PILES

B. Landelle

Table des matières

I Structure de pile

1 Présentation

Définition 1. *Une* pile *(*stack *en anglais) est une structure de données correspondant à un empilement d'éléments régi par la règle* « dernier arrivé, premier sorti » *(*LIFO *en anglais,* Last In First Out*).*

La dernière information rangée est la première disponible. La pile correspond donc exactement à l'image usuelle d'un empilement de livres ou d'assiettes : on ajoute des assiettes à la pile et on ne peut accéder directement qu'à la dernière assiette.

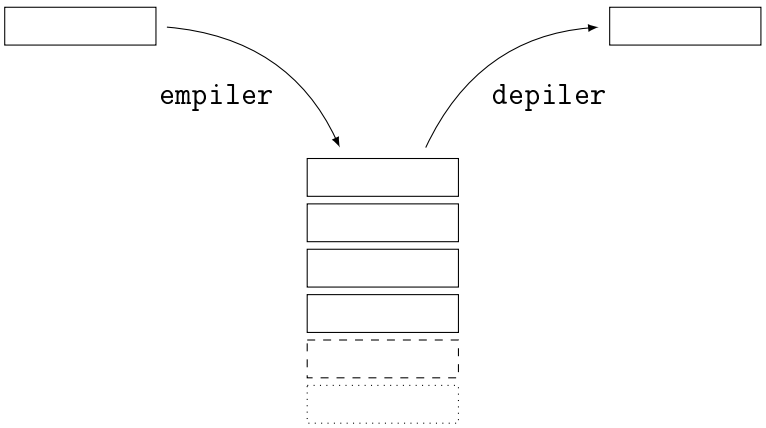

FIGURE 1 – Schéma d'une pile

Vocabulaire : Le terme *empiler* désigne l'ajout d'un élément à la pile (*push* en anglais) et le terme *dépiler* désigne le retrait du dernier élément de la pile (*pop* en anglais).

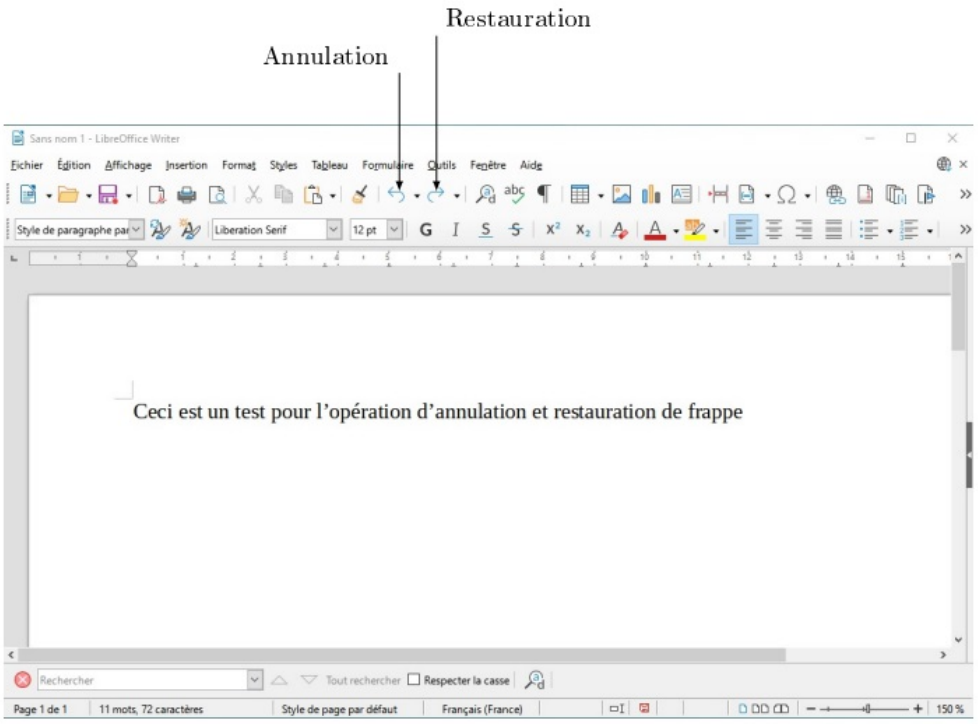

FIGURE 2 – Annulation/Restauration d'une frappe sous `LibreOffice`

Les piles sont fondamentales en informatique :

- les microprocesseurs gèrent nativement une pile ;
- les appels d'un programme vers des sous-programmes sont enregistrés dans une pile ;
- l'historique des pages web d'un navigateur est mémorisé dans une pile ;
- la commande d'annulation d'une frappe enregistre les instructions dans une pile ;
- etc. ...

2 Opérations sur les piles

Une pile peut être vue comme une structure abstraite que l'on va munir d'opérations permettant d'agir sur sa composition. Avec ces opérations, on pourra alors écrire des algorithmes utilisant des piles sans avoir besoin d'en connaître l'implémentation.

Définition 2. *Pour manipuler des piles, on utilise les* opération primitives *suivantes :*

- *la fonction* `Pile` *: crée une pile vide ;*
- *la méthode* `empiler` *: ajoute un élément au sommet de la pile ;*
- *la méthode* `depiler` *: dépile l'élément au sommet de la pile et le renvoie ;*
- *la méthode* `vide` *: indique si la pile est vide ;*

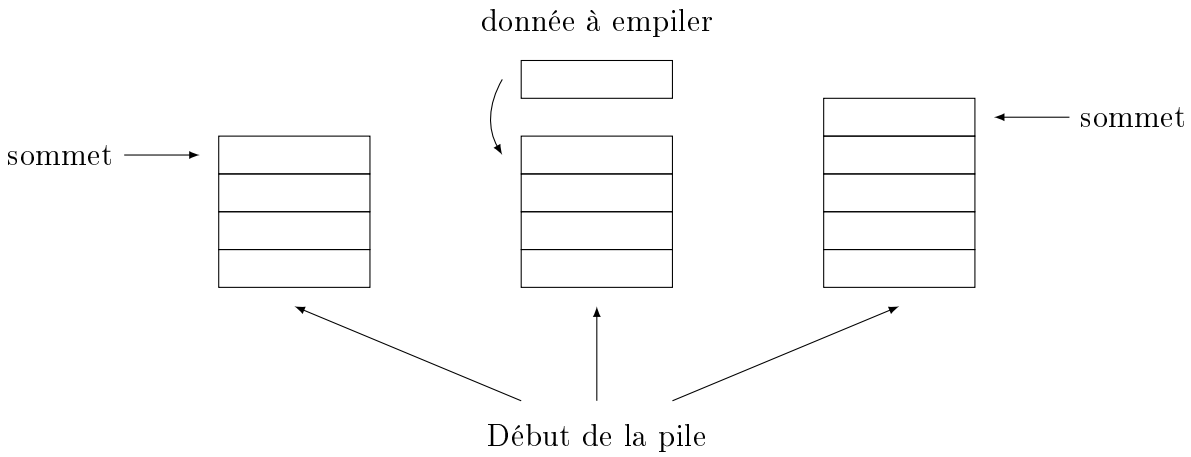

FIGURE 3 – Sommet d'une pile

Remarque : Ces opérations sont dites *primitives* : les opérations plus élaborées s'appuieront sur ces opérations primitives.

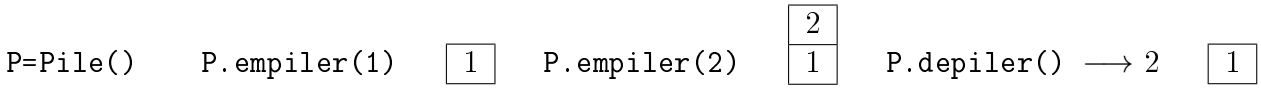

SCHÉMA 1 - Manipulation de pile

Pour définir une fonction `sommet` qui renvoie le sommet d'une pile sans la modifier, on saisit :

```
def sommet(P):
    S=P.depiler()
    P.empiler(S)
    return S
```

Pour définir une fonction `taille` qui renvoie la taille d'une pile sans la modifier, on saisit :

```
def taille(P):
    Q=Pile()
    while not P.vide():
        Q.empiler(P.depiler())
    t=0
    while not Q.vide():
        t+=1
        P.empiler(Q.depiler())
    return t
```

Exercice : Déterminer la complexité temporelle et spatiale de la fonction `taille`.

Corrigé : Notons n la taille de la pile initiale. La fonction dépile totalement la pile `P` et créée une nouvelle pile `Q` dans laquelle sont empilés les éléments de `P` puis on fait l'inverse en dépilant `Q` et en empilant ses éléments dans `P` (pour que celle-ci retrouve son état initial). On utilise également une variable annexe de comptage. La complexité spatiale est donc en $\mathrm{O}(n)$ et on effectue n passages dans chacune des deux boucles `while` d'où une complexité temporelle en $\mathrm{O}(n)$.

Exercice : Écrire, uniquement à l'aide des opérations primitives, des fonctions réalisant les instructions suivantes :

- `duplique(P)` qui, pour une pile `P` non vide, duplique l'élément au sommet de la pile ;
- `echange(P)` qui, pour une pile `P` de taille au moins égale à 2, échange les deux éléments au sommet de la pile ;
- `ins(P,elt)` qui insère, en début de pile `P`, l'élément `elt` ;
- `renverse(P)` qui inverse l'ordre des éléments de la pile `P` ;
- `cycle(P)` effectue une permutation cyclique du sommet vers le début de la pile `P`.

Corrigé : On saisit :

```
def duplique(P):
    S=P.depiler()
    P.empiler(S)
    P.empiler(S)

def echange(P):
    a=P.depiler()
    b=P.depiler()
    P.empiler(a)
    P.empiler(b)
```

```
def ins(P,elt):
    Q=Pile()
    while not P.vide():
        Q.empiler(P.depiler())
    P.empiler(elt)
    while not Q.vide():
        P.empiler(Q.depiler())

def renverse(P):
    Q=Pile()
    R=Pile()
    while not P.vide():
        Q.empiler(P.depiler())
    while not Q.vide():
        R.empiler(Q.depiler())
    while not R.vide():
        P.empiler(R.depiler())

def cycle(P):
    Q=Pile()
    S=P.depiler()
    while not P.vide():
        Q.empiler(P.depiler())
    P.empiler(S)
    while not Q.vide():
        P.empiler(Q.depiler())
```

3 Implémentation

On utilisera la classe `Pile` définie par :

```
class Pile:
    def __init__(self):
        self.items = []

    def vide(self):
        return self.items == []

    def empiler(self, item):
        self.items.append(item)

    def depiler(self):
        return self.items.pop()
```

Une variable de type `Pile` (on parle *d'instance de la classe* `Pile`) ne sera manipulable que par les opérations et méthodes primitives. La classe `Pile` repose sur le type `list`. Il s'agit d'un modèle de piles non bornées, ou plutôt de piles dont la taille n'est pas fixée *a priori*. Les piles non bornées ne sont pas réellement implémentables puisqu'aucune machine ne possède une

mémoire infinie. Il existe aussi des modèles de piles bornées : par exemple, les microprocesseurs gèrent une pile de hauteur fixée.

II Applications

1 Expression bien parenthésée

Une application simple des piles consiste en la vérification d'expressions mathématiques bien parenthésées. Par exemple, l'expression $(1 + 2) \times 3$ est correctement parenthésée tandis que $(1+2\times3$ ne l'est pas. En reprenant l'implémentation du modèle de pile proposée précédemment, on peut, sur une expression mathématique saisie sous forme de chaîne, empiler les caractères lors des ouvertures de parenthèses et les dépiler lors des fermetures ce qui permet ainsi de tester la validité de l'expression.

```
def parenth1(expr):
    P=Pile()
    for x in expr:
        if x=="(":
            P.empiler(x)
        elif x==")":
            if P.vide():
                return False
            P.depiler()
    return P.vide()
```

Expérimentation :

```
>>> parenth1("(()())")
True
>>> parenth1("(()()(((")
False
```

On pourrait tout à fait se passer d'une structure de pile et gérer un compteur de parenthèses ouvrantes. Mais l'intérêt d'une approche avec pile est son adaptabilité à des situations plus complexes comme par exemple la vérification d'expressions avec parenthèses, crochets, accolades.
Exercice : Expliquer l'utilité de l'instruction `if P.vide(): return False`.

Corrigé : Si on rencontre une parenthèse fermante alors qu'aucune parenthèse n'a été ouverte ou que toutes les parenthèses ouvertes antérieurement ont déjà été fermées, alors l'expression n'est pas valide et le test en question le détecte.

Exercice : Pour une chaîne de caractères, la méthode `count` permet de compter le nombre d'occurrences d'un caractère. On propose la fonction suivante :

```
def parenth2(expr):
    return expr.count("(")==expr.count(")")
```

que l'on expérimente sur les situations suivantes

```
>>> parenth2("(()())")
True
>>> parenth2("((())()")
False
>>> parenth2("(()))")
False
```

Ce programme vérifie-t-il la validité du parenthésage comme le programme précédent ? Si non, expliquer pourquoi et proposer une version corrigé du programme.

Corrigé : En apparence, le programme semble faire la même chose que le précédent. Toutefois, si on le teste sur la chaîne ")(", l'expression est considérée comme valide.

```
>>> parenth2(")(")
True
```

Le simple comptage de parenthèses ouvrantes et fermantes ne suffit donc pas à garantir la validité de l'expression (condition nécessaire, pas suffisante). Une manière simple de rectifier ce programme consiste à regarder si, pour tout chaîne extraite `expr[0:k]` avec `k` dans $[\![0\,;n-1]\!]$ où n désigne la taille de `expr`, le nombre de parenthèses ouvrantes est toujours supérieur ou égal au nombre de parenthèses fermantes et sur la chaîne complète `expr`, si le nombre de parenthèses ouvrantes est égal au nombre de parenthèses fermantes.

```
def parenth3(expr):
    for k in range(len(expr)):
        pre=expr[:k]
        if pre.count("(")<pre.count(")"):
            return False
    return expr.count("(")==expr.count(")")
```

Avec cette nouvelle version, l'expérimentation est concluante.

```
>>> parenth3(")(")
False
>>> parenth3("()")
True
```

2 Notation postfixée

Une autre application simple des piles consiste en la *notation postfixée*, aussi appelée *notation polonaise inverse*[1] (RPN pour *Reverse Polish Notation*). Cette notation permet d'écrire, de manière non ambiguë, des formules arithmétiques sans utiliser de parenthèses. Dans une expression en notation postfixée, les opérateurs arithmétiques sont placés après les opérandes. En pratique, une pile sert à l'empilement des opérateurs sur lesquels agissent ensuite les opérandes.

1. Elle est dérivée de la notation polonaise présentée en 1920 par le mathématicien polonais Jan Lukasiewicz.

Dans ce qui suit, les schémas de pile seront orientés avec le sommet en bas pour faciliter la lisibilité des opérations arithmétiques (notamment la division).

• Un cas simple :

$$(1+2)\times 4 \quad \text{s'écrira en RPN :} \quad 1 \quad 2 \quad + \quad 4 \quad \times$$

avec la représentation des empilements et des opérations :

$$\begin{array}{|c|}\hline 1\\\hline 2\\\hline\end{array} \xrightarrow{+} \boxed{3} \qquad \begin{array}{|c|}\hline 3\\\hline 4\\\hline\end{array} \xrightarrow{\times} \boxed{12}$$

• Un cas plus élaboré :

$$\frac{-\sqrt{2}}{\ln(3+\sin 1)} \quad \text{s'écrira en RPN :} \quad 2 \quad \text{sqrt} \quad \text{neg} \quad 3 \quad 1 \quad \sin \quad + \quad \ln \quad /$$

avec la représentation des empilements et des opérations :

$$\boxed{2} \xrightarrow{\sqrt{}} \boxed{\sqrt{2}} \xrightarrow{\text{neg}} \boxed{-\sqrt{2}} \qquad \begin{array}{|c|}\hline -\sqrt{2}\\\hline 3\\\hline 1\\\hline\end{array} \xrightarrow{\sin} \begin{array}{|c|}\hline -\sqrt{2}\\\hline 3\\\hline \sin 1\\\hline\end{array}$$

$$\xrightarrow{+} \begin{array}{|c|}\hline -\sqrt{2}\\\hline 3+\sin 1\\\hline\end{array} \xrightarrow{\ln} \begin{array}{|c|}\hline -\sqrt{2}\\\hline \ln(3+\sin 1)\\\hline\end{array} \xrightarrow{/} \boxed{-\sqrt{2}/\ln(3+\sin 1)}$$

Le lecteur désirant découvrir d'autres utilisations de ces structures de pile pourra consulter la bible en informatique qu'est l'ouvrage [1].

Références

[1] Donald Knuth, *The Art of Computer Programming, Volume 1 : Fundamental Algorithms*, Third Edition, Addison-Wesley, 1997