TP-Cours d'Informatique 14

Manipulation de piles

ITC – PCSI^2 2024-2025 – Lycée Fabert (Metz) – L. Parise – N.Wipf – S.Limal – D.Mengel

La structure de **pile** (en anglais *stack*) est une structure de données fondée sur le principe « dernier arrivé, premier sorti » (en anglais **LIFO** pour *last in, first out*), ce qui veut dire que le dernier élément ajouté à la pile est le premier à en sortir.

On peut se représenter une pile en imaginant de très lourdes briques empilées les unes sur les autres. Outre la création d'une pile vide, **voici les primitives/opérations communément utilisées pour manipuler des piles :**

- **Empiler** : ajoute un élément sur la pile. Le terme anglais correspondant est *push*.
- **Dépiler** : supprime le dernier élément ajouté à la pile le cas échéant et le renvoie. Le terme anglais correspondant est *pop*.
- **Tester si une pile est vide** : renvoie *vrai* si la pile est vide, *faux* sinon.

Il existe d'autres primitives usuelles, mais non indispensables dans la mesure où elles se déduisent des trois primitives précédentes :

- **Compter le nombre d'éléments d'une pile** renvoie le nombre d'éléments dans la pile. Selon les implémentations, peut être fait soit en temps constant soit en temps linéaire.
- **Lire l'élément au sommet de la pile** : renvoie l'élément de tête sans le dépiler. Le terme anglais correspondant est *peek* ou *top*.
- **Vider la pile** : dépiler tous les éléments. Selon l'implémentation, cela peut être fait en temps constant ou linéaire. Le terme anglais correspondant est *clear*.
- **Dupliquer l'élément de tête** et **Échanger les deux premiers éléments** : existent sur les calculatrices fonctionnant en notation polonaise inversée (NPI). Les termes anglais correspondants sont *dup* et *swap* respectivement.

Dans ce TP, nous modéliserons la structure de pile en utilisant des listes et on utilisera donc l'instruction `P=[]` pour créer une pile vide.

1. Écrire la fonction `empiler(P,x)→None` qui ajoute en tête/au somment de la pile `P` (donc de manière concrète, un objet de type **`list`** en Python) l'objet `x` de type quelconque.
2. Écrire la fonction `estVide(P)→`**`bool`** qui renvoie `True` ou `False` suivant que la pile `P` est vide ou non.
3. Écrire la fonction `depiler(P)→x` qui retire à la pile `P` **non vide** son élément de tête `x` et le renvoie. **Indication :** utiliser l'instruction **`assert`** pour vérifier que la pile passée en paramètre de cette fonction contient au moins un élément !

   Remarque : les traitements de texte permettent des actions du type « Annuler » et « Rétablir » qui utilisent une pile contenant les éléments saisis au clavier et toutes les modifications effectuées sur le texte par l'utilisateur.

Manipulation de piles

Pour répondre aux quatre questions qui suivent, on utilisera uniquement les trois fonctions précedentes : `empiler`, `depiler` et `estVide` sans oublier la création de pile `P=[]`.

4. Écrire la fonction `copier(P)→Q` qui prend comme argument la pile `P` et en renvoie une copie `Q` sans modifier `P`.

5. Écrire la fonction `inverser(P)→Q` qui prend comme argument la pile `P` et renvoie la pile `Q` contenant les éléments de `P` en ordre inverse sans modifier `P`. **Indication :** vous pouvez utiliser la fonction `copier` de la question précédente !

6. Écrire la fonction `tourner(P)→Q` qui prend comme argument la pile `P` et renvoie la pile `Q` obtenue par **permutation circulaire des éléments de `P`** : l'élément au sommet passe à la base de la pile et les autres sont décalés sans modifier `P`.

7. Écrire la fonction `lastBeFirst(P)→Q` qui prend comme argument la pile `P` et renvoie la pile `Q` obtenue en échangeant l'élément au sommet et l'élément à la base de la pile `P`, les autres gardant leur place dans la pile. Bien sûr, cette fonction de doit pas modifier la pile `P`

Vérification du parenthèsage

8. À présent, on cherche à vérifier si un texte (une chaîne de caractères) contenant des parenthèses ouvrantes et fermantes est « bien parenthésé » ce qui signifie qu'à chaque parenthèse ouvrante correspond directement une parenthèse fermante de même type. Vous avez sans doute déjà remarqué ce type de contrôle syntaxique basique lancé lorsque vous exécutez du code Python dans Pyzo : une erreur de syntaxe est relevée s'il manque une parenthèse ou si une parenthèse fermante inexpliquée est repérée :

```
>>> print(Bonjour))

    print(Bonjour))
                  ^
SyntaxError: unmatched ')'
```

Listing 1 – Problème de parenthésage

```
>>> print(Bonjour; a = 2

    print(Bonjour; a = 2
                  ^
SyntaxError: invalid syntax
```

Listing 2 – Problème de parenthésage

Dans cette question, on prendra en compte les paires de parenthèses `()`, `{}` et `[]`.

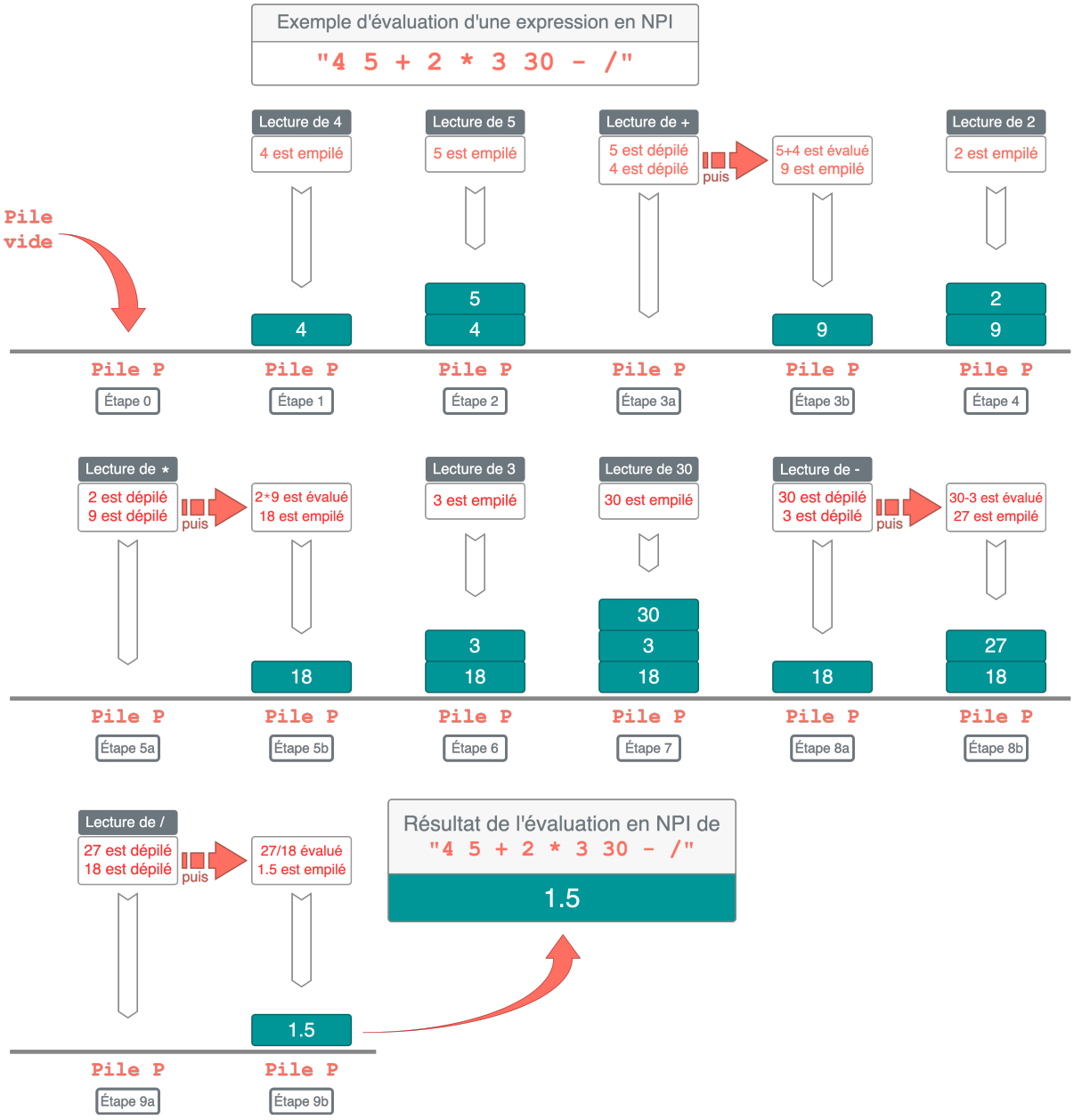
Bien sûr, le problème resterait le même pour d'autres types de parenthésage – `begin end` en Pascal – ou de balises – `<div></div>` ou `<span></span>` ou `<a></a>` en HTML.

Ainsi, les chaînes `"c(de)fg[hijki]lkj"`, `"a(defg[hijki]lk)j"` ou `"c(d{e[f]}g)p"` sont bien parenthésées mais les chaînes `"c(de[fg)hijki]lkj"`, `"a(defg[hijki]lkj"` ou `"cd{e[f]}g)p"` ne le sont pas.

Écrire une fonction `verifPar(expression:str)→bool` qui étudie le parenthésage de la chaîne de caractères `expression` et renvoie `True` si ce parenthésage est correct et `False` dans le cas contraire. **L'algorithme de cette fonction est donné sur la dernière page !**

9. La notation polonaise inverse (NPI) (en anglais *RPN* pour *Reverse Polish Notation*), également connue sous le nom de notation post-fixée, permet d'écrire de façon non ambiguë les formules arithmétiques sans utiliser de parenthèses. En NPI, les opérandes sont présentées avant les opérateurs. **Exemple :** `"5 3 2 + *"` représente l'expression mathématique $5 * (3 + 2)$.

   Les calculatrices NPI sont fondées sur l'utilisation d'une pile : une expression en NPI est lue de gauche à droite, les opérandes rencontrés sont disposés au sommet de la pile, et lorsqu'on rencontre un opérateur, celui-ci s'applique entre l'élément au sommet de la pile et son précédesseur. **Exemple :** `"2 5 -"` est évalué en $5 - 2 = 3$. Les résultats des calculs sont retournés au sommet de la pile puis la lecture de l'expression en NPI se poursuit.



Écrire la fonction `evalNPI(commande:str)→float/int` qui évalue en NPI l'expression contenue dans la chaîne de caractères `commande` et renvoie résultat de type **`float`** ou **`int`**.

Pour aller plus loin :

10. Reprendre et modifier le code de la fonction `verifPar` pour que cette fonction indique (en plus du résultat `True`/`False` renvoyé) sur quel(s) parenthèse(s) de la chaîne `expression` le problème de parenthésage a été détecté le cas échéant.

```
>>> verifPar("2+3*[4*(1+5*(7+6)+8*{6+4}/5)+12]")
True
```

```
>>> verifPar("aaa(kkk)(kkk][lll]")
Les deux types de parenthèses ci-dessous ne se correspondent pas !
aaa(kkk)(kkk][lll]
        ^    ^
False
```

```
>>> verifPar("aaa(kkk)(kkk)[lll")
Cette parenthèse n'est jamais refermée
aaa(kkk)(kkk)[lll
            ^
False
```

```
>>> verifPar("ss]ddd[")
Il y a une parenthèse fermante inutile ici
OU il manque une parenthèse ouvrante à sa gauche! :
ss]ddd[
  ^
False
```

Algorithme 1 Vérification de parenthésage

Définition préalable : DicPFermantes ← { ")" : "(", "]" : "[", "}" : "{" } # Dictionnaire des *parenthèses fermantes* autorisées et des *parenthèses ouvrantes* qui leur correspondent.

Fonction verifPar(expression : **chaîne de caractères**) : **booléen**

Rôle : Déterminer si la chaîne *expression* est bien parenthésée

Déclaration : *c* : **caractère**, pilePar : **Pile** # Pile des parenthèses ouvrantes

Début

```
1:  pilePar ← Pile Vide
2:  tant que il y a des caractères dans expression faire
3:      c ← premier caractère non encore lu dans expression
4:      si c est une parenthèse ouvrante alors
5:          empiler(PilePar,c)
6:      sinon si c est une parenthèse fermante alors
7:          si estVide(pilePar) ou que sommet(pilePar) ne correspond pas à c alors
8:              retourner False # il manque une parenthèse ouvrante OU pb de correspondance
9:          sinon
10:             depiler(pilePar)  # correspondance entre 2 parenthèses
11:         fin si
12:     fin si
13: fin tant que
14: si estVide(pilePar) alors
15:     retourner True  # expression est bien parenthésée
16: sinon
17:     retourner False # expression est mal parenthésée
18: fin si
```

Fin Fonction
