
    alt: pila -> natural
  ecuaciones
    ...
    alt(pilaVacía) = 0
    alt(apilar(p,e)) = suc(alt(p))
fespec
```

En efecto, basta ver que `alt(p)` es estrictamente decreciente hacia su cota inferior, `0`, que se alcanza si y sólo si se verifica `vacía?(p)`.

Ejercicio: Demostrar que la operación `+` de la especificación `naturales_3`, presentada más arriba, es conmutativa.

Demostrar la corrección de la implementación de un TAD es, en general, bastante más difícil que verificar la corrección de los programas usuarios del TAD. De hecho, no existe un enfoque formal que admita una solución cómoda en la práctica. Nos limitamos, por tanto, a una introducción intuitiva de la idea de implementación de un TAD.

Implementar un TAD es simular sus valores por medio de valores de otros tipos "más concretos" (tipos ya implementados), denominados **soporte de la implementación**, y simular sus operaciones por medio de las operaciones de dichos tipos más concretos.

Por ejemplo, el tipo `conjcar`, definido en la especificación `conjuntosDeCaracteres` vista antes, puede implementarse utilizando un vector de caracteres `v[1..max]` y un contador `n`, `0 ≤ n ≤ max`. Convenimos que la parte utilizada del vector son las componentes `1..n` y que no se almacenan componentes repetidas. En este caso, los valores abstractos de tipo `conjcar` se simulan mediante pares `(v,n)`, formados por un vector y un entero. Las operaciones del tipo `conjcar` (`vacío`, `poner`, `quitar`, $\cup$, $\cap$, $\in$, `esVacío`, `cardinal`) pueden simularse fácilmente mediante operaciones que trabajan con pares `(v,n)`. El problema es cómo saber que la implementación del tipo `conjcar` es correcta.

Para asegurarse de la corrección de la implementación es necesario, en primer lugar, establecer la **función de abstracción**. Esta es una correspondencia entre los valores del tipo soporte y los del tipo especificado. Si denotamos por A_{ESPEC} un álgebra isomorfa a T_{ESPEC} y por A_{SOP} el álgebra definida por el tipo soporte, la función de abstracción es una familia de aplicaciones $\Phi = \{\Phi_g\}_{g \in G}$:

$$\Phi_g: A_{SOP_g} \rightarrow A_{ESPEC_g}$$

En el ejemplo anterior, a cada par `(v,n)` le corresponde un conjunto de caracteres (abstracto). Nótese que varios pares distintos `(v,n)` pueden corresponder a un mismo conjunto (por ejemplo, si difieren sólo en el orden de los elementos almacenados en las componentes `1..n` de `v`). Además, hay pares `(v,n)` que no representan a ningún conjunto (por ejemplo, los que tienen elementos repetidos en las componentes `1..n` de `v` o los que tengan un valor negativo de `n`). En este caso el tipo soporte tiene basura desde el punto de vista del tipo implementado.

En general, las características de la función de abstracción son:

- **suprayectiva:** todos los valores de A_{ESPEC_g} deben ser representados;
- **no necesariamente inyectiva:** un valor de A_{ESPEC_g} puede tener varios valores soporte;
- **parcial:** no todo valor del tipo soporte representa un valor del tipo implementado;
- **homomórfica:** es decir, debe conmutar con las operaciones:
  $\forall\ \sigma: \rightarrow g,\ \Phi_g(\sigma^{A_{SOP}}) = \sigma^{A_{ESPEC}}$
  $\forall\ \sigma: g_1 \ldots g_n \rightarrow g,\ \Phi_g(\sigma^{A_{SOP}}(t_1,\ldots,t_n)) = \sigma^{A_{ESPEC}}(\Phi_{g_1}(t_1),\ldots,\Phi_{g_n}(t_n))$.

La suprayectividad es obviamente imposible si el cardinal del conjunto de valores del tipo representado es infinito. En ese caso, nos conformaremos con representaciones de un subconjunto de valores.

En particular, la implementación de un TAD **no debe introducir confusión**, es decir, no puede ocurrir que a dos valores distintos del tipo implementado se asocie un mismo valor del tipo soporte.

La implementación es correcta si el álgebra A_{ESPEC} es isomorfa a la que resulta del álgebra A_{SOP} tras realizar las siguientes modificaciones:

1. enriquecerla con las operaciones del tipo implementado;
2. restringirla a la subálgebra de valores que representan algún valor del tipo implementado, es decir, eliminar la basura;
3. identificar las distintas representaciones que puede tener cada valor del tipo implementado (se consigue mediante la incorporación de las ecuaciones de A_{ESPEC}).

Anexo 5: Transformación de algoritmos recursivos en iterativos

Indice

1. Transformación de algoritmos recursivos finales
2. Transformación de algoritmos recursivos lineales (no finales)
3. Transformación de algoritmos recursivos múltiples: un caso particular
4. Transformación de algoritmos recursivos múltiples: caso general

1. Transformación de algoritmos recursivos finales

El estudio de una técnica general de transformación de algoritmos recursivos en iterativos es **interesante** por diversas razones. En primer lugar, algunos compiladores generan código ineficiente al compilar programas recursivos, por ello puede ser conveniente realizar la transformación a iterativo antes de compilar. En segundo lugar, hay lenguajes de programación que no permiten el diseño de algoritmos recursivos; en esos casos es indispensable disponer de una técnica de transformación de algoritmos recursivos a iterativos. Además, la técnica presentada tiene interés didáctico puesto que integra la utilización de razonamientos formales sobre algoritmos recursivos con el uso de estructuras de datos arborescentes.

Un algoritmo recursivo se dice **lineal** si cada llamada recursiva sólo genera, como mucho, otra llamada. Si, además, esa llamada es la última operación que se efectúa, entonces el algoritmo se llama recursivo **final**.

Así, un esquema general de algoritmo recursivo final es el siguiente:

```
procedimiento r(ent x:tx)
principio
  selección
    C0(x): A0(x);
    C1(x): A1(x);
           r(sig1(x));
    C2(x): A2(x);
           r(sig2(x));
    ...
    Cn(x): An(x);
           r(sign(x))
  fselección
fin
```

En el esquema anterior, `tx` es un tipo previamente definido; `C0(x)`, `C1(x)`, …, `Cn(x)` son expresiones booleanas que dependen del valor del parámetro `x`; `A0(x)`, `A1(x)`, …, `An(x)` son secuencias de instrucciones que no incluyen llamadas al algoritmo `r`; y `sig1(x)`, …, `sign(x)` son expresiones de tipo `tx` que, obviamente, dependen del valor del parámetro `x`.

Para deducir la versión iterativa del esquema anterior, basta observar que las **sucesivas llamadas recursivas** al algoritmo `r` pueden **ordenarse en una secuencia** (puesto que cada llamada recursiva sólo genera, como mucho, otra llamada) de ejecución de instrucciones. Esas instrucciones se ejecutarán para diferentes valores `x0` (= `x`), `x1` (= `sigi(x0)`, si `Ci(x0)` es `verdad`), …, `xn` (= `sigj(xn-1)`, si `Cj(xn-1)` es `verdad`). Para los valores `xi` anteriores al último de la secuencia, hay que ejecutar las instrucciones `Aj(xi)` tales que `Cj(xi)` es `verdad`. Para el último elemento de esa secuencia, `xn` (al que se llega cuando se verifica `C0(xn)`) hay que ejecutar `A0(xn)`.

Por tanto, el siguiente algoritmo iterativo ejecuta las mismas instrucciones que el recursivo `r`:

```
procedimiento rIter(ent x:tx)
variable v:tx
principio
  v:=x;
  mientrasQue not C0(v) hacer
    selección
      C1(v): A1(v);
             v:=sig1(v);
```

```
      C2(v): A2(v);
             v:=sig2(v);
      ...
      Cn(v): An(v);
             v:=sign(v)
    fselección
  fmq;
  A0(v)
Fin
```

2. Transformación de algoritmos recursivos lineales (no finales)

El siguiente es un esquema general de algoritmo lineal:

```
procedimiento r(ent x:tx)
principio
  si C(x) entonces
    A(x)
  sino
    B1(x);
    r(sig(x));
    B2(x)
  fsi
fin
```

Como puede verse, cada llamada al algoritmo genera, como mucho, una llamada recursiva (`r(sig(x))`, cuando la expresión booleana `C(x)` es `falsa`; si `C(x)` es `verdad`, no se genera ninguna llamada recursiva).

Al igual que en el caso particular de algoritmo recursivo final (cuando `B2(x)` es una secuencia vacía de instrucciones), puede construirse una secuencia de datos `x0` (= `x`), `x1`, ..., `xn` que corresponden a los valores del parámetro `x` en las sucesivas llamadas recursivas al algoritmo `r`. Para cada uno de los valores `xi` de esa secuencia hay que ejecutar `B1(xi)` antes de tratar el resto de la secuencia y `B2(xi)` después de tratar el resto de la secuencia. En consecuencia, hay que recorrer la secuencia `x0`, `x1`, ..., `xn` dos veces; una de principio a fin y otra al revés. Para ello, cuando se recorre de principio a fin, además de ejecutar `B1(xi)` podemos almacenar el valor `xi` en una pila de datos de tipo `tx`. Así, una vez recorrida la secuencia de principio a fin, basta con ir extrayendo datos de la pila para poder recorrer la secuencia en sentido inverso y ejecutando `B2(xi)` para valores decrecientes de `i`.

En definitiva, la versión iterativa del algoritmo recursivo lineal anterior es la siguiente:

```
procedimiento rIter(ent x:tx)
variables v:tx;
          p:pila_de_tx
principio
  creaVacía(p);
  v:=x;
  mientrasQue not C(v) hacer
    B1(v);
    apilar(p,v);
    v:=sig(v)
  fmq;
  A(v);
  mientrasQue not esVacía(p) hacer
    v:=cima(p);
    desapilar(p);
    B2(v)
  fmq
fin
```

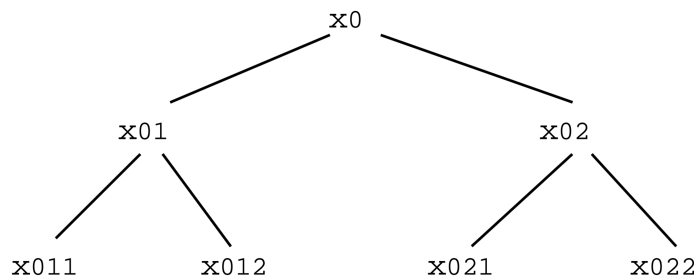
3. Transformación de algoritmos recursivos múltiples: un caso particular

En este apartado y el siguiente vamos a estudiar la transformación en iterativos de aquellos algoritmos, denominados recursivos **múltiples**, que pueden generar más de una llamada recursiva en cada llamada. Por simplificar la exposición nos centraremos en el caso de algoritmos que generan en cada llamada, a lo sumo, dos nuevas llamadas recursivas. La consideración de más de dos llamadas recursivas puede hacerse de forma análoga, sin demasiadas complicaciones.

Veremos, en primer lugar, el **caso particular** en el que en cada llamada al algoritmo se realizan una secuencia de operaciones (no recursivas), `A(x)`, y después dos llamadas recursivas consecutivas y termina el algoritmo. El esquema de tal algoritmo recursivo es el siguiente:

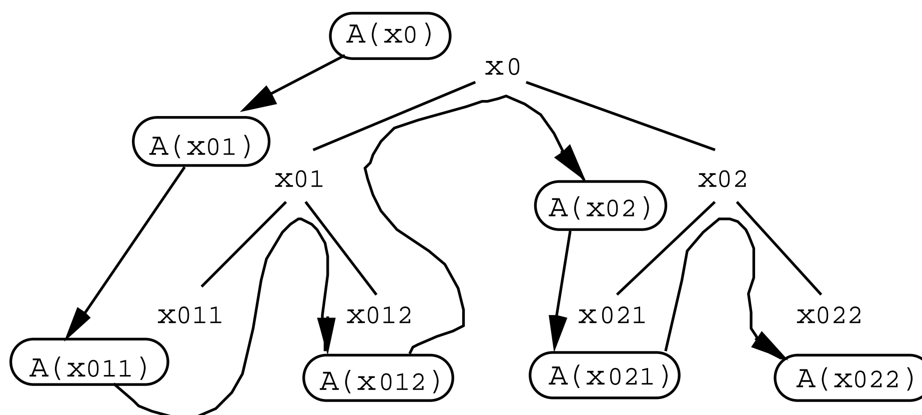
```
procedimiento r(ent x:tx)
principio
  si C(x) entonces
    A(x);
    r(sig1(x));
    r(sig2(x))
  fsi
fin
```

En este caso, los diferentes valores del parámetro `x` en todas las llamadas recursivas generadas por una llamada inicial al algoritmo `r` pueden representarse en un árbol binario de datos de tipo `tx` en la forma siguiente:



Como puede verse, se ha denotado por `xn1` el valor de `sig1(xn)` y por `xn2` el valor de `sig2(xn)`.

La ejecución del algoritmo `r` para un valor del parámetro `x` igual a `x0` puede asociarse con un **recorrido en pre-orden del árbol** anterior de forma que, al visitar el valor `xn` hay que ejecutar `A(xn)` y después hay que recorrer su subárbol izquierdo y después su subárbol derecho.



La escritura de un algoritmo iterativo de recorrido en pre-orden de un árbol binario es muy sencilla utilizando, de nuevo, una **pila auxiliar de datos de tipo `tx`** que almacenará los elementos **raíz de aquellos subárboles que están todavía por recorrer**. Inicialmente se apila la raíz del árbol. Después, mientras la pila no sea vacía, se desapila un valor, `v`, y si se verifica `C(v)` entonces se ejecutan las instrucciones asociadas a ese valor, `A(v)`, y se apilan sucesivamente la raíz del subárbol derecho, `sig2(v)`, y la raíz del subárbol izquierdo, `sig1(v)`.

```
procedimiento rIter(ent x:tx)
variables v:tx;
```

```
           p:pila_de_tx
principio
  si C(x) entonces
    creaVacía(p);
    apilar(p,x);
    mientrasQue not esVacía(p) hacer
      v:=cima(p);
      desapilar(p);
      si C(v) entonces
        A(v);
        apilar(p,sig2(v));
        apilar(p,sig1(v))
      fsi
    fmq
  fsi
fin
```

Como ejemplo de aplicación, recordemos el **algoritmo rápido de ordenación** (*quicksort*) en su versión recursiva (véase, por ejemplo, el libro de Aho *et al*):

```
constante maxN = 100 {dimensión del vector}
tipos índice = 1..maxN;
      ítem = registro
               clave:entero;
               info:tpInfo
             freg;
      tpVector = vector[1..maxN] de ítem

procedimiento ordenar(e/s v:tpVector; ent n:índice)
{Ordena los elementos 1..n del vector v por valores crecientes de clave.}

  procedimiento rápido(e/s v:tpVector; ent izq,der:índice)
  {Ordena los elementos izq..der de v por valores crecientes de clave.}
  variables i,d:índice;
            unItem,aux:ítem
  principio
    si izq ≤ der entonces
      unItem:=v[izq];
      i:=izq+1;
      d:=der;
      mientrasQue i≠d+1 hacer
        si v[i].clave ≤ unItem.clave entonces
          i:=i+1
        sino
          si v[d].clave ≥ unItem.clave entonces
            d:=d-1
          sino {(v[i].clave>unItem.clave)∧(v[d].clave<unItem.clave)}
            aux:=v[i];
            v[i]:=v[d];
            v[d]:=aux;
            i:=i+1;
            d:=d-1
          fsi
        fsi
      fmq;
      v[izq]:=v[d];
      v[d]:=unItem;
      rápido(v,izq,d-1);
      rápido(v,d+1,der)
    fsi
  fin

principio {de ordenar}
  rápido(v,1,n)
fin
```

Aplicando el esquema de transformación visto anteriormente, se obtiene la siguiente versión iterativa del algoritmo:

```
constante maxN = 100 {dimensión del vector}
tipos índice = 1..maxN;
      ítem = registro
               clave:entero;
               info:tpInfo
             freg
      tpVector = vector[1..maxN] de ítem

      par = registro
              iz,de:índice
            freg

procedimiento ordenar_iter(e/s v:tpVector; ent n:índice)
{Ordena los elementos 1..n de v por valores crecientes de clave.}
variables p:pila_de_par;
          unP:par;
          izq,der,i,d:índice;
          unItem,aux:ítem
principio
  creaVacía(p);
  unP.iz:=1;
  unP.de:=n;
  apilar(p,unP);
  mientrasQue not esVacía(p) hacer
    unP:=cima(p);
    izq:=unP.iz;
    der:=unP.de;
    desapilar(p);
    si izq ≤ der entonces
      unItem:=v[izq];
      i:=izq+1;
      d:=der;
      mientrasQue i≠d+1 hacer
        si v[i].clave ≤ unItem.clave entonces
          i:=i+1
        sino
          si v[d].clave ≥ unItem.clave entonces
            d:=d-1
          sino {(v[i].clave>unItem.clave)∧(v[d].clave<unItem.clave)}
            aux:=v[i];
            v[i]:=v[d];
            v[d]:=aux;
            i:=i+1;
            d:=d-1
          fsi
        fsi
      fmq;
      v[izq]:=v[d];
      v[d]:=unItem;
      unP.iz:=d+1;
      unP.de:=der;
      apilar(p,unP);
      unP.iz:=izq;
      unP.de:=d-1;
      apilar(p,unP)
    fsi
  fmq
fin
```

4. Transformación de algoritmos recursivos múltiples: caso general

En el caso particular anterior, se han considerado algoritmos que se corresponden con un recorrido en pre-orden del árbol binario de valores del parámetro, es decir, con una operación del tipo:

```
operación
  preOrden: arbin -> lista
ecuaciones e:elemento; ai,ad:arbin
  preOrden(vacío) = []
  preOrden(plantar(e,ai,ad)) = [e] & preOrden(ai) & preOrden(ad)
```

Ahora vamos a considerar el caso general de un algoritmo con dos llamadas recursivas. De igual forma, se puede asociar un **árbol binario de valores del parámetro** del algoritmo e identificar el problema dado con el de una **operación sobre el árbol** de tipo general como la siguiente:

```
      operación
        r: arbin -> lista
      ecuaciones e:elemento; ai,ad:arbin
```

$[e_0^1]$ `r(vacío) = L0`

$[e_0^2]$ `r(plantar(e,ai,ad)) = f(plantar(e,ai,ad),r(ai),r(ad))`

Donde `L0` representa una constante cualquiera de género `lista` y `f` es una operación cualquiera con el siguiente perfil:

```
operación
  f: arbin lista lista -> lista
```

En notación algorítmica, la operación anterior se puede implementar de forma recursiva de la siguiente forma:

```
función r(a:arbin) devuelve lista
variables ai,ad:arbin
  función L0 devuelve lista
  ...
  función f(a:arbin; L1,L2:lista) devuelve lista
  ...
principio
  si esVacío(a) entonces
    devuelve(L0)
  sino
    subIzq(a,ai);
    subDer(a,ad);
    devuelve(f(a,r(ai),r(ad)))
  fsi
fin
```

Un ejemplo de algoritmo recursivo que responde al esquema general anterior es el siguiente:

```
procedimiento r(ent x:tx)
principio
  si C(x) entonces
    A(x)
  sino
    B1(x);
    r(sig1(x));
    B2(x);
    r(sig2(x));
    B3(x)
  fsi
fin
```

El algoritmo anterior tiene asociado un recorrido del árbol de valores del parámetro `x` en el que cada nodo se visita tres veces: antes de recorrer su subárbol izquierdo, después de tratar su subárbol izquierdo, y después de tratar su subárbol derecho.

Antes de deducir la versión iterativa del algoritmo r, supongamos que existe un **valor especial del tipo lista**, que denotaremos por ⊥ y llamaremos **lista indefinida** (si resulta extraño, puede interpretarse/implementarse mediante un par o registro con dos campos: uno de tipo lista y otro booleano que dice si la lista es indefinida o no).

Vamos a emplear una **pila de pares** (es decir, de registros con dos campos) formados por: un **árbol binario no vacío** y una **lista**. Los valores del tipo par (árbol, lista) los generaremos con la siguiente operación:

```
operación
  par: arbin lista -> parArbLis
```

La **interpretación intuitiva** de la pila de pares es la siguiente:

- un par `par(a,⊥)` en la pila indica que estamos calculando `r(a)` y aún no sabemos nada sobre sus hijos;
- un par `par(a,LL)`, con LL ≠ ⊥, en la pila indica que estamos calculando `r(a)` y la solución de su hijo izquierdo ya ha sido calculada, y vale LL, por lo que ahora estamos calculando la solución de su hijo derecho.

Definimos, en primer lugar, la siguiente **operación auxiliar**:

```
      operación
        r1: lista pila -> lista
      ecuaciones L,LL:lista; a:arbin; p:pila
[e_1^1 ]   r1(L,pilaVacía) = L
[e_1^2 ]   r1(L,apilar(p,par(a,⊥))) = r1(r(subDer(a)),apilar(p,par(a,L))
[e_1^3 ]   si LL≠⊥ ⇒ r1(L,apilar(p,par(a,LL))) = r1(f(a,LL,L),p)
```

La **interpretación intuitiva** de la operación anterior es la siguiente:

- `r1(L,p)` representa lo siguiente: L es la solución para uno de los hijos del árbol a que está en la cima de p;
  - si la pareja de este árbol a es ⊥, entonces L es la solución para el hijo izquierdo de a, y lo que vamos a hacer es sustituir la cima `par(a,⊥)` por la cima `par(a,L)` y pasaremos a resolver el hijo derecho de a;
  - si, en cambio, la cima es `par(a,LL)`, con LL≠⊥, entonces LL es la solución del hijo izquierdo de a, L es la solución del hijo derecho y ya se puede calcular la solución de a como `f(a,LL,L)`.

A continuación, vamos a definir una nueva **operación auxiliar** (parcial):

```
      operación
        parcial r2: arbin lista pila -> lista
      dominio de definición a:arbin; L:lista; p:pila
        r2(a,L,p) está definido sólo si (vacío?(a)=verdad) ∨ (L=⊥)
      ecuaciones a:arbin; L:lista; p:pila
[e_2^1 ]   r2(a,⊥,p) = r1(r(a),p)
[e_2^2 ]   L≠⊥ ⇒ r2(vacío,L,p) = r1(L,p)
```

La **interpretación intuitiva** de la operación anterior es la siguiente:

- la operación r2 distingue entre cuándo aplicamos r1 para resolver un árbol sin tener todavía nada calculado (ecuación $[e_2^1]$) y cuándo la aplicamos a partir de una lista ya calculada (ecuación $[e_2^2]$).

Lema 1. Si encontramos una solución iterativa de r2, tenemos una solución iterativa de r, en la forma siguiente:

```
       [e_1^1]                         [e_2^1]
r(a)    =    r1(r(a),pilaVacía)   =    r2(a,⊥,pilaVacía)
```
♦

Lema 2. La operación `r2(a,L,p)` admite una definición recursiva final en el caso `(a≠vacío)∧(L=⊥)`. En efecto:

```
                          [e_2^1]             [e_0^2]
a≠vacío ⇒ r2(a,⊥,p)   =   r1(r(a),p)   =
```

```
= r1(f(a,r(subIzq(a)),r(subDer(a))),p)   [e_1^3] =
= r1(r(subDer(a)),apilar(p,par(a,r(subIzq(a)))))   [e_1^2] =
= r1(r(subIzq(a)),apilar(p,par(a,⊥)))   [e_2^1] =
= r2(subIzq(a),⊥,apilar(p,par(a,⊥)))
```
♦

Lema 3. La operación `r2(a,L,p)` admite una definición recursiva final en el caso `(a=vacío)∧(L=⊥)`. En efecto:

```
r2(vacío,⊥,p)  [e_2^1] =  r1(r(vacío),p)  [e_0^1] =  r1(L0,p)  [e_2^2] =  r2(vacío,L0,p)
```
♦

Lema 4. La operación `r2(a,L,p)` admite una definición recursiva final en el caso `(a=vacío)∧(L≠⊥)∧(cima(p)=par(aa,⊥))`. En efecto:

```
L≠⊥ ⇒ r2(vacío,L,apilar(p,par(a,⊥)))   [e_2^2] =
      = r1(L,apilar(p,par(a,⊥)))   [e_1^2] =
      = r1(r(subDer(a)),apilar(p,par(a,L)))   [e_2^1] =
      = r2(subDer(a),⊥,apilar(p,par(a,L)))
```
♦

Lema 5. La operación `r2(a,L,p)` admite una definición recursiva final en el caso `(a=vacío)∧(L≠⊥)∧(cima(p)=par(aa,LL))∧(LL≠⊥)`. En efecto:

```
(L≠⊥)3(LL≠⊥) ⇒ r2(vacío,L,apilar(p,par(a,LL)))   [e_2^2] =
               = r1(L,apilar(p,par(a,LL)))   [e_1^3] =
               = r1(f(a,LL,L),p)   [e_2^2] =   r2(vacío,f(a,LL,L),p)
```
♦

Lema 6. La operación `r2(a,L,p)` admite la siguiente solución trivial o directa (es decir, no recursiva) en el caso `(a=vacío)∧(L≠⊥)∧(p=pilaVacía)`:

```
L≠⊥ ⇒ r2(vacío,L,pilaVacía)  [e_2^2] =  r1(L,pilaVacía)  [e_1^1] =  L
```
♦

Ahora, utilizando los lemas 2, 3, 4, 5 y 6, se obtiene la siguiente solución recursiva final para `r2`:

```
función r2Final(a:arbin; L:lista; p:pila) devuelve lista
{Pre: esVacío(a)∨(L=⊥)}
variable LL:lista;
         unPar:registro
                 ar:arbin;
                 li:lista
               freg
principio
  selección
    (L≠⊥) and esVacía(p) {⇒ esVacío(a)}: devuelve(L); {lema 6}
    not esVacío(a) {⇒ L=⊥}: devuelve (r2(subIzq(a),L,apilar(p,par(a,⊥)))); {lema 2}
    (L=⊥) and esVacío(a): devuelve (r2(a,L0,p)); {lema 3}
    (L≠⊥) and not esVacía(p) {⇒ esVacío(a)}:
```

```
                              unPar:=cima(p);
                              a:=unPar.ar;
                              LL:=unPar.li;
                              desapilar(p);
                              si LL=⊥ entonces {lema 4}
                                devuelve (r2(subDer(a),⊥,apilar(p,par(a,L))))
                              sino {lema 5}
                                devuelve (r2(árbolVacío,f(a,LL,L),p))
                              fsi
  fselección
fin
```

El método de transformación de algoritmos recursivos finales en iterativos, visto al principio de esta lección, se puede aplicar al algoritmo `r2Final`, obteniéndose:

```
función r2Iter(a:arbin; L:lista; p:pila) devuelve lista
{Pre: esVacío(a)∨(L=⊥)}
variables ll:lista;
          unPar:registro
                  ar:arbin;
                  li:lista
                freg;
          aa:arbin
principio
  mientrasQue (L=⊥) or not esVacía(p) hacer
    selección
      not esVacío(a):           apilar(p,par(a,⊥));
                                subIzq(a,aa);
                                a:=aa;
      (L=⊥) and esVacío(a):     L:=L0;
      (L≠⊥) and not esVacía(p): unPar:=cima(p);
                                a:=unPar.ar;
                                LL:=unPar.li;
                                desapilar(p);
                                si LL=⊥ entonces
                                  apilar(p,par(a,L));
                                  subDer(a,aa);
                                  a:=aa;
                                  L:=⊥
                                sino
                                  L:=f(a,LL,L);
                                  creaVacío(a)
                                fsi
    fselección
  fmq;
  devuelve L
fin
```

Finalmente, podemos escribir la versión iterativa del algoritmo `r` utilizando `r2Iter` (de acuerdo con el lema 1):

```
función rIter(a:arbin) devuelve lista
principio
  devuelve r2Iter(a,⊥,pilaVacía)
fin
```

Se proponen los siguientes ejercicios:

- Escribir un algoritmo iterativo que implemente el “recorrido de tres visitas” de un árbol binario:

```
    operación
      vis3: arbin -> lista
    ecuaciones e:elemento; ai,ad:arbin
```

```
vis3(vacío) = []
vis3(plantar(e,ai,ad)) = [e] & vis3(ai) & [e] & vis3(ad) & [e]
```

- Escribir un algoritmo iterativo que genere una lista conteniendo todas las hojas del árbol:

```
operación
  hojas: arbin -> lista
ecuaciones e:elemento; ai,ad:arbin
  hojas(vacío) = []
  vacío?(ai) ∧ vacío?(ad) ⇒ hojas(plantar(e,ai,ad)) = [e]
  ¬vacío?(ai) ∨ ¬vacío?(ad) ⇒ hojas(plantar(e,ai,ad)) = hojas(ai) & hojas(ad)
```

- Escribir un algoritmo iterativo que resuelva el problema de las torres de Hanoi.

Anexo 6: Chuletas de sintaxis de especificación y pseudocódigo

Indice

1. Sintaxis para una especificación de TAD en lenguaje natural
2. Sintaxis del pseudocódigo

1. Sintaxis para una especificación de TAD en lenguaje natural

espec nombreEspecificación
[usa especificación1, especificación2, …]
[parámetro formal
  género nombreGénero1, …
  [operación
    [parcial] [_] nombreOperación **[_] : [** dominio **]** → rango
    {Descripción del dominio y el rango de la operación, y en su caso de las situaciones que hacen la operación parcial}
    ….
  ]
fpf]

género nombreGénero1, … *{descripción del TAD}*

operaciones

  [parcial] nombreOperación**: [** dominio **]** → rango
    {Descripción del dominio y el rango de la operación.
    *[**Parcial:** descripción de las situaciones que hacen la operación parcial]}*
    ...
  [parcial] nombreOperación _ **:** géneroArg nombreArg → rango
    {Descripción del dominio y el rango de la operación.
    *[**Parcial:** descripción de las situaciones que hacen la operación parcial]}*
    ...
  [parcial] _ nombreOperación**:** géneroArg1 nombreArg1**,** géneroArg2 nombreArg2 → rango
    {Descripción del dominio y el rango de la operación.
    *{ **Parcial:** descripción de las situaciones que hacen la operación parcial]}*
    ...

    ...
fespec

Donde:

[ ] Significa que lo que está entre los corchetes es opcional, puede que aparezca o no

Dominio es una lista de elementos separados por '**,**' y donde cada elemento describe un argumento de la operación, indicando el género y nombre del argumento.

Rango es el nombre de un género.

El símbolo **'_'** indica la posición de los argumentos respecto al nombre de la operación. Se utiliza para indicar operaciones con notación prefija sin paréntesis o con notación infija (ejemplo: ¬_ , _≤_.).

2. Sintaxis del pseudocódigo

1. Tipos de datos predefinidos

```
booleano {con los valores verdad y falso; escribimos los comentarios entre llaves}
carácter
natural  {incluímos el 0 en los naturales}
entero
real
cadena   {son secuencias de caracteres de longitud arbitraria (vacías o no)}
```

2. Definición de constantes

```
constantes
  letraA = 'A'; maxNum = 100  {usamos el punto y coma para separar}
  hoy = "martes"  {es una cadena no vacía}
  pi = 3.1416
```

3. Definición de nuevos tipos de datos

```
tipos
  mes = (ene,feb,mar,abr,may,jun,jul,ago,sep,oct,nov,dic)  {tipo enumerado}
  mesVerano = jul..sep  {tipo subrango de otro tipo discreto}
  día = 1..31
  fecha = registro   {tipo registro (o estructura), agregación de campos}
            elDía: día;
            elMes: mes;
            elAño: natural
          freg
  pluviometría = vector[mes] de real    {tipo vector (array)}
  fiestas = vector[1..maxNum] de fecha
  secFechas = fichero de fecha  {tipo fichero binario de fechas}
  entrada = fichero de texto  {tipo fichero de texto, i.e., secuencia de líneas
                               (una línea es una secuencia de caracteres)}
```

4. Declaración de variables

```
variables
  contador: natural:= 0     {podemos inicializar la variable al declararla}
  éxito,error: booleano
  nombre: cadena
  cadenaVacía: cadena:= ""  {la variable se inicializa con la cadena vacía, ""}
  cumpleaños,aniversario: fecha
  festivos,patronos: fiestas
```

5. Instrucción de asignación

```
contador:= contador+1
error:= falso; éxito:= verdad;
éxito:= not error and (contador>3) or éxito
cumpleaños.elDía:= 13
nombre:= "Juan"
festivos[2].elMes:= sucesor(ene)  {operaciones sucesor y predecesor para enumerados}
cumpleaños:= aniversario {asignación entre registros: todos los campos se asignan}
festivos:= patronos {asignación entre vectores: todas las componentes se asignan}
```

6. Operaciones con datos cadena

```
variables apellido, resto: cadena
          i: natural
          letra: carácter
```

Comparaciones (por orden alfabético, considerando todos los caracteres según su orden en la tabla del código ASCII y sus extensiones):

```
"Costa" = apellido   apellido ≠ "Po3$"    apellido < resto
apellido ≤ resto     apellido > resto     apellido ≥ resto
```

Concatenación:

```
resto:= resto + apellido  {resto toma el valor resultante de concatenar su valor
                            previo con apellido}
apellido:= "de " + apellido  {se antepone a la cadena apellido la cadena "de "}
resto:= "hol" + "a"     {resto toma el valor "hola"}
```

Longitud:

```
i:= long(aux)  {long devuelve la longitud de la cadena, es decir, su número de
                caracteres; la longitud de la cadena vacía, "", es 0}
```

Carácter i-ésimo y subcadenas:

```
letra:= apellido[i]  {es el i-ésimo carácter de la cadena; 1 ≤ i ≤ long(apellido)}

letra:= apellido[1] {primer carácter de la cadena, que debe ser no vacía}
resto:= apellido[2..long(apellido)] {subcadena de apellido desde el carácter 2
                                     hasta el último}
resto:= apellido[i..j] {subcadena desde el carácter i al j; 1 ≤ i ≤ j ≤ long(apellido)}
```

7. Instrucciones de entrada/salida

```
escribir("Introduzca su edad: ")  {escribe el mensaje en el dispositivo estándar
                                   de salida (pantalla, por ejemplo)}
leerLínea(edad) {si edad es una variable entera, lee un valor de tipo entero desde
                 el dispositivo estándar de entrada (teclado, por ejemplo), lo asigna
                 a edad y luego lee la marca de fin de línea}
leerLínea(nombre) {si nombre es una variable de tipo cadena, lee todos los caracteres
                   anteriores al fin de línea desde el dispositivo estándar de entrada
                  (teclado), los asigna a nombre y luego lee la marca de fin de línea}

escribirLínea("La edad introducida es ", edad) {escribe el mensaje y el valor de edad
                                                 y luego salta de línea}
escribirLínea("Su nombre es:", nombre){escribe el mensaje entrecomillado, después el
                                        valor de la cadena nombre y luego salta de línea}
escribir("Introduce tres letras: ") {escribe el mensaje}
leer(a,b,c)  {si a,b,c son variables de tipo carácter, lee tres caracteres y los
              asigna a las variables a,b,c}
leerLínea {lee la marca de fin de línea}
```

8. Instrucciones condicionales

```
si <condición> entonces
  <secuencia de acciones>
fsi

si <condición> entonces
  <secuencia de acciones>
sino
  <secuencia de acciones>
fsi

si <condición> entonces
  <secuencia de acciones>
sino_si <condición> entonces
  <secuencia de acciones>
sino_si <condición> entonces
  <secuencia de acciones>
...
sino
  <secuencia de acciones>
fsi
```

```
selección
  <condición 1>: <secuencia de instrucciones>;
  <condición 2>: <secuencia de instrucciones>;
  ...
  <condición n>: <secuencia de instrucciones>;
  [ otrosCasos: <secuencia de instrucciones> ]
fselección
{se requiere que las condiciones sean excluyentes entre sí;
 una vez ejecutada la secuencia de instrucciones de la primera
 condición verdadera se termina la instrucción de selección}
```

9. Instrucciones iterativas

```
para <variable_de_tipo_discreto>:=<valor_inicial> hasta <valor_final> hacer
  <secuencia de acciones>
fpara

para <variable_de_tipo_disc>:=<valor_inicial> descendiendo hasta <valor_final> hacer
  <secuencia de acciones>
fpara

mientrasQue <condición> hacer
  <secuencia de acciones>
fmq

repetir
  <secuencia de acciones>
hastaQue <condición>
```

10. Procedimientos y funciones

```
procedimiento <nombre>(ent <parámetros_1>:<tipo_1>;
                       sal <parámetros_2>:<tipo_2>;
                       e/s <parámetros_3>:<tipo_3> ... )
{ent,sal,e/s significa parámetro de entrada, salida, entrada y salida, respec.}
<declaraciones locales de constantes, tipos, variables, proced., funciones...>
principio
  <secuencia de acciones>
fin
{un procedimiento es una acción o instrucción virtual;
 el programa principal es un procedimiento sin parámetros}

función <nombre>(<parám_1>:<tipo_1>; <parám_2>:<tipo_2> ...) devuelve <tipo_fun>
<declaraciones locales de constantes, tipos, variables, proced., funciones...>
principio
  <secuencia de acciones>
  devuelve <valor_de_tipo_fun>   {tras devolver el valor la función termina}
fin
{una función es un valor virtual, es decir, se evalúa dentro de una expresión
 y el resultado es un valor}
```

11. Módulos

```
módulo tablas
importa <lista de módulos que necesita usar>
exporta
  {parte pública: constantes, nombres de tipos, encabezamientos de proced. y func...}
  ...
implementación
  {parte privada: se incluyen las def. de los tipos cuyos nombres aparecen en la
   parte pública, otros tipos privados, el código de procedimientos y funciones...}
  ...
fin
```

12. Módulos genéricos

```
módulo genérico listasGenéricas
parámetros
  tipo elemento
  con función "<"(e1,e2:elemento) devuelve booleano
exporta
  tipo lista
  procedimiento crear(sal l: lista)
  procedimiento añadirÚltimo(e/s l: lista; ent e:elemento)
  ...
implementación
  ...
fin
```

13. Uso de módulos genéricos

a. Para definir otro tipo también genérico:

```
módulo genérico pilasGenéricas
importa listasGenéricas
. . .
parámetro tipo elemento
exporta
  tipo pila
. . .
implementación
  tipo pila = listasGenéricas.lista {el tipo exportado en listasGenéricas}

  procedimiento crear(sal p:pila)
  principio
    listasGenéricas.crear(p)
  fin
. . .

fin {del módulo}
```

b. Concretar un modulo genérico y utilizarlo:

```
. . .
importa pilasGenéricas;
modulo pila_naturales = pilasGenéricas(natural); {que ofrece el tipo: pila}
{admitimos también esta otra sintaxis aleternativa:}
modulo pila_naturales concreta pilasGenéricas(natural)
. . .
{para declarar variables o parámetros:}
p: pila;    {o bien: p:pila_naturales.pila}
{para utilizar sus operaciones:}
crear(p);   {o bien: pila_naturales.crear(p)}
. . .
```

14. Datos puntero

```
tipo pila = ↑unDato {tipo puntero (o referencia) a unDato}
     unDato = registro
                dato:natural;
                siguiente:pila
              freg
variables p,q: pila
```

Valor especial (constante) de cualquier tipo puntero: `nil` (ninguna dirección)

```
p:=nil
```

Instrucciones con punteros:

```
nuevoDato(p);
p↑.dato:=3;
```

```
q:=p;
disponer(p);

si p=q entonces ...
```

15. Instrucciones de creación y uso de ficheros

A continuación se muestra un esquema básico del trabajo con ficheros de texto o binarios, junto con las instrucciones disponibles en pseudocódigo para trabajar con ellos.

- **Ficheros de texto**

```
variables
   f: fichero de texto
   d: carácter
   nombre: cadena
...
principio
{Asociar la variable f con el fichero externo de nombre "Leeme.txt":}
asociar(f, "Leeme.txt");

{Inicializar el fichero para escritura: el fichero externo es creado vacío}
iniciarEscritura(f);

   {Para escribir en un fichero de texto se considerarán también disponibles las
mismas instrucciones que para escribir en pantalla o salida estándar, con el mismo
significado y la misma sintaxis salvo porque se les añade un primer parámetro que es
la variable fichero. Ejemplo:}
escribir(f,d); {escribe en f el caracter d al final del fichero}
escribir(f,nombre); {escribe en f la cadena nombre al final del fichero}
. . .

   {Inicializar para lectura el fichero asociado con f, la posición de lectura para
el fichero f se situa para que el primer dato que se lea sea el primero del fichero:}
   iniciarLectura(f);

   {Para saber si ya no quedan más datos que leer en el fichero (se ha leído ya el
último dato), está disponible la función: finFichero(f) (devuelve booleano
indicándolo)}
mientrasQue not finFichero(f) hacer
      {Para leer de fichero de texto se considerarán también disponibles las mismas
   instrucciones que para leer de teclado o entrada estándar, con el mismo significado
   y la misma sintaxis salvo porque se les añade un primer parámetro que es la
   variable fichero. Ejemplo:}
      leer(f,d); {lee el siguiente dato del fichero f y lo deja en d (lee carácter),
   y deja disponible para ser leído el siguiente dato en el fichero}
         . . .
fmq;
{Eliminar la asociación entre f y el fichero externo:}
   disociar(f);
...
Fin
```

- **Ficheros binarios**

Similares a los ficheros de texto, pero en ellos la unidad mínima de información para la lectura o escritura es un dato del tipo especificado en la creación de la variable fichero (y por supuesto no están estructurados en líneas).

```
variables
   ff: fichero de fecha  {tipo fichero binario de fechas. En el se leerán o escribirán
datos de tipo fecha}
   dia: fecha

...
principio
{asociar la variable ff con el fichero externo de nombre "misDatos.dat":}
asociar(ff, "misDatos.dat");
```

```
   {Inicializar para escritura: el fichero externo es creado vacío y la posición de
escritura en el fichero se mantendrá siempre al final del mismo}
iniciarEscritura(ff);
escribir(ff,dia); {escribe en ff el dato dia, siempre al final del fichero}
. . .
   iniciarLectura(ff); {Inicializar para lectura el contenido del fichero asociado
con ff, la posición de lectura para el fichero ff queda justo por delante del primer
dato del fichero}

   {Para saber si ya no quedan más datos que leer en el fichero (se ha leído ya el
último dato), está disponible la función: finFichero(ff) (devuelve booleano
indicándolo)}
   mientrasQue not finFichero(ff) hacer
      {Para leer un dato de un fichero binario:}
      leer(ff,dia); {lee la siguiente fecha en el fichero ff y la deja en dia, y deja
                     disponible para ser leído el siguiente dato del fichero}
   fmq;
   {eliminar la asociación entre ff y el fichero externo:}
   disociar(ff);
   ...

fin
```

Bibliografía

Bibliografía Básica:

Weiss, M.A.: *Data Structures and Algorithm Analysis in C++*, 4th Edition, Pearson/Addison Wesley, 2014.

Hernández, Z.J. y otros: *Fundamentos de Estructuras de Datos. Soluciones en Ada, Java y C++*, Thomson, 2005.

Shaffer, Clifford A.: *Data Structures and Algorithm Analysis in C++*, Third Edition, Dover Publications, 2013.

Ejercicios:

Martí Oliet, N., Ortega Mallén, Y., Verdejo López, J.A.: *Estructuras de datos y métodos algorítmicos: 213 ejercicios resueltos*. 2ª Edición, Ed. Garceta, 2013.

Joyanes, L., Zahonero, I., Fernández, M. y Sánchez, L.: *Estructura de datos. Libro de problemas*, McGraw Hill, 1999.

Bibliografía sobre C++:

Stroustrup, B.: *The C++ Programming Language*, 4th Edition, Addison-Wesley, 2013.

Bibliografía Complementaria:

Franch Gutiérrez, X.: *Estructuras de Datos. Especificación, Diseño e Implementación*, 3ª edición, Ed. Edicions UPC, 2001.

Mehta, D.P. y Sahni, S.: *Handbook of Data Structures and Applications*, Chapman & Hall/CRC, 2005.

Aho, A. V., Hopcroft, J. E. y Ullman, J. D.: *Estructuras de Datos y Algoritmos*, Addison-Wesley, 1988.

Knuth, D. E.: *El arte de programar ordenadores, volumen III: clasificación y búsqueda*, Ed. Reverté, 1987.

Wirth, N.: *Algoritmos + Estructuras de Datos = Programas*, Ediciones del Castillo, S.A., 1986.

Peña, R.: *Diseño de Programas. Formalismo y Abstracción*, Pearson Educación, 2005.