Programmation et Algorithmique

École Polytechnique

Jean Berstel et Jean-Éric Pin

Avant-propos

Ce polycopié est utilisé pour le cours INF 421 intitulé *Les bases de la programmation et de l'algorithmique*. Ce cours fait suite au cours INF 311 *Introduction à l'informatique* et précède le cours INF 431 intitulé *Fondements de l'informatique*.

Ce polycopié reprend et développe les chapitres 10 et 11 du polycopié de Robert Cori, Jean-Jacques Lévy et François Morain *Les bases de la programmation et de l'algorithmique*. Il fait également suite au polycopié de François Morain *Les bases de l'informatique et de la programmation*. Pour qu'il n'y ait pas confusion avec ce dernier polycopié, nous avons intitulé le nôtre plus simplement *Programmation et algorithmique*.

Comme le lecteur pourra le constater, nous avons suivi pour l'essentiel le canevas des chapitres 10 et 11 du polycopié de Robert Cori, Jean-Jacques Lévy et François Morain, en incluant des applications et des développements sur certaines variantes. Nous avons aussi insisté, au début, sur le concept de référence, notion qui se retrouve d'ailleurs, sous le terme de pointeur, dans d'autres langages de programmation.

Le thème principal du cours est, du côté de la programmation, la conception et la mise en œuvre de nouveaux types. Le langage Java le permet de deux façons, par la création de tableaux, et par la définition de classes. Quant à l'algorithmique, c'est la description et l'emploi de structures de données dynamiques qui sont au centre de ce cours. Nous traitons en détail les listes et les arbres. Leur usage dans la représentation de séquences et d'ensembles ordonnés est particulièrement développé. Les applications concernent la gestion des partitions (« union-find » en anglais), la compression de textes, les tétrarbres (« quadtrees ») et des problèmes géométriques qui ne sont pas tous décrits dans cette première version du polycopié. Les structures plus élaborées, comme les graphes, seront étudiées dans le cours 431.

Nous tenons à remercier en tout premier lieu Robert Cori, Jean-Jacques Lévy et François Morain qui nous ont autorisé à reprendre à notre compte certains éléments de leurs polycopiés. Nous remercions aussi nos collègues de l'École Polytechnique, et plus particulièrement Philippe Chassignet, Emmanuel Coquery, Fabrice Le Fessant, Laurent Mauborgne, Éric Schost, Alexandre Sedoglavic et Nicolas Sendrier qui ont largement contribué à l'élaboration et à la mise au point de ce cours. Enfin nous sommes particulièrement reconnaissants à Catherine Bensoussan et au service de reprographie de l'École Polytechnique pour leur travail exemplaire.

Les auteurs peuvent être contactés par courrier électronique aux adresses suivantes :

`berstel@univ-mlv.fr`

`Jean-Eric.Pin@liafa.jussieu.fr`

On peut aussi consulter les pages Web des auteurs :

`http://www-igm.univ-mlv.fr/~berstel`

`http://liafa.jussieu.fr/~jep`

Table des matières

Chapitre I

Compléments de programmation

Ce premier chapitre nous permettra de préciser certains points cruciaux de la programmation en Java. Une première section est consacrée à l'étude détaillée des références, ou adresses. La seconde section fait le point sur les méthodes et les variables statiques. Nous terminons par quelques rappels sur les méthodes et les variables déclarés final.

1 Références

L'utilisation des adresses se retrouve dans divers langages de programmation (Pascal, C, C++, Java, etc.) et peut paraître un peu déroutante au début. Elle devient limpide dès que l'on en a assimilé les principes de base et c'est la raison pour laquelle nous y consacrons cette section.

1.1 Types et références

En Java, les données d'un programme se répartissent en deux catégories selon leur type : les types *primitifs* et les autres. Les types primitifs comprennent les types d'entiers (byte, short, int, long), les types des réels (float, double), le type des caractères (char) et le type des booléens (boolean). Les autres types sont les types de tableaux ou d'objets. Par exemple, String est un type d'objets, et int[] est le type d'un tableau d'entiers. La construction de tableaux et la définition de nouvelles classes sont les deux moyens de créer de nouveaux types à partir de types donnés. Ainsi, une définition comme

```
class Personne
{
  String nom;
  int age;
}
```

définit un nouveau type d'objets. Ces deux mécanismes s'emboîtent. Ainsi,

```
class Annuaire
{
  Personne[] liste;
}
```

définit un type d'objets ayant pour membres un tableau d'objets de type Personne.

La manipulation des données dans un programme Java dépend de leur type. La *référence* d'une donnée est l'adresse mémoire où est logée la donnée. La nature précise de l'adresse (entier, manière de numéroter, etc) nous intéresse peu. Une référence est de plus typée par le type de la donnée.

Règle. *Les données de type primitif sont manipulées par valeur, les données des autres types sont manipulées par référence.*

Ainsi, le contenu d'une variable de type `int` est un entier, le contenu d'une variable de type `char` est un caractère, et plus généralement le contenu d'une variable de type primitif est une valeur de ce type. En revanche, le contenu d'une variable de type non primitif est *l'adresse* d'une donnée de ce type. La *manipulation* de références est bien plus limitée que la manipulation de types de base : on peut seulement les comparer (par == et !=) et les affecter. L'affectation est une affectation de références, et il n'y a donc pas de copie des données référencées ! On peut illustrer ce phénomène par un exemple concret. Si un ami vous annonce son déménagement et vous donne ses nouvelles coordonnées, une simple modification dans votre carnet d'adresses vous permettra de continuer à lui écrire ou à lui téléphoner. Bien entendu, cette modification n'a pas eu pour effet le déménagement ou le changement de la ligne téléphonique de votre ami. C'est la même chose en Java. Lorsqu'on modifie une référence, cela n'entraîne pas le déplacement physique des données, qu'il faudra donc réaliser indépendamment si on le souhaite.

Voici un premier exemple.

```
public static void main(String[] args)
{
  byte[] t = {5, 2, 6};
  System.out.println("Tableau " + t) ;
}
```

Le résultat est

```
> Tableau [B@f12c4e
```

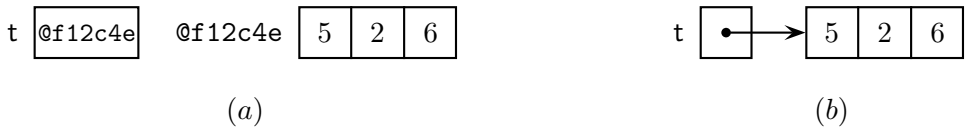
La deuxième chaîne s'interprète comme suit : les deux premiers caractères représentent le type d'un tableau d'octets (le crochet ouvrant [signifie « tableau » et B signifie « octet')', le symbole '@' signifie « adresse », la fin est l'adresse en hexadécimal.

Remarque. (Cette remarque peut être ignorée en première lecture) Lors d'une impression, la méthode println cherche une représentation des données sous la forme d'une chaîne de caractères. Celle-ci est fournie par la méthode toString qui est définie pour tout objet. Par défaut, cette méthode retourne une chaîne de caractères formée du nom de la classe, de '@' et de l'écriture hexadécimale du code de hachage de l'objet. Par défaut, le code de hachage d'un objet est obtenu en hachant l'adresse de l'objet. Dans notre exemple, f12c4e est l'écriture hexadécimale du code de hachage de l'adresse du tableau t. Fréquemment, une classe redéfinit la méthode toString. Par exemple, la classe String définit cette méthode pour qu'elle retourne la chaîne de caractères contenue dans la donnée. Ainsi, la chaîne " Tableau " est effectivement affichée telle quelle.

Une figure bien choisie aide beaucoup à la compréhension du comportement des références. Pour ce faire, la valeur associée à une variable n est dessinée dans une boîte étiquetée par n. Dans la figure 1.1, on représente une variable entière n dont la valeur est 2. Quand la valeur d'une variable est une référence, c'est-à-dire l'adresse d'une donnée, la valeur numérique de cette adresse est remplacée par une flèche dirigée vers l'emplacement de la donnée.

FIG. 1.1 – Une variable n contenant la valeur 2.

Dans la figure 1.2, on représente une variable de tableau d'octets t. La valeur de cette variable est le début de la zone mémoire réservée au tableau d'octets.



(a) (b)

FIG. 1.2 – Une variable t contenant l'adresse d'un tableau de trois octets : (a) situation effective, (b) description symbolique.

Quand une variable vaut null, cette valeur est indiquée par une petite croix, comme dans la figure 1.3.

FIG. 1.3 – Une variable t contenant la valeur null.

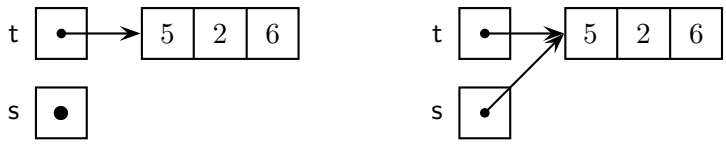
Comme nous l'avons dit, les opérations sur les adresses sont restreintes. Voici un exemple d'affectation.

```
public static void main(String[] args)
{
  byte[] t = {5, 2, 6};
  byte[] s;
  s = t;
  System.out.println("Tableau " + t + " " + s) ;
}
```

Le résultat est

```
> Tableau [B@f12c4e [B@f12c4e
```

Avant l'affectation s = t, la valeur de s est indéfinie. Après l'affectation, les variables désignent le même emplacement dans la mémoire (voir figure 1.4).



(a) (b)

FIG. 1.4 – (a) avant affectation ; (b) après affectation.

En effet, l'instruction d'affectation s = t copie le contenu de t dans s comme toute affectation, et le contenu est l'adresse du tableau.

Continuons notre exploration. En toute logique, puisque après l'affectation s = t, les variables s et t désignent le même emplacement mémoire, toute modification peut être faite indistinctement via s ou t. En particulier, une modification via s se repère aussi via t. Le programme

```
public static void main(String[] args)
```

```
{
  byte[] t = {5, 2, 6};
  byte[] s = t;
  s[1] = 8;
  System.out.println(t[1]) ;
}
```

affiche en effet le nombre 8.

A contrario, si l'on recopie les valeurs contenues dans le tableau t dans un autre tableau, les données sont séparées.

```
public static void main(String[] args)
{
  byte[] t = {5, 2, 6};
  byte[] s = new byte[3];
  for (int i = 0; i < t.length; i++)
    s[i] = t[i];
  System.out.println(s + " " + t) ;
  System.out.println(s == t) ;
}
```

donne

```
> [B@f12c4e [B@93dee9
> false
```

En effet, avant la boucle for, le tableau s ne contient que des valeurs nulles (figure 1.5).

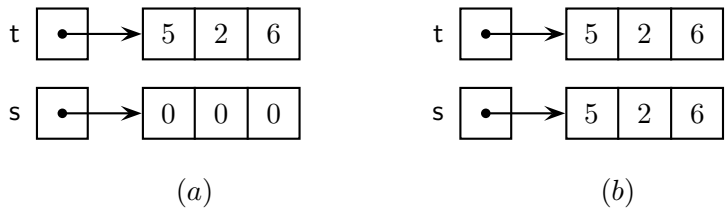


FIG. 1.5 – (*a*) avant copie ; (*b*) après copie.

1.2 Références d'objets

Ce qui a été illustré sur des tableaux s'applique aussi aux objets, définis comme instances de classes. Reprenons la classe des « personnes » déjà introduite plus haut.

```
class Personne
{
  String nom;
  int age;
}
```

On s'en sert comme dans la méthode suivante.

```
public static void main(String[] args)
{
  Personne p = new Personne();
  p.nom = "Dominique";
  p.age = 22;
  System.out.println(p);
```

```
    System.out.println(p.nom + ", " + p.age + " ans");
  }
```

avec le résultat

```
> Personne@cac268
> Dominique, 22 ans
```

La première ligne affichée est le contenu de p qui est une référence ; cette référence est retournée par l'instruction new Personne(). La deuxième ligne donne le contenu des champs de l'objet dont p contient la référence. L'objet créé par le programme est représenté dans la figure 1.6, avant et après remplissage.

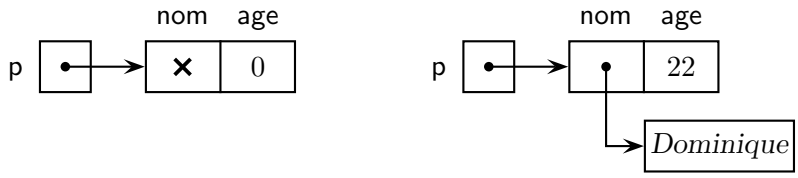


FIG. 1.6 – Personne créée par new Personne(), et après remplissage des champs.

Comme déjà dit plus haut, null est une valeur de référence particulière. Ce n'est la référence d'aucun objet. On peut affecter null à toute variable référence, mais aucun champ et aucune méthode d'objet n'est accessible par une variable qui contient cette valeur spéciale. Une telle tentative provoque la levée d'une exception de la classe NullPointerException.

L'opérateur new est utilisé pour la création d'un nouvel objet.

Règle. *À la création, les champs d'un objet sont initialisés à une valeur par défaut. Cette valeur est 0 pour les types primitifs numériques, false pour le type booléen, et null pour tout champ objet.*

Et en effet, le programme

```
  public static void main(String[] args)
  {
    Personne p = new Personne();
    System.out.println(p + " nom: " + p.nom + ", age: " + p.age);
  }
```

affiche

```
> Personne@93dee9 nom: null, age: 0
```

Compliquons un peu. Nous avons dit que les deux opérations de créations de nouveaux types sont les classes et les tableaux. Créons donc un tableau de trois personnes. Là également, le tableau contient, au départ, des valeurs par défaut, donc null puisqu'il s'agit d'un tableau d'objets.

```
  public static void main(String[] args)
  {
    Personne[] t = new Personne[3];
    Personne p = new Personne();
    p.nom = "Dominique";
    p.age = 22;
    t[1] = p;
    for (int i = 0; i < t.length; i++)
      System.out.print(t[i] + " ");
  }
```

Le résultat est

```
> null Personne@93dee9 null
```

La situation est décrite dans la figure 1.7.

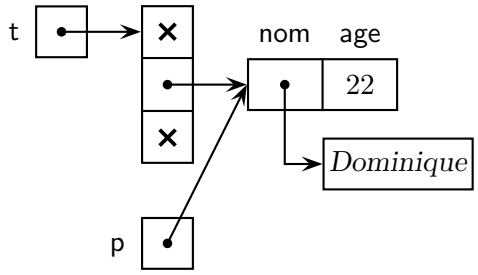


FIG. 1.7 – Tableau de trois personnes. La deuxième entrée est affectée par copie de la valeur de p.

1.3 Constructeurs

Une classe Java est un bloc d'un programme. Une classe définit un nouveau type non primitif, et peut contenir

- des attributs (aussi appelés variables),
- des méthodes,
- des initialisations,
- d'autres classes (classes internes ou imbriquées).

Parmi les méthodes, nous connaissons les méthodes de classe, ou méthodes statiques (justement celles qualifiées static) et les méthodes d'instance. Les *constructeurs* sont d'autres méthodes qui permettent de créer des objets tout en affectant aux champs des valeurs. Leur syntaxe est un peu particulière. Un constructeur

- a le même nom que la classe,
- est sans type de retour (pas même void !).

Un constructeur est appelé par new. Un constructeur particulier, appelé *constructeur par défaut*, est fourni à toute classe : c'est le constructeur sans arguments, celui que nous avons utilisé jusqu'alors. Il initialise tous les champs de l'objet créé à leur valeur par défaut.

Plusieurs constructeurs peuvent coexister. Ceci est dû à la possibilité de *surcharger* un nom de méthode : deux méthodes sont surchargées (ou l'une est une surcharge de l'autre), si elles ont le même nom, mais diffèrent par le nombre ou par le type de leurs arguments.

Reprenons la classe Personne. Le constructeur par défaut initialise nom à null et age à 0. Voici une nouvelle implantation, avec deux autres constructeurs :

```
class Personne
{
  String nom;
  int age;

  Personne() {}

  Personne(String n, int a)
  {
    nom = n;
    age = a;
  }
```

```
  Personne(String n)
  {
    nom = n;
  }
}
```

Le programme

```
public static void main(String[] args)
{
  Personne p = new Personne("Luc", 23);
  Personne q = new Personne("Anne");
  System.out.println(p.nom + ", " + p.age + " ans.");
  System.out.println(q.nom + ", " + q.age + " ans.");
}
```

affiche

```
> Luc, 23 ans.
> Anne, 0 ans.
```

Notons que le constructeur par défaut cesse d'être fourni implicitement dès qu'un autre constructeur est défini. Si l'on veut quand même disposer de ce constructeur, il faut le redéfinir — comme nous l'avons fait.

Voici un autre exemple. Nous manipulons un ensemble de personnes au moyen d'un annuaire, où on peut ajouter des personnes, en enlever, et demander si une personne y figure. La classe Annuaire est formée d'un tableau de Personne et d'un entier n qui donne la taille de l'annuaire.

```
class Annuaire
{
  final static int nMax = 12;
  Personne[] liste = new Personne[nMax];
  int n = 0;
  ...
}
```

La constante nMax sert à donner une taille maximale. Les méthodes à définir sont

```
class Annuaire
{
  ...
  boolean ajouter(String nom, int age) {...}
  boolean enlever(String nom) {...}
  int index(String nom) {...}
  boolean estDans(String nom) {...}
  void afficher() {...}
}
```

On peut s'en servir dans un programme comme

```
public static void main(String[] args)
{
  Annuaire a = new Annuaire();
  a.ajouter("Luc", 22);
  a.ajouter("Anne", 21);
  a.afficher();
  System.out.println();
```

```
    boolean x = a.estDans("Luc"); // retourne true
    a.enlever("Luc");
    a.afficher();
  }
```

et le résultat est

```
> Luc, 22 ans.
> Anne, 21 ans.
>
> Anne, 21 ans.
```

La méthode index retourne l'entier i tel que liste[i] a pour champ nom l'argument passé en paramètre, si un tel entier existe, et -1 sinon. Les méthodes ajouter et enlever retournent un booléen qui indique si l'opération a réussi.

```
class Annuaire
{
  final static int nMax = 12;
  Personne[] liste = new Personne[nMax];
  int n = 0;

  boolean ajouter(String nom, int age)
  {
    if (estDans(nom) || n == nMax)
      return false;
    liste[n++] = new Personne(nom, age);
    return true;
  }

  boolean enlever(String nom)
  {
    int i = index(nom);
    if (i == -1)
      return false;
    for (int j = i + 1; j < n; j++)
      liste[j - 1] = liste[j];
    n--;
    return true;
  }

  int index(String nom)
  {
    for (int i = 0; i < n; i++)
      if (liste[i].nom.equals(nom))
        return i;
    return -1;
  }

  boolean estDans(String nom)
  {
    return (index(nom) != -1);
  }

  void afficher()
  {
```

```
      for (int i = 0; i < n; i++)
      {
        Personne p = liste[i];
        System.out.println(p.nom + ", " + p.age + " ans.");
      }
    }
  }
```

Ce programme, comme beaucoup d'autres semblables, doit traiter le problème de l'adjonction d'un élément à un tableau qui est déjà plein. La politique de l'autruche consiste à ignorer ce cas. Plus sérieusement, on peut considérer deux solutions : *traiter* le problème, ou le *signaler*. Pour signaler le problème, on utilise la méthode ci-dessus : on ne fait pas l'adjonction, et on retourne la valeur false. Une version plus sophistiquée, basée sur les *exceptions*, sera décrite plus loin. Pour *traiter* le problème du tableau plein, il n'y a guère qu'une solution : augmenter sa taille. Ceci se fait en dupliquant le tableau dans un tableau plus grand, comme décrit ci-dessous (la méthode est à inclure dans la classe Annuaire) :

```
void doublerListe()
{
  Personne[] nouvelleListe = new Personne[2*liste.length];
  for (int i = 0; i < liste.length; i++)
    nouvelleListe[i] = liste[i];
  liste = nouvelleListe;
}
```

Bien voir que l'on recopie le *contenu* de l'ancien tableau dans le nouveau, puis que l'on copie l'*adresse* du nouveau tableau dans la variable contenant l'ancienne adresse. Avec cette méthode, la méthode ajouter peut se récrire de la façon suivante :

```
boolean ajouter(String nom, int age)
{
  if (estDans(nom))
    return false;
  if (n == liste.length)
    doublerListe();
  liste[n++] = new Personne(nom, age);
  return  true;
}
```

En fait, il est si souvent nécessaire de recopier des tableaux qu'il existe une méthode toute prête proposée par Java (voir la méthode System.arraycopy()). De plus, Java dispose de classes qui gèrent des tableaux dont la taille augmente automatiquement en cas de besoin.

2 Méthodes et variables statiques

Comme pour les références, la distinction entre méthodes (ou variables) statiques et non statiques ne présente plus de difficulté si on a assimilé les principes de bases.

2.1 Variables statiques

Une variable déclarée static est commune à tous les objets de la classe. Pour cette raison, on parle de *variable de classe* et on peut l'utiliser avec la syntaxe NomdeClasse.nomDeVariable. Par exemple, dans Math.Pi, Pi est une variable final static de la classe Math. En revanche, une variable non statique aura une instance par objet. Une variable statique peut aussi être manipulée

à travers un objet de la classe, comme toute variable. Ce faisant, on accède toujours à la même variable.

Comme une variable statique est attachée à une classe, et non à un objet, on a la règle essentielle suivante.

Règle. *Toute modification d'une variable statique peut être faite indistinctement par n'importe quel objet de la classe. Cette modification est visible par tous les objets de la classe.*

Pour illustrer ce phénomène, considérons une classe contenant une variable statique et une variable non statique.

```
class Paire
{
  static int x;
  int y;
}
```

On peut alors utiliser la variable statique x, sous la forme Paire.x, sans déclarer d'objet, mais écrire Paire.y provoquerait une erreur. Par contre, si on déclare un objet s de type Paire, on a accès aux champs s.x et s.y.

```
class Test
{
  static void main()
  {
    Paire s = new Paire();
    System.out.println("s.x = " + s.x + ", s.y = " + s.y +
                       ", Paire.x = " + Paire.x);
  }
}
```

Le résultat est

```
> s.x = 0, s.y = 0, Paire.x = 0
```

Modifions légèrement la classe Test.

```
class Test
{
  static void main()
  {
    Paire s = new Paire();
    Paire t = new Paire();
    t.x = 2;
    t.y = 3;
    System.out.println("t.x = " + t.x + ", t.y = " + t.y);
    System.out.println("s.x = " + s.x + ", s.y = " + s.y);
  }
}
```

Le résultat est

```
> t.x = 2, t.y = 3
> s.x = 2, s.y = 0
```

Comme on le voit, l'instruction $t.x = 2$ a modifié la valeur de la variable statique x. De ce fait, la valeur retournée par s.x est également modifiée. En revanche, l'instruction $t.y = 3$ n'a eu aucune influence sur la valeur de s.y. Pour clarifier de tels comportements, il est d'ailleurs préférable de remplacer l'instruction $t.x = 2$ par $Paire.x = 2$. Même s'il y a plusieurs objets de la classe Paire, il

n'y a qu'une seule instance de la variable statique x. Les expressions Paire.x, s.x et t.x désignent la même donnée à la même adresse. Une dernière précision : à l'intérieur de la classe Paire, la variable statique Paire.x peut s'écrire plus simplement x, car la classe courante est implicite.

Pour bien illustrer ce qui précède, voici une légère variation du programme précédent.

```
class Paire
{
  static int[] x;
  int y;
}

class StaticTest
{
  public static void main(String[] args)
  {
    Paire.x = new int[1];
    Paire.x[0] = 7;

    System.out.println(
      "Paire.x = " + Paire.x + ", Paire.x[0] = " + Paire.x[0]);
    Paire s = new Paire();
    s.y = 3;
    System.out.println("s.x = " + s.x + ", s.y = " + s.y);
    Paire t = new Paire();
    t.y = 4;
    System.out.println("t.x = " + t.x + ", t.y = " + t.y);
  }
}
```

Ce programme affichera

```
> Paire.x = [I@93dee9, Paire.x[0] = 7
> s.x = [I@93dee9, s.y = 3
> t.x = [I@93dee9, t.y = 4
```

2.2 Méthodes statiques

Une méthode déclarée static peut être utilisée sans référence à un objet particulier. On parle alors de *méthode de classe*. Une méthode statique n'a pas directement accès aux variables non statiques. On peut appeler une méthode statique par NomdeClasse.nomDeMethode(). Par exemple, dans Math.abs(), abs() est une méthode statique de la classe Math.

Une méthode qui n'est pas déclarée static est toujours utilisée en référence à un objet. On parle alors de *méthode d'objet*. On peut appeler une méthode non statique par nomObjet.nomDeMethode(). Par exemple, la classe System contient une variable de classe de nom out. L'objet System.out appelle une méthode d'objet de nom println().

Pour simplifier l'écriture des méthodes d'objet, on utilise souvent le mot clé this, qui désigne l'objet courant. Il n'a jamais la valeur null, et sa valeur ne peut être modifiée. Utiliser this dans une méthode statique produit une erreur à la compilation.

Pour illustrer ces définitions, introduisons simultanément dans la classe Paire des méthodes de classe et des méthodes d'objet.

```
class Paire
{
  static int x;
```

```
  int y;

  void affiche_x() // méthode d'objet
  {
    System.out.println("s.x = " + this.x);
  }

  void affiche_y() // méthode d'objet
  {
    System.out.println("s.y = " + this.y);
  }

  static void affiche2_x() // méthode de classe
  {
    System.out.println("s.x = " + x);
  }

  static void affiche3_y(Paire s)
  {
    System.out.println("s.y = " + s.y);
  }
}
```

On remarquera la syntaxe utilisant this dans affiche_x et dans affiche_y et l'utilisation de la variable statique x dans affiche2_x. On notera également la différence entre la méthode sans paramètre affiche_y et la méthode affiche3_y qui prend en paramètre un objet s de type Paire. Attention en effet au piège suivant :

```
// Attention, cette méthode est incorrecte ...
  static void affiche2_y()
  {
    System.out.println("s.y = " + y);
  }
// ... car y n'est pas "static" !
```

Maintenant, le programme

```
class Test
{
  static void main()
  {
    Paire s = new Paire();
    Paire t = new Paire();
    t.x = 2;
    t.y = 3;
    s.affiche_x();
    s.affiche_y();
    Paire.affiche2_x();
    s.affiche2_x();
    Paire.affiche3_y(s);
  } // s.affiche3_y(s) serait maladroit
}
```

produit le résultat suivant :

```
> s.x = 2
> s.y = 0
> s.x = 2
> s.x = 2
> s.y = 0
```

3 Méthodes et variables final

La déclaration d'une variable final doit toujours être accompagnée d'une initialisation. Une méthode final ne peut être surchargée.

Une variable final ne peut être modifiée. Attention toutefois ! Si la variable est un tableau, donc une référence, cette référence ne change plus, mais les valeurs du tableau peuvent être modifiées !

```
class Test
{
  static final int n = 100;
  static final int[] a = {1, 2, 3};

  public static void main(String args[])
  {
    a[0] = 5; // OK
    n = 9; // Erreur ! Variable finale
  }
}
```

Les méthodes d'une classe final sont implicitement final. La classe String est une classe final.

4 La classe String

La manipulation des chaînes de caractères est simplifiée en Java par la classe String. Cette classe, dont nous venons de dire qu'elle est finale, permet les opérations usuelles sur les chaînes de caractères. Toutefois, les objets de la classe String sont *immutables*, c'est-à-dire qu'ils ne peuvent être modifiés. Une opération sur une chaîne de caractère se traduit donc, lors d'une modification, par la création d'un nouvel objet. Dans la suite de ce polycopié, on parlera souvent d'opérations destructrices et non destructrices ; dans ce contexte, les opérations de la classe String sont non destructrices. Il existe une variante, destructrice, des chaînes de caractères, la classe StringBuffer.

Les méthodes de la classe String sont nombreuses. Voici quelques-unes des plus fréquemment utilisées :

```
int length()
char charAt(int index)
String toLowerCase()
String toUpperCase()
```

Ainsi

```
"abc".charAt(1)
```

retourne le caractère b, et

```
"X2001".toLowerCase()
```

retourne la chaîne x2001. D'autres méthodes sont utiles pour la comparaison, l'examen, la modification. En voici quelques exemples :

```
boolean equals(String s)
int compareTo(String s)
boolean startsWith(String prefix)
boolean endsWith(String suffix)
int indexOf(String facteur)
int lastIndexOf(String facteur)
```

L'égalité de deux chaînes de caractères se teste par equals. Le test par == teste l'égalité des adresses. La classe String est très optimisée en Java, et souvent les chaînes composées des mêmes caractères sont en fait rangées qu'une seule fois en mémoire (ce qui est possible puisqu'elles sont inmodifiables !). De ce fait, l'égalité du contenu implique souvent l'égalité des adresses et peut donc être testé de cette manière.

La méthode compareTo compare la chaîne avec l'argument, pour l'ordre alphabétique. Elle retourne un entier négatif, 0, ou un entier positif selon que la chaîne appelante est plus petite, égale ou plus grande que la chaîne passée en argument. Ainsi

```
"X2001".compareTo("W1001")
```

retourne un nombre positif. Bien entendu, compareTo retourne 0 si et seulement si equals retourne true. La méthode indexOf (resp. lastIndexOf) retourne le plus petit (resp. plus grand) indice où la chaîne en argument commence comme facteur de la chaîne appelante, et -1 si l'argument n'est pas facteur. Ainsi

```
"X2000001".indexOf("00")
"X2000001".lastIndexOf("00")
```

donne 2 et 5. Enfin, voici quelques opérations de construction :

```
String valueOf(int i)
String substring(int debut, int fin)
String concat(String s)
String replace(char x, char y)
```

Il existe en fait neuf versions de la méthode valueOf, qui différent par le type de l'argument (celle citée ci-dessus a pour argument un int). Elles retournent toutes une chaîne de caractères représentant la valeur passée en paramètre. La méthode concat retourne une nouvelle chaîne de caractères formée de la chaîne appelante et de la chaîne en paramètre. Enfin, notons la possibilité de créer un objet de la classe String implicitement, par l'énumération de son écriture. Ainsi, les chaînes comme "X2000001" sont des objets de la classe String. La méthode remplace remplace toutes les occurrences du caractère x par le caractère y. Le mot "remplace" n'est pas très bon, puis que c'est une nouvelle chaîne de caractères qui est crée.

Parfois, il est utile de disposer de la suite des caractères d'un String sous forme d'un tableau. La méthode toCharArray() retourne un tel tableau ; réciproquement, le constructeur String(char[] tableau) construit une chaîne à partir d'un tableau de caractères.

```
char[] tableau = "X2000001".toCharArray();
tableau[1]++;
String s = new String(tableau);
```

La chaîne retournée est bien sûr "X3000001".

Retour sur la méthode toString(). La méthode toString() fournit une chaîne de caractères contenant une description textuelle de l'objet qui l'appelle. Comme nous l'avons dit plus haut, cette méthode retourne par défaut une chaîne de caractères formée du nom de la classe, de '@' et de l'écriture hexadécimale du code de hachage de l'objet. Pour les objets définis par des tableaux, le nom est formé du préfixe [pour les tableaux, et du codage suivant du nom des types :

`B`	byte	`J`	long
`C`	char	`L`*nom de classe*`;`	objet
`D`	double	`S`	short
`F`	float	`Z`	boolean
`I`	int		

Considérons l'exemple suivant :

```
class Exemple
{
  public static void main(String[] args)
  {
    Exemple e = new Exemple();
    Exemple[] t = new Exemple[12];
    Exemple[][] u = new Exemple[12][24];
    System.out.println(e) ;
    System.out.println(t) ;
    System.out.println(u) ;
  }
}
```

Le résultat est :

```
> Exemple@7182c1
> [LExemple;@3f5d07
> [[LExemple;@fabe9
```

et en effet, la variable `u` est un tableau dont les éléments sont des tableaux dont les éléments sont de type Exemple.

Chapitre II

Structures séquentielles

Les structures de données en algorithmique sont souvent complexes et de taille variable. Elles sont souvent aussi *dynamiques*, au sens où elles évoluent, en forme et en taille, en cours d'exécution d'un programme. Les tableaux présentent, dans cette optique, un certain nombre de limitations qui en rendent parfois l'usage peu naturel.

Tout d'abord, un tableau est de taille fixe, déterminé à sa création. Il faut donc choisir une taille suffisante dès le début, ce qui peut amener à un gaspillage considérable de place. Et même si l'on a vu large, il arrive que la place manque. Il faut alors procéder à la création d'un tableau plus grand et à une recopie, ce qui représente une perte de temps qui peut être importante.

Un tableau a une structure fixe, linéaire. Une modification de l'ordre, par exemple par insertion d'un élément, oblige à déplacer d'autres éléments du tableau, ce qui peut être très coûteux. Un tableau est donc rigide, et peu économique si l'ensemble des éléments qu'il contient est appelé à évoluer en cours d'exécution.

Dans ce chapitre et dans le chapitre suivant, nous introduisons quelques structures *dynamiques*. Elles sont utilisées de façon très intensive en programmation. Leur but est de gérer un ensemble fini d'éléments dont le nombre n'est pas fixé *a priori* et peut évoluer.

Les structures dynamiques doivent répondre à un certain nombre de critères. Elles doivent permettre une bonne gestion de la mémoire, par un usage économe de celle-ci. Mais elles doivent aussi être simples à utiliser, pour ne pas échanger cette économie en mémoire contre une difficulté d'emploi, ou un temps d'exécution disproportionné.

La solution présentée ici, et généralement utilisée, repose sur trois principes :

- Allocation de la mémoire au fur et à mesure des besoins. On dispose alors de toute la place nécessaire sans en gaspiller.
- Définition récursive des structures. Elle permettent en contrepoint une programmation simple par des méthodes elle-mêmes récursives, calquées sur leur structure.
- Enfin, un petit nombre d'opérateurs d'accès et de modification, à partir desquels on construit des méthodes plus élaborées par composition.

Allouer de la mémoire signifie réserver de la mémoire pour l'utilisation du programme. Après utilisation, on peut ensuite désallouer cette mémoire. Dans des langages tels que PASCAL ou C, il faut gérer soigneusement l'allocation et la désallocation. En JAVA, c'est un peu plus simple : il faut toujours allouer de la mémoire, mais JAVA se charge de désallouer la mémoire qui ne sert plus.

On notera que la description ci-dessus repose sur des principes qui ne dépendent pas du langage de programmation choisi. En Java, et dans d'autres langages de haut niveau, on peut même spécifier une structure dynamique par une description purement abstraite de ses propriétés. Ce style de programmation utilisant des types abstraits relève des cours ultérieurs (cours 431 et cours de majeure).

Nous étudierons dans ce cours deux types de structures dynamiques. Ce chapitre est consacré

aux structures dites *séquentielles* : listes, piles et files d'attente. Le chapitre suivant traite des arbres et de leur utilisation. D'autres structures dynamiques, en particulier les graphes, sont traitées dans le cours « Fondements de l'informatique ».

Les structures considérées servent à manipuler des ensembles. Les éléments de ces ensembles peuvent être de différentes sortes : nombres entiers ou réels, chaînes de caractères, ou des objets plus complexes comme des processus, des expressions, des matrices. Le nombre d'éléments des ensembles varie au cours de l'exécution du programme, par ajout et suppression d'éléments en cours de traitement. Plus précisément, les opérations que l'on s'autorise sur un ensemble E sont les suivantes :

- *tester* si l'ensemble E est vide.
- *ajouter* l'élément x à l'ensemble E.
- *vérifier* si l'élément x appartient à l'ensemble E.
- *supprimer* l'élément x de l'ensemble E.

1 Listes chaînées

Une *liste chaînée* est composée d'une suite finie de couples formés d'un élément et de l'adresse (référence) vers l'élément suivant.

Il s'agit d'une structure de base de la programmation, fort ancienne. Un usage systématique en est déjà fait dans le langage Lisp (*LISt Processing*), conçu par John MacCarthy en 1960. Une version plus récente de ce langage, Scheme, utilise principalement cette structure. Les listes sont aussi prédéfinies en Caml. Dans la plupart des langages à objets, comme C++ et Java, on trouve des classes gérant les listes.

La recherche d'un élément dans une liste s'apparente à un classique « jeu de piste » dont le but est de retrouver un objet caché : on commence par avoir des informations sur un « site » où pourrait se trouver cet objet. En ce lieu, on découvre des informations, et une indication sur un autre site (l'adresse d'un autre site) où l'objet pourrait se trouver, et ainsi de suite. Le point de départ du jeu de piste est l'adresse du premier site qu'il faut visiter. Une version contemporaine de ce jeu consisterait en une suite de pages Web contenant une information et un *lien* vers la page suivante. Il y a une analogie évidente entre lien et référence.

Notons que l'ordre de la suite des éléments d'une liste chaînée n'est pas connu globalement, puisqu'on ne connaît que le successeur d'un élément. C'est ce caractère local qui permet une grande souplesse d'organisation : il suffit de modifier, en un seul site, la référence au site suivant pour réorganiser la suite des pages Web.

Les opérations usuelles sur les listes sont les suivantes :

- *créer* une liste vide.
- *tester* si une liste est vide.
- *ajouter* un élément en tête de liste.
- *rechercher* un élément dans une liste.
- *supprimer* un élément dans une liste.

D'autres opérations sont : retourner une liste, concaténer deux listes, fusionner deux listes, etc.

En Java, les « sites » sont des objets qui possèdent deux champs : un champ pour le contenu, et un champ qui contient la référence vers le site suivant. L'accès à la liste se fait par la référence au premier site. Il est de tradition d'appeler *cellule* ce que nous avons appelé site.

1.1 Définitions

Les déclarations sont les suivantes. On suppose ici que les éléments stockés dans la liste sont des entiers. Bien entendu, le schéma de déclaration serait le même pour une liste de caractères, de booléens, ou d'objets de type quelconque.

```
class Liste
{
  int contenu;
  Liste suivant;

  Liste (int x, Liste a)
  {
    contenu = x;
    suivant = a;
  }
}
```

Une liste est représentée par la référence au premier objet de type Liste. La liste vide, sans élément, a pour référence null. La figure 1.1 représente la forme générale d'une liste. La liste est composée de cellules. Chacune contient, dans son champ contenu, l'un des éléments de la liste. Le champ suivant contient la référence à la cellule suivante. La croix dans la dernière cellule indique que sa valeur est null, convention pour indiquer qu'il n'y a pas d'élément suivant. Une liste est repérée par la référence à sa première cellule.

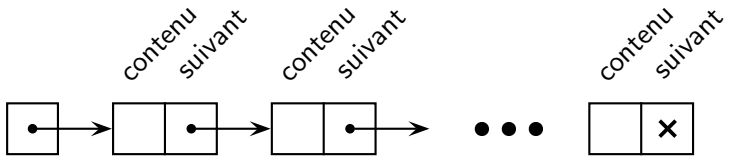


FIG. 1.1 – Forme générale d'une liste.

L'instruction new Liste (x, a) construit une nouvelle cellule dont les champs sont x et a et retourne la référence à cette cellule. Revenons à notre jeu de piste. Supposons que l'adresse du premier site soit a. L'instruction new Liste (x, a) revient à insérer, au début du jeu, un nouveau site, dont le contenu est x. Il contient, comme indication du site suivant, l'adresse de l'ancien point de départ, soit a. Le nouveau point de départ est le site fraîchement inséré.

Créer une liste de trois entiers, par exemple de $2, 7$ et 11 se fait par usage répété de l'instruction d'insertion :

```
Liste a = new Liste(2, new Liste (7, new Liste (11, null)))
```

La liste de cet exemple est représentée dans la figure 1.2.

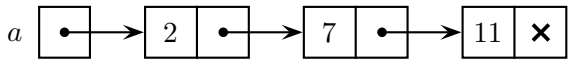


FIG. 1.2 – Une liste de trois éléments $2, 7, 11$ représentées par trois cellules.

Venons-en aux opérations sur les listes. La méthode listeVide retourne une liste vide, c'est-à-dire null.

```
static Liste listeVide ()
{
  return null;
}
```

La fonction qui teste si une liste est vide s'écrit :

```
static boolean estVide (Liste a)
{
  return a == null;
}
```

Il faut bien dire que ces fonctions sont si simples qu'on les remplace souvent par leur définition dans la pratique.

Ajouter un élément en début de liste se fait par :

```
static Liste ajouter (int x, Liste a)
{
  return new Liste (x, a);
}
```

L'effet de cette opération est représenté dans la figure 1.3.

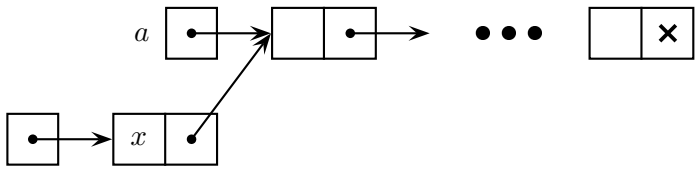


FIG. 1.3 – Ajout d'un élément x au début d'une liste a.

Ainsi, créer une liste contenant seulement `x` se fait de l'une des quatre façons équivalentes :

```
new Liste(2, null)              ajouter(x, null)
new Liste(2, listeVide())       ajouter(x, listeVide())
```

Voici un autre exemple :

```
static Liste suite(int n)
{
  Liste a = null;
  for (int i = n; i >=1; i--)
    a = ajouter(i, a);
  return a;
}
```

Cette fonction retourne une liste contenant les entiers de 1 à n, dans cet ordre. Noter le parcours décroissant des entiers dans la boucle.

Deux fonctions d'accès sont importantes : celle retournant le contenu, et celle retournant la référence à la suite de la liste. Ces fonctions ne doivent être appelées que sur des listes non vides.

```
static int tete(Liste a)
{
  return a.contenu;
}

static Liste queue(Liste a)
{
  return a.suivant;
}
```

Là encore, on utilise souvent dans la pratique l'expression plutôt que la fonction. Les trois fonctions ajouter, tete et queue sont liées par les équations suivantes.

$$\begin{aligned} \mathsf{tete}(\mathsf{ajouter}(x, a)) &= x \\ \mathsf{queue}(\mathsf{ajouter}(x, a)) &= a \\ \mathsf{ajouter}(\mathsf{tete}(a), \mathsf{queue}(a)) &= a; \quad (a \neq \mathsf{null}) \end{aligned}$$

Elles permettent de simplifier toute expression composée de ces trois fonctions en une forme réduite.

Les trois fonctions ajouter, tete et queue permettent de réaliser beaucoup d'opérations sur les listes. Elles ne permettent pas, en revanche, de modifier la valeur d'un champ, que ce soit le contenu ou la référence à la cellule suivante. C'est pourquoi on appelle ces fonctions, et les opérations qu'elles permettent de réaliser, des opérations *non destructives*, en opposition aux opérations qui modifient la valeur, qui détruisent donc les valeurs précédentes, et qui sont appelées *destructives*. L'usage d'opérations non destructives pour modifier une donnée amène à la *recopie* des parties à modifier.

1.2 Fonctions simples

Voici quelques exemples de fonctions simples et utiles. Elles sont basées sur le parcours des éléments d'une liste. La *longueur* d'une liste est le nombre d'éléments qui la composent. En voici une réalisation :

```
static int longueur(Liste a)
{
  if (estVide(a)) return 0;
  return 1 + longueur(queue(a));
}
```

On peut l'écrire de manière plus condensée, en utilisant une expression conditionnelle :

```
static int longueur(Liste a)
{
  return (estVide(a)) ? 0 : 1 + longueur(queue(a));
}
```

On peut aussi l'écrire de manière itérative, et par accès direct aux champs :

```
static int longueurI(Liste a)
{
  int n = 0;
  while (a != null)
  {
    n++;
    a = a.suivant;
  }
  return n;
}
```

Autre exemple similaire, l'*affichage* d'une liste. On choisit ici d'afficher les éléments sur une ligne, séparés par un blanc.

```
static void afficher(Liste a)
{
  if (estVide(a))
    System.out.println();
```

```
  else
  {
    System.out.print(a.contenu + " ");
    afficher(a.suivant);
  }
}
```

Notons que si les deux dernières instructions de la méthode précédente sont interverties, la liste est affichée dans l'ordre inverse ! Bien sûr, on peut écrire notre fonction de façon itérative :

```
static void afficherI(Liste a)
{
  while (a != null)
  {
    System.out.print(a.contenu + " ");
    a = a.suivant;
  }
  System.out.println();
}
```

1.3 Opérations

Copie

La copie d'une liste est facile à réaliser. En voici une version récursive.

```
static Liste copier(Liste a)
{
  if (a == null)
    return a;
  return ajouter(tete(a), copier(queue(a));
}
```

Recherche

La *recherche* d'un élément dans une liste peut aussi se faire itérativement et récursivement. C'est encore un algorithme basé sur un parcours, mais le parcours est interrompu dès que l'élément cherché est trouvé. Voici l'écriture itérative :

```
static boolean estDansI(int x, Liste a) // itératif
{
  while (a != null)
  {
    if (a.contenu == x)
      return true;
    a = a.suivant;
  }
  return false;
}
```

La boucle while peut être transformée de manière mécanique en une boucle for. Dans notre cas, on obtient l'écriture suivante :

```
static boolean estDansI(int x, Liste a) // itératif
{
  for(; a!= null; a = a.suivant)
```

```
    if (a.contenu == x)
      return true;
  return false;
}
```

Voici enfin une première version récursive. Elle distingue trois cas : une liste vide, où la recherche échoue, une liste dont la tête est l'élément cherché, où la recherche réussit, et le cas restant, traité par récursion.

```
static boolean estDans(int x, Liste a) // récursif
{
  if (a == null)
    return false;
  if (a.contenu == x)
    return true;
  return estDans(x, a.suivant);
}
```

Voici une deuxième version, utilisant à la fois les fonctions d'accès et une expression booléenne :

```
static boolean estDans(int x, Liste a) // récursif
{
  return !estVide(a) && (tete(a) == x || estDans(x, queue(a)));
}
```

En Java, l'évaluation d'expressions booléennes se fait de gauche à droite et est toujours *paresseuse*. Ceci signifie que lors de l'évaluation d'une expression booléenne du type b1 || b2, où b1 et b2 sont des expressions booléennes, Java évalue d'abord b1. Si le résultat de cette évaluation est true, il en déduit que la valeur de b1 || b2 est true, sans même évaluer b2. Par contre, si le résultat est false, il poursuit en évaluant b2. Dans le cas d'une expression du type b1 && b2, le même principe paresseux s'applique : si b1 est évalué à false, Java en conclut que b1 && b2 vaut false sans évaluer b2.

Dans notre cas, le deuxième facteur de la conjonction n'est donc pas évalué si la liste est vide. On voit donc là un intérêt subtil de l'évaluation paresseuse, en sus du gain de temps. Si l'évaluation n'était pas paresseuse, comme c'était le cas en PASCAL, un des ancêtres de JAVA, l'évaluation de tete(a) provoquerait une erreur...

Suppression

La *suppression* d'un élément x dans une liste consiste à éliminer la première cellule — si elle existe — qui contient cet élément x (voir figure 1.4). Cette élimination s'effectue en modifiant la valeur du champ suivant du prédécesseur : le successeur du prédécesseur de la cellule de x devient le successeur de la cellule de x. Un traitement particulier doit être fait si l'élément à supprimer est le premier élément de la liste, car sa cellule n'a pas de prédécesseur. Nous commençons par

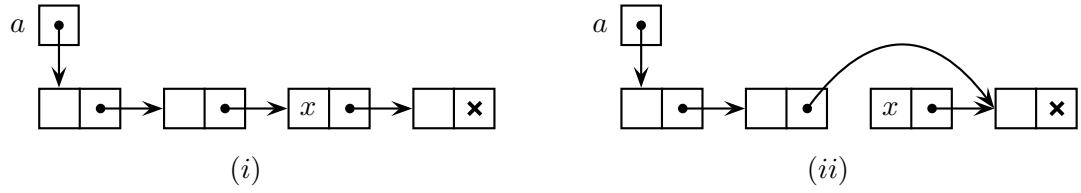


FIG. 1.4 – Suppression d'un élément x dans une liste ; (i) avant suppression, (ii) après suppression.

une écriture itérative.

```
static Liste supprimerI(int x, Liste a)
{
  if (a == null)
    return null;
  if (a.contenu == x)
    return a.suivant;
  Liste prec = a, cour = prec.suivant;
  for (; cour != null; prec = cour, cour = prec.suivant)
    if (cour.contenu == x)
    {
      prec.suivant = cour.suivant;
      return a;
    }
  return a;
}
```

Cette méthode maintient deux variables prec et cour qui contiennent les références vers la cellule courante et la cellule précédente. On peut d'ailleurs vérifier que chaque fois, cour = prec.suivant. Comme la première cellule n'a pas de précédent, elle est traitée séparément. Si la cellule courante contient l'élément cherché, cette cellule est éliminée en « sautant par dessus ». L'instruction prec.suivant = cour.suivant a pour conséquence que la cellule référencée par cour n'est plus accessible dans la liste.

Voici la même méthode, en écriture récursive :

```
static Liste supprimer(int x, Liste a)
{
  if (a == null)
    return null;
  if (a.contenu == x)
    return a.suivant;
  Liste s = supprimer(x, a.suivant);
  a.suivant = s;
  return a;
}
```

Dans cette version, si les deux tests initiaux échouent, on procède récursivement sur la queue de la liste. Le résultat est une liste qui est la nouvelle queue de la liste de départ. Il est important de noter que, dans ce cas, ce n'est pas la queue de la liste que l'on retourne mais la liste elle-même. En effet, supprimer un site au milieu d'un jeu de piste ne modifie pas le début !

Les deux lignes

```
...
Liste s = supprimer(x, a.suivant);
a.suivant = s;
...
```

peuvent bien entendu être condensées en une seule :

```
...
a.suivant = supprimer(x, a.suivant);
...
```

Les deux implantations précédentes de la méthode de suppression modifient la liste de départ, elles sont donc destructives. Une méthode de suppression non destructive conserve la liste de départ, et construit une nouvelle liste formée du début modifié de la liste, suivie de la partie commune.

La figure 1.5 montre l'effet de la suppression d'un élément x dans une liste a : la partie de la liste qui précède l'élément supprimé est recopiée. Le résultat est la liste b, la liste de départ n'est pas modifiée, et les deux listes se partagent les cellules qui suivent la cellule contenant x.

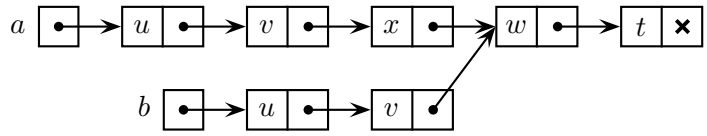


FIG. 1.5 – Suppression d'un élément x dans une liste sans destruction.

Voici une écriture, n'employant que les opérateurs non destructifs.

```
static Liste supprimer(int x, Liste a)
{
  if (estVide(a))
    return a;
  if (tete(a) == x)
    return queue(a);
  return ajouter(tete(a), supprimer(x, queue(a)));
}
```

Deux exercices pour finir...

(1) Modifier les programmes précédents pour supprimer *tous* les éléments égaux à x dans la liste.

(2) Écrire une méthode `purge` qui, appliquée à une liste chaînée, rend une liste dans laquelle les doublons ont été éliminés.

Concaténation

La *concaténation* de deux listes a et b produit une liste obtenue en ajoutant les éléments de la liste b à la fin de la liste a (voir figure 1.6).

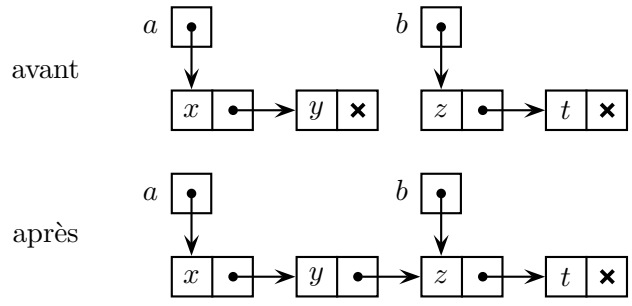


FIG. 1.6 – Concaténation de deux listes.

Cette opération illustre fort bien l'alternative entre une construction destructive et non destructive. Une version non destructive laisse les listes a et b inchangées après concaténation. Ceci oblige à effectuer une copie de la première liste (voir figure 1.7).

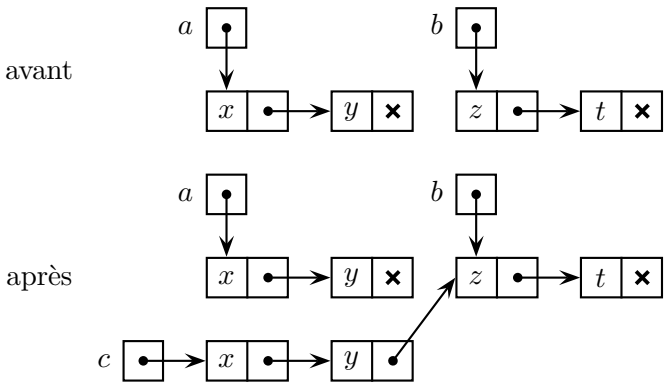


FIG. 1.7 – Concaténation de deux listes avec copie de la première.

Une version destructive allonge simplement la liste a par adjonction de la liste b comme valeur du champ suivant de la dernière cellule de a. Cette opération simple recèle un piège : si les listes a et b ont des cellules en commun, cet « allongement » provoque un cycle dans la liste ! Ce piège est illustré sur la figure 1.8.

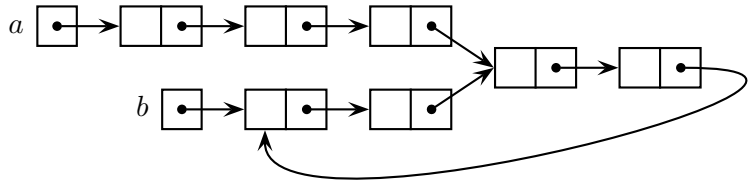


FIG. 1.8 – Concaténation de deux listes partageant des cellules.

Dans les applications, on utilise souvent la version destructive, parce qu'elle est en temps constant si l'on dispose de l'adresse de la dernière cellule.

Voici une première implantation de la concaténation, non destructive et récursive :

```
static Liste concat(Liste a, Liste b)
{
  if (estVide(a))
    return b;
  return ajouter(tete(a), concat(queue(a), b));
}
```

Voici une deuxième implantation, celle-ci destructive et récursive. Rappelons que a et b ne doivent avoir aucune cellule en commun, sinon on obtient une liste qui est — en partie au moins — circulaire !

```
static Liste fusion(Liste a, Liste b)
{ // a != b cellule par cellule
  if (estVide(a))
    return b;
  a.suivant = fusion(queue(a), b);
  return  a;
}
```

Voici une troisième implantation, destructive et itérative, avec la même restriction. On cherche d'abord la dernière cellule d'une liste.

```
static Liste dernier(Liste a)
{
  if (a == null)
    return null;
  while (a.suivant != null)
    a = a.suivant;
  return a;
}

static Liste fusionI(Liste a, Liste b)
{ // a != b cellule par cellule
  if (estVide(a))
    return b;
  dernier(a).suivant = b;
  return  a;
}
```

Exercice : comment peut-on tester que deux listes partagent une cellule ?

Inversion

Voici une autre opération. *Inverser* une liste consiste à construire une liste qui contient les éléments de la liste dans l'ordre opposé. Voici une première implantation, itérative et non destructive.

```
static Liste inverserI(Liste a)
{
  Liste b = null;
  while (a != null)
  {
    b = ajouter(a.contenu, b);
    a = a.suivant;
  }
  return b;
}
```

La même méthode est facile à écrire récursivement, en considérant qu'il s'agit d'un cas particulier d'une méthode plus générale :

```
static Liste passer(Liste a, Liste b)
{
  if (a == null)
    return b;
  return passer(queue(a), ajouter(tete(a), b));
}
```

La méthode passer recopie la liste a et l'insère en ordre inversé au début de la liste b. On le voit bien dans le corps de la méthode, où l'élément tete(a) est ajouté, en tête, dans la liste b, comme illustré sur la figure 1.9. L'inversion d'une liste est alors toute simple :

```
static Liste inverser(Liste a)
{
  return passer(a, null);
}
```

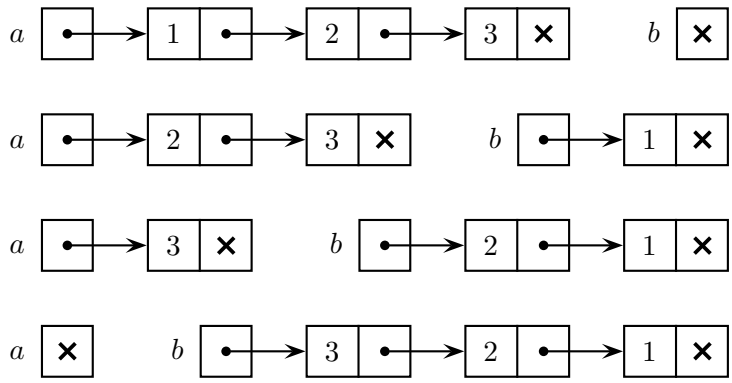


FIG. 1.9 – Inversion d'une liste.

Les deux méthodes non destructives construisent un nouvel exemplaire de la liste. On peut aussi inverser la liste « en place », c'est-à-dire sans copie, en « inversant le sens des flèches ». Le fonctionnement est illustré sur la figure 1.10. On décrit la situation juste avant l'exécution de l'instruction a.suivant = b.

```
static Liste inverserID (Liste a)
{
  Liste b = null;
  while (a != null)
  {
    Liste c = a.suivant;
    a.suivant = b; // inverse
    b = a;
    a = c;
  }
 return b;
}
```

La complexité de ces méthodes est linéaire, bien que des générations d'étudiants aient proposé les algorithmes quadratiques les plus divers...

1.4 Une application : les polynômes

Nous considérons ici des polynômes en une variable à coefficients entiers. On représente d'habitude les polynômes par un tableau de coefficients. Comme le tableau est de taille fixe mais que le degré du polynôme est éminemment variable, on complète le tableau par un entier dont la valeur est le degré. Cette représentation est couramment utilisée pour la manipulation des polynômes, par exemple dans les systèmes de calcul formel, et permet des calculs rapides. La taille de la représentation est proche du degré en général, et les opérations arithmétiques se font en temps proportionnel à la taille (ou à la plus grande des tailles) pour l'addition, la soustraction, la dérivation, etc., et au carré de ces valeurs pour la multiplication[1] par exemple.

Or, il arrive souvent qu'un polynôme ait presque tous ses coefficients nuls, comme $X^{1000} - 1$. Il n'est pas difficile d'élever ce polynôme au carré, manifestement en moins d'un million (1000^2) d'opérations ! On appelle polynôme « creux » un polynôme qui a « peu » de coefficients non nuls. Pour de tels polynômes, il est manifestement plus intéressant de disposer d'une représentation qui permette une manipulation qui ne dépend pas du degré du polynôme mais de sa taille

[1]Il existe en fait des algorithmes de meilleure complexité pour la multiplication.

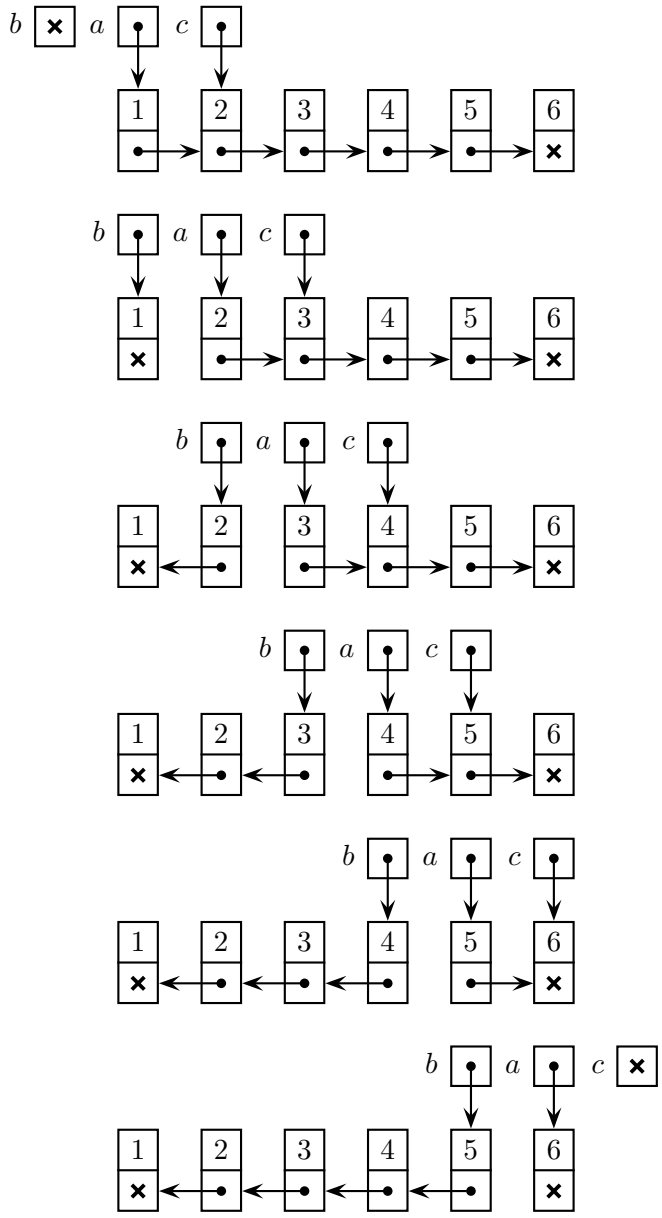


FIG. 1.10 – Inversion d'une liste « en place ».

mesurée par le nombre de ses termes non nuls. Dans les systèmes de calcul formel déjà évoqués, une telle représentation existe aussi, et il y a des routines internes de passage de l'une à l'autre des représentations en fonction de leur meilleure efficacité.

Dans cette section, nous utiliserons une représentation des polynômes par des listes chaînées. Elle aura les avantages déjà mentionnés pour les polynômes creux, à savoir des temps de calcul paramétrés par la taille. La programmation ne s'avère guère plus compliquée que pour la représentation classique. Un polynôme est représenté par une liste chaînée de monômes non nuls par degré décroissant. Chaque cellule correspond à un monôme non nul du polynôme et l'ordre des monômes est important. Chaque cellule contient donc deux données : le degré du monôme, et son coefficient (non nul !). Le polynôme nul est représenté par la liste vide. Par exemple, le polynôme $3X^5 - 4X^2 + 1$ est représenté par la liste chaînée de la figure 1.11.

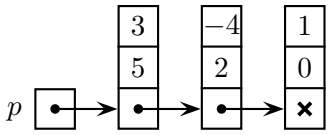


FIG. 1.11 – Liste chaînée pour le polynôme $3X^5 - 4X^2 + 1$.

Nous commençons par la description de la structure, et du constructeur.

```
class Pol
{
  int coeff, degre;
  Pol suivant;

  Pol(int c, int d, Pol p)
  {
    coeff = c;
    degre = d;
    suivant = p;
  }
}
```

Le constructeur correspond assez précisément à celui des listes ; la seule différence est la présence de deux paramètres, correspondant aux deux valeurs de chaque cellule. Ainsi, le polynôme $3X^5 - 4X^2 + 1$ est construit par

```
Pol p = new Pol(3, 5, new Pol(-4, 2, new Pol(1,0, null)));
```

Voici, pour nous mettre en appétit, une méthode qui retourne le polynôme dérivé du polynôme passé en argument.

```
static Pol deriver(Pol p)
{
  if (p == null || p.degre == 0)
    return null ;
  Pol r = deriver(p.suivant);
  return new Pol(p.coeff * p.degre, p.degre - 1, r);
}
```

La dérivation étant une opération linéaire, on dérive monôme par monôme, et on regroupe le résultat.

L'addition est réalisée comme suit :

```
static Pol additionner(Pol p, Pol q)
{
  if (p == null)
    return q;
  if (q == null)
    return p;
  if (p.degre > q.degre)
  {
    Pol r = additionner(p.suivant, q);
    return new Pol(p.coeff, p.degre, r);
  }
```

```
    if (q.degre > p.degre)
    {
      Pol r = additionner(p, q.suivant);
      return new Pol(q.coeff, q.degre, r);
    }
    // cas restant : p.degre == q.degre
    Pol r = additionner(p.suivant, q.suivant);
    int coeff = p.coeff + q.coeff;
    if (coeff == 0)
      return r;
    return new Pol(coeff, p.degre, r);
  }
```

Si l'un des deux polynômes est nul, le résultat est l'autre polynôme. Sinon, on compare les degrés des monômes de plus haut degré des deux polynômes. S'ils sont différents, il suffit d'ajouter le monôme de plus haut degré à la somme des polynômes restants. Si en revanche ils sont égaux, on les additionne — sauf s'ils ont des coefficients opposés, auquel cas on les ignore.

La multiplication de deux polynômes se ramène, en partie du moins, à l'addition. Il est facile de multiplier un polynôme par un monôme non nul : il suffit de multiplier les coefficients et d'ajouter les degrés. La méthode suivante réalise cette opération :

```
static Pol multiParMonome(Pol p, Pol m)
{ // seul le premier monôme de m est considéré
  if (p == null)
    return null;
  Pol r = multiParMonome(p.suivant, m);
  return new Pol(p.coeff * m.coeff, p.degre + m.degre, r);
}
```

Nous avons procédé de manière non destructive car, comme on le verra, le polynôme `p` intervient dans plusieurs appels de la même méthode.

La multiplication par un polynôme quelconque se ramène à la multiplication par les monômes composant ce polynôme :

```
static Pol multiplier(Pol p, Pol q)
{
  if (q == null)
    return null;
  Pol r, s;
  r = multiParMonome(p, q);
  s = multiplier(p, q.suivant);
  return additionner(r, s);
}
```

On multiplie le polynôme `p` successivement par les monômes composant `q` (cette « succession » est réalisée ici de façon récursive). Les résultats de ces multiplications sont ensuite additionnés.

Enfin, il convient d'afficher un polynôme. Il est important que le résultat de l'affichage corresponde à ce que l'on attend. Signalons quelques conventions à respecter :

- ne pas afficher un coefficient s'il vaut 1 (ou -1) sauf pour le terme constant ;
- ne pas afficher l'exposant s'il vaut 1 ;
- ne pas afficher le symbole de variable X pour le terme constant ;
- ne pas afficher le signe '+' devant le terme de plus haut degré si le coefficient est positif, mais bien devant les autres termes.

Voici une réalisation. Elle utilise une méthode qui affiche le signe :

```
static void afficherSigne(int n)
{
  System.out.print((n >= 0) ? " + " : " - ");
}
```

Cette méthode est utilisée pour tous les termes sauf le premier :

```
static void afficherMonome(Pol p, boolean premierMonome)
{
  int a = p.coeff;
  if (premierMonome && (a < 0))
    System.out.print("-"); // Pour avoir -5X^2 + 3X - 1
  else if (!premierMonome)
    afficherSigne(a);      // Evite + 5X
  if ((a != 1 && a != -1) || p.degre == 0)      // Evite 1X^3
    System.out.print(Math.abs(a));
  if (p.degre > 0)         // Evite X^0
    System.out.print("X");
  if (p.degre > 1)         // Evite X^1
    System.out.print("^" + p.degre);
}
```

Enfin, l'affichage d'un polynôme complet se fait par :

```
static void afficher(Pol p)
{
  if (p == null)
    System.out.print(0);
  else
  {
    afficherMonome(p, true);
    p = p.suivant;
    for (; p != null; p = p.suivant)
      afficherMonome(p, false);
  }
}
```

Dans le cas d'un polynôme non nul, le premier monôme est affiché avec la valeur true pour le deuxième paramètre, les autres avec la valeur false.

Voici une implantation un peu différente. Cette fois, nous allons produire un résultat directement utilisable par TEX pour afficher le résultat habituel. Deux éléments de syntaxe de TEX suffisent à comprendre le programme. Tout d'abord, tout ce qui est en mode mathématique est encadré par des signes $. Ensuite, pour écrire X^{23} en TEX, on écrit `$X^{23}$`.

Notre programme construit un String directement exploitable par TEX. Ici, la méthode monome ne retourne pas le signe qui est calculé directement par la méthode signe.

```
static String signe(int n)
{
  return (n >= 0) ? " + " : " - ";
}

static String monome(Pol p)
{
  String m = "";
  int a = p.coeff;
```

```
    if ((a != 1 && a != -1) || p.degre == 0)      // Evite 1X^3
      m += Math.abs(a);
    if (p.degre > 0)      // Evite X^0
      m += "X";
    if (p.degre > 1)      // Evite X^1
      m += "^{" + p.degre + "}";
    return m;
  }

  static String ecrire(Pol p)
  {
    if (p == null)
      return "$0$";
    else
    {
      String m = "$";
      if (p.coeff < 0)
        m += signe(p.coeff);
      m += monome(p);
      p = p.suivant;
      for (; p != null; p = p.suivant)
        m += signe(p.coeff) + monome(p);
      return m+"$";
    }
  }
```

Par exemple, le polynôme de la figure 1.11 s'écrit `$3X^{5} - 4X^{2} + 1$`.

2 Listes circulaires

Dans une liste chaînée ordinaire, l'adresse du champ suivant de la dernière cellule est null. C'est d'ailleurs comme cela que l'on reconnaît la fin d'une liste. Dans une *liste circulaire*, le champ suivant de la dernière cellule contient la référence de la première cellule. Cette variante présente des avantages dans le cas où tous les éléments représentés dans une liste ont la même importance. Deux exemple concrets : un *polygone* est donné par la suite de ses points. Il est naturel de ranger cette suite dans une liste circulaire. Une *permutation* est un produit de cycles. Chaque cycle est naturellement représenté par une liste circulaire.

On repère une liste circulaire non vide par la référence à une cellule appelée l'*entrée*.

2.1 Définitions

La structure d'une cellule d'une liste circulaire est la même que pour une liste ordinaire (séquentielle). Pour simplifier, nous supposons que les éléments rangés dans la liste sont des entiers.

```
class ListeC
{
  int contenu;
  ListeC suivant;
  ...
}
```

La *liste circulaire vide* est représenté par null. Une liste circulaire qui contient un seul élément est un *singleton*. Le champ suivant de la cellule d'un singleton contient la référence à la cellule elle-même. On peut donc créer une liste singleton par :

```
static ListeC singleton(int x)
{
  ListeC a = new ListeC();
  a.contenu = x;
  a.suivant = a;
  return a;
}
```

La fonction qui teste si une liste est un singleton s'écrit :

```
static boolean estSingleton (ListeC a)
{
  return a != null && a == a.suivant;
}
```

Quand on ajoute un élément à une liste circulaire qui n'est pas vide, la nouvelle cellule doit être insérée dans la liste. Pour cela, il faut indiquer quelle est la cellule suivante, mais aussi la cellule dont cette nouvelle cellule est la suivante (sa « précédente »). Pour rendre l'opération de complexité constante, on insère donc cette cellule juste *après* l'entrée.

```
static ListeC ajouter(int x, ListeC a)
{
  if (a == null)
    return singleton(x);
  ListeC n = new ListeC();
  n.contenu = x;
  n.suivant = a.suivant;
  a.suivant = n;
  return a;
}
```

Ainsi, une liste circulaire composée des entiers $4, 7, 11$ se crée par :

```
ListeC a = ajouter(4, ajouter(7, ajouter(11, null)));
```

Noter que le contenu de la cellule d'entrée est 11, c'est-à-dire le premier élément inséré. Les méthodes singleton et ajouter font appel au constructeur par défaut ListeC(). Il est plus dans l'esprit de Java de définir des constructeurs spécifiques. Nous en introduisons deux, l'un pour les singletons, et l'autre pour l'adjonction.

```
ListeC(int x)
{ // singleton
  contenu = x;
  suivant = this;
}

ListeC(int x, ListeC a)
{
  contenu = x;
  if (a == null)
    suivant = this;
  else
    suivant = a;
```

```
}
```

Avec ces constructeurs, on réécrit les méthodes singleton et ajouter comme suit :

```
static ListeC singleton(int x)
{
  return new ListeC(x);
}
```

et

```
static ListeC ajouter(int x, ListeC a)
{
    if (a == null)
        return singleton(x);
    a.suivant = new ListeC(x, a.suivant);
    return a;
}
```

Noter que l'on ne retourne pas, dans ajouter, la référence à la cellule nouvelle.

2.2 Opérations

Parcours

Le parcours d'une liste circulaire est, lui aussi, différent du parcours d'une liste séquentielle. Voici quelques exemples. Commençons par le calcul de la longueur d'une liste circulaire.

```
static int longueurI(ListeC a)
{
  if (a == null)
    return 0;
  int n = 0;
  ListeC b = a;
  do
  {
    b = b.suivant;
    n++;
  }
  while (b != a);
  return n;
}
```

On remarque l'usage de la boucle `do ... while`. Dans une telle boucle, le corps est exécuté *avant* le test de continuation. Cette boucle est nécessaire dans une liste circulaire (non vide) car le test d'arrêt est la visite d'une cellule particulière. Pour que cette cellule soit quand-même visitée, il faut commencer par elle et ne faire le test d'arrêt qu'après la visite.

En revanche, l'instruction `b = b.suivant;` peut se déplacer en début ou en fin de la boucle `do ... while`. Si elle est au début, la cellule de tête sera examinée en dernier, si elle est à la fin de la boucle, la cellule de tête sera la première visitée. Dans notre optique où la première cellule contient le dernier élément de la liste, c'est donc de cette façon que le parcours se fait naturellement.

Et les méthodes récursives ? Elles s'écrivent naturellement, mais en prenant soin du test d'arrêt. En fait, et comme dans certains exemples de listes séquentielles, on introduit un paramètre supplémentaire, de « fin de liste » dans une fonction auxiliaire. Ainsi, la fonction longueur à deux arguments retourne la longueur de la liste débutant en d et se terminant en a.

```
static int longueur(ListeC a)
{
  if (a == null)
    return 0;
  return longueur(a.suivant, a);
}
static int longueur(ListeC d, ListeC a)
{
  if (d == a)
    return 1;
  else
    return 1 + longueur(d.suivant, a);
}
```

Autre exemple similaire, l'affichage d'une liste circulaire. On choisit d'afficher les éléments sur une ligne, séparés par un blanc.

```
static void afficher(ListeC a)
{
  if (a != null)
    afficher(a.suivant, a);
  System.out.println();
}

static void afficher(ListeC d, ListeC a)
{
  System.out.print(d.contenu + " ");
  if (d != a)
    afficher(d.suivant, a);
}
```

Bien sûr, on peut écrire cette fonction itérativement.

Voici un exemple d'emploi des listes circulaires : on cherche à calculer la somme des carrés des différences entre éléments consécutifs. Pour être précis, si $(x_1, \ldots, x_n)$ sont les éléments de la liste, on cherche

$$(x_2 - x_1)^2 + \ldots + (x_n - x_{n-1})^2 + (x_1 - x_n)^2 \,.$$

Dans une liste séquentielle ordinaire, le dernier terme de cette somme doit être calculé séparément. Ce n'est pas le cas pour une liste circulaire.

```
static int sommeCarresDifferences(ListeC a)
{
  if (a  == null)
    return 0;
  int somme = 0;
  ListeC b = a;
  do
  {
    b = b.suivant;
    int x = b.contenu;
    int y = b.suivant.contenu;
    somme += (y-x)*(y-x);
  }
  while (b != a);
```

```
    return somme;
  }
```

Observer que dans le cas d'un singleton, on obtient bien zéro.

Fusion

La *fusion* de deux listes circulaires est très simple. Elle correspond à la concaténation de listes séquentielles.

```
static ListeC fusion(ListeC a, ListeC b)
{
  if (a == null)
    return b;
  if (b == null)
    return a;
  Liste c = a.suivant;
  a.suivant = b;
  b.suivant = c;
  return a;
}
```

Deux listes circulaires sont similaires, si elles sont formées des mêmes cellules. Pour tester que deux listes sont similaires, il suffit de vérifier qu'une cellule quelconque de l'une des listes ne figure pas dans l'autre.

```
static boolean sontSimilaires(ListeC a, ListeC b)
{
  if (a  == null || b == null)
    return a == b;
  ListeC c = a;
  do
  {
    c = c.suivant;
    if (c == b)
      return true;
  }
  while (c != a);
  return false;
}
```

Exercice : quel est l'effet de la fusion de deux listes similaires ?

2.3 Exemples

Permutations

Rappelons qu'une *permutation* d'un ensemble fini E est une bijection de E dans E. La décomposition en *cycles* d'une permutation p s'obtient de la manière suivante. Pour construire un cycle, on choisit un élément de $x \in E$, puis on calcule la suite des éléments $x, p(x), p^2(x), p^3(x), \ldots$ jusqu'à ce qu'on revienne sur l'élément x (on revient nécessairement sur x, car E est fini). On répète ce processus pour former le cycle suivant à partir d'un élément qui n'est pas apparu dans le premier cycle. Le processus s'arrête lorsque tous les éléments de E sont apparus dans un cycle. Les cycles obtenus sont deux à deux disjoints. On représente la permutation en juxtaposant ces

cycles. Par exemple, pour la permutation p donnée par

i	0	1	2	3	4	5	6	7	8	9
$p(i)$	3	0	8	1	9	4	6	2	5	7

la décomposition en cycles est (0 3 1)(2 8 5 4 9 7)(6).

Passons à la programmation. Chaque cycle est représenté par une liste circulaire.

```
static ListeC[] enCycles(int[] p)
{
  int n = p.length;
  boolean[] vu = new boolean[n];
  ListeC[] c = new ListeC[n];
  int nc = 0;
  for (int m = 0; m < n; m++)
    if (!vu[m])
    {
      ListeC a = singleton(m);
      vu[m] = true;
      for (int j = p[m]; j != m; j = p[j])
      {
        a = ajouter(j, a);
        vu[j] = true;
        a = a.suivant;
      }
      c[nc++] = a;
    }
  return c;
}
```

L'affichage se fait en parcourant le tableau `c`, et on s'arrête lorsque l'on rencontre une liste vide.

```
static void afficherCycles(ListeC[] c)
{
  int n = c.length;
  for (int m = 0; m < n && c[m] != null; m++)
  {
    ListeC b = c[m];
    System.out.print("(");
    do
    {
      System.out.print(b.contenu);
      if (b.suivant != c[m])
        System.out.print(" ");
      b = b.suivant;
    }
    while (b!= c[m]);
    System.out.print(")");
  }
  System.out.println();
}
```

3 Variations sur les listes

On utilise aussi d'autres variantes des listes. Un excellent exercice de programmation consiste à adapter à ces différentes variantes les programmes décrits pour les listes chaînées.

Les listes *gardées* contiennent une cellule spéciale, qui sert en quelque sorte de marqueur. On distingue les listes *gardées à la fin*, dans lesquelles la cellule spéciale indique la fin de la liste, et les listes *gardées au début*, dans lesquelles la cellule spéciale indique le début de la liste, mais ne contient aucun élément. On peut bien sûr avoir des listes *doublement gardées* qui contiennent deux cellules spéciales pour marquer le début et la fin.

Les listes *doublement chaînées* sont des listes qui contiennent une référence vers la liste suivante et une référence vers la liste précédente. On peut les déclarer comme suit :

```
class Liste
{
  int contenu;
  Liste suivant;
  Liste precedent;

  Liste (int x, Liste s, Liste p)
  {
    contenu = x;
    suivant = s;
    precedent = p;
  }
}
```

4 Hachage

Nous abordons dans cette section l'un des problèmes les plus fréquemment rencontrés en informatique, la recherche d'information dans un ensemble géré de façon dynamique.

Pour formaliser ce problème, on suppose que les éléments de l'ensemble sont constitués de plusieurs champs. L'un de ces champs, appelé *clé*, est élément d'un ensemble totalement ordonné U, appelé l'*univers* des clés. En pratique, une clé est le plus souvent un nombre entier ou une chaîne de caractères. Dans le premier cas, l'univers est un intervalle d'entiers, muni de l'ordre naturel ; dans le second cas, l'univers est un ensemble de chaînes de caractères, tel que l'ensemble des chaînes de longueur ≤ 10, et il est muni de l'ordre lexicographique.

Les autres champs des éléments sont des informations auxiliaires. Pour avoir accès à ces informations, on recherche l'élément à l'aide de sa clé. Par exemple, le numéro de sécurité sociale est une clé qui permet d'avoir accès à divers types d'information sur un individu (sexe, age, date de naissance, etc.)

On souhaite non seulement retrouver une information sur un élément en recherchant sa clé, mais aussi pouvoir insérer ou supprimer un élément. On est ainsi amené à implanter un type abstrait de données, appelé *dictionnaire*, qui opère sur un ensemble totalement ordonné à l'aide des opérations suivantes :

- rechercher un élément
- insérer un élément
- supprimer un élément

Les tables de hachage, présentées ci-dessous, permettent une implantation des dictionnaires particulièrement efficace, mais d'autres implantations sont envisageables. Nous en verrons en particulier une autre lors de l'étude des arbres binaires de recherche.

Précisons un point avant de commencer. Il arrive souvent dans la pratique que la clé permette d'identifier complètement l'élément cherché, comme dans le cas du numéro de sécurité sociale. Nous nous placerons toutefois dans le cadre plus général où plusieurs éléments peuvent avoir la même clé. Ce type de clé est également d'utilisation courante. Par exemple, on peut rechercher sur l'ordinateur tous les noms de fichier commençant par la chaîne de caractères `Poly.`. On trouve tous les fichiers utilisés par LaTeX pour composer ce polycopié : `Poly.aux`, `Poly.bbl`, `Poly.bib`, `Poly.blg`, `Poly.dvi`, `Poly.idx`, `Poly.ind`, `Poly.log`, `Poly.pdf`, `Poly.ps`, `Poly.tex`, `Poly.toc`.

4.1 Adressage direct

Cette technique très efficace ne peut malheureusement s'appliquer que dans des cas très particuliers. Il faut d'une part que l'univers des clés soit de la forme $\{0, \ldots, n-1\}$, où n est un entier pas trop grand, et d'autre part que deux éléments distincts aient des clés distinctes. Il suffit alors d'utiliser un tableau de taille n pour représenter l'ensemble des éléments.

Par exemple, supposons que chaque donnée soit constituée par un couple (rang, note) où rang est le rang de classement et note est la note d'un élève pour le cours 421.

```
class Note
{
  int rang;
  int note;
}
```

La clé est le rang de classement d'un élève. On représente donc l'ensemble des données par un tableau t[500] où t[i] est un objet de la classe Note. Si l'élève de rang i a suivi l'un des cours 421, t[i].note donne la note de l'élève. Sinon, la valeur est t[i] est null. Les trois méthodes utilisées par un dictionnaire s'implantent sans difficulté en temps $O(1)$.

```
class AdressageDirect
{
  static Note[] t = new Note[500];

  static Note chercher(int k)
  {
    return t[k];
  }

  static void inserer(int k, Note n)
  {
    t[k] = n;
  }

  static void supprimer(int k)
  {
    t[k] = null;
  }
}
```

4.2 Tables de hachage

Supposons maintenant que l'ensemble K des clés soit beaucoup plus petit que l'univers U de toutes les clés possibles : par exemple, dans le cas d'un annuaire téléphonique, l'ensemble K est l'ensemble des noms des abonnés, alors que U est l'ensemble des chaînes de caractères. Même

en se limitant aux chaînes de longueur ≤ 16, voire ≤ 8, l'univers reste beaucoup trop vaste pour que la méthode par adressage direct puisse s'appliquer.

L'utilisation d'une table de hachage va nous permettre de ranger les clés dans un tableau T de taille m très inférieure à la taille de U. On se donne pour cela une fonction

$$h : U \to \{0, \ldots, m-1\}$$

appelée *fonction de hachage*. L'idée est de ranger l'élément de clé k non pas dans $T[k]$, comme dans l'adressage direct, mais dans $T[h(k)]$. Nous reviendrons dans la section 4.5 sur le choix, relativement délicat, de la fonction de hachage, mais nous devons faire face dès à présent à une difficulté : comme il est possible d'avoir $h(k) = h(k')$ pour deux clés différentes, il se peut que la cas $h(k)$ soit déjà occupée, comme dans la figure 4.12.

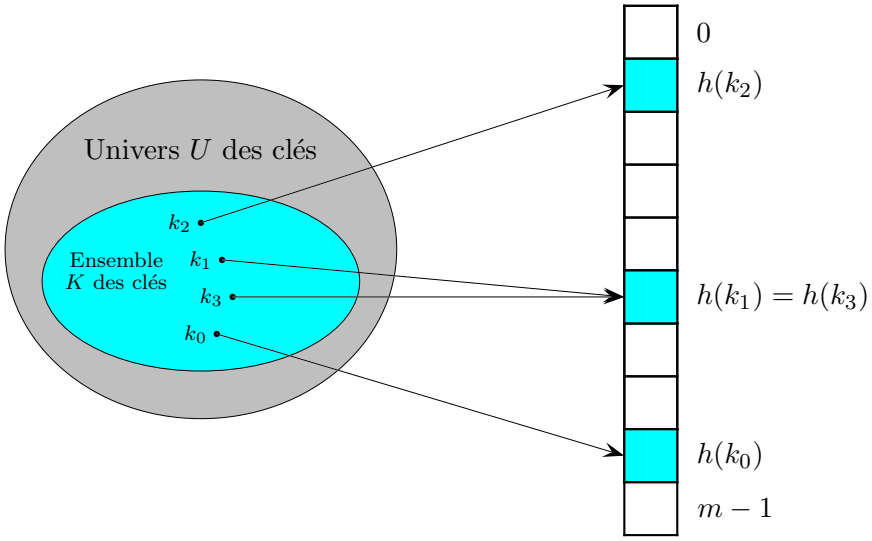


FIG. 4.12 – Une table de hachage.

Dans ce cas, où doit-on ranger l'élément de clé k ? Nous allons proposer deux solutions différentes à ce problème de *collision* : le chaînage et l'adressage ouvert.

4.3 Résolution des collisions par chaînage

Le principe est très simple : elle consiste à mettre dans une liste chaînée toutes les clés qui ont même valeur de hachage. Si une valeur de hachage ne correspond à aucune clé, on aura une liste vide. Cette technique est illustrée sur la figure 4.13.

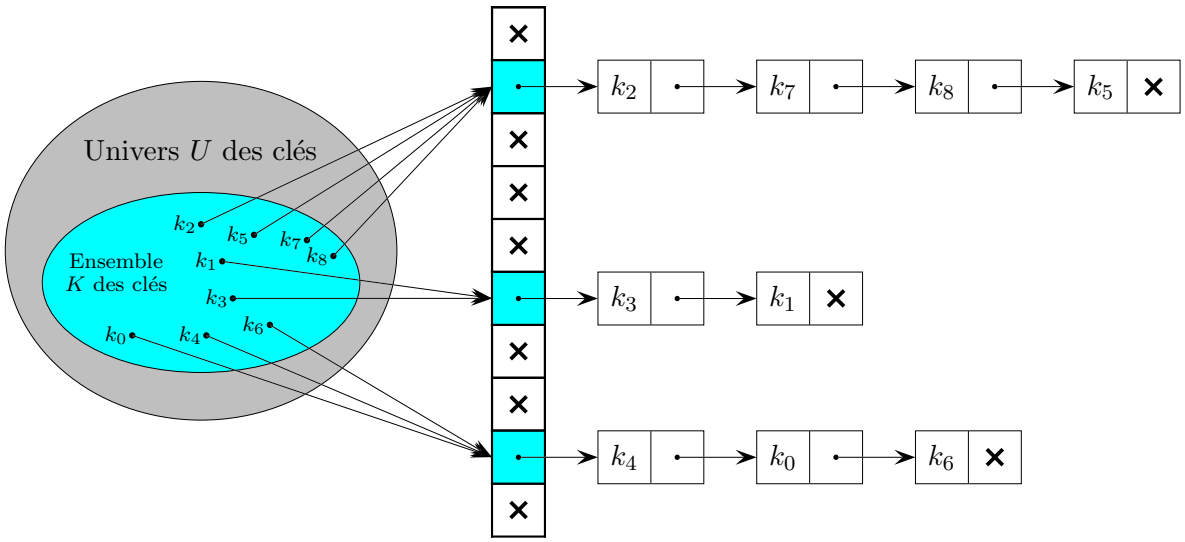


FIG. 4.13 – Résolution des collisions par chaînage.

Les trois méthodes des dictionnaires sont alors modifiées comme suit :

```
class Chainage
{
  static final int m = 20;
  static Liste[] t = new Liste[m];

  static int hachage(Element x)
  { ... }

  static boolean estDans(Element x)
  {
    int k = hachage(x.cle);
    return Liste.estDans(x, t[k]);
  }

  static void inserer(Element x)
  {
    int k = hachage(x.cle);
    t[k] = Liste.ajouter(x, t[k]);
  }

  static void supprimer(Element x)
  {
    int k = hachage(x.cle);
    t[k] = Liste.supprimer(x, t[k]);
  }
}
```

Si l'on suppose que la fonction de hachage se calcule en temps $O(1)$, l'insertion se fait dans le cas le pire en temps $O(1)$ car il suffit d'insérer un élément en tête de liste. On démontre que la recherche d'un élément se fait en moyenne en temps $O(1+\alpha)$, où $\alpha = n/m$ est le *taux de remplissage* de la table (n est le nombre de clés à ranger et m est la taille de la table).

4.4 Adressage ouvert

Dans l'adressage ouvert, on prévoit une table de taille supérieure au nombre de clés à ranger. Le taux de remplissage de la table est donc ici toujours inférieur à 1. Nous verrons qu'en pratique, un taux de remplissage compris entre 50% et 95% convient, mais qu'il est préférable de ne pas dépasser cette dernière valeur.

Un élément de clé k est donc rangé dans la case $h(k)$ de la table. Si cette case est déjà occupée (collision), la solution la plus simple consiste à ranger la clé dans la case $h(k)+1$, si elle est libre, ou à défaut dans la case $h(k)+2$, etc. Ces calculs sont effectués modulo la taille m de la table.

Toutefois cette solution trop simpliste provoque des phénomènes de regroupement qui sont faciles à formaliser. Considérons par exemple la table ci-dessous, où les cases encore libres sont en blanc :

FIG. 4.14 – Un phénomène de regroupement.

En supposant que $h(k)$ soit distribué uniformément, la probabilité pour que la case i soit choisie à la prochaine insertion est la suivante

$$P(0)=1/19 \quad P(2)=2/19 \quad P(3)=1/19 \quad P(8)=5/19 \quad P(9)=1/19$$
$$P(12)=3/19 \quad P(13)=1/19 \quad P(14)=1/19 \quad P(16)=2/19 \quad P(18)=2/19$$

Comme on le voit, la case 8 a la plus grande probabilité d'être occupée, ce qui accentuera le phénomène de regroupement des cases 4-7.

Plusieurs solutions ont été proposées pour éviter ce problème. Le *hachage quadratique* consiste à choisir successivement, en cas de collision, les cases $h(k)+1$, $h(k)+4$, $h(k)+9$, $h(k)+16$, etc. Mais là encore, des phénomènes de regroupement peuvent se produire.

La meilleure solution consiste à utiliser un *double hachage*. On se donne deux fonctions de hachage $h : U \to \{0,\ldots,m-1\}$ et $h' : U \to \{0,\ldots,r-1\}$. En cas de collision, on choisit successivement les cases $h(k)+h'(k)$, $h(k)+2h'(k)$, $h(k)+3h'(k)$, etc. En pratique, on peut prendre $h'(k) = 8-(k \mod 8)$ ou $h'(k) = 1+(k \bmod (m-1))$.

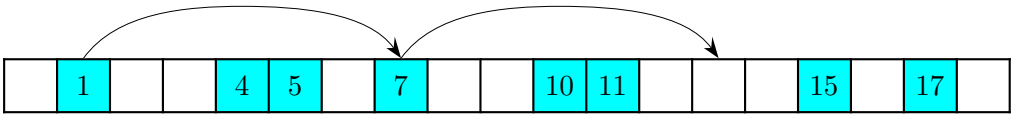


FIG. 4.15 – Insertion par double hachage. On prend $h(k) = k \bmod 19$ et $h'(k) = 1+(k \bmod 18)$. Si $k = 77$, on a $h(k) = 1$ et $h'(k) = 6$. Les cases 1 et 7 étant occupées, l'élément sera inséré dans la case 13.

Pour rechercher un élément de clé k, on procède de la même façon, en examinant successivement le contenu des cases $h(k)+h'(k)$, $h(k)+2h'(k)$, $h(k)+3h'(k)$, etc. La recherche s'arrête lorsque l'élément a été trouvé (succès) ou lorsqu'on a atteint une case vide (échec).

On démontre (sous des hypothèses de distribution uniforme) que si $\alpha = n/m$ est le taux de remplissage, le temps moyen de recherche pour un hachage double est au plus $-\ln(1-\alpha)/\alpha$ en cas de succès et à $1/(1-\alpha)$ en cas d'échec. Le tableau ci-dessous donne quelques valeurs

numériques :

Taux de remplissage	50%	80%	90%	99%
Succès	1,39	2,01	2,56	4,65
Echec	2	5	10	100

Comme on le voit $\alpha = 80\%$ est un excellent compromis. Insistons sur le fait que les valeurs de ce tableau sont indépendantes de n. Par conséquent, avec un taux de remplissage de 80%, il suffit en moyenne de deux essais pour retrouver un élément, même avec dix milliards de clés !

4.5 Choix des fonctions de hachage

Rappelons qu'une fonction de hachage est une fonction $h : U \to \{0, \dots m-1\}$. Une bonne fonction de hachage doit se rapprocher le plus possible d'une répartition uniforme. Formellement, cela signifie que si on se donne une probabilité P sur U et que l'on choisit aléatoirement chaque clé k dans U, on ait pour $0 \leq j \leq m-1$,

$$\sum_{\{k|h(k)=j\}} P(k) = \frac{1}{m}$$

En pratique, on connaît rarement la probabilité P et on se limite à des heuristiques.

Le cas que nous traitons ci-dessous est celui où l'univers U est un sous-ensemble des entiers naturels, car on peut toujours en pratique se ramener à ce cas. Par exemple, si les clés sont des chaînes de caractères, on peut interpréter chaque caractère comme un entier. Plusieurs codages sont d'ailleurs possibles : le plus fréquent est de choisir le code ASCII du caractère, ce qui donne un entier sur 7 bits, compris entre 0 et 127. Mais on pourrait aussi prendre son unicode, qui est un entier sur 16 bits, compris entre 0 et 16383. Enfin, si on est assuré de n'avoir que des caractères entre a et z, et si on ne tient pas compte des majuscules, on peut coder les 26 caractères de l'alphabet en utilisant seulement des entiers de 0 à 25. Ainsi le caractère `j` sera-t-il codé par 106 dans le premier et le second cas et par 10 dans le dernier cas, puisque `j` est la dixième lettre de l'alphabet.

Chaque caractère est maintenant codé par un entier compris entre 0 et $B-1$, où B dépend du codage choisi : 128 pour le codage ASCII, 16384 pour l'unicode et 26 pour le codage alphabétique. Il suffit maintenat d'interpréter une chaîne de caractères comme un entier écrit en base B. Par exemple, dans le codage ASCII, le code de `a` est 97, celui de `j` est 106 et celui de `v` est 118.

`j`	`a`	`v`	`a`
106	97	118	97

La chaîne `java` est donc codée par l'entier $106 \times 128^3 + 97 \times 128^2 + 118 \times 128 + 97 = 223902561$.

Revenons aux fonctions de hachage sur les entiers. Une technique courante consiste à prendre pour $h(k)$ le reste de la division de k par m :

$$h(k) = k \mod m$$

Mais dans ce cas, certaines valeurs de m sont à éviter. Par exemple, si on prend $m = 2^r$, $h(k)$ ne dépendra que des r derniers bits de k. Si on code des chaînes de caractères et si on prend $m = 2^B - 1$, l'interversion de deux caractères passera inapercue. En effet, comme $(a2^B + b) - (b2^B + a) = (a-b)(2^B - 1)$, on a

$$a2^B + b \equiv b2^B + a \pmod m$$

En pratique, une bonne valeur pour m est un nombre premier pas trop proche d'une puissance de 2.

Une autre technique consiste à prendre

$$h(k) = \lfloor m(Ck - \lfloor Ck \rfloor) \rfloor$$

où C est une constante réelle telle que $0 < C < 1$. Cette méthode a l'avantage de pouvoir s'appliquer à toutes les valeurs de m. Il est même conseillé dans ce cas de prendre $m = 2^r$ pour faciliter les calculs. Pour le choix de C, Knuth recommande le nombre d'or, $C = (\sqrt{5} - 1)/2 \approx 0,618$.

Chapitre III

Piles et files

Une *pile* est une structure de données où les insertions et les suppressions se font toutes du même côté. Une telle structure est aussi appelée LIFO (last-in first-out).

Une *file* est une structure où les insertions se font d'un côté et les suppressions de l'autre côté. Une telle structure est aussi appelée FIFO (first-in first-out).

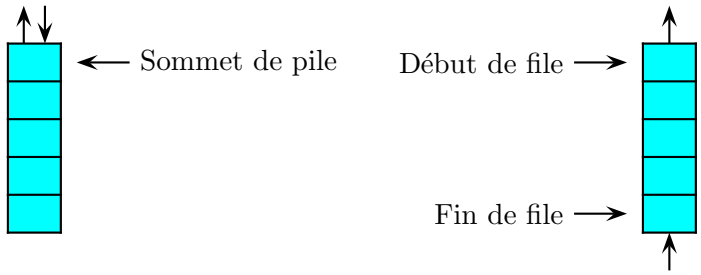


FIG. 0.1 – Une pile (insertion et extraction du même côté), et une file (insertion en fin, extraction au début).

Les piles et les files se rencontrent fréquemment. Ainsi, les appels de méthodes dans l'exécution d'un programme sont gérés par une pile (la pile d'exécution). Au sommet de la pile se trouve la dernière méthode appelée, dont l'exécution est en cours. Juste en dessous, on trouve la méthode qui a appelé la méthode en cours d'exécution, et ainsi de suite. Le fond de la pile est constitué de la méthode main par laquelle l'exécution du programme a été lancée. Lorsque la méthode en cours se termine, elle est enlevée de la pile d'exécution, et le programme continue avec la méthode appelante. Au contraire, l'appel d'une nouvelle méthode provoque l'empilement, au sommet de la pile, de cette méthode. Dans une méthode récursive, la même méthode peut se trouver empilée plusieurs fois.

Une autre situation où l'on rencontre, au moins conceptuellement, une pile, est fournie par un éditeur de texte évolué qui possède un couple d'actions « annuler-répéter » (ou « undo-redo »). Chaque action d'édition est empilée, et la commande « undo » défait la dernière action, celle qui se trouve au sommet de la pile. On peut imaginer une deuxième pile (ce n'est en général pas ainsi que le mécanisme est réalisé) où sont empilées les actions que l'on vient d'annuler. Alors, une commande « undo » empile l'action défaite sur la deuxième pile, et une commande « redo » l'enlève de la deuxième pile et la remet sur la première.

Comme exemple plus élémentaire, on verra que l'évaluation d'une expression arithmétique peut également se faire avec une pile. De façon générale, on peut simuler toute méthode récursive par une méthode itérative utilisant une pile.

De par leurs caractéristiques (les ajouts et les suppressions se font du même côté), les piles modélisent tout système où l'entrée et la sortie se font par le même lieu : wagons sur une voie

de garage, cabine de téléphérique, etc.

Les files servent à gérer, tout d'abord, les files d'attente. De telles files modélisent par exemple les processus opérant sur un système : chaque processus dispose des ressources du processeur quand son tour arrive. Souvent, les processus sont munis de priorités qui leur permettent de passer avant d'autres ; il s'agit alors de ce qu'on appelle des files de priorité (on les verra plus loin). Tout système qui traite des requêtes par ordre d'arrivée (système de réservation, d'inscription etc) gère une file. Contrairement à une pile, une file dispose de deux points de communication, l'un pour l'entrée, l'autre pour la sortie.

1 Piles

Les opérations usuelles sur les piles sont les suivantes :
- *créer* une pile vide.
- *tester* si une pile est vide.
- *ajouter* un élément à la pile. Il est ajouté au sommet de la pile.
- *trouver* l'élément qui est au sommet de la pile.
- *supprimer* un élément de la pile. C'est l'élément au sommet de la pile qui est supprimé.

Trois méthodes définissent ces opérations. La méthode $\mathsf{ajouter}(x, p)$ permet d'ajouter l'élément x à la pile p et la fonction $\mathsf{valeur}(p)$ retourne la valeur qui se trouve au sommet de la pile. Enfin $\mathsf{supprimer}(p)$ enlève de la pile l'élément qui se trouve à son sommet. Les deux dernières opérations demandent bien sûr que la pile ne soit pas vide. Ces méthodes sont liées par les équations suivantes.

$$\mathsf{valeur}(\mathsf{ajouter}(x, p)) = x$$
$$\mathsf{supprimer}(\mathsf{ajouter}(x, p)) = p$$

1.1 Implantation

Deux structures sont utilisées pour implanter une pile : les tableaux et les listes chaînées. Les tableaux ont l'avantage de la simplicité, mais manquent un peu de souplesse, car il faut gérer la taille du tableau contenant les éléments. C'est pourtant la méthode la plus fréquemment utilisée, et c'est celle que nous mettons en place (figure 1.2). La seconde méthode est laissée au lecteur à titre d'exercice.

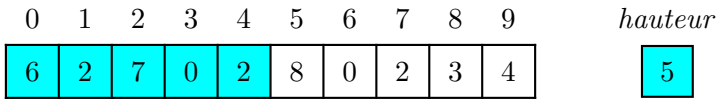


FIG. 1.2 – Une pile implantée par un tableau et un entier. Seules les parties ombrées sont significatives.

On considère des piles d'entiers, pour simplifier. Une pile contient donc un tableau d'entiers, dont nous fixons la taille arbitrairement à une valeur constante. Seul le début du tableau (un segment initial) est utilisé, et un indice variable indique la *hauteur* de la pile, c'est-à-dire le nombre d'éléments présents dans la pile. Ils occupent les places d'indices $0, 1, \ldots$ *hauteur* $- 1$ dans le tableau, et la hauteur est donc aussi l'indice de la première place disponible. Dans la figure 1.2, seules les 5 premières valeurs sont significatives.

```
class Pile
{
  static final int maxP = 10; // assez grand ?

  int hauteur;
  int[] contenu;

  Pile()
  {
    hauteur = 0;
    contenu = new int[maxP];
  }
}
```

À la création, une pile contient donc un tableau de 10 entiers, tous initialisés à 0 (figure 1.3).

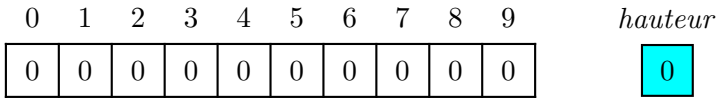


FIG. 1.3 – Une pile à la création.

Trois méthodes de gestion simples pour commencer.

```
static boolean estVide(Pile p)
{
  return p.hauteur == 0;
}

static boolean estPleine(Pile p)
{
  return p.hauteur == Pile.maxP;
}

static void vider(Pile p)
{
  p.hauteur = 0;
}
```

Par exemple, la méthode vider, appliquée à la pile de la figure 1.2, produit la pile de la figure 1.4. On notera que le tableau d'une pile vide ne contient pas nécessairement que des valeurs nulles !

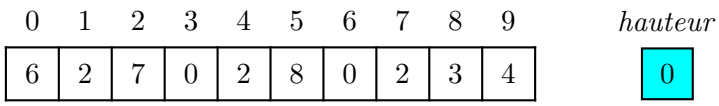


FIG. 1.4 – La pile de la figure 1.2, après avoir été « vidée ».

Les trois méthodes propres à la manipulation des piles s'écrivent comme suit :

```
static int valeur(Pile p)
{
  return p.contenu[p.hauteur - 1];
}

static void supprimer(Pile p)
{
  --p.hauteur;
}

static void ajouter(int x, Pile p)
{
  p.contenu[p.hauteur++] = x;
}
```

Les figures 1.5 et 1.6 illustrent l'ajout et la suppression.

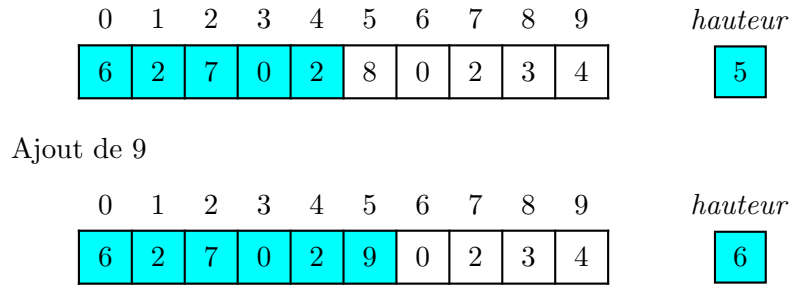


FIG. 1.5 – Ajout de 9 à la pile.

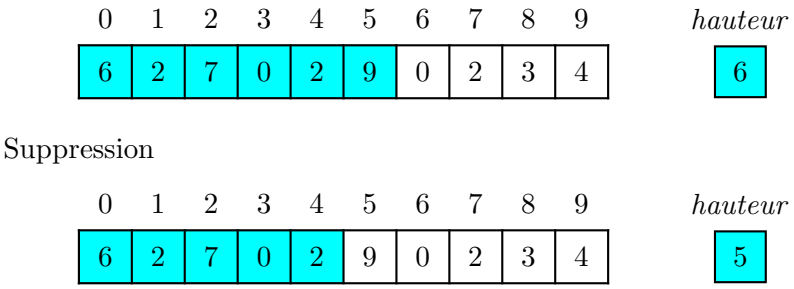


FIG. 1.6 – Suppression dans la pile. C'est l'élément en haut de pile qui est supprimé.

Les six méthodes précédentes peuvent être incluses dans la classe Pile, qui prend alors la forme suivante

```
class Pile
{
  static final int maxP = 10;

  int hauteur;
  int[] contenu;
```

```
  Pile()  { ... }
  static boolean estVide(Pile p)  { ... }
  static boolean estPleine(Pile p)  { ... }
  static void vider(Pile p)  { ... }
  static int valeur(Pile p)  { ... }
  static void supprimer(Pile p)  { ... }
  static void ajouter(int x, Pile p)  { ... }
}
```

Une petite subtilité : dans la méthode estPleine, on peut omettre le préfixe Pile pour la constante maxP puisqu'il s'agit implicitement de la classe courante. Rappelons sur un exemple l'utilisation pratique des méthodes statiques :

```
class Usage
{       // usage statique
  Pile p = new Pile();
  Pile.ajouter(7, p);
}
```

En fait, les six méthodes précédentes s'appliquent toujours à la pile courante, donnée en argument. Il est donc naturel d'en faire des méthodes d'objet. Dans cette optique, la classe Pile peut s'écrire :

```
class Pile
{
  static final int maxP = 10;

  int hauteur;
  int[] contenu;

  Pile()
  {
    hauteur = 0;
    contenu = new int[maxP];
  }

  boolean estVide()
  {
    return hauteur == 0;
  }

  boolean estPleine()
  {
    return hauteur == maxP;
  }

  void vider()
  {
    hauteur = 0;
  }

  int valeur()
  {
    return contenu[hauteur - 1];
  }
```

```
  void supprimer()
  {
    --hauteur;
  }

  void ajouter(int x)
  {
    contenu[hauteur++] = x;
  }
```

Cette fois, l'utilisation se fait ainsi :

```
class UsageNS
{    // usage non statique
  Pile p = new Pile();
  p.ajouter(7);
}
```

2 Les exceptions

Tout programmeur — même chevronné — fait des erreurs. Elles sont de deux catégories : les erreurs de syntaxe qui sont détectées et signalées à la compilation, et les erreurs dites de logique, qui provoquent, dans le meilleur des cas, un arrêt imprévu du programme avec un message souvent assez mystérieux. Par exemple, le programme

```
class Exceptions
{
  public static void main(String args[])
  {
    int[] a = {2, 3, 5};
    System.out.println(a[3]);
  }
}
```

provoque le message d'erreur

```
> Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
>         at Exceptions.main(Exceptions.java:6)
```

La première ligne indique la nature de l'erreur trouvée, qui est ArrayIndexOutOfBoundsException. Ce qui signifie en français que l'on est sorti des limites du tableau[1]. La deuxième indique qu'elle s'est produite dans la méthode main, à la ligne 6 du fichier. Voici un exemple plus instructif. Le programme

```
class Lecture
{
  static int lire(String s)
  {
    return Integer.parseInt(s);
  }
  public static void main(String[] args)
  {
    int i = lire(args[0]);
    System.out.println(i*i);
```

[1]Rappelons à ce propos qu'en JAVA, l'indexation des tableaux commence à 0...

```
    }
}
```

prend le paramètre passé sur la ligne de commande, le transforme en entier dans la méthode lire, et affiche son carré. Ainsi,

```
> java Lecture 7
```

affiche 49. En revanche,

```
> java Lecture m
```

provoque l'affichage que voici :

```
> Exception in thread "main" java.lang.NumberFormatException: m
>         at java.lang.Integer.parseInt(Integer.java:414)
>         at java.lang.Integer.parseInt(Integer.java:463)
>         at Lecture.lire(Lecture.java:3)
>         at Lecture.main(Lecture.java:6)
```

La première ligne indique le type d'erreur qui s'est produite, ici un problème de format de nombre. Ensuite, est affiché le contenu de la pile d'exécution : l'erreur est détectée dans une méthode parseInt de la classe Integer (à la ligne 414 du fichier Integer.java), méthode appelée par une autre méthode parseInt (il y en a effectivement deux) du même fichier. Cette méthode a été appelée à la ligne 3 de notre fichier, dans la méthode lire, qui a elle-même été appelée à la ligne 6 de notre fichier, dans la méthode main. On voit ici à la fois la pile d'exécution, et la « remontée » des erreurs, ce qui permet de les retracer.

Le langage Java, comme la plupart des langages de programmation évolués, permet de gérer les erreurs à travers un mécanisme spécifique appelé le mécanisme des *exceptions*. Une erreur qui se produit dans un programme n'aboutit alors pas à un arrêt immédiat, mais à la création et la propagation, à travers les méthodes concernées, d'un objet particulier, appelé une *exception*, créé pour la circonstance, et qui contient une information sur l'erreur. Le programmeur peut prévoir d'intercepter cet objet (le *capter*) et d'effectuer un traitement approprié. Si aucun traitement n'est prévu, la propagation continue et provoque finalement l'arrêt du programme. Dans l'exemple ci-dessus, l'objet créé est de la classe NumberFormatException. Ici, un traitement de cette erreur par le programme de l'utilisateur aurait été naturel.

Dans le cas d'une pile, trois erreurs peuvent se produire, selon que l'on veut
- dépiler une pile vide,
- demander la valeur d'une pile vide,
- empiler sur une pile pleine.

Une pile ne peut traiter directement ces erreurs : une pile est un outil, mis en place et exploité en général par une autre classe, et il n'y a pas moyen, pour une pile, d'agir sur le code de la classe utilisatrice qui lui est inconnu. Bien entendu, la classe qui utilise une pile devrait systématiquement tester qu'une pile n'est pas vide avant de dépiler, et qu'elle n'est pas pleine avant d'empiler. Le mécanisme des exceptions peut remédier à un éventuel oubli. La pile, convenablement programmée, crée une exception (on dit *lève* une exception) en cas d'erreur. La méthode appelante peut traiter l'exception (on dit qu'elle *capte* l'exception) ou décider de l'ignorer.

2.1 Syntaxe des exceptions

Une exception est un objet de la classe Exception ou de l'une de ses classes dérivées. Deux constructeurs existent dans la classe Exception, l'un sans argument, et un autre prenant en paramètre une chaîne de caractères qui constitue le message à transmettre. Ce message peut être récupéré par la méthode String getMessage(). Pour créer des exceptions spécifiques aux piles, il suffit de définir une classe propre. Par exemple

```
class PileException extends Exception
{
  PileException(String message)
  {
    super("Pile " + message + " !");
  }
}
```

Ici, super indique qu'on utilise le constructeur de la superclasse (Exception) avec la valeur indiquée du paramètre.

2.2 Levée d'exceptions

Créer et propager une exception (*lever* une exception) se fait par le mot-clé throw. Ainsi, pour lever une nouvelle exception de la classe PileException on écrit

```
throw new PileException("vide");
```

Vu la définition du constructeur, le message de l'exception est Pile vide !. Récrivons la méthode valeur de la classe Pile :

```
int valeur() throws PileException
{
  if (estVide())
    throw new PileException("vide");
  return contenu[hauteur - 1];
}
```

La méthode signale qu'elle est susceptible de lever des exceptions dans son en-tête, où elle indique de quel type sont les exceptions. Si la pile est vide, l'exception est levée. L'effet de la levée d'une exception est le suivant :

- sortie immédiate de la méthode en cours,
- recherche dans la pile des appels d'une méthode qui *capte* ce type d'exceptions.

S'il n'y a pas de capture, on sort du programme.

2.3 Capture des exceptions

La syntaxe de la capture est la suivante. L'ensemble des instructions susceptibles de lever une exception que l'on veut capter est soumise à une exécution contrôlée en l'entourant d'un expression `try{...}`. Cette expression est suivie d'une expression `catch(TypeExpression e) {...}` où la partie entre accolades contient le traitement à exécuter si l'exception du type indiquée s'est produite. Voici un exemple :

```
Pile p = new Pile();
...
try
{
  System.out.println(p.valeur());
}

catch (PileException e)
{
  System.out.println(e.getMessage());
}
```

En cas d'erreur dans le bloc try, le bloc catch est exécuté. Son argument, objet de la classe PileException, est créé lors de la levée de l'exception. La méthode getMessage est commune à toutes les exceptions ; elle retourne la chaîne de caractères formée lors de la création de l'exception. S'il n'y a pas d'erreur dans le bloc try, le bloc catch est ignoré.

Revenons à notre programme de lecture d'un entier. On peut entourer le calcul d'un bloc d'exécution contrôlée, comme par exemple :

```
class CLecture
{
  static int lire(String s)
  {
    return Integer.parseInt(s);
  }

  public static void main(String[] args)
  {
    try
    {
      int i = lire(args[0]);
      System.out.println(i*i);
    }

    catch(NumberFormatException e)
    {
      System.out.println("C'est raté!");
    }
  }
}
```

En cas de format incorrect, c'est le texte contenu dans le bloc catch qui est affiché, et l'exécution se termine normalement.

Dans une implantation d'une pile tenant compte des exceptions, les trois méthodes qui sont susceptibles d'en lever se récrivent comme suit :

```
int valeur() throws PileException
{
  if (estVide())
    throw new PileException("vide");
  return contenu[hauteur - 1];
}

void supprimer() throws PileException
{
  if (estVide())
    throw new PileException("vide");
  --hauteur;
}

void ajouter(int x) throws PileException
{
  if (estVide())
    throw new PileException("pleine");
  contenu[hauteur++] = x;
}
```

3 Une application : l'évaluation d'expressions arithmétiques

L'analyse et l'évaluation des expressions arithmétiques figurent parmi les problèmes algorithmiques les plus anciens qui se sont posés aux chercheurs — qui n'étaient alors pas encore appelés des informaticiens — lors du développement des premiers compilateurs. C'est de cette époque que date d'ailleurs l'introduction du concept de pile en tant que structure de données.

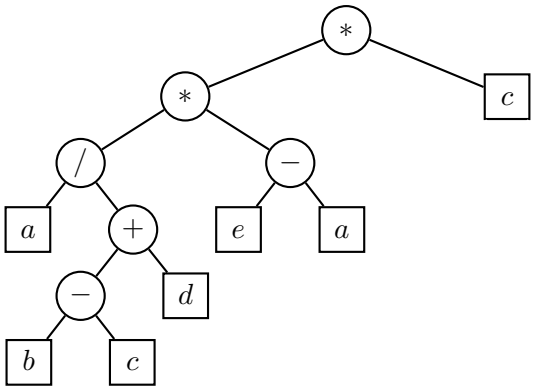


FIG. 3.7 – L'arbre d'une expression arithmétique.

Une expression arithmétique — comme $a/(b-c+d)*(e-a)*c$ — est en fait une expression structurée, contenant dans son parenthésage la description de son mode d'évaluation. Sa structure hiérarchique se décrit parfaitement dans un arbre, comme celui de la figure 3.7. La valeur de l'expression résulte d'une opération ultime, dont la nature est indiquée à la racine de l'arbre, appliquée au résultat de l'évaluation des deux sous-arbres.

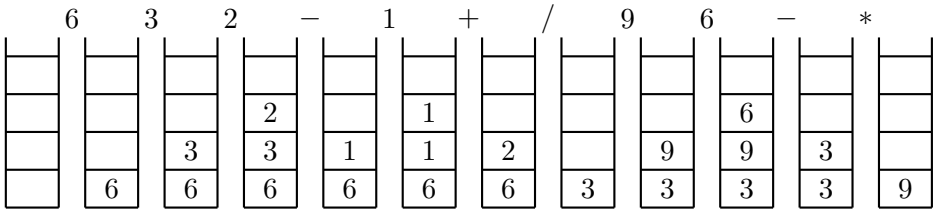


FIG. 3.8 – Evolution de la pile lors de l'évaluation d'une expression postfixée.

Avant de décrire un algorithme d'évaluation d'expressions arithmétiques données sous leur forme traditionnelle, nous considérons le problème plus simple de leur évaluation lorqu'elles sont présentées sous forme postfixée. Dans notre exemple, il s'agit de l'expression $abc-d+/ea-*c*$. Rappelons que dans la forme postfixée d'une expression, on écrit les deux opérandes avant l'opérateur. On voit assez facilement que les parenthèses ne sont alors plus nécessaires tant que chaque opérateur a une arité (un nombre d'opérandes) fixe. On ne pourrait pas écrire aussi simplement une expression qui comporterait des opérateurs d'arité variable puisque, par exemple, $ab-+$ peut être vu à la fois comme l'écriture de $+(a-b)$ et de $a+(-b)$. Nous verrons ultérieurement (Section 3.3 du chapitre IV) comment réaliser de façon simple le passage de la forme parenthésée à la forme postfixée.

L'évaluation d'une expression arithmétique sous forme postfixée se fait très simplement de la gauche vers la droite. Chaque opérateur prend pour argument les deux dernières valeurs calculées, et le résultat de l'évaluation se substitue à ces valeurs. Il est donc naturel d'utiliser

une pile pour y ranger les valeurs intermédiaires. La figure 3.8 illustre ce fonctionnement sur un exemple.

Pour l'implantation, on suppose que les opérandes de l'expression sont des entiers. On utilise une pile d'entiers pour ranger les valeurs intermédiaires, obtenues par évaluation des sous-expressions. L'expression est une suite de termes, chaque terme étant un entier ou un symbole d'opération. On décrira donc la donnée d'une expression par un tableau de termes, avec la définition suivante :

```
class Terme
{
  boolean estNombre;
  int nombre;
  char operateur;
}
```

L'évaluation d'une expression se ramène à l'évaluation successive des termes de l'expression, et l'évaluation d'un terme prend deux formes, en fonction de la nature du terme : s'il s'agit d'un nombre, il est simplement ajouté à la pile ; si au contraire il s'agit d'un opérateur (binaire dans notre spécification), on dépile deux entiers de la pile, on évalue la fonction correspondant à l'opérateur sur ces entiers, et on empile le résultat. Voici l'écriture en Java :

```
static Pile evaluer(Terme x, Pile p) throws PileException
{
  if (x.estNombre)
    p.ajouter(x.nombre);
  else
  {
    int a = p.valeur();
    p.supprimer();
    int b = p.valeur();
    p.supprimer();
    int c = evaluer(b, x.operateur, a);
    p.ajouter(c);
  }
  return p;
}
```

On aurait bien sûr pu déclarer cette méthode de type de retour void, mais notre façon de faire reflète mieux l'évolution de la pile. Le calcul numérique se fait dans la méthode suivante :

```
static int evaluer(int x, char operateur, int y)
{
  switch(operateur)
  {
    case '+': return x + y;
    case '-': return x - y;
    case '*': return x * y;
    case '/': return x / y;
  }
  return -1;
}
```

Rappelons que Java exige une valeur de retour dans tous les cas, c'est-à-dire même dans le cas « impossible » d'un opérateur dont la valeur serait différente des quatre cas énumérés. Enfin, la mise en œuvre de ces méthodes se fait par :

```
static int evaluer(Terme[] expression) throws PileException
{
  Pile p = new Pile();
  for (int i = 0; i < expression.length; i++)
    p = evaluer(expression[i], p);
  return p.valeur();
}
```

C'est quand on lance l'évaluation d'une expression que l'on capte les exceptions, comme dans

```
try
{
  int valeur = evaluer(expression);
  System.out.println("La valeur de l'expression est " + valeur);
}
catch (PileException e)
{
  System.out.println("Erreur dans l'evaluation de l'expression");
}
```

4 Files

Les opérations usuelles sur les files sont les suivantes :
- *créer* une file vide.
- *tester* si une file est vide.
- *ajouter* un élément à la file. Il est ajouté à la fin de la file.
- *trouver* l'élément qui est au début de la file.
- *supprimer* un élément de la file. C'est l'élément au début de la file qui est supprimé.

Trois fonctions définissent ces opérations. La méthode $\mathsf{ajouter}(x, f)$ permet d'ajouter l'élément x à la fin de la file f, la fonction $\mathsf{valeur}(f)$ retourne la valeur qui se trouve au début de la file. Enfin $\mathsf{supprimer}(f)$ enlève de la file l'élément qui se trouve à son début. Les deux dernières opérations demandent bien sûr que la file ne soit pas vide. Les méthodes sont liées par les équations suivantes.

$$\mathsf{valeur}(\mathsf{ajouter}(x, f)) = \begin{cases} \mathsf{valeur}(f) & \text{si } f \neq \mathsf{null} \\ x & \text{si } f = \mathsf{null} \end{cases}$$

$$\mathsf{supprimer}(\mathsf{ajouter}(x, f)) = \begin{cases} \mathsf{ajouter}(x, \mathsf{supprimer}(f)) & \text{si } f \neq \mathsf{null} \\ f & \text{si } f = \mathsf{null} \end{cases}$$

4.1 Implantation

Comme pour les piles, deux techniques sont utilisées pour implanter une file : les tableaux et les listes chaînées. Nous présentons d'abord la technique des tableaux, puis celle des listes chaînées.

Implantation par tableaux

Les éléments à ajouter s'insèrent en *fin* de file, les éléments sont extraits au *début* de la file. Chaque élément reçoit donc un numéro d'ordre quand il est inséré. C'est le numéro indiquant la *fin* de la file. Quand le numéro d'ordre d'un élément est égal au *début* de la file, cet élément est prêt à être extrait. La *fin* est incrémentée avant insertion, le *début* est incrémenté après

extraction. On optimise la gestion de la place en prenant les numéros modulo la taille du tableau qui contient les éléments. En d'autres termes, on utilise un tableau circulaire.

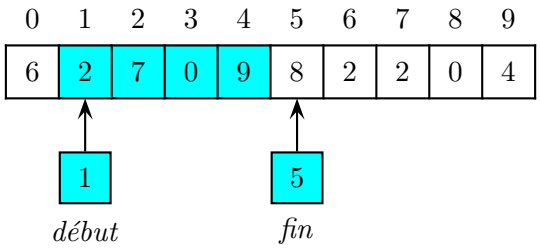


FIG. 4.9 – Une file implantée par un tableau et un deux entiers. Elle contient les entiers $2, 7, 0, 9$.

On considère des tableaux d'entiers, pour simplifier, et on fixe la taille à une valeur arbitraire constante. Les éléments présents dans la file ont pour ensemble de numéros l'intervalle semi-ouvert $[d, f[$, où d est le début, et f la fin de la file. Par exemple, la file de la figure 4.9 contient les valeurs $2, 7, 0, 9$, l'indice de début est 1, et l'indice de fin est 5. Cet intervalle doit être vu circulairement. Par exemple, la file de la figure 4.10 contient les valeurs $2, 0, 4, 6, 2, 7$, l'indice de début est 7, et l'indice de fin est 3.

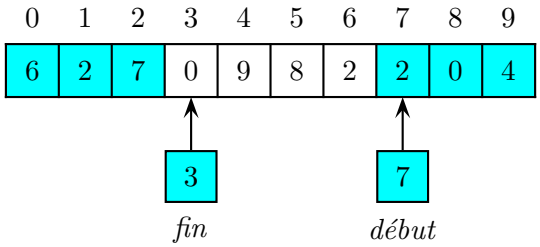


FIG. 4.10 – La file contient les entiers $2, 0, 4, 6, 2, 7$.

L'intervalle est vide si $d = f$, mais il n'est pas nécessaire que d soit nul (voir figure 4.11).

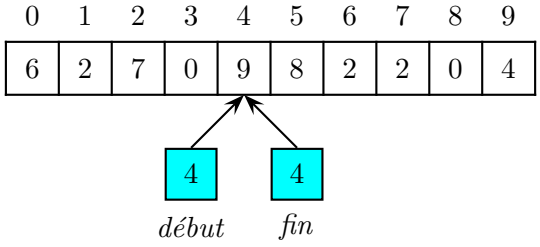


FIG. 4.11 – Une file vide.

Pour distinguer cette situation du cas d'une file pleine, on s'interdit d'utiliser la totalité du tableau, et on laisse toujours une entrée vide (cf. figure 4.12). Ainsi, la file est pleine si $f + 1 \equiv d \bmod n$, où n est la taille de la file. Une autre technique consisterait à utiliser un booléen pour indiquer que la file est pleine.

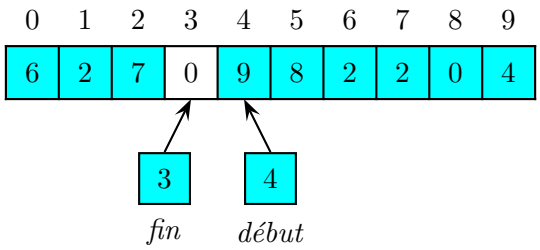


FIG. 4.12 – Une file pleine.

Passons à la réalisation en Java.

```
class File
{
  static final int maxF = 10; // assez grand ?

  int debut, fin;
  int[] contenu;

  File()
  {
    debut = 0;
    fin = 0;
    contenu = new int[maxF];
  }
}
```

À la création, une file contient donc un tableau de 10 entiers, tous initialisés à 0. Trois méthodes simples pour commencer.

```
boolean estVide()
{
  return debut == fin;
}

boolean estPleine()
{
  return (fin + 1) % maxF == debut;
}

void vider()
{
  debut = fin;
}
```

Noter que le test qu'une file est pleine se fait selon la formule indiquée. Les trois méthodes propres à la manipulation des files s'écrivent comme suit :

```
int valeur()
{
  return contenu[debut];
}

void supprimer()
{
  debut = (debut + 1) % maxF;
}
```

```
void ajouter(int x)
{
  fin = (fin + 1) % maxF;
  contenu[fin] = x;
}
```

On peut, bien entendu, lever une exception dans chacune de ces méthodes lorsque la file est vide ou pleine.

FIG. 4.13 – Adjonction de 11 à la file.

FIG. 4.14 – Suppression dans la file.

Implantation par listes chaînées

Une autre façon de gérer les files consiste à utiliser une liste chaînée pour représenter la suite des éléments dans la file. On utilise alors deux attributs qui contiennent les références de la première et de la dernière cellule. L'avantage de cette représentation est qu'une telle file n'est jamais pleine (sauf si l'on sature l'espace de travail de la machine !).

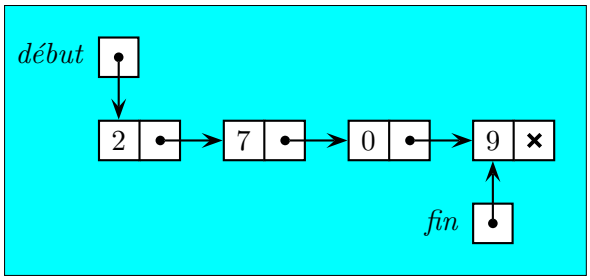


FIG. 4.15 – Une file implantée par une liste. Elle contient $2, 7, 0, 9$.

Voici comme se programme une telle classe.

```
class File
{
  Liste debut, fin;
```

```
  boolean estVide()
  {
    return debut == null;
  }

  boolean estPleine()
  {
    return false;
  }

  void vider()
  {
    debut = null;
  }

  int valeur()
  {
    return debut.contenu;
  }

  void supprimer()
  {
    debut = debut.suivant;
  }

  void ajouter(int x)
  {
    if (estVide())
      fin = debut = new Liste(x, null);
    else
      fin = fin.suivant = new Liste(x, null);
  }
}
```

La seule difficulté est l'ajout d'un élément. Il faut distinguer le cas de la file vide, où les deux attributs debut et fin doivent être modifiés (cf. figure 4.16).

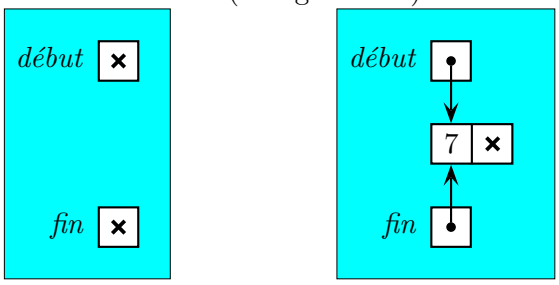


FIG. 4.16 – La file vide, et une file contenant un élément.

Le constructeur de file n'a pas besoin de figurer, puisque les attributs sont initialisés à null par défaut.

Choix de l'implantation

Nous avons présenté deux implantations des files, l'une par tableau, l'autre par liste chaînée. Le choix de l'implantation dépend de plusieurs paramètres. Par exemple, si l'on connaît une borne raisonnable à la taille maximale de la file, on peut choisir la réalisation par tableau. Il

est plus important de remarquer que les deux implantations se présentent de la même manière au programmeur, par le même jeu de méthodes de manipulation. On peut donc faire abstraction de l'implantation effective en considérant qu'une file est caractérisée par les méthodes de manipulation, et que la seule façon d'y accéder, c'est à travers ces méthodes. Les méthodes constituent l'*interface* entre la réalisation et l'utilisateur. La file devient un type de données *abstrait*. Java offre la possibilité de matérialiser ce concept par une structure syntaxique propre, qui sera étudiée en cours de majeure.

Chapitre IV

Arbres

Ce chapitre est consacré aux arbres, l'un des concepts algorithmiques les plus importants de l'informatique. Les arbres servent à représenter un ensemble de données structurées hiérarchiquement. Plusieurs notions distinctes se cachent en fait sous cette terminologie : arbres libres, arbres enracinés, arbres binaires, etc. Ces définitions sont précisées dans la section 1.

Nous présentons plusieurs applications des arbres : les arbres de décision, les files de priorité, le tri par tas et l'algorithme baptisé « union-find », qui s'applique dans une grande variété de situations. Les arbres binaires de recherche seront traités dans le chapitre suivant.

1 Définitions

Pour présenter les arbres de manière homogène, quelques termes empruntés aux graphes s'avèrent utiles. Nous présenterons donc les graphes, puis successivement, les arbres libres, les arbres enracinés et les arbres ordonnés.

1.1 Graphes

Un *graphe* $G = (S, A)$ est un couple formé d'un ensemble de *nœuds* S et d'un ensemble A d'*arcs*. L'ensemble A est une partie de $S \times S$. Les nœuds sont souvent représentés par des points dans le plan, et un arc $a = (s, t)$ par une ligne orientée joignant s à t. On dit que l'arc a part de s et va à t. Un chemin de s à t est une suite $(s = s_0, \ldots, s_n = t)$ de nœuds tels que, pour $1 \leqslant i \leqslant n$, (s_{i-1}, s_i) soit un arc. Le nœud s_0 est *l'origine* du chemin et le nœud s_n son *extrémité*. L'entier n est la *longueur* du chemin. C'est un entier positif ou nul. Un *circuit* est un chemin de longueur non nulle dont l'origine coïncide avec l'extrémité.

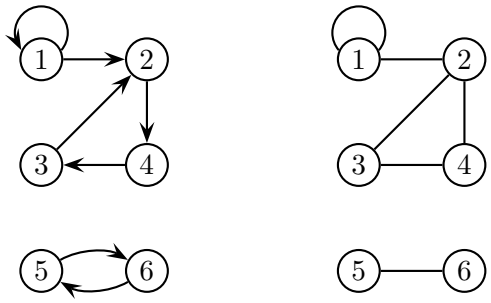


FIG. 1.1 – À gauche un graphe, à droite un graphe non orienté.

À côté de ces graphes, appelés aussi *graphes orientés* (« digraph » en anglais, pour « directed graph »), il existe la variante des graphes *non orientés*. Au lieu de couples de nœuds, on considère

des *paires* $\{s, t\}$ de nœuds. Un graphe non orienté est donné par un ensemble de ces paires, appelées *arêtes*. Les concepts de chemin et circuit se transposent sans peine à ce contexte.

Un chemin est *simple* si tous ses nœuds sont distincts. Un graphe est *connexe* si deux quelconques de ses nœuds sont reliés par un chemin.

1.2 Arbres libres

Dans la suite de chapitre, nous présentons des familles d'arbres de plus en plus contraints. La famille la plus générale est formée des arbres libres. Un *arbre libre* est un graphe non orienté non vide, connexe et sans circuit. La proposition suivante est laissée en exercice.

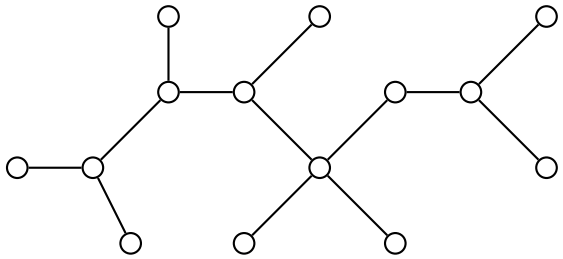


FIG. 1.2 – Un arbre « libre ».

Proposition 1 *Soit $G = (S, A)$ un graphe non orienté non vide. Les conditions suivantes sont équivalentes :*

(1) G est un arbre libre,

(2) Deux nœuds quelconques de S sont connectés par un chemin simple unique,

(3) G est connexe, mais ne l'est plus si l'on retire une arête quelconque,

(4) G est sans circuit, mais ne l'est plus si l'on ajoute une arête quelconque,

(5) G est connexe, et $\mathrm{Card}(A) = \mathrm{Card}(S) - 1$,

(6) G est sans circuit, et $\mathrm{Card}(A) = \mathrm{Card}(S) - 1$.

1.3 Arbres enracinés

Un *arbre enraciné* ou *arbre* (« rooted tree » en anglais) est un arbre libre muni d'un nœud distingué, appelé sa *racine*. Soit T un arbre de racine r. Pour tout nœud x, il existe un chemin simple unique de r à x. Tout nœud y sur ce chemin est un *ancêtre* de x, et x est un *descendant* de y. Le *sous-arbre* de racine x est l'arbre contenant tous les descendants de x. L'avant-dernier nœud y sur l'unique chemin reliant r à x est le *parent* (ou le *père* ou la *mère*) de x, et x est un *enfant* (ou un *fils* ou une *fille*) de y. L'*arité* d'un nœud est le nombre de ses enfants. Un nœud sans enfant est une *feuille*, un nœud d'arité strictement positive est appelé *nœud interne*. La *hauteur* d'un arbre T est la longueur maximale d'un chemin reliant sa racine à une feuille. Un arbre réduit à un seul nœud est de hauteur 0.

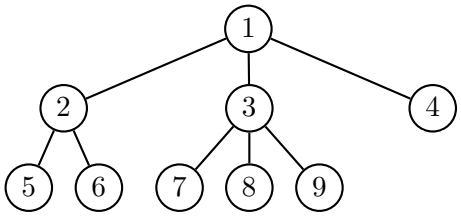


FIG. 1.3 – Un arbre « enraciné ».

Les arbres admettent aussi une définition récursive. Un arbre sur un ensemble fini de nœuds est un couple formé d'un nœud particulier, appelé sa racine, et d'une partition des nœuds restants en un ensemble d'arbres. Par exemple, l'arbre de la figure 1.3 correspond à la définition

$$T = (1, \{(2, \{(5), (6)\}), (3, \{(7), (8), (9)\}), (4)\})$$

Cette définition récursive est utile dans les preuves et dans la programmation. On montre ainsi facilement que si tout nœud interne d'un arbre est d'arité au moins 2, alors l'arbre a strictement plus de feuilles que de nœuds internes.

Une *forêt* est un ensemble d'arbres.

1.4 Arbres ordonnés

Un arbre *ordonné* est un arbre dans lequel l'ensemble des enfants de chaque nœud est totalement ordonné.

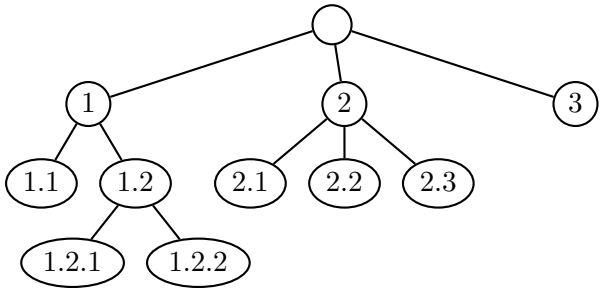


FIG. 1.4 – L'arbre ordonné de la table des matières d'un livre.

Par exemple, un livre, structuré en chapitres, sections, etc., se présente comme un arbre ordonné (voir figure 1.4). Les enfants d'un nœud d'un arbre ordonné sont souvent représentés, dans un programme, par une liste attachée au nœud. Un autre solution est d'associer, à chaque nœud, un tableau de fils. C'est une solution moins souple si le nombre de fils est destiné à changer. Enfin, on verra plus loin une autre représentation au moyen d'arbres binaires.

2 Union-Find, ou gestion des partitions

Comme premier exemple de l'emploi des arbres et des forêts, nous considérons un problème célèbre, et à ce jour pas encore entièrement résolu, appelé le problème *Union-Find*. Rappelons qu'une *partition* d'un ensemble E est un ensemble de parties non vides de E, deux à deux disjointes et dont la réunion est E. Étant donné une partition de l'ensemble $\{0, \ldots, n-1\}$, on veut résoudre les deux problèmes que voici :

- trouver la classe d'un élément (*find*)
- faire l'union de deux classes (*union*).

Nous donnons d'abord une solution du problème Union-Find, puis nous donnerons quelques exemples d'application.

2.1 Une solution du problème

En général, on part d'une partition où chaque classe est réduite à un singleton, puis on traite une suite de requêtes de l'un des deux types ci-dessus.

Avant de traiter ce problème, il faut imaginer la façon de représenter une partition. Une première solution consiste à représenter la partition par un tableau `classe`. Chaque classe est

identifiée par un entier par exemple, et classe[x] contient le numéro de la classe de l'élément x (cf. figure 2.5).

x	0	1	2	3	4	5	6	7	8	9
classe[x]	2	3	1	4	4	1	2	4	1	4

FIG. 2.5 – Tableau associé à la partition $\{\{2,5,8\},\{0,6\},\{1\},\{3,4,7,9\}\}$.

Trouver la classe d'un élément se fait en temps constant, mais fusionner deux classes prend un temps $O(n)$, puisqu'il faut parcourir tout le tableau pour repérer les éléments dont il faut changer la classe. Une deuxième solution, que nous détaillons maintenant, consiste à choisir un représentant dans chaque classe. Fusionner deux classes revient alors à changer de représentant pour les éléments de la classe fusionnée. Il apparaît avantageux de représenter la partition par une forêt. Chaque classe de la partition constitue un arbre de cette forêt. La racine de l'arbre est le représentant de sa classe. La figure 2.6 montre la forêt associée à une partition.

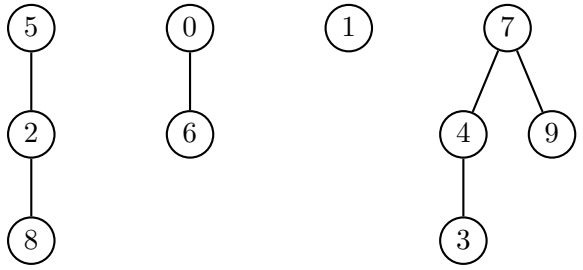


FIG. 2.6 – Forêt associée à la partition $\{\{2,5,8\},\{0,6\},\{1\},\{3,4,7,9\}\}$.

Une forêt est représentée par un tableau d'entiers pere (cf. Figure 2.7). Chaque nœud est représenté par un entier, et l'entier pere[x] est le père du nœud x. Une racine r n'a pas de parent. On convient que, dans ce cas, pere[r] = r.

x	0	1	2	3	4	5	6	7	8	9
pere[x]	0	1	5	4	7	5	0	7	2	7

FIG. 2.7 – Tableau associé à la forêt de la figure 2.6.

On suppose donc défini un tableau

```
int[] pere = new int[n];
```

Ce tableau est initialisé à l'identité par

```
static void initialisation()
{
  for (int i = 0; i < pere.length ; i++)
    pere[i] = i;
}
```

Chercher le représentant de la classe contenant un élément donné revient à trouver la racine de l'arbre contenant un nœud donné. Ceci se fait par la méthode suivante :

```
static int trouver(int x)
```

```
{
  while (x != pere[x])
    x = pere[x];
  return x;
}
```

L'union de deux arbres se réalise en ajoutant la racine de l'un des deux arbres comme nouveau fils à la racine de l'autre :

```
static void union(int x, int y)
{
  int r = trouver(x);
  int s = trouver(y);
  if (r != s)
    pere[r] = s;
}
```

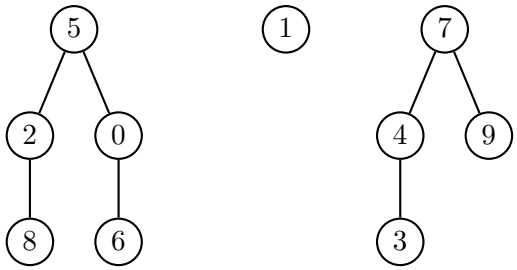


FIG. 2.8 – La forêt de la figure 2.6 après l'union pondérée de 5 et 0.

Il n'est pas difficile de voir que chacune de ces deux méthodes est de complexité $O(h)$, où h est la hauteur l'arbre (la plus grande des hauteurs des deux arbres). En fait, on peut améliorer l'efficacité de l'algorithme par la règle suivante (voir figure 2.8) :

Règle. *Lors de l'union de deux arbres, la racine de l'arbre de moindre taille devient fils de la racine de l'arbre de plus grande taille.*

Pour mettre en œuvre cette stratégie, on utilise un tableau supplémentaire qui mémorise la taille des arbres, qui doit être initialisé à 1 :

```
int[] taille = new int[n];
```

La nouvelle méthode d'union s'écrit alors :

```
static void unionPondérée(int x, int y)
{
  int r = trouver(x);
  int s = trouver(y);
  if (r == s)
    return;
  if (taille[r] > taille[s])
  {
    pere[s] = r;
    taille[r] += taille[s];
  }
  else
  {
```

```
      pere[r] = s;
      taille[s] += taille[r];
    }
  }
```

L'intérêt de cette méthode vient de l'observation suivante :

Lemme 2 *La hauteur d'un arbre à n nœuds créé par union pondérée est au plus* $1 + \lfloor \log_2 n \rfloor$.

Preuve. Par récurrence sur n. Pour $n = 1$, il n'y a rien à prouver. Si un arbre est obtenu par union pondérée d'un arbre à m nœuds et d'un arbre à $n - m$ nœuds, avec $1 \leqslant m \leqslant n/2$, sa hauteur est majorée par

$$\max(1 + \lfloor \log_2(n-m) \rfloor, 2 + \lfloor \log_2 m \rfloor)\,.$$

Comme $\log_2 m \leqslant \log_2(n/2) = \log_2 n - 1$, cette valeur est majorée par $1 + \lfloor \log_2 n \rfloor$. □

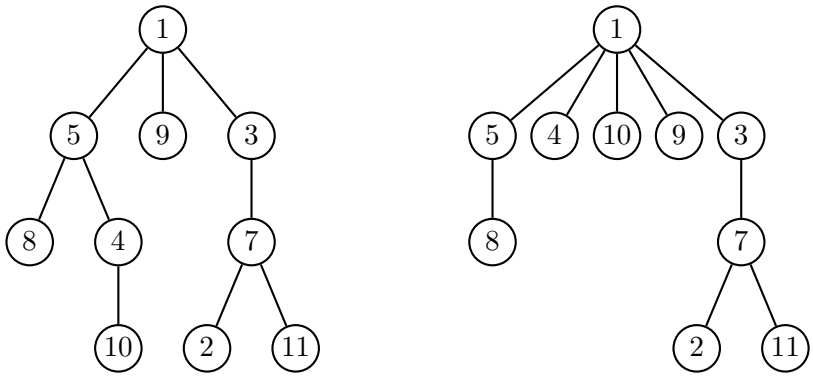


FIG. 2.9 – « Trouver » 10 avec compression du chemin.

Une deuxième stratégie, appliquée cette fois-ci lors de la méthode trouver permet une nouvelle amélioration considérable de la complexité. Elle est basée sur la règle de compression de chemins suivante :

Règle. *Après être remonté du nœud x à sa racine r, on refait le parcours en faisant de chaque nœud rencontré un fils de r.*

La figure 2.9 montre la transformation d'un arbre par une compression de chemin. Chacun des nœuds 10 et 4 devient fils de 1. L'implantation de cette règle se fait simplement.

```
static int trouverAvecCompression(int x)
{
  int r = trouver(x);
  while (x != r)
  {
    int y = pere[x];
    pere[x] = r;
    x = y;
  }
  return r;
}
```

L'ensemble des deux stratégies permet d'obtenir une complexité presque linéaire.

Théorème 3 (Tarjan) *Avec l'union pondérée et la compression des chemins, une suite de* $n-1$ « unions » *et de* m « trouver » *(*$m \geqslant n$*) se réalise en temps* $O(n+m\alpha(n,m))$*, où* α *est l'inverse d'une sorte de fonction d'Ackermann.*

En fait, on a $\alpha(n,m) \leqslant 2$ pour $m \geqslant n$ et $n < 2^{65536}$ et par conséquent, l'algorithme précédent se comporte, d'un point de vue pratique, comme un algorithme linéaire en $n+m$. Pourtant, Tarjan a montré qu'il *n'est pas* linéaire et on ne connaît pas à ce jour d'algorithme linéaire.

2.2 Applications de l'algorithme Union-Find

Un premier exemple est la construction d'un *arbre couvrant* un graphe donné. Au départ, chaque nœud constitue à lui seul un arbre. On prend ensuite les arêtes, et on fusionne les arbres contenant les extrémités de l'arête si ces extrémités appartiennent à des arbres différents.

Un second exemple concerne les problèmes de connexion dans un réseau. Voici un exemple de tel problème. Huit ordinateurs sont connectés à travers un réseau. L'ordinateur 1 est connecté au 3, le 2 au 3, le 5 au 4, le 6 au 3, le 7 au 5, le 1 au 6 et le 7 au 8. Est-ce que les ordinateurs 4 et 6 peuvent communiquer à travers le réseau ? Certes, il n'est pas très difficile de résoudre ce problème à la main, mais imaginez la même question pour un réseau dont la taille serait de l'ordre de plusieurs millions. Comment résoudre ce problème efficacement ? C'est la même solution que précédemment ! On considère le réseau comme un graphe dont les nœuds sont les ordinateurs. Au départ, chaque nœud constitue à lui seul un arbre. On prend ensuite les arêtes (i.e. les connexions entre deux ordinateurs), et on fusionne les arbres contenant les extrémités de l'arête si ces extrémités appartiennent à des arbres différents.

3 Arbres binaires

La notion d'arbre binaire est assez différente des définitions précédentes. Un *arbre binaire* sur un ensemble fini de nœuds est soit vide, soit l'union disjointe d'un nœud appelé sa *racine*, d'un arbre binaire appelé *sous-arbre gauche*, et d'un arbre binaire appelé *sous-arbre droit*. Il est utile de représenter un arbre binaire non vide sous la forme d'un triplet $A = (A_g, r, A_d)$.

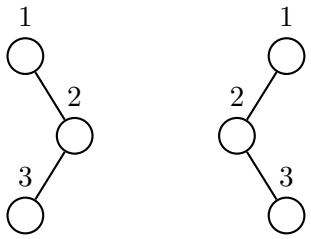


FIG. 3.10 – Deux arbres binaires différents.

Par exemple, l'arbre binaire sur la gauche de la figure 3.10 est $(\emptyset, 1, ((\emptyset, 3, \emptyset), 2, \emptyset))$, alors que l'arbre sur la droite de la figure 3.10 est $((\emptyset, 2, (\emptyset, 3, \emptyset)), 1, \emptyset)$. Cet exemple montre qu'un arbre binaire n'est pas simplement un arbre ordonné dont tous les nœuds sont d'arité au plus 2.

La *distance* d'un nœud x à la racine ou la *profondeur* de x est égale à la longueur du chemin de la racine à x. La *hauteur* d'un arbre binaire est égale à la plus grande des distances des feuilles à la racine.

Proposition 4 *Soit* A *un arbre binaire à* n *nœuds, de hauteur* h*. Alors* $h+1 \geqslant \log_2(n+1)$*.*

Preuve. Il y a au plus 2^i nœuds à distance i, donc $n \leqslant 2^{h+1} - 1$. □

Un arbre binaire est *complet* si tout nœud a 0 ou 2 fils.

Proposition 5 *Dans un arbre binaire complet, le nombre de feuilles est égal au nombre de nœuds internes, plus* 1.

Preuve. Notons, pour simplifier, $f(A)$ le nombre de feuilles et $n(A)$ le nombre de nœuds internes de l'arbre binaire complet A. Il s'agit de montrer que $f(A) = n(A) + 1$.

Le résultat est vrai pour l'arbre binaire de hauteur 0. Considérons un arbre binaire complet $A = (A_g, r, A_d)$. Les feuilles de A sont celles de A_g et de A_d et donc $f(A) = f(A_g) + f(A_d)$. Les nœuds internes de A sont ceux de A_g, ceux de A_d et la racine, et donc $n(A) = n(A_g) + n(A_d) + 1$. Comme A_g et A_d sont des arbres complets de hauteur inférieure à celle de A, la récurrence s'applique et on a $f(A_g) = n(A_g) + 1$ et $f(A_d) = n(A_d) + 1$. On obtient finalement $f(A) = f(A_g) + f(A_d) = (n(A_g) + 1) + (n(A_d) + 1) = n(A) + 1$. □

On montre aussi que, dans un arbre binaire complet, il y a un nombre pair de nœuds à chaque niveau, sauf au niveau de la racine.

3.1 Compter les arbres binaires

La figure 3.11 montre les arbres binaires ayant 1, 2, 3 et 4 nœuds.

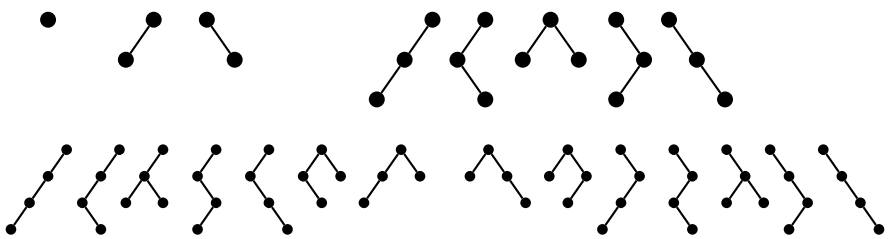


FIG. 3.11 – Les premiers arbres binaires.

Notons b_n le nombre d'arbres à n nœuds. On a donc $b_0 = b_1 = 1$, $b_2 = 2$, $b_3 = 5$, $b_4 = 14$. Comme tout arbre A non vide s'écrit de manière unique sous forme d'un triplet (A_g, r, A_d), on a pour $n \geqslant 1$ la formule

$$b_n = \sum_{i=0}^{n-1} b_i b_{n-i-1}$$

La série génératrice $B(x) = \sum_{n \geqslant 0} b_n x^n$ vérifie donc l'équation

$$xB^2(x) - B(x) + 1 = 0\,.$$

Comme les b_n sont positifs, la résolution de cette équation donne

$$b_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{n!(n+1)!}$$

Les nombres b_n sont connus comme les *nombres de Catalan*. L'expression donne aussi $b_n \sim \pi^{-1/2} 4^n n^{-3/2} + O(4^n n^{-5/2})$.

3.2 Arbres binaires et mots

Nous présentons quelques concepts sur les mots qui servent à plusieurs reprises. D'abord, ils mettent en évidence des liens entre les parcours d'arbres binaires et certains ordres. Ensuite, ils seront employés dans des algorithmes de compression.

Mots

Un *alphabet* est un ensemble de *lettres*, comme $\{0, 1\}$ ou $\{a, b, c, d, r\}$. Un *mot* est une suite de lettres, comme 0110 ou *abracadabra*. La *longueur* d'un mot u, notée $|u|$, est le nombre de lettres de u : ainsi, $|0110| = 4$ et $|abracadabra| = 11$. Le mot vide, de longueur 0, est noté ε. Etant donnés deux mots, le mot obtenu par *concaténation* est le mot formé des deux mots juxtaposés. Le produit de concaténation est noté comme un produit. Si $u = abra$ et $v = cadabra$, alors $uv = abracadabra$. Un mot p est *préfixe* (propre) d'un mot u s'il existe un mot v (non vide) tel que $u = pv$. Ainsi, ε, abr, $abrac$ sont des préfixes propres de $abraca$. Un ensemble de mots P est *préfixiel* si tout préfixe d'un mot de P est lui-même dans P. Par exemple, les ensembles $\{\varepsilon, 1, 10, 11\}$ et $\{\varepsilon, 0, 00, 01, 000, 001\}$ sont préfixiels.

Ordres sur les mots

Un ordre total sur l'alphabet s'étend en un ordre total sur l'ensemble des mots de multiples manières. Nous considérons deux ordres, l'ordre lexicographique et l'ordre des mots croisés (« radix order » ou « shortlex » en anglais).

L'ordre lexicographique, ou ordre du dictionnaire, est défini par $u <_{\text{lex}} v$ si seulement si u est préfixe de v ou u et v peuvent s'écrire $u = pau'$, $v = pbv'$, où p est un mot, a et b sont des lettres, et $a < b$. L'ordre des mots croisés est défini par $u <_{\text{mc}} v$ si et seulement si $|u| < |v|$ ou $|u| = |v|$ et $u <_{\text{lex}} v$. Par exemple, on a

$$bar <_{\text{mc}} car <_{\text{mc}} barda <_{\text{mc}} radar <_{\text{mc}} abracadabra$$

Codage des arbres binaires

Chaque arête (p, f) d'un arbre A binaire est étiquetée par 0 si f est fils gauche de p, et par 1 si f est fils droit. L'étiquette du chemin qui mène de la racine à un nœud est le mot formé des étiquettes de ses arêtes. Le *code* de l'arbre A est l'ensemble des étiquettes des chemins issus de la racine. Cet ensemble est clairement préfixiel, et réciproquement, tout ensemble fini préfixiel de mots formés de 0 et 1 est le code d'un arbre binaire. La correspondance est, de plus, bijective.

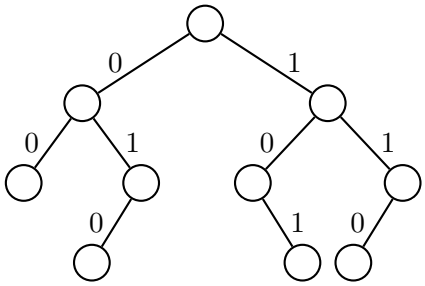


FIG. 3.12 – Le code de l'arbre est $\{\varepsilon, 0, 1, 00, 01, 10, 11, 010, 101, 110\}$.

Le code $c(A)$ d'un arbre binaire A se définit d'ailleurs simplement par récurrence : on a $c(\emptyset) = \{\varepsilon\}$ et si $A = (A_g, r, A_d)$, alors $c(A) = 0c(A_g) \cup \{\varepsilon\} \cup 1c(A_d)$.

3.3 Parcours d'arbre

Un *parcours d'arbre* est une énumération des nœuds de l'arbre. Chaque parcours définit un ordre sur les nœuds, déterminé par leur ordre d'apparition dans cette énumération.

On distingue les parcours de gauche à droite, et les parcours de droite à gauche. Dans un parcours de gauche à droite, le fils gauche d'un nœud précède le fils droit (et vice-versa pour un parcours de droite à gauche). Ensuite, on distingue les parcours en profondeur et en largeur.

Le *parcours en largeur* énumère les nœuds niveau par niveau. Ainsi, le parcours en largeur de l'arbre 3.13 donne la séquence $a, b, c, d, e, f, g, h, i, k$. On remarque que les codes des nœuds correspondants, ε, 0, 1, 00, 01, 10, 11, 010, 101, 110, sont en ordre croissant pour l'ordre des mots croisés. C'est en fait une règle générale.

Règle. *L'ordre du parcours en largeur correspond à l'ordre des mots croisés sur le code de l'arbre.*

On définit trois *parcours en profondeur* privilégiés qui sont

- le *parcours préfixe* : tout nœud est suivi des nœuds de son sous-arbre gauche puis des nœuds de son sous-arbre droit, en abrégé NGD (Nœud, Gauche, Droite).
- le *parcours infixe* : tout nœud est précédé des nœuds de son sous-arbre gauche et suivi des nœuds de son sous-arbre droit, en abrégé GND (Gauche, Nœud, Droite).
- le *parcours suffixe*, ou *postfixe* : tout nœud est précédé des nœuds de son sous-arbre gauche puis des nœuds de son sous-arbre droit, en abrégé GDN (Gauche, Droite, Nœud).

Les ordres correspondant sont appelés *ordres préfixe, infixe et suffixe*. Considérons l'arbre de la figure 3.13.

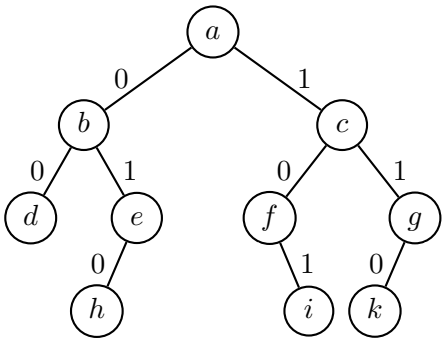


FIG. 3.13 – Un arbre binaire. Les nœuds sont nommés par des lettres.

Le parcours préfixe donne les nœuds dans l'ordre $a, b, d, e, h, c, f, i, g, k$, le parcours infixe donne la suite $d, b, h, e, a, f, i, c, k, g$ et le parcours suffixe donne $d, h, e, b, i, f, k, g, c, a$. De manière formelle, les parcours préfixe (infixe, suffixe) sont définis comme suit. Si A est l'arbre vide, alors $\mathrm{pref}(A) = \mathrm{inf}(A) = \mathrm{suff}(A) = \varepsilon$; si $A = (A_g, r, A_d)$, et $e(r)$ est le nom de r, alors

$$\begin{aligned} \mathrm{pref}(A) &= e(r)\mathrm{pref}(\mathrm{A_g})\mathrm{pref}(\mathrm{A_d}) \\ \mathrm{inf}(A) &= \mathrm{inf}(\mathrm{A_g})e(r)\mathrm{inf}(\mathrm{A_d}) \\ \mathrm{suff}(A) &= \mathrm{suff}(\mathrm{A_g})\mathrm{suff}(\mathrm{A_d})e(r) \end{aligned}$$

Règle. *Le parcours préfixe d'un arbre correspond à l'ordre lexicographique sur le code de l'arbre. Le parcours suffixe correspond à l'opposé de l'ordre lexicographique si l'on convient que* $1 < 0$.

Qu'on se rassure, il y a aussi une interprétation pour le parcours infixe, mais elle est un peu plus astucieuse ! À chaque nœud x de l'arbre, on associe un nombre formé du code du chemin menant à x, suivi de 1. Ce code complété est interprété comme la partie fractionnaire d'un nombre entre 0 et 1, écrit en binaire. Pour l'arbre de la figure 3.13, les nombres obtenus sont

donnés dans la table suivante

a	.1	=	1/2
b	.01	=	1/4
c	.11	=	3/4
d	.001	=	1/8
e	.011	=	3/8
f	.101	=	5/8
g	.111	=	7/8
h	.0101	=	5/16
i	.1011	=	11/16
k	.1101	=	13/16

L'ordre induit sur les mots est appelé l'*ordre fractionnaire*.

Règle. *L'ordre infixe correspond à l'ordre fractionnaire sur le code de l'arbre.*

La programmation de ces parcours sera donnée au chapitre V.

3.4 Une borne inférieure pour les tris par comparaisons

Voici une application surprenante des arbres à l'analyse de complexité. Il existe de nombreux algorithmes de tri, certains dont la complexité dans le pire des cas est en $O(n^2)$ comme les tris par sélection, par insertion ou à bulles, d'autres en $O(n^{3/2})$ comme le tri Shell, et d'autres en $O(n \log n)$ comme le tri fusion ou le tri par tas, que nous verrons page 89. On peut se demander s'il est possible de trouver un algorithme de tri de complexité inférieure dans le pire des cas.

Avant de résoudre cette question, il faut bien préciser le modèle de calcul que l'on considère. Un *tri par comparaison* est un algorithme qui trie en n'utilisant que des comparaisons. On peut supposer que les éléments à trier sont deux-à-deux distincts. Le modèle utilisé pour représenter un calcul est un *arbre de décision*. Chaque comparaison entre éléments d'une séquence à trier est représentée par un nœud interne de l'arbre. Chaque nœud pose une question. Le fils gauche correspond à une réponse négative, le fils droit à une réponse positive (figure 3.14).

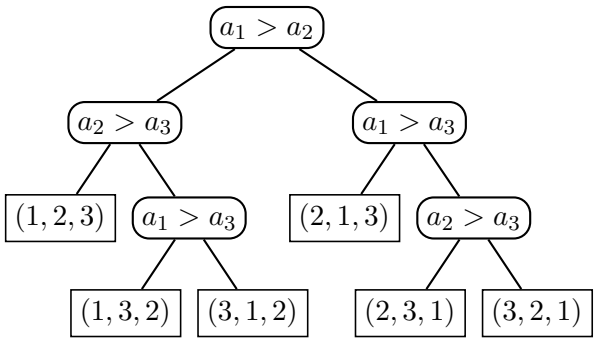


FIG. 3.14 – Exemple d'arbre de décision pour le tri.

Les feuilles représentent les permutations des éléments à effectuer pour obtenir la séquence triée. Le nombre de comparaisons à effectuer pour déterminer cette permutation est égale à la longueur du chemin de la racine à la feuille.

Nous prouvons ici :

Théorème 6 *Tout algorithme de tri par comparaison effectue $\Omega(n \log n)$ comparaisons dans le pire des cas pour trier une suite de n éléments.*

Preuve. Tout arbre de décision pour trier n éléments a $n!$ feuilles, représentant toutes les permutations possibles. La hauteur de l'arbre est donc minorée par $\log(n!)$. Or $\log(n!) = O(n \log n)$ par la formule de Stirling. □

Deux précisions pour clore cette parenthèse sur les tris. Tout d'abord, le résultat précédent n'est plus garanti si l'on change de modèle. Supposons par exemple que l'on veuille classer les notes (des entiers entre 0 et 20) provenant d'un paquet de 400 copies. La façon la plus simple et la plus efficace consiste à utiliser un tableau `T` de taille 21, dont chaque entrée `T[i]` sert à compter les notes égales à i. Il suffit alors de lire les notes une par une et d'incrémenter le compteur correspondant. Une fois ce travail accompli, le tri est terminé : il y a `T[0]` notes égales à 0, suivi de `T[1]` notes égales à 1, etc. Cet algorithme est manifestement linéaire et ne fait aucune comparaison ! Pourtant, il ne contredit pas notre résultat. Nous avons en effet utilisé implicitement une information supplémentaire : toutes les valeurs à trier appartiennent à l'intervalle $[0, 20]$. Cet exemple montre qu'il faut bien réfléchir aux conditions particulières avant de choisir un algorithme.

Seconde remarque, on constate expérimentalement que l'algorithme de tri rapide (QuickSort), dont la complexité dans le pire des cas est en $O(n^2)$, est le plus efficace en pratique. Comment est-ce possible ? Tout simplement parce que notre résultat ne concerne que la complexité dans le pire des cas. Or QuickSort est un algorithme en $O(n \log n)$ en moyenne.

4 Files de priorité

Nous avons déjà rencontré les files d'attente. Les files de priorité sont des files d'attente où les éléments ont un niveau de priorité. Le passage devant le guichet, ou le traitement de l'élément, se fait en fonction de son niveau de priorité. L'implantation d'une file de priorité est un exemple d'utilisation d'arbre, et c'est pourquoi elle trouve naturellement sa place ici.

De manière plus formelle, une *file de priorité* est un type abstrait de données opérant sur un ensemble ordonné, et muni des opérations suivantes :

- trouver le plus grand élément
- insérer un élément
- retirer le plus grand élément

Bien sûr, on peut remplacer « le plus grand élément » par « le plus petit élément » en prenant l'ordre opposé. Plusieurs implantations d'une file de priorité sont envisageables : par tableau ou par liste, ordonnés ou non. Nous allons utiliser des *tas*. La table IV.1 présente la complexité des opérations des files de priorités selon la structure de données choisie.

Implantation	Trouver max	Insérer	Retirer max
Tableau non ordonné	$O(n)$	$O(1)$	$O(n)$
Liste non ordonnée	$O(n)$	$O(1)$	$O(1)^{(*)}$
Tableau ordonné	$O(1)$	$O(n)$	$O(1)$
Liste ordonnée	$O(1)$	$O(n)$	$O(1)$
Tas	$O(1)$	$O(\log n)$	$O(\log n)$

TAB. IV.1 – Complexité des implantations de files de priorité.

*Dans cette table, le coût de la suppression dans une liste non ordonnée est calculé en supposant l'élément déjà trouvé.

4.1 Tas

Un arbre binaire est *tassé* si son code est un segment initial pour l'ordre des mots croisés. En d'autres termes, dans un tel arbre, tous les niveaux sont entièrement remplis à l'exception peut-être du dernier niveau, et ce dernier niveau est rempli « à gauche ». La figure 4.15 montre un arbre tassé.

Proposition 7 *La hauteur d'un arbre tassé à n nœuds est $\lfloor \log_2 n \rfloor$.*

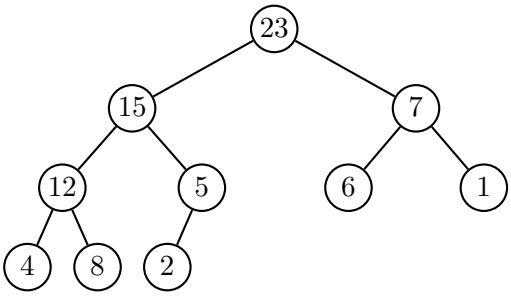


FIG. 4.15 – Un arbre tassé.

Un *tas* (en anglais « heap ») est un arbre binaire tassé tel que le contenu de chaque nœud soit supérieur ou égal à celui de ses fils. Ceci entraîne, par transitivité, que le contenu de chaque nœud est supérieur ou égal à celui de ses descendants.

L'arbre de la figure 4.15 est un tas.

4.2 Implantation d'un tas

Un tas s'implante facilement à l'aide d'un simple tableau. Les nœuds d'un arbre tassé sont numérotés en largeur, de gauche à droite. Ces numéros sont des indices dans un tableau (cf figure 4.16).

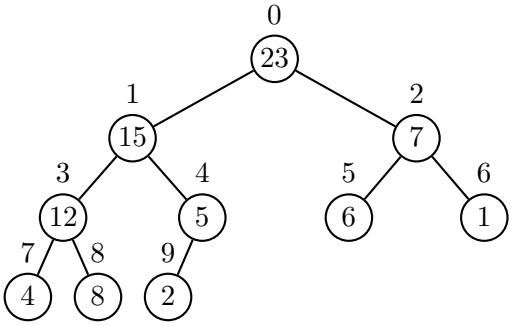


FIG. 4.16 – Un arbre tassé, avec la numérotation de ses nœuds.

L'élément d'indice i du tableau est le contenu du nœud de numéro i. Dans notre exemple, le tableau est :

i	0	1	2	3	4	5	6	7	8	9
a_i	23	15	7	12	5	6	1	4	8	2

Le fait que l'arbre soit tassé conduit à un calcul très simple des relations de filiation dans l'arbre

(n est le nombre de ses nœuds) :

racine	:	nœud 0
parent du nœud i	:	nœud $\lfloor (i-1)/2 \rfloor$
fils gauche du nœud i	:	nœud $2i+1$
fils droit du nœud i	:	nœud $2i+2$
nœud i est une feuille	:	$2i+1 \geqslant n$
nœud i a un fils droit	:	$2i+2 < n$

L'*insertion* d'un nouvel élément v se fait en deux temps : d'abord, l'élément est ajouté comme contenu d'un nouveau nœud à la fin du dernier niveau de l'arbre, pour que l'arbre reste tassé. Ensuite, le contenu de ce nœud, soit v, est comparé au contenu de son père. Tant que le contenu du père est plus petit que v, le contenu du père est descendu vers le fils. À la fin, on remplace par v le dernier contenu abaissé (voir figure 4.17).

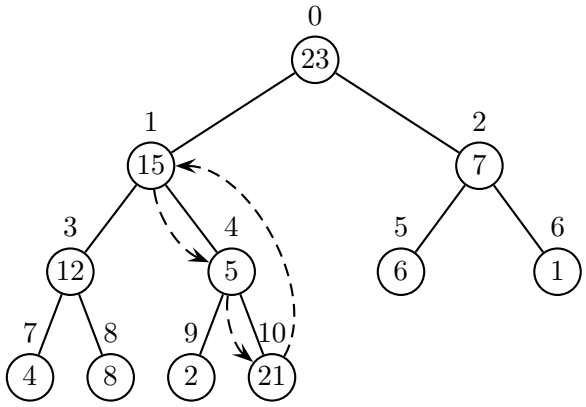


FIG. 4.17 – Un tas, avec remontée de la valeur 21 après insertion.

La *suppression* se fait de manière similaire. D'abord, le contenu du nœud le plus à droite du dernier niveau est transféré vers la racine, et ce nœud est supprimé. Ceci garantit que l'arbre reste tassé. Ensuite, le contenu v de la racine est comparé à la plus grande des valeurs de ses fils (s'il en a). Si cette valeur est supérieure à v, elle est remontée et remplace le contenu du père. On continue ensuite avec le fils. Par exemple, la suppression de 16 dans l'arbre de gauche de la figure 4.18 conduit d'abord à l'arbre de droite de cette figure, et enfin au tas de la figure 4.19

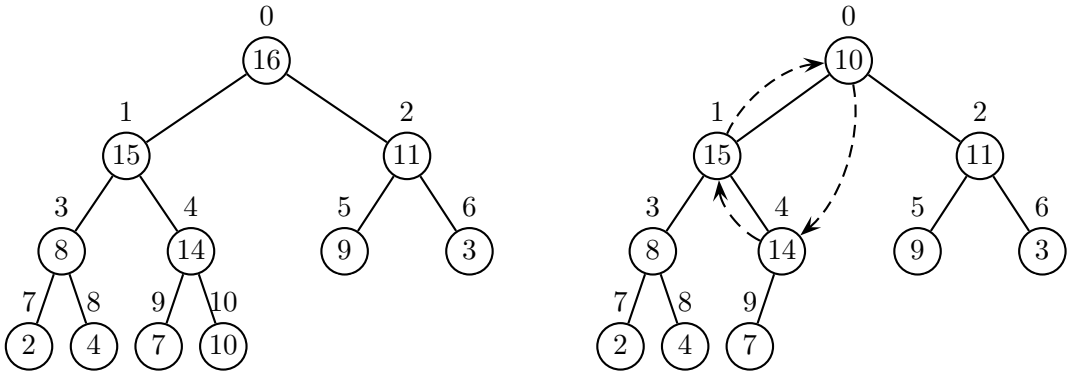


FIG. 4.18 – Un tas, et la circulation des valeurs pendant la suppression de 16.

La complexité de chacune de ces opérations est majorée par la hauteur de l'arbre qui est, elle, logarithmique en la taille.

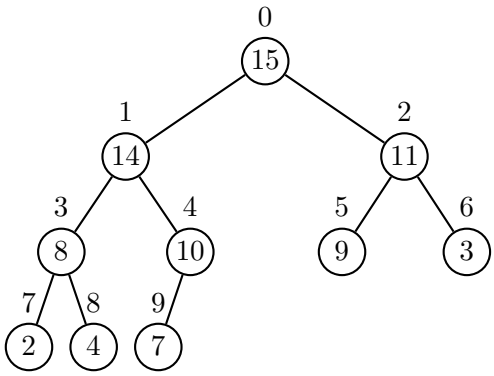


FIG. 4.19 – Le tas de la figure 4.18 après suppression de 16.

Un tas est naturellement présenté comme une classe, fournissant les trois méthodes maximum(), inserer(), supprimer(). On range les données dans un tableau interne. Un constructeur permet de faire l'initialisation nécessaire. Voici le squelette :

```
class Tas
{
  int[] a;
  int nTas = 0;

  Tas(int n)
  {
    nTas = 0;
    a = new int[n];
  }

  int maximum()
  {
    return a[0];
  }

  void ajouter(int v) {...}
  void supprimer() {...}
```

Avant de donner l'implantation finale, voici une première version à l'aide de méthodes qui reflètent les opérations de base.

```
void ajouter(int v)
{
  int i = nTas;
  ++nTas;
  while (!estRacine(i) && cle(parent(i)) < v)
  {
    cle(i) = cle(parent(i));
    i = parent(i);
  }
  cle(i) = v;
}
```

De même pour la suppression :

```
void supprimer()
{
```

```
    --nTas;
    cle(0) = cle(nTas);
    int v = cle(0);
    int i = 0;
    while (!estFeuille(i))
    {
      int j = filsG(i);
      if (existeFilsD(i) && cle(filsD(i)) > cle(filsG(i)))
        j = filsD(i);
      if (v >= cle(j)) break;
      cle(i) = cle(j);
      i = j;
    }
    cle(i) = v;
  }
```

Il ne reste plus qu'à remplacer ce pseudo-code par les instructions opérant directement sur le tableau.

```
  void ajouter(int v)
  {
    int i = nTas;
    ++nTas;
    while (i > 0 && a[(i-1)/2] <= v)
    {
      a[i] = a[(i-1)/2];
      i = (i-1)/2;
    }
    a[i] = v;
  }
```

On notera que, puisque la hauteur d'un tas à n nœuds est $\lfloor \log_2 n \rfloor$, le nombre de comparaisons utilisée par la méthode ajouter est en $O(\log n)$.

```
  void supprimer()
  {
    int v = a[0] = a[--nTas];
    int i = 0;
    while (2*i + 1 < nTas)
    {
      int j = 2*i + 1;
      if (j +1 < nTas && a[j+1] > a[j])
        ++j;
      if (v >= a[j])
        break;
      a[i] = a[j];
      i = j;
    }
    a[i] = v;
  }
}
```

Là encore, la complexité de la méthode supprimer est en $O(\log n)$.

On peut se servir d'un tas pour trier : on insère les éléments à trier dans le tas, puis on les extrait un par un. Ceci donne une méthode de tri appelée *tri par tas* (« heapsort » en anglais).

```
static int[] triParTas(int[] a)
{
  int n = a.length;
  Tas t = new Tas(n);
  for (int i = 0; i < n; i++)
    t.ajouter(a[i]);
  for (int i = n - 1; i >= 0; --i)
  {
    int v = t.maximum();
    t.supprimer();
    a[i] = v;
  }
  return a;
}
```

La complexité de ce tri est, dans le pire des cas, en $O(n \log n)$. En effet, on appelle n fois chacune des méthodes ajouter et supprimer.

4.3 Arbres de sélection

Une variante des arbres tassés sert à la *fusion* de listes ordonnées. On est en présence de k suites de nombres ordonnées de façon décroissante (ou croissante), et on veut les fusionner en une seule suite croissante. Cette situation se présente par exemple lorsqu'on veut fusionner des données provenant de capteurs différents.

Un algorithme « naïf » opère de manière la suivante. À chaque étape, on considère le premier élément de chacune des k listes (c'est le plus grand dans chaque liste), puis on cherche le maximum de ces nombres. Ce nombre est inséré dans la liste résultat, et supprimé de la liste dont il provient. La recherche du maximum de k éléments coûte $k-1$ comparaisons. Si la somme des longueurs des k listes de données est n, l'algorithme naïf est donc de complexité $O(nk)$.

Il semble plausible que l'on puisse gagner du temps en « mémorisant » les comparaisons que l'on a faites lors du calcul d'un maximum. En effet, lorsque l'on calcule le maximum suivant, seule une donnée sur les k données comparées a changé. L'ordre des $k-1$ autres éléments reste le même, et on peut économiser des comparaisons si l'on connaît cet ordre au moins partiellement. La solution que nous proposons est basée sur un tas qui mémorise partiellement l'ordre. On verra que l'on peut effectuer la fusion en temps $O(k + n \log k)$. Le premier terme correspond au « prétraitement », c'est-à-dire à la mise en place de la structure particulière.

L'*arbre de sélection* est un arbre tassé à k feuilles. Chaque feuille est en fait une liste, l'une des k listes à fusionner (voir figure 4.20). La hauteur de l'arbre est $\log k$. Chaque nœud de l'arbre contient, comme clé, le maximum des clés de ses deux fils (ou le plus grand élément d'une liste). En particulier, le maximum des éléments en tête des k listes (en grisé sur la figure 4.20) se trouve $\log k$ fois dans l'arbre, sur les nœuds d'un chemin menant à la racine. L'extraction d'un plus grand élément se fait donc en temps constant. Cette extraction est suivie d'un mise-à-jour : En descendant le chemin dont les clés portent le maximum, on aboutit à la liste dont la valeur de tête est ce maximum. L'élément est remplacé, dans la liste, par son suivant. Ensuite, on remonte vers la racine en recalculant, pour chaque nœud rencontré, le valeur de la clé, avec la nouvelle valeur du fils mis-à-jour. À la racine, on trouve le nouveau maximum (voir figure 4.21). La mise-à-jour se fait donc en $O(\log k)$ opérations. La mise en place initiale de l'arbre est en temps k.

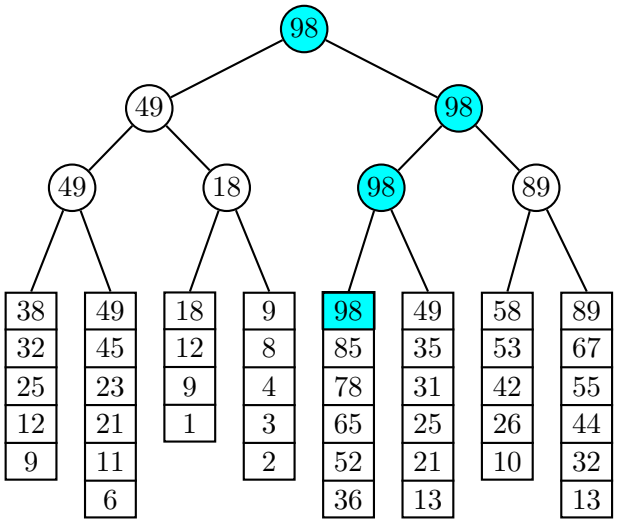


FIG. 4.20 – Un arbre de sélection pour 8 listes.

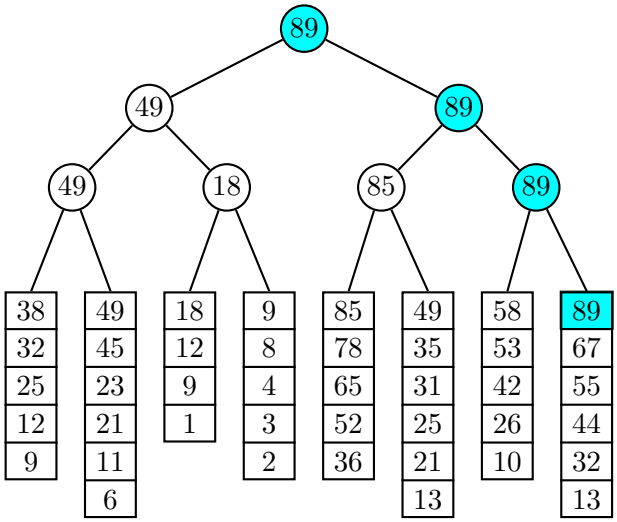


FIG. 4.21 – Après extraction du plus grand élément et recalcul.

5 Codage de Huffman

5.1 Compression des données

La compression des données est un problème algorithmique aussi important que le tri. On distingue la compression sans perte, où les données décompressées sont identiques aux données de départ, et la compression avec perte. La compression sans perte est importante par exemple dans la compression de textes, de données scientifiques ou du code compilé de programmes. En revanche, pour la compression d'images et de sons, une perte peut être acceptée si elle n'est pas perceptible, ou si elle est automatiquement corrigée par le récepteur.

Parmi les algorithmes de compression sans perte, on distingue plusieurs catégories : les algorithmes statistiques codent les caractères fréquents par des codes courts, et les caractères rares par des codes longs ; les algorithmes basés sur les dictionnaires enregistrent toutes les chaînes de caractères trouvées dans une table, et si une chaîne apparaît une deuxième fois, elle est remplacée

par son indice dans la table. L'algorithme de Huffman est un algorithme statistique. L'algorithme de Ziv-Lempel est basé sur un dictionnaire. Enfin, il existe des algorithmes « numériques ». Le codage arithmétique remplace un texte par un nombre réel entre 0 et 1 dont les chiffres en écriture binaire peuvent être calculés au fur et à mesure du codage. Le codage arithmétique repose aussi sur la fréquence des lettres. Ces algorithmes seront étudiés dans le cours 431 ou en majeure.

5.2 Algorithme de Huffman

L'algorithme de Huffman est un algorithme statistique. Les caractères du texte clair sont codés par des chaînes de bits. Le choix de ces codes se fait d'une part en fonction des fréquences d'apparition de chaque lettre, de sorte que les lettres fréquentes aient des codes courts, mais aussi de façon à rendre le décodage facile. Pour cela, on choisit un code préfixe, au sens décrit ci-dessous. La version du codage de Huffman que nous détaillons dans cette section est le codage dit « statique » : les fréquences n'évoluent pas au cours de la compression, et le code reste fixe. C'est ce qui se passe dans les modems, ou dans les fax.

Une version plus sophistiquée est le codage dit « adaptatif », présenté brièvement dans la section 5.3. Dans ce cas, les fréquences sont mises à jour après chaque compression de caractère, pour chaque fois optimiser le codage.

Codes préfixes et arbres

Un ensemble P de mots non vides est un *code préfixe* si aucun des mots de P n'est préfixe propre d'un autre mot de P.

Par exemple, l'ensemble $\{0, 100, 101, 111, 1100, 1101\}$ est un code préfixe. Un code préfixe P est *complet* si tout mot est préfixe d'un produit de mots de P. Le code de l'exemple ci-dessus est complet. Pour illustrer ce fait, prenons le mot 101101010111110110011 : il est préfixe du produit

$$101 \cdot 101 \cdot 0 \cdot 101 \cdot 0 \cdot 111 \cdot 10 \cdot 1100 \cdot 11\mathit{1}$$

L'intérêt de ces définitions vient des propositions suivantes.

Proposition 8 *Un ensemble fini de mots sur* $\{0, 1\}$ *est un code préfixe si et seulement s'il est le code des feuilles d'un arbre binaire.*

Rappelons qu'un arbre binaire est dit *complet* si tous ses nœuds internes ont arité 2.

Proposition 9 *Un ensemble fini de mots sur* $\{0, 1\}$ *est un code préfixe complet si et seulement s'il est le code des feuilles d'un arbre binaire complet.*

La figure 5.22 reprend l'arbre de la figure 3.12. Le code des feuilles est préfixe, mais il n'est pas complet.

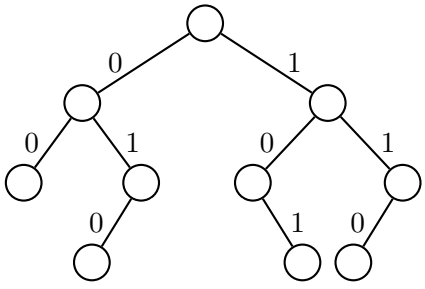


FIG. 5.22 – Le code des feuilles est $\{00, 010, 101, 110\}$.

En revanche, le code des feuilles de l'arbre de la figure 5.23 est complet.

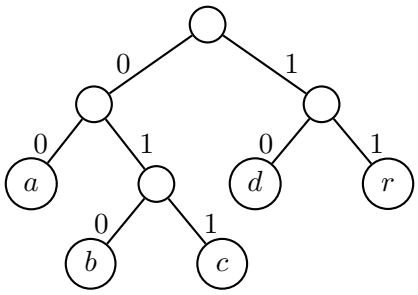


FIG. 5.23 – Le code des feuilles est $\{00, 010, 011, 10, 11\}$.

Le codage d'un texte par un code préfixe (complet) se fait de la manière suivante : à chaque lettre est associée une feuille de l'arbre. Le code de la lettre est le code de la feuille, c'est-à-dire l'étiquette du chemin de la racine à la feuille. Dans le cas de l'arbre de la figure 5.23, on associe les lettres aux feuilles de la gauche vers la droite, ce qui donne le tableau de codage suivant :

a	$\rightarrow$	00
b	$\rightarrow$	010
c	$\rightarrow$	011
d	$\rightarrow$	10
r	$\rightarrow$	11

Le codage consiste simplement à remplacer chaque lettre par son code. Ainsi, *abracadabra* devient 00010110001100100001011O. Le fait que le code soit préfixe permet un décodage instantané : il suffit d'entrer la chaîne caractère par caractère dans l'arbre, et de se laisser guider par les étiquettes. Lorsque l'on est dans une feuille, on y trouve le caractère correspondant, et on recommence à la racine. Quand le code est complet, on est sûr de pouvoir toujours décoder un message, éventuellement à un reste près qui est un préfixe d'un mot du code.

Le problème qui se pose est de minimiser la taille du texte codé. Avec le code donné dans le tableau ci-dessus, le texte *abracadabra*, une fois codé, est composé de 25 bits. Si l'on choisit un autre codage, comme par exemple

a	$\rightarrow$	0
b	$\rightarrow$	10
c	$\rightarrow$	1101
d	$\rightarrow$	1100
r	$\rightarrow$	111

on observe que le même mot se code par 01011101101011000101110 et donc avec 23 bits. Bien sûr, la taille du résultat ne dépend que de la fréquence d'apparition de chaque lettre dans le texte source. Ici, il y a 5 lettres a, 2 lettres b et r, et 1 fois la lettre c et d.

Construction de l'arbre

L'algorithme de Huffman construit un arbre binaire complet qui donne un code optimal, en ce sens que la taille du code produit est minimale parmi la taille de tous les codes produits à l'aide de codes préfixes complets.

Initialisation On construit une forêt d'arbres binaires formés chacun d'une seul feuille. Chaque feuille correspond à une lettre du texte, et a pour valeur la fréquence d'apparition de la lettre dans le texte.

Itération On fusionne deux des arbres dont la fréquence est minimale. Le nouvel arbre a pour fréquence la somme des fréquences de ses deux sous-arbres.

Arrêt On termine quand il ne reste plus qu'un seul arbre qui est l'arbre résultat.

La fréquence d'un arbre est, bien sûr, la somme des fréquences de ses feuilles. Voici une illustration de cet algorithme sur le mot *abracadabra*. La première étape conduit à la création des 5 arbres (feuilles) de la figure 5.24. Les feuilles des lettres c et d sont fusionnées (figure 5.25), puis cet arbre avec la feuille b (figure 5.26), etc. Le résultat est représenté dans la figure 5.28.

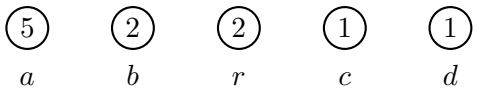


FIG. 5.24 – Les 5 arbres réduits chacun à une feuille.

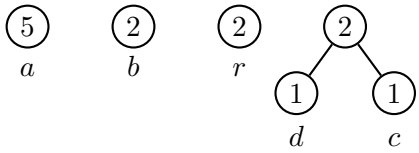


FIG. 5.25 – Les feuilles des lettres c et d sont fusionnées.

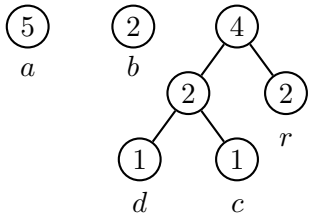


FIG. 5.26 – L'arbre est fusionné avec la feuille de r.

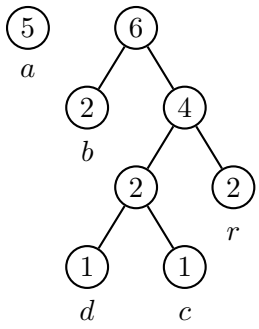


FIG. 5.27 – La feuille de b est fusionnée avec l'arbre.

Notons qu'il y a beaucoup de choix dans l'algorithme : d'une part, on peut choisir lequel des deux arbres devient sous-arbre gauche ou droit. Ensuite, les deux sous-arbres de fréquence minimale ne sont peut-être pas uniques, et là encore, il y a des choix pour la fusion. On peut prouver que cet algorithme donne un code optimal. Il en existe de nombreuses variantes. L'une d'elle consiste à grouper les lettres par deux (digrammes) ou même par bloc plus grands, notamment s'il s'agit de données binaires. Les techniques de compression avec dictionnaire (compression de Ziv-Lempel) ou le codage arithmétique donnent en général de meilleurs taux de compression.

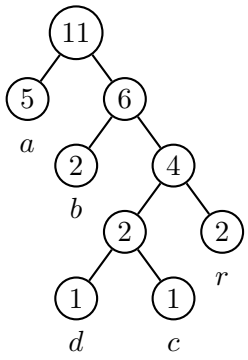


FIG. 5.28 – L'arbre après la dernière fusion.

Choix de la représentation des données

Si le nombre de lettres figurant dans le texte est n, l'arbre de Huffman est de taille $2n-1$. On le représente ici par un tableau pere, où, pour $x \geqslant 0$ et $y > 0$ pere[x] = y si x est fils droit de y et pere[x] = -y si x est fils gauche de y. Chaque caractère c a une fréquence d'apparition dans le texte, notée freq[c]. Seuls les caractères qui figurent dans le texte, i.e. de fréquence non nulle, vont être représentés dans l'arbre.

La création de l'arbre est contrôlée par un tas (mais un « tas-min », c'est-à-dire un tas avec extraction du minimum). La clé de la sélection est la fréquence des lettres et la fréquence cumulée dans les arbres. Une autre méthode rencontrée est l'emploi de deux files.

Cette représentation des données est bien adaptée au problème considéré, mais est assez éloignée des représentations d'arbres que nous verrons dans le chapitre suivant. Réaliser une implantation à l'aide de cette deuxième représentation est un bon exercice.

Implantation

La classe Huffman a deux données statiques : le tableau pere pour représenter l'arbre, et un tableau freq qui contient les fréquences des lettres dans le texte, et aussi les fréquences cumulées dans les arbres. On part du principe que les 26 lettres de l'alphabet peuvent figurer dans le texte (dans la pratique, on prendra plutôt un alphabet de 128 ou de 256 caractères).

```
class Huffman
{
  final static int N = 26, M = 2*N - 1; // nombre de caractères
  static int[] pere = new int[M];
  static int[] freq  = new int[M];

  public static void main(String[] args)
  {
    String s = args[0];
    calculFrequences(s);
    creerArbre();
    String[] table = faireTable();
    afficherCode(s,table);
    System.out.println();
  }
}
```

La méthode main décrit l'opération : on calcule les fréquences des lettres, et le nombre de lettres de fréquence non nulle. On crée ensuite un tas de taille appropriée, avec le tableau des

fréquences comme clés. L'opération principale est creerArbre() qui crée l'arbre de Huffman. La table de codage est ensuite calculée et appliquée à la chaîne à compresser.

Voyons les diverses étapes. Le calcul des fréquences et du nombre de lettres de fréquence non nulle ne pose pas de problème. On notera toutefois l'utilisation de la méthode charAt de la classe String, qui donne l'unicode du caractère en position i. Comme on a supposé que l'alphabet était a–z et que les unicodes de ces caractères sont consécutifs, l'expression s.charAt(i) - 'a' donne bien le rang du i-ème caractère dans l'alphabet. Il faudrait bien sûr modifier cette méthode si on prenait un alphabet à 256 caractères.

```
static void calculFrequences(String s)
{
  for (int i = 0; i < s.length(); i++)
    freq[s.charAt(i) - 'a']++;
}

static int nombreLettres()
{
  int n = 0;
  for (int i = 0; i < N; i++)
    if (freq[i] > 0)
      n++;
  return n;
}
```

La méthode de création d'arbre utilise très exactement l'algorithme exposé plus haut : d'abord, les lettres sont insérées dans le tas ; ensuite, les deux éléments de fréquence minimale sont extraits, et un nouvel arbre, dont la fréquence est la somme des fréquences des deux arbres extraits, est inséré dans le tas. Comme il y a n feuilles dans l'arbre, il y a $n-1$ créations de nœuds internes.

```
static void creerArbre()
{
  int n = nombreLettres();
  Tas tas = new Tas(2*n-1, freq);
  for (int i = 0; i < N; ++i)
    if (freq[i] >0)
      tas.ajouter(i);
  for (int i = N; i < N+n-1; ++i)
  {
    int x = tas.minimum();
    tas.supprimer();
    int y = tas.minimum();
    tas.supprimer();
    freq[i] = freq[x] + freq[y];
    pere[x] = -i;
    pere[y] = i;
    tas.ajouter(i);
  }
}
```

Le calcul de la table de codage se fait par un parcours d'arbre :

```
static String code(int i)
{
```

```
  if (pere[i] == 0) return "";
  return code(Math.abs(pere[i])) + ((pere[i] < 0) ? "0" : "1");
}

static String[] faireTable()
{
  String[] table = new String[N];
  for (int i = 0; i < N; i++)
    if (freq[i] >0)
      table[i] = code(i);
  return table;
}
```

Il reste à examiner la réalisation du « tas-min ». Il y a deux différences avec les tas déjà vus : d'une part, le maximum est remplacé par le minimum (ce qui est négligeable), et d'autre part, ce ne sont pas les valeurs des éléments eux-mêmes (les lettres du texte) qui sont utilisées comme comparaison, mais une valeur qui leur est associée (la fréquence de la lettre). Les méthodes des tas déjà vues se récrivent très facilement dans ce cadre.

```
class Tas
{
  int[] a;  // contient les caractères
  int nTas = 0;
  int[] freq; // contient les fréquences des caractères

  Tas(int taille, int[] freq)
  {
    this.freq = freq;
    nTas = 0;
    a = new int[taille];
  }

  int minimum()
  {
    return a[0];
  }

  void ajouter(int v)
  {
    int i = nTas;
    ++nTas;
    while (i > 0 && freq[a[(i-1)/2]] >= freq[v])
    {
      a[i] = a[(i-1)/2];
      i = (i-1)/2;
    }
    a[i] = v;
  }

  void supprimer()
  {
    int v = a[0] = a[--nTas];
    int i = 0;
    while (2*i + 1 < nTas)
    {
```

```
        int j = 2*i + 1;
        if (j +1 < nTas && freq[a[j+1]] < freq[a[j]])
          ++j;
        if (freq[v] <= freq[a[j]])
          break;
        a[i] = a[j];
        i = j;
      }
      a[i] = v;
    }
  }
```

Une fois la table du code calculée, encore faut-il la transmettre avec le texte comprimé, pour que le destinaire puisse décompresser le message. Transmettre la table telle quelle est redondant puisque l'on transmet tous les chemins de la racine vers les feuilles. Il est plus économique de faire un parcours préfixe de l'arbre, avec la valeur littérale des lettres représentées aux feuilles. Par exemple, pour l'arbre de la figure 5.28, on obtient la représentation `01[a]01[b]001[d]1[c]1[r]`. Une méthode plus simple est de transmettre la suite de fréquence, quitte au destinataire de reconstruire le code. Cette deuxième méthode admet un raffinement (qui montre jusqu'où peut aller la recherche de l'économie de place) qui consiste à non pas transmettre les valeurs exactes des fréquences, mais une séquence de fréquences fictives (à valeurs numériques plus petites, donc plus courtes à transmettre) qui donne le même code de Huffman. Par exemple, les deux arbres

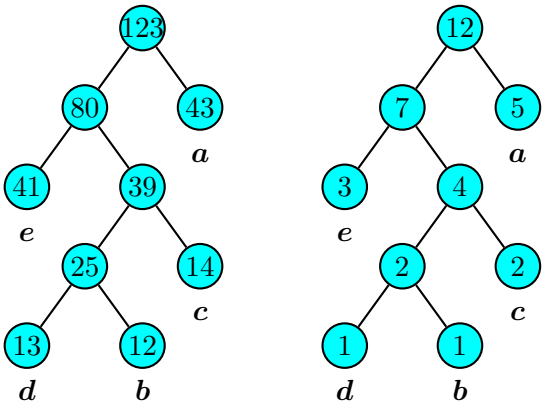


FIG. 5.29 – Deux suites de fréquences qui produisent le même arbre.

de la figure 5.29 ont des fréquences différentes, mais le même code. Il est bien plus économique de transmettre la suite de nombres $5, 1, 2, 1, 3$ que la suite initiale $43, 13, 12, 14, 41$.

5.3 Algorithme de Huffman adaptatif

Les inconvénients de la méthode de compression de Huffman sont connus :
- Il faut lire le texte entièrement avant de lancer la compression.
- Il faut aussi transmettre le code trouvé.

Une version adaptative de l'algorithme de Huffman corrige ces défauts. Le principe est le suivant :
- Au départ, toutes les lettres ont même fréquence (nulle) et le code est uniforme.
- Pour chaque lettre x, le code de x est envoyé, puis la fréquence de la lettre est augmentée et l'arbre de Huffman est recalculé.

Evidemment, le code d'une lettre change en cours de transmission : quand la fréquence (relative) d'une lettre augmente, la longueur de son code diminue. Le code d'une lettre s'adapte à sa fréquence. Le destinataire du message compressé mime le codage de l'expéditeur : il maintient

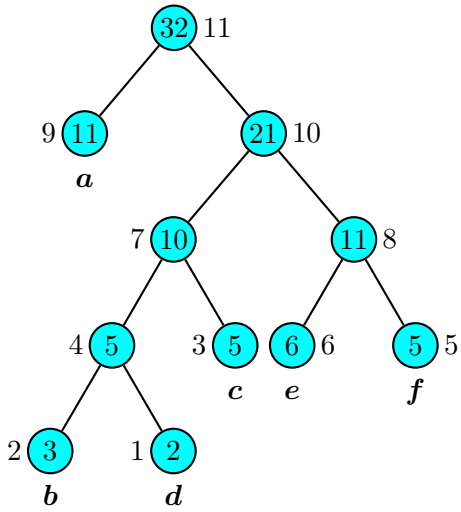


FIG. 5.30 – Un arbre de Huffman avec un parcours compatible.

un arbre de Huffman qu'il met à jour comme l'expéditeur, et il sait donc à tout moment quel est le code d'une lettre.

Les deux inconvénients du codage de Huffman (statique) sont corrigés par cette version. Il n'est pas nécessaire de calculer globalement les fréquences des lettres du texte puisqu'elles sont mises à jour à chaque pas. Il n'est pas non plus nécessaire de transmettre la table de codage puisqu'elle est recalculée par le destinataire.

Il existe par ailleurs une version « simplifiée » du codage de Huffman où les fréquences utilisées pour la construction de l'arbre ne sont pas les fréquences des lettres du texte à comprimer, mais les fréquences moyennes rencontrées dans les textes de la langue considérée. Bien entendu, le taux de compression s'en ressent si la distribution des lettres dans le texte à comprimer s'écarte de cette moyenne. C'est dans ce cas que l'algorithme adaptatif s'avère particulièrement utile.

Notons enfin que le principe de l'algorithme adaptatif s'applique à tous les algorithmes de compression statistiques. Il existe, par exemple, une version adaptative de l'algorithme de compression arithmétique.

L'algorithme adaptatif opère, comme l'algorithme statique, sur un arbre. Aux feuilles de l'arbre se trouvent les lettres, avec leurs fréquences. Aux nœuds de l'arbre sont stockées les fréquences cumulées, c'est-à-dire la somme des fréquences des feuilles du sous-arbre dont le nœud est la racine (voir figure 5.30). L'arbre de Huffman est de plus muni d'une liste de parcours (un ordre total) qui a les deux propriétés suivantes :

- le parcours est compatible avec les fréquences (les fréquences sont croissantes dans l'ordre du parcours),
- deux nœuds frères sont toujours consécutifs.

On démontre que, dans un arbre de Huffman, on peut toujours trouver un ordre de parcours possédant ces propriétés. Dans la figure 5.30, les nœuds sont numérotés dans un tel ordre. L'arbre de Huffman évolue avec chaque lettre codée, de la manière suivante. Soit x le caractère lu dans le texte clair. Il correspond à une feuille de l'arbre. Le code correspondant est envoyé, puis les deux opérations suivantes sont réalisées.

- Partant de la feuille du caractère, sa fréquence est incrémentée, et on remonte vers la racine en incrémentant les fréquences dans les nœuds rencontrés.
- Avant l'incrémentation, chaque nœud est permuté avec le *dernier nœud* (celui de plus grand numéro) de même fréquence dans l'ordre de parcours.

La deuxième opération garantit que l'ordre reste compatible avec les fréquences. Supposons que

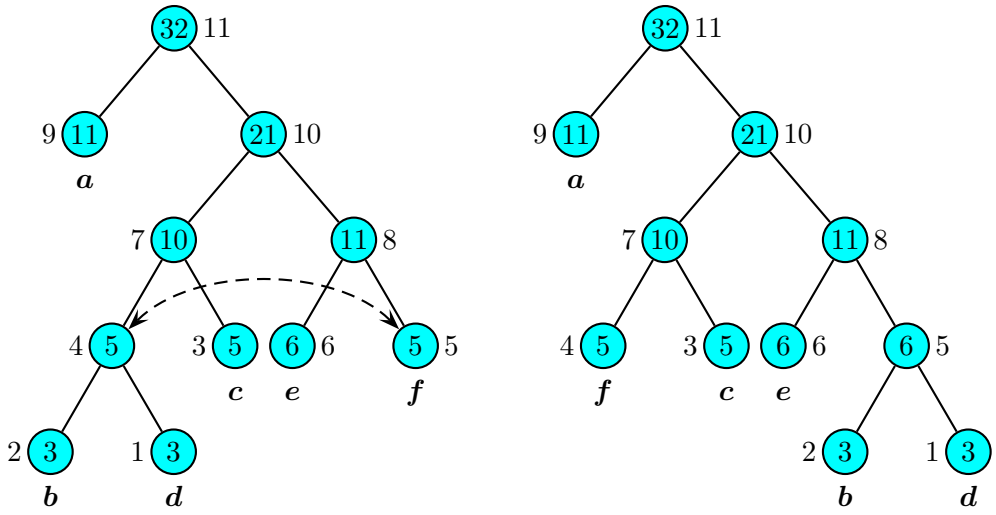


FIG. 5.31 – Après incrémentation de d (à gauche), et après permutation des nœuds (à droite).

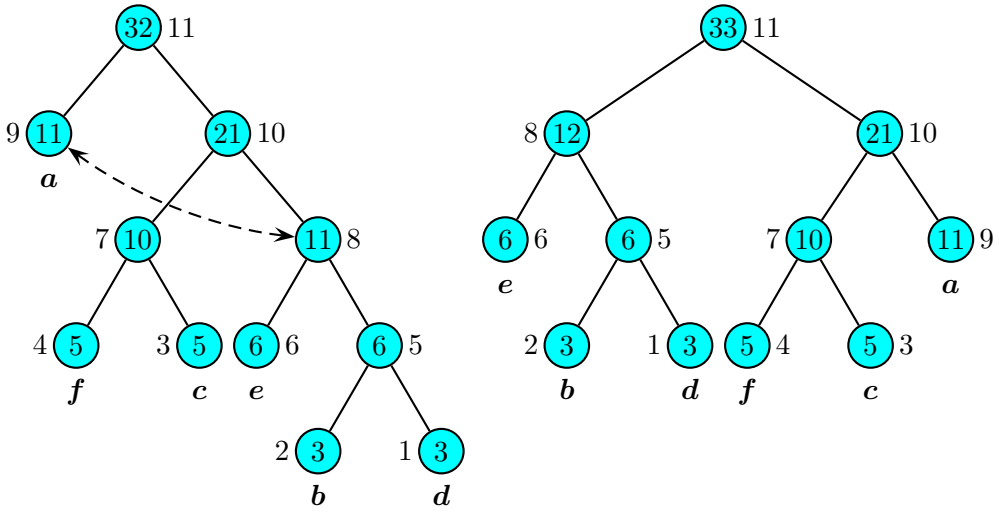


FIG. 5.32 – Avant permutation des nœuds 8 et 9 (à gauche) , et arbre final (à droite).

la lettre suivante du texte clair à coder, dans l'arbre de la figure 5.30, soit une lettre d. Le code émis est alors 1001. La fréquence de la feuille d est incrémentée. Le père de la feuille d a pour fréquence 5. Son numéro d'ordre est 4, et le plus grand nœud de fréquence 5 a numéro d'ordre 5. Les deux nœuds sont échangés (figure 5.31). De même, avant d'incrémenter le nœud de numéro 8, il est échangé avec le nœud de numéro 9, de même fréquence (figure 5.32). Ensuite, la racine est incrémentée.

Pour terminer, voici une table de quelques statistiques concernant l'algorithme de Huffman. Ces statistiques ont été obtenues à une époque où il y avait encore peu de données disponibles sous format électronique, et les contes de Grimm en faisaient partie... Comme on le voit, le taux de compression obtenu par les deux méthodes est sensiblement égal, car les données sont assez

homogènes.

	Contes de Grimm	Texte technique
Taille (bits)	700 000	700 000
Huffman	439 613	518 361
Taille code	522	954
Total	440135	519315
Adaptatif	440164	519561

Si on utilise un codage par digrammes (groupe de deux lettres), les statistiques sont les suivantes :

	Contes de Grimm	Texte technique
Taille (bits)	700 000	700 000
Huffman	383 264	442 564
Taille code	10 880	31 488
Total	394 144	474 052
Adaptatif	393 969	472 534

On voit que le taux de compression est légèrement meilleur.

Chapitre V

Arbres binaires

Dans ce chapitre, nous traitons d'abord les arbres binaires de recherche, puis les arbres équilibrés.

1 Implantation des arbres binaires

Un arbre binaire qui n'est pas vide est formé d'un nœud, sa *racine*, et de deux sous-arbres binaires, l'un appelé le *fils gauche*, l'autre le *fils droit*. Nous nous intéressons aux arbres contenant des informations. Chaque nœud porte une information, appelée son *contenu*. Un arbre non vide est donc entièrement décrit par le triplet (fils gauche, contenu de la racine, fils droit). Cette définition récursive se traduit en une spécification de programmation. Il suffit de préciser la nature du contenu d'un nœud. Pour simplifier, nous supposons que le contenu est un entier. On obtient alors la définition.

```
class Arbre
{
  int contenu;
  Arbre filsG, filsD;

  Arbre(Arbre g, int c, Arbre d)
  {
    filsG = g;
    contenu = c;
    filsD = d;
  }
}
```

L'arbre vide est, comme d'habitude, représenté par `null`. Un arbre réduit à une feuille, de contenu `x`, est créé par

```
new Arbre(null, x, null)
```

L'arbre de la figure 1.1 est créé par

```
new Arbre(
  new Arbre(
    new Arbre(null, 3, null),
    5,
    new Arbre(
      new Arbre(
       new Arbre(null, 6, null),
       8,
```

```
        null)
       12,
       new Arbre(null, 13, null))),
  20,
  new Arbre(
    new Arbre(null, 21, null),
    25,
    new Arbre(null, 28, null)))
```

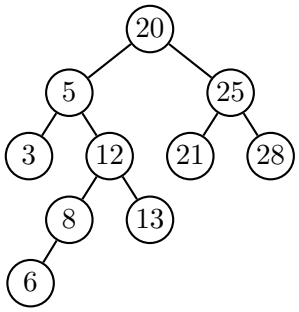


FIG. 1.1 – Un arbre binaire portant des informations aux nœuds.

Avant de poursuivre, reprenons le schéma déjà utilisé pour les listes. Quatre fonctions caractérisent les arbres : composer, cle, filsGauche, filsDroit. Elles s'implantent facilement :

```
static Arbre composer(Arbre g, int c, Arbre d)
{
  return new Arbre(g, c, d);
}

static int cle(Arbre a)
{
  return a.contenu;
}

static Arbre filsGauche(Arbre a)
{
  return a.filsG;
}

static Arbre filsDroit(Arbre a)
{
  return a.filsD;
}
```

Les quatre fonctions sont liées par les équations suivantes :

$$\begin{aligned} \mathsf{cle}(\mathsf{composer}(g,c,d)) &= c \\ \mathsf{filsGauche}(\mathsf{composer}(g,c,d)) &= g \\ \mathsf{filsDroit}(\mathsf{composer}(g,c,d)) &= d \\ \mathsf{composer}(\mathsf{filsGauche}(a), \mathsf{cle}(a), \mathsf{filsDroit}(a)) &= a; \quad (a \neq \mathsf{null}) \end{aligned}$$

Comme pour les listes, ces quatre fonctions sont à la base d'opérations non destructives.

La définition récursive des arbres binaires conduit naturellement à une programmation récursive, comme pour les listes. Voici quelques exemples : la *taille* d'un arbre, c'est-à-dire le nombre $t(a)$ de ses nœuds, s'obtient par la formule

$$t(a) = \begin{cases} 0 & \text{si } a \text{ est vide} \\ 1 + t(a_g) + t(a_d) & \text{sinon.} \end{cases}$$

où sont notés a_g et a_d les sous-arbres gauche et droit de a. D'où la méthode

```
static int taille(Arbre a)
{
  if (a == null)
    return 0;
  return 1 + taille(a.filsG) + taille(a.filsD);
}
```

Des formules semblables donnent le nombre de feuilles ou la hauteur d'un arbre. Nous illustrons ce style de programmation par les *parcours* d'arbre définis page 81. Les trois parcours en profondeur s'écrivent :

```
static void parcoursPréfixe(Arbre a)
{
  if (a == null)
    return;
  System.out.print(a.contenu + " ");
  parcoursPréfixe(a.filsG);
  parcoursPréfixe(a.filsD);
}

static void parcoursInfixe(Arbre a)
{
  if (a == null)
    return;
  parcoursInfixe(a.filsG);
  System.out.print(a.contenu + " ");
  parcoursInfixe(a.filsD);
}

static void parcoursSuffixe(Arbre a)
{
  if (a == null)
    return;
  parcoursSuffixe(a.filsG);
  parcoursSuffixe(a.filsD);
  System.out.print(a.contenu + " ");
}
```

Le *parcours en largeur* d'un arbre binaire s'écrit simplement avec une *file*. Le *parcours préfixe* s'écrit lui aussi simplement de manière itérative, avec une *pile*. Nous reprenons les classes Pile et File du chapitre II, sauf que ce sont, cette fois-ci, des piles ou des files d'arbres. On écrit alors

```
static void parcoursPréfixeI(Arbre a)
{
  if (a == null)
    return;
```

```
  Pile p = new Pile();
  p.ajouter(a);
  while (!p.estVide())
  {
    a = p.valeur();
    p.supprimer();
    System.out.print(a.contenu + " ");
    if (a.filsD != null)
      p.ajouter(a.filsD);
    if (a.filsG != null)
      p.ajouter(a.filsG);
  }
}
static void parcoursLargeurI(Arbre a)
{
  if (a == null)
    return;
  File f = new File();
  f.ajouter(a);
  while (!f.estVide())
  {
    a = f.valeur();
    f.supprimer();
    System.out.print(a.contenu + " ");
    if (a.filsG != null)
      f.ajouter(a.filsG);
    if (a.filsD != null)
      f.ajouter(a.filsD);
  }
}
```

1.1 Implantation des arbres ordonnés par arbres binaires

Rappelons qu'un arbre est *ordonné* si la suite des fils de chaque nœud est ordonnée. Il est donc naturel de représenter les fils dans une liste chaînée. Un nœud contient aussi la référence à son fils aîné, c'est-à-dire à la tête de la liste de ses fils. Ainsi, chaque nœud contient deux références, celle à son fils aîné, et celle à son frère cadet. En d'autres termes, la structure des arbres binaires convient parfaitement, sous réserve de rebaptiser filsAine le champ filsG et frereCadet le champ filsD.

```
class ArbreOrdonne
{
  int contenu;
  Arbre filsAine, frereCadet;
  Arbre(Arbre g, int c, Arbre d)
  {
    filsAine = g;
    contenu = c;
    frereCadet = d;
  }
}
```

Cette représentation est aussi appelée « fils gauche — frère droit » (voir figure 1.2).

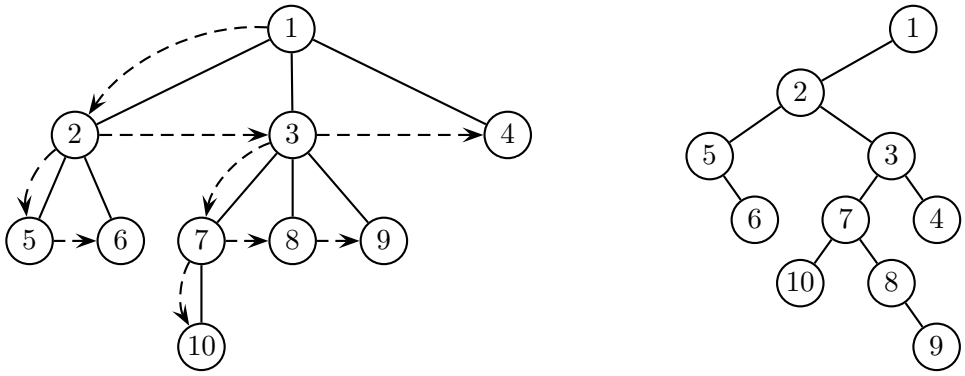


FIG. 1.2 – Représentation d'un arbre ordonné par un arbre binaire.

Noter que la racine de l'arbre binaire n'a pas de fils droit. En fait, cette représentation s'étend à la représentation, par un seul arbre binaire, d'une forêt ordonnée d'arbres ordonnés.

2 Arbres binaires de recherche

Les arbres binaires servent à gérer des informations. Chaque nœud contient une donnée prise dans un certain ensemble. Nous supposons dans cette section que cet ensemble est totalement ordonné. Ceci est le cas par exemple pour les entiers et pour les mots.

Un arbre binaire a est un *arbre binaire de recherche* si, pour tout nœud s de a, les contenus des nœuds du sous-arbre gauche de s sont strictement inférieurs au contenu de s, et que les contenus des nœuds du sous-arbre droit de s sont strictement supérieurs au contenu de s (cf. figure 2.3).

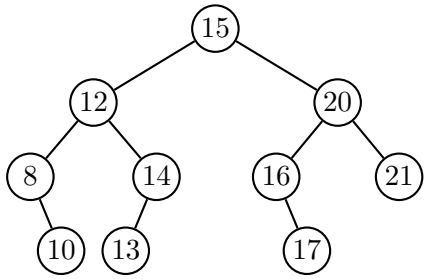


FIG. 2.3 – Un arbre binaire de recherche.

Une petite mise en garde : contrairement à ce que l'analogie avec les tas pourrait laisser croire, il ne suffit pas de supposer que, pour tout nœud s de l'arbre, le contenu du fils gauche de s soit strictement inférieur au contenu de s, et que le contenu du fils droit de s soit supérieur au contenu de s. Ainsi, dans l'arbre de la figure 2.3, si on change la valeur 13 en 11, on n'a plus un arbre de recherche...

Une conséquence directe de la définition est la régle suivante :

Règle. *Dans un arbre binaire de recherche, le parcours infixe fournit les contenus des nœuds en ordre croissant.*

Une seconde règle permet de déterminer dans certains cas le nœud qui précède un nœud donné dans le parcours infixe.

Règle. *Si un nœud possède un fils gauche, son prédécesseur dans le parcours infixe est le nœud le plus à droite dans son sous-arbre gauche. Ce nœud n'est pas nécessairement une feuille, mais il n'a pas de fils droit.*

Ainsi, dans l'arbre de la figure 2.3, la racine possède un fils gauche, et son prédécesseur est le nœud de contenu 14, qui n'a pas de fils droit...

Les arbres binaires de recherche fournissent, comme on le verra, une implantation souvent efficace d'un type abstrait de données, appelé *dictionnaire*, qui opère sur un ensemble totalement ordonné à l'aide des opérations suivantes :

- rechercher un élément
- insérer un élément
- supprimer un élément

Plusieurs implantations d'un dictionnaire sont envisageables : par tableau, par liste ordonnés ou non, par tas, et par arbre binaire de recherche. La table V.1 rassemble la complexité des opérations de dictionnaire selon la structure de données choisie.

Implantation	Rechercher	Insérer	Supprimer
Tableau non ordonné	$O(n)$	$O(1)$	$O(n)$
Liste non ordonnée	$O(n)$	$O(1)$	$O(1)^{(*)}$
Tableau ordonné	$O(\log n)$	$O(n)$	$O(n)$
Liste ordonnée	$O(n)$	$O(n)$	$O(1)^{(*)}$
Tas	$O(n)$	$O(\log n)$	$O(n)$
Arbre de recherche	$O(h)$	$O(h)$	$O(h)$

TAB. V.1 – Complexité des implantations de dictionnaires

L'entier h désigne la hauteur de l'arbre. On voit que lorsque l'arbre est bien équilibré, c'est-à-dire lorsque la hauteur est proche du logarithme de la taille, les opérations sont réalisables de manière particulièrement efficace.

2.1 Recherche d'une clé

Nous commençons l'implantation des opérations de dictionnaire sur les arbres binaires de recherche par l'opération la plus simple, la recherche. Plutôt que d'écrire une méthode booléenne qui teste la présence d'un élément dans l'arbre, nous écrivons une méthode qui retourne l'arbre dont la racine porte l'élément cherché s'il figure dans l'arbre, et `null` sinon. Comme toujours, il y a le choix entre une méthode récursive, calquée sur la définition récursive des arbres, et une méthode itérative, cheminant dans l'arbre. Nous présentons les deux, en commençant par la méthode récursive. Pour chercher si un élément x figure dans un arbre A, on commence par comparer x au contenu c de la racine de A. S'il y a égalité, on a trouvé la réponse ; sinon il y a deux cas selon que $x < c$ et $x > c$. Si $x < c$, alors x figure peut-être dans le sous-arbre gauche A_g de A, mais certainement pas dans le sous-arbre droit A_d. On élimine ainsi de la recherche tous les nœuds du sous-arbre droit. Cette méthode n'est pas sans rappeler la recherche dichotomique. La méthode s'écrit récursivement comme suit :

*Dans cette table, le coût de la suppression dans une liste non ordonnée est calculé en supposant l'élément déjà trouvé.

```
static Arbre chercher(int x, Arbre a)
{
  if (a == null || x == a.contenu)
    return a;
  if (x < a.contenu)
    return chercher(x, a.filsG);
  return chercher(x, a.filsD);
}
```

Cette méthode retourne null si l'arbre a ne contient pas x. Ceci inclut le cas où l'arbre est vide. Voici la méthode itérative.

```
static chercherI(int x, Arbre a)
{
  while(a != null && x != a.contenu)
    if (x < a.contenu)
      a = a.filsG;
    else
      a = a.filsD;
  return a;
}
```

On voit que la condition de *continuation* dans la méthode itérative chercherI est la négation de la condition d'*arrêt* de la méthode récursive, ce qui est logique.

2.2 Insertion d'une clé

L'adjonction d'un nouvel élément à un arbre modifie l'arbre. Nous sommes confrontés au même choix que pour les listes : soit on construit une nouvelle version de l'arbre (version non destructive), soit on modifie l'arbre existant (version destructive). Nous présentons une méthode récursive dans les deux versions. Dans les deux cas, si l'entier figure déjà dans l'arbre, on ne l'ajoute pas une deuxième fois. Voici la version destructive.

```
static Arbre inserer(int x, Arbre a)
{
  if (a == null)
    return new Arbre(null, x, null);
  if (x < a.contenu)
    a.filsG = inserer(x, a.filsG);
  else if (x > a.contenu)
    a.filsD = inserer(x, a.filsD);
  return a;
}
```

Voici la version non destructive.

```
static Arbre inserer(int x, Arbre a)
{
  if (a == null)
    return new Arbre(null, x, null);
  if (x < a.contenu)
  {
    Arbre b = inserer(x, a.filsG);
    return new Arbre(b, a.contenu, a.filsD);
  }
```

```
    else if (x > a.contenu)
    {
      Arbre b = inserer(x, a.filsD);
      return new Arbre(a.filsG, a.contenu, b);
    }
    return a;
  }
```

2.3 Suppression d'une clé

La *suppression* d'une clé dans un arbre est une opération plus complexe. Elle s'accompagne de la suppression d'un nœud. Comme on le verra, ce n'est pas toujours le nœud qui porte la clé à supprimer qui sera enlevé. Soit s le nœud qui porte la clé x à supprimer. Trois cas sont à considérer selon le nombre de fils du nœud x :

- si le nœud s est une feuille, alors on l'élimine ;
- si le nœud s possède un seul fils, on élimine s et on « remonte » ce fils.
- si le nœud s possède deux fils, on cherche le prédécesseur t de s. Celui-ci n'a pas de fils droit. On remplace le *contenu* de s par le contenu de t, et on élimine t.

La suppression de la feuille qui porte la clé 13 est illustrée dans la figure 2.4

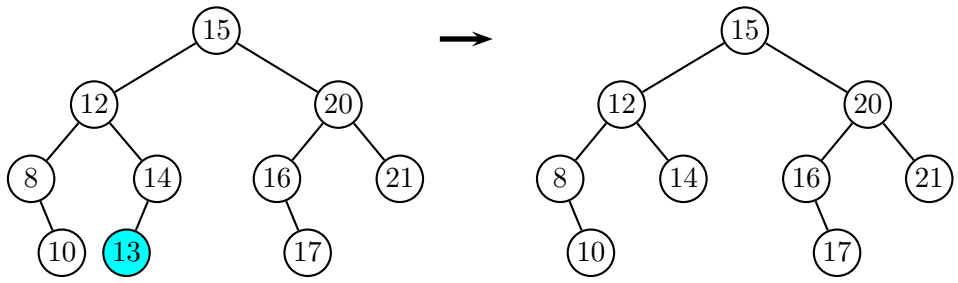


FIG. 2.4 – Suppression de la clé 13 par élimination d'une feuille.

La figure 2.5 illustre la « remontée » : le nœud s qui porte la clé 16 n'a qu'un seul enfant. Cet enfant devient l'enfant du père de s, le nœud de clé 20.

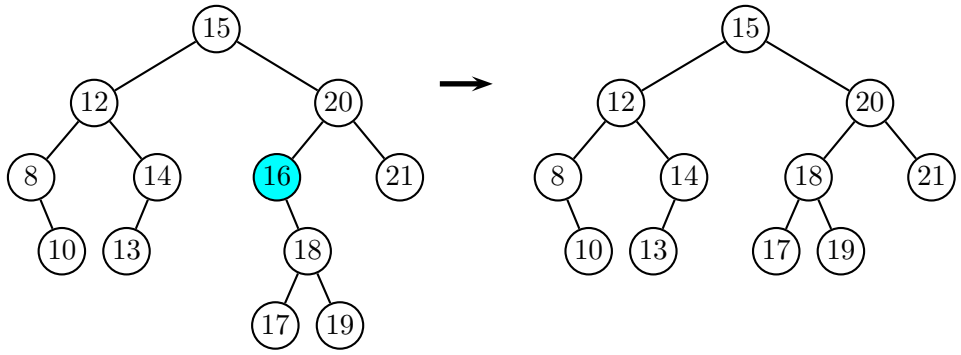


FIG. 2.5 – Suppression de la clé 16 par remontée du fils.

Le cas d'un nœud à deux fils est illustré dans la figure 2.6. La clé à supprimer se trouve à la racine de l'arbre. On ne supprime pas le nœud, mais seulement sa clé, en remplaçant la clé par

une autre clé. Pour conserver l'ordre sur les clés, il n'y a que deux choix : la clé du prédécesseur dans l'ordre infixe, ou la clé du successeur. Nous choisissons la première solution. Ainsi, la clé 14 est mise à la racine de l'arbre. Nous sommes alors ramenés au problème de la suppression du nœud du prédécesseur et de sa clé. Comme le prédécesseur est le nœud le plus à droite du sous-arbre gauche, il n'a pas de fils droit, donc il a zéro ou un fils, et sa suppression est couverte par les deux premiers cas.

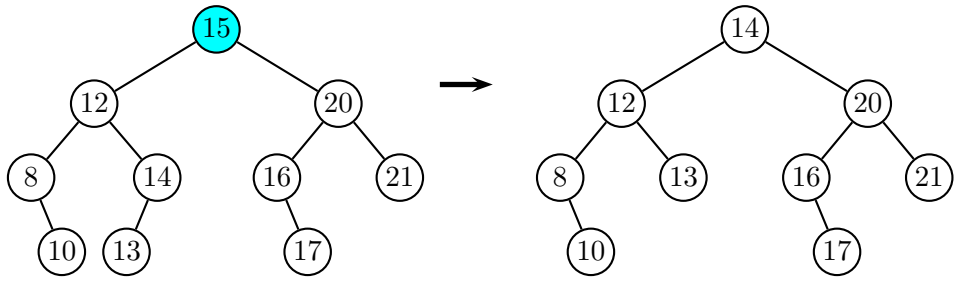


FIG. 2.6 – Suppression de la clé 15 par substitution de la clé 14 et suppression de ce nœud.

Trois méthodes coopèrent pour la suppression. La première, suppression, recherche le nœud portant la clé à supprimer ; la deuxième, suppressionRacine, effectue la suppression selon les cas énumérés ci-dessus. La troisième, dernierDescendant est une méthode auxiliaire ; elle calcule le précédesseur d'un nœud qui a un fils gauche.

```
static Arbre supprimer(int x, Arbre a)
{
  if (a == null)
    return a;
  if (x == a.contenu)
    return supprimerRacine(a);
  if (x < a.contenu)
    a.filsG = supprimer(x, a.filsG);
  else
    a.filsD = supprimer(x, a.filsD);
  return a;
}
```

La méthode suivante supprime la clé de la racine de l'arbre.

```
static Arbre supprimerRacine(Arbre a)
{
  if (a.filsG == null)
    return a.filsD;
  if (a.filsD == null)
    return a.filsG;
  Arbre f = dernierDescendant(a.filsG);
  a.contenu = f.contenu;
  a.filsG = supprimer(f.contenu, a.filsG);
}
```

La dernière méthode est toute simple :

```
static Arbre dernierDescendant(Arbre a)
{
  if (a.filsD == null)
```

```
    return a;
  return dernierDescendant(a.filsD);
}
```

La récursivité croisée entre les méthodes supprimer et supprimerRacine est déroutante au premier abord. En fait, l'appel à supprimer à la dernière ligne de supprimerRacine conduit au nœud prédécesseur de la racine de l'arbre, appelé f. Comme ce nœud n'a pas deux fils, il n'appelle pas une deuxième fois la méthode supprimerRacine...

Il est intéressant de voir une réalisation itérative de la suppression. Elle démonte entièrement la « mécanique » de l'algorithme. En fait, chacune des trois méthodes peut séparément être écrite de façon récursive.

```
static Arbre supprimer(int x, Arbre a)
{
  Arbre b = a;
  while (a != null && x != a.contenu)
    if (x < a.contenu)
      a = a.filsG;
    else
      a = a.filsD;
  if (a != null)
    a = supprimerRacine(a);
  return b;
}
```

Voici la deuxième.

```
static Arbre supprimerRacine(Arbre a)
{
  if (a.filsG == null)
    return a.filsD;
  if (a.filsD == null)
    return a.filsG;
  Arbre b = a.filsG;
  if (b.filsD == null)
  { // cas (i)
    a.contenu = b.contenu;
    a.filsG = b.filsG;
  }
  else
  { // cas (ii)
    Arbre p = avantDernierDescendant(b);
    Arbre f = p.filsD;
    a.contenu = f.contenu;
    p.filsD = f.filsG;
  }
  return a;
}
```

Et voici le calcul de l'avant-dernier descendant :

```
static Arbre avantDernierDescendant(Arbre a)
{
  while (a.filsD.filsD != null)
    a = a.filsD;
```

```
    return a;
  }
```

Décrivons plus précisément le fonctionnement de la méthode supprimerRacine. La première partie permet de se ramener au cas où la racine de l'arbre a a deux fils. On note b le fils gauche de a, et pour déterminer le prédécesseur de la racine de a, on cherche le nœud le plus à droite dans l'arbre b. Deux cas peuvent se produire :

(i) la racine de b n'a pas de fils droit, ou

(ii) la racine de b a un fils droit.

Les deux cas sont illustrés sur les figures 2.7 et 2.8.

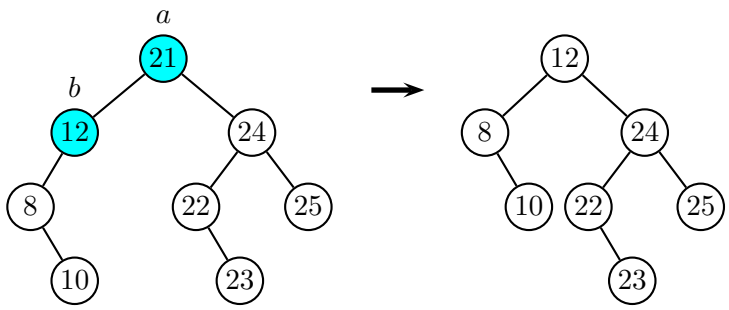


FIG. 2.7 – Suppression de la racine, version itérative, cas (i).

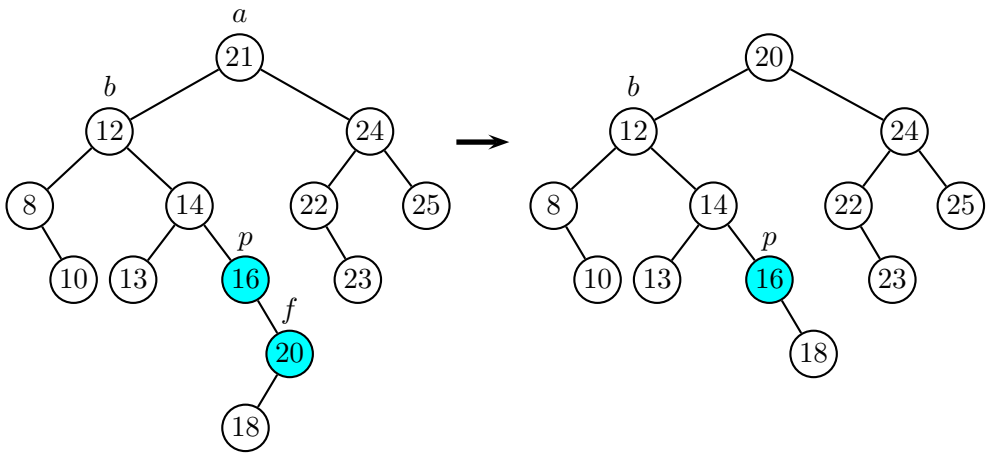


FIG. 2.8 – Suppression de la racine, version itérative, cas (ii).

Dans le premier cas, la clé de la racine de b est transférée à la racine de a, et b est remplacée par son sous-arbre gauche (qui peut d'ailleurs être vide). Dans le deuxième cas, on cherche l'avant-dernier descendant, noté p, de b sur la branche droite de b, au moyen de la méthode avantDernierDescendant. Cela peut être b lui-même, ou un de ses descendants (notons que dans le cas (i), l'avant-dernier descendant n'existe pas, ce qui explique le traitement séparé opéré dans ce cas). Le sous-arbre droit f de p n'est pas vide par définition. La clé de f est transférée à la racine de a, et f est remplacé par son sous-arbre gauche — ce qui fait disparaître la racine de f.

2.4 Hauteur moyenne

Il est facile de constater, sur nos implantations, que la recherche, l'insertion et la suppression dans un arbre binaire de recherche se font en complexité $O(h)$, où h est la hauteur de l'arbre.

Le cas le pire, pour un arbre à n nœuds, est $O(n)$. En ce qui concerne la *hauteur moyenne*, deux cas sont à considérer. La première des propositions s'applique aux arbres, la deuxième aux permutations.

Proposition 10 *Lorsque tous les arbres binaires à n nœuds sont équiprobables, la hauteur moyenne d'un arbre binaire à n nœuds est en $O(\sqrt{n})$.*

Proposition 11 *Lorsque toutes les permutations de $\{1, \ldots, n\}$ sont équiprobables, la hauteur moyenne d'un arbre binaire de recherche obtenu par insertion des entiers d'une permutation dans un arbre initialement vide est $O(n \log n)$.*

La différence provient du fait que plusieurs permutations peuvent donner le même arbre. Par exemple les permutations $2, 1, 3$ et $2, 3, 1$ produisent toutes les deux l'arbre de la figure 2.9.

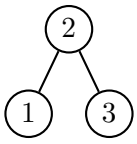


FIG. 2.9 – L'arbre produit par la suite d'insertions $2, 1, 3$, ou par la suite $2, 3, 1$.

3 Arbres équilibrés

Comme nous l'avons déjà constaté, les coûts de la recherche, de l'insertion et de la suppression dans un arbre binaire de recherche sont de complexité $O(h)$, où h est la hauteur de l'arbre. Le cas le pire, pour un arbre à n nœuds, est $O(n)$. Ce cas est atteint par des arbres très déséquilibrés, ou « filiformes ». Pour éviter que les arbres puissent prendre ces formes, on utilise des opérations plus ou moins simples, mais peu coûteuses en temps, de transformation d'arbres. À l'aide de ces transformations on tend à rendre l'arbre le plus régulier possible dans un sens qui est mesuré par un paramètre dépendant en général de la hauteur. Une famille d'arbres satisfaisant une condition de régularité est appelée une famille d'arbres *équilibrés*. Plusieurs espèces de tels arbres ont été développés, notamment les arbres AVL, les arbres 2-3, les arbres rouge et noir, ainsi qu'une myriade de variantes. Dans les langages comme Java ou C++, des modules de gestion d'ensembles sont préprogrammés. Lorsqu'un ordre total existe sur les éléments de ces ensembles, ils sont en général gérés, en interne, par des arbres rouge et noir.

3.1 Arbres AVL

La famille des arbres AVL est nommée ainsi d'après leurs inventeurs, Adel'son-Velskii et Landis, qui les ont présentés en 1962. Au risque de paraître vieillots, nous décrivons ces arbres plus en détail parce que leur programmation peut être menée jusqu'au bout, et parce que les principes utilisés dans leur gestion se retrouvent dans d'autres familles plus complexes.

Un arbre binaire est un *arbre AVL* si, pour tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1.

Rappelons qu'une feuille est un arbre de hauteur 0, et que l'arbre vide a la hauteur -1. L'arbre vide, et l'arbre réduit à une feuille, sont des arbres AVL.

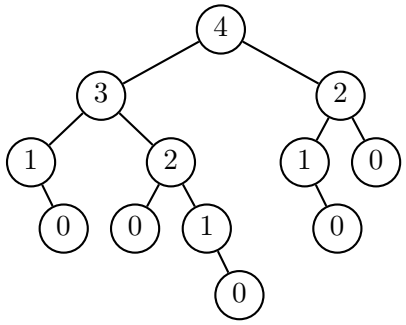


FIG. 3.10 – Un arbre AVL, avec les hauteurs aux nœuds.

L'arbre de la figure 3.10 porte, dans chaque nœud, la hauteur de son sous-arbre.

Un autre exemple est fourni par les *arbres de Fibonacci*, qui sont des arbres binaires A_n tels que les sous-arbres gauche et droit de A_n sont respectivement A_{n-1} et A_{n-2}. Les premiers arbres de Fibonacci sont donnés dans la figure 3.11.

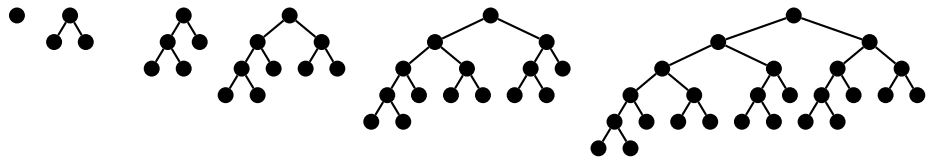


FIG. 3.11 – Les premiers arbres de Fibonacci.

L'intérêt des arbres AVL résulte du fait que leur hauteur est toujours logarithmique en fonction de la taille de l'arbre. En d'autres termes, la recherche, l'insertion et la suppression (sous réserve d'un éventuel rééquilibrage) se font en temps logarithmique. Plus précisément, on a la propriété que voici.

Proposition 12 *Soit A un arbre AVL ayant n nœuds et de hauteur h. Alors*

$$\log_2(1+n) \leqslant 1+h \leqslant \alpha \log_2(2+n)$$

avec $\alpha = 1/\log_2((1+\sqrt{5})/2) \leqslant 1.44$.

Preuve. On a toujours $n \leqslant 2^{h+1}-1$, donc $\log_2(1+n) \leqslant 1+h$. Soit $N(h)$ le nombre minimum de nœuds d'un arbre AVL de hauteur h. Alors

$$N(h) = 1 + N(h-1) + N(h-2)$$

car un arbre minimal aura un sous-arbre de hauteur $h-1$ et l'autre sous-arbre de hauteur seulement $h-2$. La suite $F(h) = 1+N(h)$ vérifie $F(0) = 2$, $F(1) = 3$, $F(h+2) = F(h+1)+F(h)$ pour $h \geqslant 0$, donc

$$F(h) = \frac{1}{\sqrt{5}}(\Phi^{h+3} - \Phi^{-(h+3)})$$

où $\Phi = (1+\sqrt{5})/2$. Il en résulte que $1+n \geqslant F(h) > \frac{1}{\sqrt{5}}(\Phi^{h+3}-1)$, soit en passant au logarithme en base Φ, $h+3 < \log_\Phi(\sqrt{5}(2+n)) < \log_2(2+n)/\log_2\Phi + 2$. □

Par exemple, pour un arbre AVL qui a 100000 nœuds, la hauteur est comprise entre 17 et 25. C'est le nombre d'opérations qu'il faut donc pour rechercher, insérer ou supprimer une donnée dans un tel arbre.

Rotations

Nous introduisons maintenant une opération sur les arbres appelée *rotation*. Les rotations sont illustrées sur la figure 3.12. Soit $A = (B, q, W)$ un arbre binaire tel que $B = (U, p, V)$. La *rotation gauche* est l'opération

$$((U, p, V), q, W) \to (U, p, (V, q, W))$$

et la rotation droite est l'opération inverse. Les rotations gauche (droite) ne sont donc définies que pour les arbres binaires non vides dont le sous-arbre gauche (resp. droit) n'est pas vide.

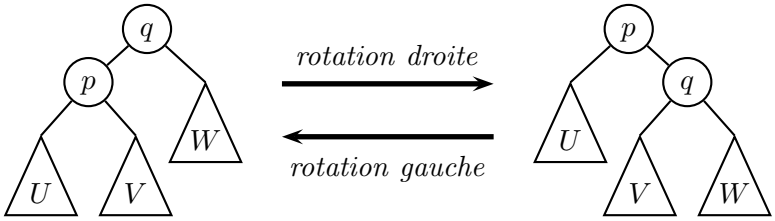


FIG. 3.12 – Rotation.

Remarquons en passant que pour l'arbre d'une expression arithmétique, si les symboles d'opération p et q sont les mêmes, les rotations expriment que l'opération est associative.

Les rotations ont la propriété de pouvoir être implantées en temps constant (voir ci-dessous), et de préserver l'ordre infixe. En d'autres termes, si A est un arbre binaire de recherche, tout arbre obtenu à partir de A par une suite de rotations gauche ou droite d'un sous-arbre de A reste un arbre binaire de recherche. En revanche, comme le montre la figure 3.13, la propriété AVL n'est pas conservée par rotation.

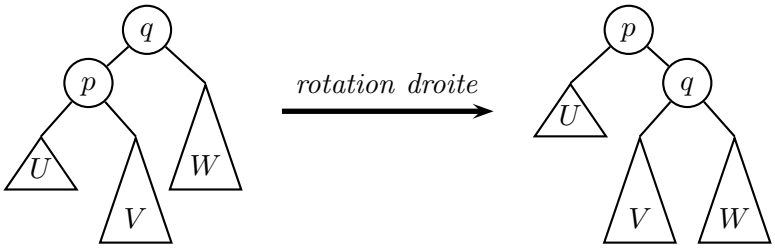


FIG. 3.13 – Les rotations ne préservent pas la propriété AVL.

Pour remédier à cela, on considère une *double rotation* qui est en fait composée de deux rotations. La figure 3.14 décrit une double rotation droite, et montre comment elle est composée d'une rotation gauche du sous-arbre gauche suivie d'un rotation droite. Plus précisément, soit $A = ((U, p, (V, q, W)), r, X)$ un arbre dont le sous-arbre gauche possède un sous-arbre droit. La *double rotation droite* est l'opération

$$A = ((U, p, (V, q, W)), r, X) \to A' = ((U, p, V), q, (W, r, X))$$

Vérifions qu'elle est bien égale à la composition de deux rotations. D'abord, une rotation gauche de $B = (U, p, (V, q, W))$ donne $B' = ((U, p, V), q, W)$, et l'arbre $A = (B, r, X)$ devient $A'' = (B', r, X)$; la rotation droite de A'' donne en effet A'. On voit qu'une double rotation droite

diminue la hauteur relative des sous-arbres V et W, et augmente celle de X. La double rotation gauche est définie de manière symétrique.

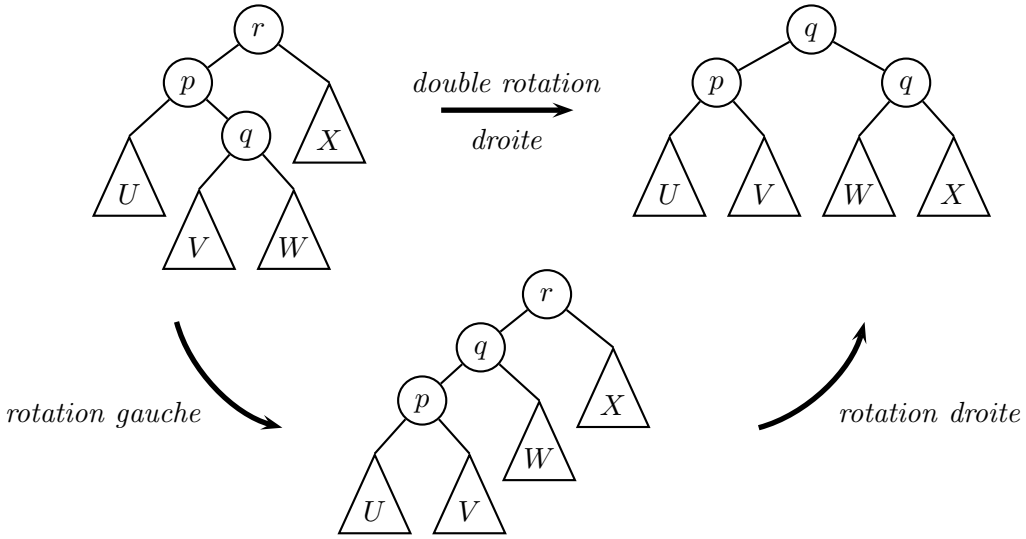


FIG. 3.14 – Rotations doubles.

Implantation des rotations

Voici une implantation non destructive d'une rotation gauche.

```
static Arbre rotationG(Arbre a)   // non destructive
{
  Arbre b = a.filsD;
  Arbre c = new Arbre(a.filsG, a.contenu, b.filsG);
  return new Arbre(c, b.contenu, b.filsD);
}
```

La fonction suppose que le sous-arbre gauche, noté `b`, n'est pas vide. La rotation gauche destructive est aussi simple à écrire.

```
static Arbre rotationG(Arbre a)   // destructive
{
  Arbre b = a.filsD;
  a.filsD = b.filsG;
  b.filsG = a;
  return b;
}
```

Les double rotations s'écrivent par composition.

Insertion et suppression dans un arbre AVL

L'insertion et la suppression dans un arbre AVL peuvent transformer l'arbre en un arbre qui ne satisfait plus la contrainte sur les hauteurs. Dans la figure 3.15, un nœud portant l'étiquette 50 est inséré dans l'arbre de gauche. Après insertion, on obtient l'arbre du milieu qui n'est plus AVL. Une double rotation autour de la racine suffit à rééquilibrer l'arbre.

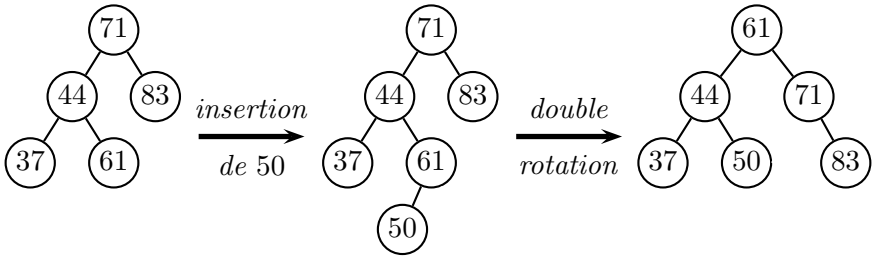


FIG. 3.15 – Insertion suivie d'une double rotation.

Cette propriété est générale. Après une insertion (respectivement une suppression), il suffit de rééquilibrer l'arbre par des rotations ou double rotations le long du chemin qui conduit à la feuille où l'insertion (respectivement la suppression) a eu lieu. L'algorithme est le suivant :

Algorithme. *Soit A un arbre, G et D ses sous-arbres gauche et droit. On suppose que $|h(G) - h(D)| = 2$. Si $h(G) - h(D) = 2$, on fait une rotation droite, mais précédée d'une rotation gauche de G si $h(g) < h(d)$ (on note g et d les sous-arbres gauche et droit de G). Si $h(G) - h(D) = -2$ on opère de façon symétrique.*

On peut montrer en exercice qu'il suffit d'une seule rotation ou double rotation pour rééquilibrer un arbre AVL après une insertion. Cette propriété n'est plus vraie pour une suppression.

Implantation : la classe Avl

Pour l'implantation, nous munissons chaque nœud d'un champ supplémentaire qui contient la hauteur de l'arbre dont il est racine. Pour une feuille par exemple, ce champ a la valeur 0. Pour l'arbre vide, qui est représenté par null et qui n'est donc pas un objet, la hauteur vaut -1. La méthode H sert à simplifier l'accès à la hauteur d'un arbre.

```
class Avl
{
  int contenu;
  int hauteur;
  Avl filsG, filsD;

  Avl(Avl g, int c, Avl d)
  {
    filsG = g;
    contenu = c;
    filsD = d;
    hauteur = 1 + Math.max(H(g),H(d));
  }

  static int H(Avl a)
  {
    return (a == null) ? -1 : a.hauteur;
  }

  static void calculerHauteur(Avl a)
  {
    a.hauteur = 1 + Math.max(H(a.filsG), H(a.filsD));
  }
  ...
}
```

La méthode calculerHauteur recalcule la hauteur d'un arbre à partir des hauteurs de ses sous-arbres. L'usage de H permet de traiter de manière unifiée le cas où l'un de ses sous-arbres serait l'arbre vide. Les rotations sont reprises de la section précédente. On utilise la version non destructive qui réévalue la hauteur. Ces méthodes et les suivantes font toutes partie de la classe Avl.

```
static Avl rotationG(Avl a)
{
  Avl b = a.filsD;
  Avl c = new Avl(a.filsG, a.contenu, b.filsG);
  return new Avl(c, b.contenu, b.filsD);
}
```

La méthode principale implante l'algorithme de rééquilibrage exposé plus haut.

```
static Avl equilibrer(Avl a)
{
  a.hauteur = 1 + Math.max(H(a.filsG), H(a.filsD));
  if(H(a.filsG) - H(a.filsD) == 2)
  {
    if (H(a.filsG.filsG) < H(a.filsG.filsD))
      a.filsG = rotationG(a.filsG);
    return rotationD(a);
  } //else version symétrique
  if (H(a.filsG) - H(a.filsD) == -2)
  {
    if (H(a.filsD.filsD) < H(a.filsD.filsG))
      a.filsD = rotationD(a.filsD);
    return rotationG(a);
  }
  return a;
}
```

Il reste à écrire les méthodes d'insertion et de suppression, en prenant soin de rééquilibrer l'arbre à chaque étape. On reprend simplement les méthodes déjà écrites pour un arbre binaire de recherche général. Pour l'insertion, on obtient

```
static Avl inserer(int x, Avl a)
{
  if (a == null)
    return new Avl(null, x, null);
  if (x < a.contenu)
    a.filsG = inserer(x, a.filsG);
  else if (x > a.contenu)
    a.filsD = inserer(x, a.filsD);
  return equilibrer(a); //seul changement
}
```

La suppression s'écrit comme suit

```
static Avl supprimer(int x, Avl a)
{
  if (a == null)
    return a;
  if (x == a.contenu)
```

```
    return supprimerRacine(a);
  if (x < a.contenu)
    a.filsG = supprimer(x, a.filsG);
  else
    a.filsD = supprimer(x, a.filsD);
  return equilibrer(a); // seul changement
}

static Avl supprimerRacine(Avl a)
{
  if (a.filsG == null && a.filsD == null)
    return null;
  if (a.filsG == null)
    return equilibrer(a.filsD);
  if (a.filsD == null)
    return equilibrer(a.filsG);
  Avl b = dernierDescendant(a.filsG);
  a.contenu = b.contenu;
  a.filsG = supprimer(a.contenu, a.filsG);
  return equilibrer(a); // seul changement
  }

static Avl dernierDescendant(Avl a) // inchangée
{
  if (a.filsD == null)
    return a;
  return dernierDescendant(a.filsD);
}
```

3.2 B-arbres et arbres a-b

Dans cette section, nous décrivons de manière succinte les arbres a-b. Il s'agit d'une des variantes d'arbres équilibrés qui ont la propriété que toutes leurs feuilles sont au même niveau, les nœuds internes pouvant avoir un nombre variable de fils (ici entre a et b). Dans cette catégorie d'arbres, on trouve aussi les B-arbres et en particulier les arbres 2-3-4. Les arbres rouge et noir (ou bicolores) sont semblables.

L'intérêt des arbres équilibrés est qu'ils permettent des modifications en temps logarithmique. Lorsque l'on manipule de très grands volumes de données, il survient un autre problème, à savoir l'accès proprement dit aux données. En effet, les données ne tiennent pas en mémoire vive, et les données sont donc accessibles seulement sur la mémoire de masse, un disque en général. Or, un seul accès disque peut prendre, en moyenne, environ autant de temps que 200 000 instructions. Les B-arbres ou les arbres a-b servent, dans ce contexte, à minimiser les accès au disque.

Un disque est divisé en pages (par exemple de taille 512, 2048, 4092 ou 8192 octets). La page est l'unité de transfert entre mémoire centrale et disque. Il est donc rentable de grouper les données par blocs, et de les manipuler de concert.

Les données sont en général repérées par des clés, qui sont rangées dans un arbre. Si chaque accès à un nœud requiert un accès disque, on a intérêt à avoir des nœuds dont le nombre de fils est voisin de la taille d'une page. De plus, la hauteur d'un tel arbre — qui mesure le nombre d'accès disques nécessaire — est alors très faible. En effet, si chaque nœud a de l'ordre de 1000 fils, il suffit d'un arbre de hauteur 3 pour stocker un milliard de clés.

Nous considérons ici des arbres de recherche qui ne sont plus binaires, mais d'arité plus grande. Chaque nœud interne d'un tel arbre contient, en plus des références vers ses sous-arbres,

des *balises*, c'est-à-dire des valeurs de clé qui permettent de déterminer le sous-arbre où se trouve l'information cherchée. Plus précisément, si un nœud interne possède $d+1$ sous-arbres $A_0, \ldots, A_d$, alors il est muni de d balises $k_1, \ldots, k_d$ telles que

$$c_0 \leqslant k_1 < c_1 \leqslant \cdots \leqslant k_d < c_d$$

pour toute séquence de clés $(c_0, \ldots, c_d)$, où chaque c_i est une clé du sous-arbre A_i (cf. figure 3.16).

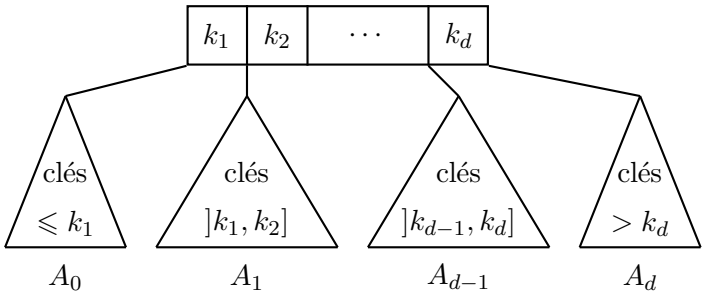


FIG. 3.16 – Un nœud interne muni de balises et ses sous-arbres.

Il en résulte que pour chercher une clé c, on détermine le sous-arbre A_i approprié en déterminant lequel des intervalles $]-\infty, k_1]$, $]k_1, k_2]$,. . .,$]k_{d-1}, k_d]$, $]k_d, \infty[$ contient la clé.

Arbres *a-b*

Soient $a \geqslant 2$ et $b \geqslant 2a-1$ deux entiers. Un arbre *a-b* est un arbre de recherche tel que
- les feuilles ont toutes la même profondeur,
- la racine a au moins 2 fils (sauf si l'arbre est réduit à sa racine) et au plus b fils,
- les autres nœuds internes ont au moins a et au plus b fils.

Les arbres 2-3 sont les arbres obtenus quand a et b prennent leurs valeurs minimales : tout nœud interne a alors 2 ou 3 fils.

Les B-arbres sont comme les arbres *a-b* avec $b = 2a-1$ mais avec une interprétation différente : les informations sont aussi stockées aux nœuds internes, alors que, dans les arbres que nous considérons, les clés aux nœuds internes ne servent qu'à la navigation.

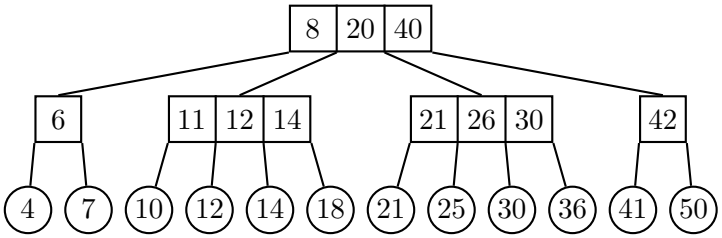


FIG. 3.17 – Un arbre 2-4. Un nœud a entre 2 et 4 fils.

L'arbre de la figure 3.17 représente un arbre 2-4. Les nœuds internes contiennent les balises, et les feuilles contiennent les clés. L'intérêt des arbres *a-b* est justifié par la proposition suivante.

Proposition 13 *Si un arbre a-b de hauteur h contient n feuilles, alors*

$$\log n / \log b \leqslant h < 1 + \log(n/2) / \log a\,.$$

Preuve. Tout nœud a au plus b fils. Il y a donc au plus b^h feuilles. Tout nœud autre que la racine a au moins a fils, et la racine en a au moins 2. Au total, il y a au moins $2a^{h-1}$ feuilles, donc $2a^{h-1} \leqslant n \leqslant b^h$. □

Il résulte de la proposition précédente que la hauteur d'un arbre a-b ayant n feuilles est en $O(\log n)$. La complexité des algorithmes décrit ci-dessous (insertion et suppression) sera donc elle aussi en $O(\log n)$.

Insertion dans un arbre a-b

La recherche dans un arbre a-b repose sur le même principe que celle utilisée pour les arbres binaires de recherche : on parcourt les balises du nœud courant pour déterminer le sous-arbre dans lequel il faut poursuivre la recherche. Pour l'insertion, on commence par déterminer, par une recherche, l'emplacement de la feuille où l'insertion doit avoir lieu. On insère alors une nouvelle feuille, et une balise appropriée : la balise est la plus petite valeur de la clé à insérer et de la clé à sa droite.

Reste le problème du rééquilibrage qui se pose lorsque le nombre de fils d'un nœud dépasse le nombre autorisé. Si un nœud a $b+1$ fils, alors il est éclaté en deux nœuds qui se partagent les fils de manière équitable : le premier nœud reçoit les $\lfloor (b+1)/2 \rfloor$ fils de gauche, le deuxième les fils de droite. Noter que $b+1 \geqslant 2a$, et donc chaque nouveau nœud aura au moins a fils. Les balises sont également partagées, et la balise centrale restante est transmise au nœud père, pour qu'il puisse à son tour procéder à l'insertion des deux fils à la place du nœud éclaté. L'insertion

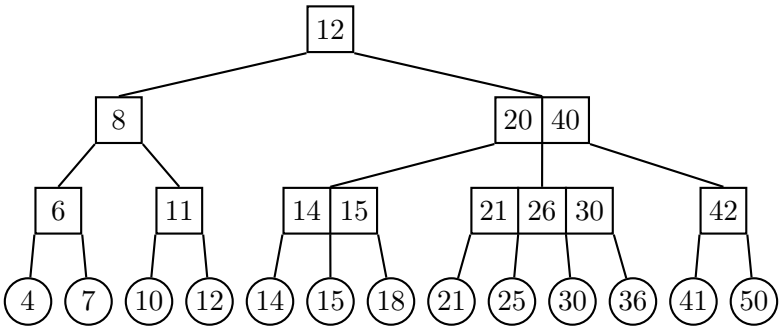


FIG. 3.18 – Un arbre 2-4. Un nœud a entre 2 et 4 fils.

de la clé 15 dans l'arbre 3.17 produit l'arbre de la figure 3.18. Cette insertion se fait par un double éclatement. D'abord, le nœud aux balises $11, 12, 14$ de l'arbre 3.17 est éclaté en deux. Mais alors, la racine de l'arbre 3.17 a un nœud de trop. La racine elle-même est éclatée, ce qui fait augmenter la hauteur de l'arbre.

Il est clair que l'insertion d'une nouvelle clé peut au pire faire éclater les nœuds sur le chemin de son lieu d'insertion à la racine — et la racine elle-même. Le coût est donc borné logarithmiquement en fonction du nombre de feuilles dans l'arbre.

Suppression dans un arbre a-b

Comme d'habitude, la suppression est plus complexe. Il s'agit de fusionner un nœud avec un nœud frère lorsqu'il n'a plus assez de fils, c'est-à-dire si son nombre de fils descend au dessous de a. Mais si son frère (gauche ou droit) a beaucoup de fils, la fusion des deux nœuds risque de conduire à un nœud qui a trop de fils et qu'il faut éclater. On groupe ces deux opérations sous la forme d'un *partage* (voir figure 3.19).

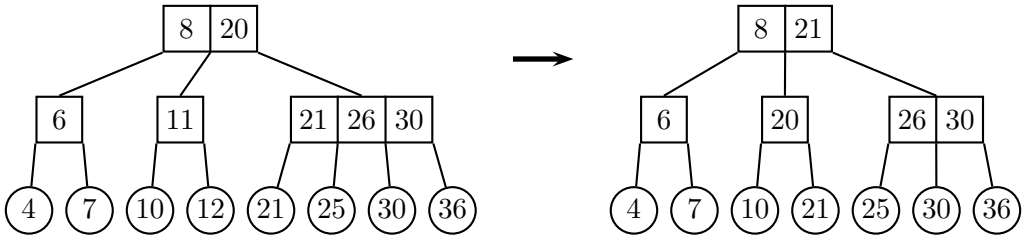


FIG. 3.19 – La suppression de 12 est absorbée par un partage.

Plus précisément, l'algorithme est le suivant :
- Supprimer la feuille, puis la balise figurant sur le chemin de la feuille à la racine.
- Si les nœuds ainsi modifiés ont toujours a fils, l'arbre est encore a-b. Si un nœud possède $a-1$ fils examiner les frères *adjacents*.
  - Si l'un des frères possède au moins $a+1$ fils, faire un partage avec ce frère.
  - Sinon, les frères adjacents ont a fils, et la fusion avec l'un des deux produit un nœud ayant $2a-1 \leqslant b$ fils.

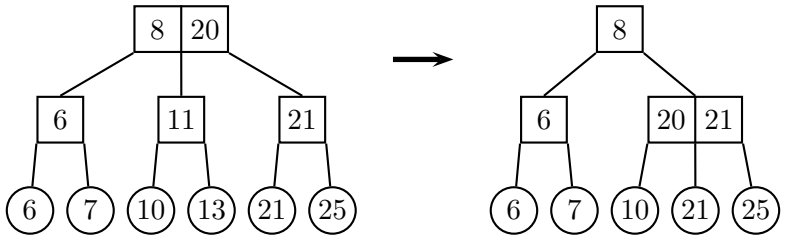


FIG. 3.20 – La suppression de 13 entraîne la fusion des deux nœuds de droite.

Là encore, le coût est majoré par un logarithme du nombre de feuilles. Il faut noter qu'il s'agit du comportement dans le cas le plus défavorable. On peut en effet démontrer

Proposition 14 *On considère une suite quelconque de n insertions ou suppressions dans un arbre* 2-4 *initialement vide. Alors le nombre total d'éclatements, de partage et de fusions est au plus* $3n/2$.

Ceci signifie donc qu'en moyenne, $1,5$ opérations suffisent pour rééquilibrer l'arbre, et ce, quel que soit sa taille ! Des résultats analogues valent pour des valeurs de a et b plus grandes.

Chapitre VI

Applications

Les arbres sont utilisés très fréquemment en informatique et nous avons déjà présenté plusieurs de leurs applications. Nous avons sélectionné dans ce chapitre trois applications supplémentaires. La première est la recherche dans un nuage de points, qui relève de la recherche dans une base de données. La seconde est l'utilisation de tétrarbres en géométrie algorithmique. La troisième est une application spécifique des tétrarbres à un problème bien connu des physiciens, le problème des N corps.

1 Recherche dans un nuage de points

Nous considérons le problème suivant : étant donné un ensemble de points dans le plan, déterminer ceux qui sont dans une région donnée.

Par exemple, on veut trouver les villes à moins de 100 km de Tours, ou déterminer les personnes entre 25 et 29 ans qui gagnent entre 1000 et 2000 euros par mois.

Ce deuxième exemple illustre bien la situation : on dispose d'une base de données qui chacune, correspond à un point dans un espace à n dimensions. On formule, pour cette base, des *requêtes* qui reviennent en général à définir un parallélépipède, et on cherche les données à l'intérieur de ce volume.

Nous allons nous restreindre ici au plan (on verra que cela n'est pas vraiment une restriction). On suppose que les requêtes définissent des rectangles, et nous allons compter le nombre de points qui se trouvent à l'intérieur d'un rectangle donné. Le décompte est plus simple que l'énumération parce que dans ce cas, il suffit d'additionner des nombres au lieu de concaténer des listes.

Nous supposons que les données de la base ne changent pas. On peut donc effectuer un *prétraitement* sur la base pour la mettre dans une forme particulière qui permet de satisfaire rapidement les requêtes. Nous allons supposer que les données sont rangées dans un arbre binaire dont nous allons décrire les particularités. Il en résultera qu'une requête pourra être satisfaite en temps $O(\log n)$ pour un nuage à n points, contre $O(n)$ pour une recherche naïve. Le coût du prétraitement est de $O(n \log n)$, et la recherche par arbre est donc avantageuse dès qu'il y a $\log n$ requêtes (les constantes doivent être ajustées bien sûr).

1.1 Construction de l'arbre

L'arbre est construit par subdivision successive du plan par des demi-droites passant par un point. La première division est faite par une droite, disons horizontale, passant par un point. Dans l'exemple, le point est A. L'ensemble des points est partitionné en points *au-dessus* du point de partage — ces points seront dans le sous-arbre *droit* et les points *au-dessous* du point de partage qui seront représentés dans le sous-arbre gauche. À l'étape suivante, de manière générale une étape sur deux, le partage se fait par une demi-droite *verticale*, le point à gauche étant dans

le sous-arbre gauche, et les autres dans le sous-arbre droit. Dans l'exemple de la figure 1.1, le deuxième point de partage est B, et le points C et F seront dans le sous-arbre gauche.

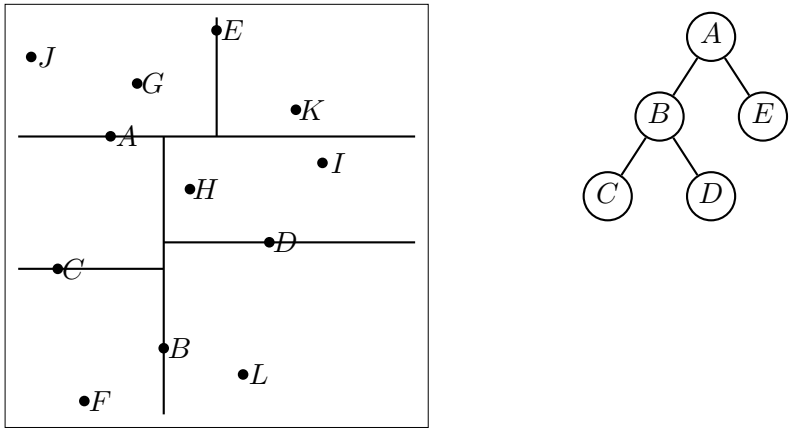


FIG. 1.1 – Un nuage de douze points et une partie de son arbre.

Voici une réalisation. On utilise des arbres binaires dont le contenu des nœuds sont des points. La classe Point est définie simplement ainsi :

```
class Point
{
  int x, y;

  Point (int a, int b)
  {
    x = a;
    y = b;
  }
}

class Arbre
{
  Point p;
  Arbre filsG, filsD;
  Arbre(Arbre g, Point v, Arbre d)
  {
    filsG = g;
    p = v;
    filsD = d;
  }
}
```

On ajoute un point à l'arbre par l'algorithme d'insertion usuel. Le partage entre sous-arbre gauche et droit se fait alternativement selon l'abscisse (quand la ligne de partage est verticale) ou selon les ordonnées (quand la ligne est horizontale). On fait donc appel à une méthode qui dépend d'un booléen indiquant la situation dans laquelle on se trouve :

```
static Arbre ajouter(Point p, Arbre a, boolean vertical)
{
  if (a == null)
    return new Arbre(null, p, null);
```

```
    boolean estAGauche =
      (vertical) ? (p.x < a.p.x) : (p.y < a.p.y);
    if (estAGauche)
    {
      Arbre g = ajouter(p, a.filsG, !vertical);
      return new Arbre(g, a.p, a.filsD);
    }
    else
    {
      Arbre d = ajouter(p, a.filsD, !vertical);
      return new Arbre(a.filsG, a.p, d);
    }
  }
```

Pour l'insertion, on utilise la méthode suivante :

```
static Arbre ajouter(Point p, Arbre a)
{
  return ajouter(p, a, false);
}
```

puisque en effet le niveau de la racine est horizontal.

1.2 Recherche de points

Etant donné un rectangle, on cherche (figure 1.2) le nombre de points du nuage contenu dans ce rectangle.

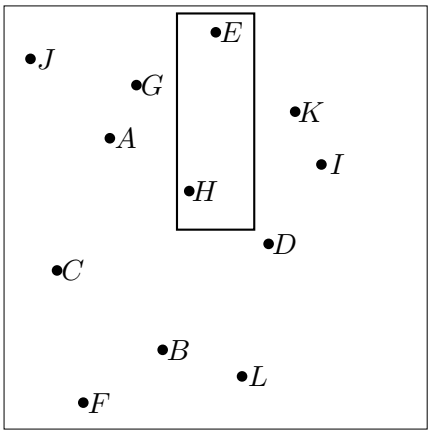


FIG. 1.2 – Nombre de points dans le rectangle.

Le calcul se fait récursivement, en comptant le nombre de points du sous-arbre gauche et du sous-arbre droit qui sont à l'intérieur du rectangle, et en ajoutant le point à la racine s'il convient. La classe des rectangles est sans surprise

```
class Rectangle
{
  int g, d, b, h;
}
```

Et voici comment on compte :

```
static int compter(Rectangle r, Arbre a, boolean vertical)
{
  if (a == null)
    return 0;
  int total = 0;
  boolean aGauche = r.g <= a.p.x;
  boolean aDroite = a.p.x < r.d;
  boolean enBas = r.b <= a.p.y;
  boolean enHaut = a.p.y < r.h;
  boolean min, max;
  min = (vertical) ? aGauche : enBas;
  max = (vertical) ? aDroite : enHaut;
  if (min)
    total += compter(r, a.filsG, !vertical);
  if (aGauche && aDroite && enBas && enHaut)
    total++;
  if (max)
    total += compter(r, a.filsD, !vertical);
  return total;
}
```

Les tests faits dans la fonction servent à éliminer le sous-arbre gauche (respectivement droit) : par exemple, si le partage est vertical, il est inutile d'explorer le sous-arbre gauche si l'abscisse `a.p.x` du point à la racine de l'arbre `a` est inférieure au bord gauche du rectangle.

2 Tétrarbres

Les *tétrarbres* (en anglais « quadtrees ») sont une structure de données utilisée pour structurer des données dans le plan. Une version tridimensionnelle (« octrees ») existe. Les applications sont nombreuses en infographie, en compression d'images, ou en géométrie algorithmique. Nous donnons, dans la section suivante, une application en physique.

Une image de taille $2^n \times 2^n$ est subdivisée en quadrants jusqu'à l'obtention de carrés monochromes. Les quadrants sont lus dans l'ordre indiqué sur la figure 2.3.

1	2
0	3

FIG. 2.3 – L'ordre de lecture des quadrants.

La hiérarchie ainsi obtenue est représentée dans un arbre où chaque nœud interne a 4 fils. Un tel arbre est un *tétrarbre*. Considérons l'image de la figure 2.4. Elle est représentée par l'arbre de la figure 2.5. Nous considérons ici des figures bicolores. Une image peut alors être représentée par une matrice de booléens. Un tétrarbre est soit une feuille monochrome, soit un nœud avec quatre sous-arbres. On est donc conduit à la définition récursive suivante :

```
class Tetrarbre
{
  boolean couleur;
  boolean estFeuille;
  Tetrarbre so, no, ne, se;
```

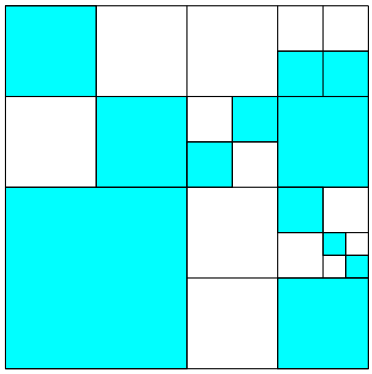


FIG. 2.4 – Une image 16×16.

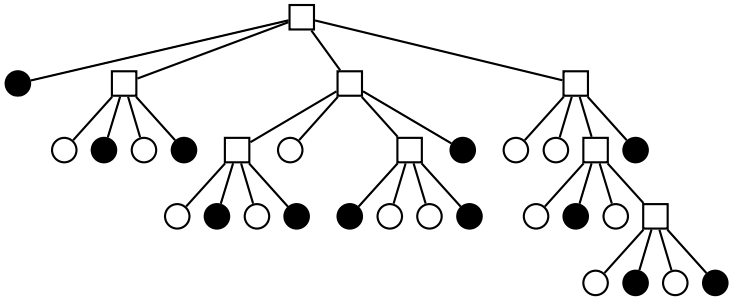


FIG. 2.5 – Le tétrarbre associé à l'image précédente.

```
  Tetrarbre(boolean couleur)
  {
    this.estFeuille = true;
    this.couleur = couleur;
  }

  Tetrarbre(Tetrarbre so, Tetrarbre no, Tetrarbre ne, Tetrarbre se)
  {
    this.estFeuille = false;
    this.so = so; this.no = no; this.ne = ne; this.se = se;
  }
}
```

Les deux constructeurs servent à former une feuille ou un nœud interne. Pour la traduction d'une image en un tétrarbre, nous supposons que cette image est donnée sous la forme d'un tableau booléen boolean[][] image. Si la taille de l'image est 1, elle est réduite à un seul pixel, et donne naissance à un tétrarbre réduit à une feuille. Sinon, l'image est divisée en quatre quadrants, et un tétrarbre est construit pour chaque quadrant. Ces quatre arbres sont combinés en un tétrarbre, sauf s'ils sont tous les quatre des feuilles de la même couleur. Dans ce deuxième cas, une seule feuille de cette couleur est créée. Voici le test utile pour cette vérification :

```
static boolean estMonochrome(Tetrarbre so, Tetrarbre no,
                             Tetrarbre ne, Tetrarbre se)
{
  return
    so.estFeuille && no.estFeuille && ne.estFeuille && se.estFeuille
```

```
      && so.couleur == no.couleur && no.couleur == ne.couleur
      && ne.couleur == se.couleur;
  }
```

On peut alors écrire la méthode de transformation :

```
  static Tetrarbre faireArbre(boolean[][] image)
  {
    return faireArbre(image.length, 0, 0, image);
  }

  static Tetrarbre faireArbre(int taille, int i, int j, boolean[][] image)
  {
    if (taille == 1)
      return new Tetrarbre(image[i][j]);
    int t = taille/2;
    Tetrarbre so = faireArbre(t, i+t, j ,image);
    Tetrarbre no = faireArbre(t, i, j ,image);
    Tetrarbre ne = faireArbre(t, i, j+t ,image);
    Tetrarbre se = faireArbre(t, i+t, j+t ,image);
    if (estMonochrome(so, no, ne, se))
      return new Tetrarbre(no.couleur);
    return new Tetrarbre(so, no, ne, se);
  }
```

Pour écrire un tétrarbre, un codage simple du parcours préfixe est utile : chaque sous-arbre qui n'est pas une feuille est écrit entre crochets, et une feuille est représentée par sa couleur. On obtient ainsi, pour l'arbre de la figure 2.5, le code

```
[X[-X-X][[X-X-]-[X--X]X][--[-X-[-X-X]]X]]
```

Ce codage est construit par la méthode suivante :

```
  static String parcours(Tetrarbre a)
  {
    if (a == null)
      return "" ;
    if (a.estFeuille)
      return (a.couleur) ? "X" : "-";
    return "[" + parcours(a.so) + parcours(a.no) +
      parcours(a.ne) + parcours(a.se) + "]";
  }
```

On peut encore raccourcir la chaîne en omettant les crochets fermants qui sont inutiles mais qui rendent la lecture plus claire.

3 Le problème des N corps

Dans cet exemple, nous montrons l'utilisation des tétrarbres dans la simulation du problème des N corps. Pour simplifier la présentation de l'algorithme, nous travaillerons dans un univers plan, mais l'adaptation à un univers 3D n'offre pas de difficultés. Le problème des N corps consiste à calculer les trajectoires de N corps ou particules qui interagissent entre eux sous l'effet de la gravité. Ce problème n'a pas de solution analytique dans le cas général, on a donc recours à une simulation numérique.

Le principe de ce type de simulation est de calculer les forces qui s'exercent, à un instant t donné, sur chaque corps, puis d'estimer les vitesses et les positions de tous les corps à l'instant

$t + dt$. Pour calculer les forces, on doit, pour chacun des N corps, calculer $N - 1$ interactions gravitationnelles. Le coût du calcul exact des forces est donc de l'ordre de N^2. Ceci limite sérieusement l'intérêt de la simulation.

Une approximation physique simple va permettre une amélioration de l'efficacité de la simulation. Pour cela on constate que pour un corps c, l'attraction exercée sur c par un ensemble C de corps peut être approchée, pourvu que C soit suffisamment éloigné de c, par l'attraction exercée sur c par le centre de masse de C (barycentre des corps de C muni de la masse de ces corps). C'est cette approche qui va être implantée.

3.1 Données

L'*univers* $\mathcal{U}$ est constitué de N corps, contenus dans un carré de côté $d_{\mathcal{U}}$ et centré à l'origine. L'univers est structuré en tétrarbre de sorte que chaque cellule de l'arbre contienne 0 ou 1 corps. Pour cela, l'arbre est d'abord assimilé à la cellule formée du carré d'origine. Chaque cellule qui contient deux corps ou plus est subdivisée en quatre sous-cellules carrées de même taille, la subdivision s'arrête lorsque chaque cellule de l'arbre contient 0 ou 1 corps.

La figure 3.6 présente par exemple douze corps organisés en tétrarbre.

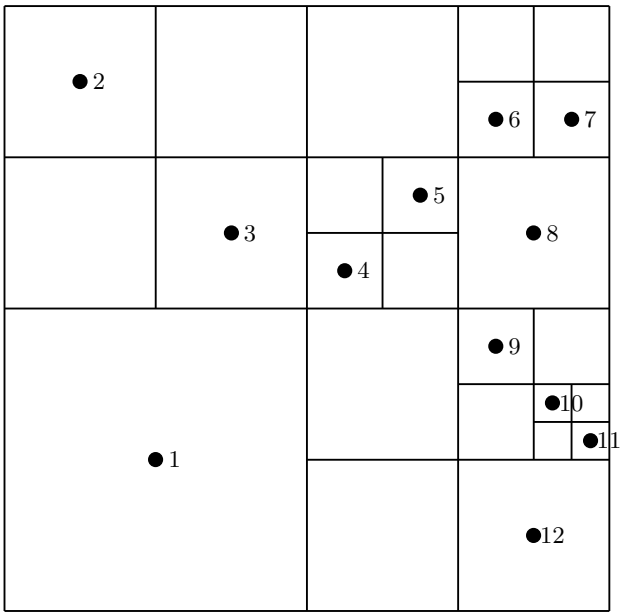


FIG. 3.6 – Douze corps organisés en tétrarbre.

3.2 Vecteurs

Un vecteur de l'espace plan est représenté par une classe, munie de quelques opérations sur les vecteurs :

```
class Vecteur
{
  double x, y;

  Vecteur(double x, double y)
  {
    this.x = x;
    this.y = y;
```

```
}
Vecteur(){}
void increment(Vecteur b)
{
  x += b.x;
  y += b.y;
}
void increment(double s, Vecteur b)
{
  x += s * b.x;
  y += s * b.y;
}
void increment(double s)
{
  x *= s;
  y *= s;
}
static Vecteur somme(Vecteur a, Vecteur b)
{
  return new Vecteur(a.x + b.x, a.y + b.y);
}
static Vecteur diff(Vecteur a, Vecteur b)
{
  return new Vecteur(a.x - b.x, a.y - b.y);
}

double carreNorme()
{
  return x*x + y*y;
}
Vecteur calculerOrigine(int s, double taille)
{
  double t = taille/4;
  Vecteur n;
  switch (s)
  {
    case 0 :
      n = new Vecteur(x - t, y - t);
    break;
    case 1:
      n = new Vecteur(x - t, y + t);
    break;
    case 2:
      n = new Vecteur(x + t, y + t);
    break;
    default:
      n = new Vecteur(x + t, y - t);
  }
```

```
    return n;
  }
}
```

Les méthodes sont simples à comprendre, à l'exception de la méthode calculerOrigine dont le sens et l'utilité seront expliqués plus loin. On a utilisé la possibilité de surcharger les méthodes dans increment qui vient en trois versions dont chacune a son utilité. Les méthodes statiques somme et diff retournent un nouveau vecteur, alors que les méthodes increment modifient le vecteur appelant.

3.3 Corps

Un *corps* est donné par sa masse, et trois vecteurs : sa position, sa vitesse et son accélération.

```
class Corps
{
  double masse;
  Vecteur position, vitesse, acceleration;

  Corps(double masse, Vecteur position,
    Vecteur vitesse, Vecteur acceleration)
  {
    this.masse = masse;
    this.position = position;
    this.vitesse = vitesse;
    this.acceleration = acceleration;
  }

  Vecteur acceleration(Corps d) {...}
  Vecteur acceleration(Arbre a) {...}
}
```

Les méthodes qui restent à écrire concernent le calcul des forces exercées sur un corps par les autres corps. Ce sera fait plus loin.

3.4 Arbre

Un nœud d'un tétrarbre est soit une feuille, soit un nœud d'arité 4. Une feuille est soit vide, et représentée par null, soit contient un seul corps. Un nœud interne a quatre arbres fils, et contient le centre de masse des corps de son arbre. Il apparaît aussi utile de stocker, dans chaque nœud non vide, la géométrie de la cellule qu'il représente, c'est-à-dire le centre et le côté du carré qu'il couvre. Ceci amène à la définition suivante :

```
class Arbre
{
  double cote; // cote du carre
  Vecteur origine; // centre du carré
  boolean estFeuille; // si c'est une feuille
  Corps corps; // si c'est un noeud
  double masse; // du barycentre
  Vecteur barycentre;
  Arbre[] fils = new Arbre[4];

  Arbre(double cote, Vecteur origine, Corps corps) {...}
```

```
  static int indiceQuadrant(Vecteur o, Vecteur p) {...}
  void insererCorps(Corps c) {...}
  void barycentres() {...}
  Vecteur gravitation(Corps c) {...}
}
```

La création d'une feuille est prise en charge par un constructeur :

```
Arbre(double cote, Vecteur origine, Corps corps)
{
  estFeuille = true;
  this.cote = cote;
  this.origine = origine;
  this.corps = corps;
}
```

La création d'un nœud se fait par transformation d'une feuille. C'est en effet l'insertion d'un deuxième corps dans la cellule jusqu'alors occupée par un seul corps qui transforme cette feuille en nœud. La transformation se fait en deux étapes : d'abord, le corps contenu dans la feuille est affecté à l'un des quatre sous-arbres ; ensuite, le deuxième corps est inséré. Cette insertion peut récursivement subdiviser l'arbre, si le nouveau corps se trouve dans une cellule déjà occupée. La méthode d'insertion d'un corps dans l'arbre s'écrit :

```
void insererCorps(Corps c)
{
  Vecteur origineF; // origine fils

  if (estFeuille)
  {
    estFeuille = false;
    int t = indiceQuadrant(origine, corps.position);
    origineF = origine.calculerOrigine(t, cote);
    fils[t] = new Arbre(cote/2, origineF, corps);
  }
  int s = indiceQuadrant(origine, c.position);
  if (fils[s] == null)
  {
    origineF = origine.calculerOrigine(s, cote);
    fils[s] = new Arbre(cote/2, origineF, c);
  }
  else
    fils[s].insererCorps(c);
}
```

Si l'insertion se fait dans une feuille, on réaménage la feuille en un nœud interne. Pour cela, on calcule d'abord l'indice du quart de carré où est situé le corps du nœud, par indiceQuadrant. On calcule l'origine de ce quart de carré par calculerOrigine, et on insère une nouvelle feuille à cet endroit. Si l'insertion se fait dans un nœud, la démarche est la même : calcul du quart de carré où sera situé ce corps ; si ce sous-arbre est vide, insertion d'une feuille, et sinon appel récursif de l'insertion sur ce sous-arbre.

La méthode indiceQuadrant est une méthode auxiliaire statique de la classe Arbre, alors que la méthode calculerOrigine se trouve dans la classe Vecteur :

```
static int indiceQuadrant(Vecteur o, Vecteur p)
{
```

```
    if (p.x < o.x)
      return (p.y < o.y) ? 0 : 1;
    return (p.y > o.y) ? 2 : 3;
  }
```

Le calcul des centres de masse consiste en le calcul de la masse totale et du barycentre. Ce calcul se fait très bien récursivement :

```
  void barycentres()
  {
    if (estFeuille)
    {
      masse = corps.masse;
      barycentre = corps.position;
    }
    else
    {
      double masseTot = 0;
      Vecteur bary = new Vecteur();
      for (int i = 0; i < 4; i++)
        if (fils[i] != null)
        {
          fils[i].barycentres();
          masseTot += fils[i].masse;
          bary.increment(fils[i].masse, fils[i].barycentre);
        }
      bary.increment(1/masseTot);
      masse = masseTot;
      barycentre = bary;
    }
  }
```

On peut tout juste noter qu'après ce calcul, le centre de masse est aussi défini pour une feuille. Ceci va simplifier les calculs à venir.

3.5 Calcul des forces

Selon les lois de la mécanique, l'accélération d'un corps c soumis à l'attraction des autres corps de l'univers est calculée en sommant les contributions de tous les autres corps :

$$a_c = \sum_{d \neq c} a_{c \leftarrow d}$$

avec

$$a_{c \leftarrow d} = \frac{m_d}{\|p_d - p_c\|^3}(p_d - p_c)$$

Ici et dans la suite, p_c désigne la position du corps c et m_d la masse du corps d.

On considère maintenant un corps c et un nœud du tétrarbre, correspondant à une cellule de côté δ. Soit C l'ensemble des corps contenus dans cette cellule, p_C son barycentre et m_C la somme de leurs masses. On estime que la force gravitationnelle exercée par les corps de C sur c peut être assimilée à celle exercée par leur centre de masse quand la longueur δ est inférieure à $\alpha\|p_C - p_c\|$, où α est une constante. L'expression

$$\sum_{d \in C} a_{c \leftarrow d}$$

est alors remplacée par

$$a_{c \leftarrow C} = \frac{m_C}{\|p_d - p_C\|^3}(p_d - p_C)\,.$$

Voici d'abord deux méthodes acceleration de la classe Corps. Elles calculent l'accélération exercée sur le corps appelant par un corps d ou par le centre de masse de la racine de l'arbre a.

```
class Corps
{
  ...
  Vecteur acceleration(Corps d)
  {
    // acceleration de d sur this
    Vecteur dc = Vecteur.diff(d.position, position);
    double norme = dc.carreNorme();
    double s = d.masse/(Math.sqrt(norme)*norme);
    dc.increment(s);
    return dc;
  }

  Vecteur acceleration(Arbre a)
  {
    // acceleration de l'arbre a sur this
    Vecteur dc = Vecteur.diff(a.barycentre, position);
    double norme = dc.carreNorme();
    double s = a.masse/(Math.sqrt(norme)*norme);
    dc.increment(s);
    return dc;
  }
}
```

L'accélération d'un corps c est estimée en sommant les contributions des autres corps au cours d'un parcours de l'arbre représentant l'univers. La méthode gravitation calculant ce vecteur est récursive sur l'arbre. Il est donc naturel de la définir dans la classe Arbre. Elle prend en argument le corps c. La constante α est une donnée de la classe Univers dont il sera question plus loin.

```
class Arbre
{
  ...
  Vecteur gravitation(Corps c)
  {
    // a non null;
    if (estFeuille)
    if (corps == c)
      return new Vecteur(0,0);
    else
      return c.acceleration(corps);
    Vecteur d = Vecteur.diff(c.position, barycentre);
    // c est "loin"
    if (Math.sqrt(d.carreNorme()) > Univers.alpha * cote)
      return c.acceleration(this);
    // c est proche
    Vecteur s =  new Vecteur();
    for (int i = 0; i < 4; i++)
```

```
      if (fils[i] != null)
        s.increment(fils[i].gravitation(c));
    return s;
  }
}
```

3.6 Univers

La classe Univers contient l'ensemble des données, et la stratégie d'évaluation des mouvements. L'univers contient donc les corps, mais aussi le tétrarbre qui les organise.

```
class Univers
{
  static double alpha = 2;
  static double deltatemps = 0.1;
  Corps[] corps;
  Arbre a;
  double cote;

  Univers(double cote, Corps[] corps)
  {
    this.cote = cote;
    this.corps = corps;
  }

  void ajusterAccelerationVitessePosition()
  {
    int N = corps.length;
    for (int i = 0; i < N; i++)
      corps[i].acceleration = a.gravitation(corps[i]);
    for (int i = 0; i < N; i++)
    {
      corps[i].vitesse.increment(deltatemps, corps[i].acceleration);
      corps[i].position.increment(deltatemps, corps[i].vitesse);
    }
  }

  void construireArbre()
  {
    a = null;
    Vecteur origine = new Vecteur();
    int N = corps.length;
    a = new Arbre(cote, origine, corps[0]);
    for (int i = 1; i < N; i++)
      a.insererCorps(corps[i]);
    a.barycentres();
  }

  void simuler()
  {
    construireArbre();
    ajusterAccelerationVitessePosition();
  }
}
```

La méthode simuler réalise un pas de la simulation. À chaque pas, le tétrarbre est reconstruit. Ensuite, les accélérations sont calculées, d'où on déduit les vitesses et les nouvelles positions. Le temps intervient, et a été arbitrairement fixé à un dixième de seconde. La complexité dans le pire des cas est $O(n^2)$, mais en pratique, l'arbre est de hauteur $O(\log n)$.

Quelques résultats théoriques et expérimentaux pour finir. Pour une distribution *uniforme* en 3D, on montre que le nombre d'interactions à calculer est de l'ordre de $\frac{28\pi\alpha^3}{9}(n \log_2 n)$, soit $263(n \log_2 n)$ pour $\alpha = 3$.

En pratique, la simulation d'un système de 5000 particules est 10 fois plus rapide qu'avec l'algorithme direct. De plus, l'algorithme est facilement parallélisable. En 1994, on a simulé 107 particules pour 103 pas de calcul (une semaine de calcul sur 500 processeurs...) On connaît par ailleurs un algorithme en $O(n)$, qui repose sur d'autres principes. Il reste que la simulation réaliste d'une galaxie nécessite des millions d'étoiles...

Bibliographie

[1] A. V. Aho, J. E. Hopcroft et J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[2] K. Arnold, J. Gosling et H. David, *The Java Programming Language*, Addison Wesley, 2000.

[3] D. Beauquier, J. Berstel et P. Chrétienne, *Éléments d'algorithmique*, Masson, 1992.

[4] J. Bentley, *Programming Pearls*, Addison Wesley, 1999.

[5] J. Berstel et D. Perrin, *Theory of Codes*, Academic Press, 1985.

[6] J. Berstel, J.-E. Pin et M. Pocchiola, *Mathématiques et informatique*, vol. 1 et 2, McGraw-Hill, 1991.

[7] T. H. Cormen, C. E. Leiserson et R. L. Rivest, *Introduction à l'algorithmique*, Dunod, 1994.

[8] G. H. Gonnet et R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, Addison Wesley, 1991.

[9] M. Goossens, F. MittelBach et S. Alexander, *The LaTeX Companion*, Addison Wesley, 1994.

[10] M. Goossens, S. Rahtz et F. MittelBach, *The LaTeX Graphics Companion : Illustrating Documents with TeX and Postscript*, Addison Wesley, 1997.

[11] J. Gosling, B. Joy, G. Steele et B. Gilad, *The Java Language Specification*, Addison Wesley, 2000.

[12] D. E. Knuth, *Fundamental Algorithms. The Art of Computer Programming, vol. 1*, Addison Wesley, 1968.

[13] D. E. Knuth, *Seminumerical Algorithms. The Art of Computer Programming, vol. 2*, Addison Wesley, 1969.

[14] D. E. Knuth, *Sorting and Searching. The Art of Computer Programming, vol. 3*, Addison Wesley, 1973.

[15] D. E. Knuth, *The TeXbook*, Addison Wesley, 1984.

[16] D. E. Knuth, *The Metafont Book*, Addison Wesley, 1986.

[17] M. Lothaire, *Combinatorics on Words*, Cambridge University Press, 1982.

[18] P. Niemeyer et J. Peck, *Exploring Java*, O'Reilly & Associates, Inc., 1996.

[19] R. Sedgewick et P. Flajolet, *Introduction à l'analyse d'algorithmes*, International Thomson Publishing, FRANCE, 1996.

[20] J. van Leeuwen, *Handbook of Theoretical Computer Science*, vol. A : Algorithms and Data Structures, Elsevier, 1990.

[21] J. van Leeuwen, *Handbook of Theoretical Computer Science*, vol. B : Formal Models and Semantics, Elsevier, 1990.

Index