Leçon d'informatique : Piles

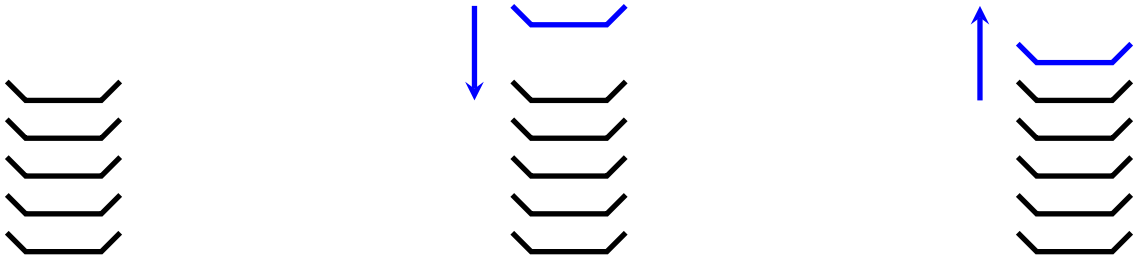
S. Benlhajlahsen - PCSI$_1$

Sommaire

I Présentation

I.A Exemple simple

Un exemple simple pour commencer est une pile d'assiettes à nettoyer. Rajouter une assiette se fera sur le dessus de la pile : on dira **empiler**. Enlever une assiette se fera toujours par celle du dessus (la dernière « arrivée ») : on dira **dépiler**.



pile d'assiettes — Empiler — Dépiler

LIFO : La dernière assiette posée sur la pile sera la première que l'on va nettoyer. Ce principe est à la base d'une pile : « dernier arrivé, premier sorti » (LIFO en anglais pour *last in, first out*)

I.B Autres exemples

Fontion "*Undo*" d'un éditeur de texte : Quand on écrit un texte sur un éditeur[a], il est généralement possible de revenir sur les anciennes versions de celui-ci. Cela permet généralement de revenir sur une erreur de frappe. Généralement, on l'obtient par la commande CTRL+Z.
On comprend donc que l'éditeur va mémoriser les versions successives du texte sous la forme d'une pile. La fonction "*Undo*" ou « annuler la frappe » va dépiler les versions en partant des dernières.

Remarque : Il existe aussi la fonction "Redo" qui fait le contraire de "Undo" pour aller vers des versions plus récentes.

a. éditeur de texte, de *mail*, traitement de texte.

Navigateur *web* : Lorsqu'on utilise un navigateur *web*[a], on page de page en page par les liens hypertexte. Le navigateur va mémoriser les pages visitées sous la forme d'une pile. L'utilisateur dépile la page courante pour accéder à la page précédente en cliquant le bouton « Afficher la page précédente ».

a. *Browser* en anglais. Par exemple, *Google Chrome*, *Safari* ou *Brave*.

Algorithmes : Nous verrons plus tard :

- les algorithmes récursifs qui utilisent une pile d'appels. On verra d'ailleurs que cette pile d'appels aura une « profondeur » limitée.
- l'algorithme de recherche en profondeur qui utilise une pile pour mémoriser les nœuds visités.

II Définition et implémentation

II.A définition

Une pile pile [1] est une **structure de données abstraite** basée sur le principe « dernier arrivé, premier sorti » (LIFO en anglais pour *last in, first out*).

Distinction entre structures de données et implémentation : On confond souvent une structure de données ou un algorithme avec son implémentation.
Par exemple, on connaît l'algorithme de Fibonacci. L'implémentation sera la « réalisation concrète de celui-ci » qui dépendra du langage utilisé.
De la même manière, un tableau de données pourra être implémenté sous la forme d'un fichier **csv**, **.xls** (Excel) ou une tableau Python.

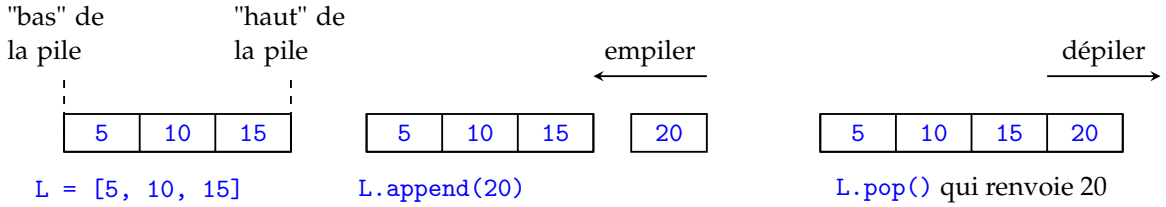
Conclusion : La pile est bien un type **abstrait**. On pourra l'implémenter de différentes manières (liste, chaîne de caractère, `deque` ou même créer une classe/objet spécifique).

II.B Primitives

De quoi avons-nous besoin ? Quelles sont les fonctions de bases (ou primitives) dont on a besoin ?

- **Empiler** : ajoute un élément sur la pile. Le terme anglais correspondant est *push* ;
- **Dépiler** : enlève un élément de la pile et le renvoie. Le terme anglais correspondant est *pop*.
- **La pile est-elle vide ?** : renvoie `True` si la pile est vide, `False` sinon.
- **Nombre d'éléments de la pile ?** : renvoie le nombre d'éléments dans la pile.
- **Quel est l'élément de tête ?** : renvoie l'élément de tête sans le dépiler.

II.C Implémentation d'une pile par une liste



Si on implémente une pile à l'aide d'une liste, on peut donc :

- **empiler** qui consiste à rajouter un élément en haut de la pile. Ici, le haut de la pile correspond à l'élément le plus à droite. On pourra donc empiler avec une méthode `append` ou une concaténation.

```
1 def empiler1(L, a):
2     # empile a au bout de la liste
3     L.append(a)
4 L1 = [5, 10, 15]
5 empiler1(L1, 20)
6 print(L1)
```

```
1 def empiler2(L, a):
2     # empile a au bout de la liste
3     return L + [a]
4 L1= [5, 10, 15]
5 L1 = empiler2(L1 , 20)
6 print(L1)
```

- dépiler qui consiste à enlever le dernier élément. La méthode `pop()` [2] peut faire ce travail. On doit tout de même faire attention aux cas d'une liste vide.

---

1. *stack* en anglais
2. ou `pop(-1)` car c'est le dernier terme.

```python
def depiler1(L):
    # depiler et renvoyer le dernier element
    return L.pop()
L1 = [5, 10, 15, 20]
a = depiler1(L1)
print(a) # renvoie 20
```

```python
def depiler2(L):
    if len(L)!=0:
        return L.pop()
L2 = []
a = depiler2(L2)
print(a) # renvoie none
```

- le booléen `len(L)==0` vaut `True` ou `False` suivant que la pile est vide ou non.

III Applications

III.A Parenthésage

On veut étudier le parenthésage d'expressions arithmétiques ou du code source d'un programme. Il est utile de vérifier que les parenthèses sont bien écrites et aussi de déterminer la parenthèse ouvrante associée à une parenthèse fermante.

Remarque : La plupart des éditeurs font du *brace matching*, c'est-à-dire qu'ils vont indiquer les parenthèses associées.

```
a = 2
b = 5
c = (2*(a+4)+(b+2))
```

Par exemple $1+2*(7-(4-3)*((2-5)+2*((12/4-8)+2*3)))$ est bien parenthésée et la parenthèse ouvrante après $(4-3)*$ est associée à l'avant dernière parenthèse fermante.
On considère une chaîne de caractère représentant l'expression et on s'intéresse uniquement aux parenthèses classiques `"("` et `")"`.
Pour savoir si une expression est bien parenthésée, il faut vérifier qu'il y a autant de parenthèses ouvrantes que de parenthèses fermantes et que chaque parenthèse fermante est associée à une parenthèse ouvrante placée avant elle. Celle-ci est la dernière parenthèse ouvrante du texte non encore fermée. On voit apparaître l'utilité d'une pile.

- On lit les caractères un par un,
- quand on voit une parenthèse ouvrante, on empile sa position ;
- quand on lit une parenthèse fermante, on dépile un élément, il sera la position de la parenthèse ouvrante associée.
- Si on doit dépiler, alors que la pile est vide ou si la pile est non vide à la fin, c'est que l'expression n'est pas bien parenthésée.

Dans la chaîne `"1+2*(7-(4-3)*((2-5)+2*((12/4-8)+2*3)))"` correspondant à l'expression ci-dessus les opérations de la pile seront

- on empile 4,
- on empile 7
- on dépile 7 qui est associé à 11
- on empile 13
- on empile 14
- on dépile 14 qui est associé à 18
- on empile 22
- on empile 23
- on dépile 23 qui est associé à 30
- on dépile 22 qui est associé à 35
- on dépile 13 qui est associé à 36
- on dépile 4 qui est associé à 37.

On peut donner le résultat sous la forme d'une liste de couples, pour l'exemple ci-dessus on aurait, comme résultat, la liste `[(7, 11), (14, 18), (23, 30), (22, 35), (13, 36), (4, 37)]`.

Exercice 1 : Liste des parenthèses associées
Écrire une fonction `listePar` qui reçoit une expression **supposée bien parenthésée** et qui retourne la liste des couples

d'indices de parenthèses associées.
Réponse en figure 1.

Exercice 2 : Test de parenthésage
Écrire une fonction `bienPar` qui reçoit une expression qui n'est plus supposée bien parenthésée et qui renvoie `True` ou `False` selon que la liste est bien parenthésée ou non.
Réponse en figure 2.

III.B Notation post-fixée (ou polonaise inversée)

Notation polonaise ou pré-fixée : En 1920, le mathématicien polonais Jan Lukasiewicz propose la notation pré-fixé (ou polonaise) qui consiste à considérer les opérations comme des opérateurs à 2 variables, $7+11$ s'écrit $+\ 7\ 11$. L'avantage est que les parenthèses deviennent inutiles : $(7-2).\sin(2x+\pi/3)$ devient $*\ -\ 7\ 2\ \sin\ +\ *\ 2\ x\ /\ \pi\ 3$.

Notation polonaise inversée ou post-fixée : La notation polonaise inverse ou notation post-fixée a été proposée par le philosophe et informaticien australien Charles Leonard Hamblin dans le milieu des années 1950.
Elle a été diffusée dans le public comme interface utilisateur avec les calculatrices de bureau de Hewlett-Packard (HP-9100), puis avec la calculatrice scientifique HP-35 en 1972.
Elle consiste simplement à placer l'opérateur **après** les deux opérandes, $7+11$ s'écrit $7\ 11\ +$. L'expression $32-2(12-3(7-2))$ peut s'écrire $32\ 2\ 12\ 3\ 7\ 2\ -\ *\ -\ *\ -$

L'avantage est que, combinée à une pile, cette notation permet d'effectuer les calculs sans faire référence à une quelconque adresse mémoire.

- On lit l'expression terme-à-terme :
- si on lit une valeur numérique, elle est empilée,
- si on lit une opération ♣,
  on dépile les deux derniers opérandes a et b
  et on empile le résultat du calcul a♣b.

Exercice 3 : Un exemple :
Simuler l'évolution de la pile avec l'expression $32\ 2\ 12\ 3\ 7\ 2\ -\ *\ -\ *\ -$
Réponse en figure 3.

Exercice 4 : Simulation d'une calculatrice :
Écrire une fonction `npi(liste)` qui prend en entrée une liste contenant des valeurs numériques et des chaînes `'+', '-', '*', '/'` et qui calcule la valeur de l'expression en NPI. On considère que les nombres sont des flottants, la division est la division réelle classique.
Pourquoi est-il difficile de lire directement une chaîne de caractères pour l'évaluer ?
Réponse en figure 4.

```python
def listePar(chaine):
    pile = nouvelle_pile()
    n = len(chaine)
    res = []
    for i in range(n):
        x = chaine[i]
        if x=="(":
            empiler(pile,i)
        if x==")":
            j = depiler(pile) # position de l'ouvrante
            res.append((j,i))
    return res
```

FIGURE 1 – Réponse pour l'exercice 1.

```python
# Il suffit de compter les parenthèses ouvrantes non encore fermées ;
# on n'a pas besoin d'une pile.
def bienPar(chaine):
    n = len(chaine)
    ouvrantes = 0
    for i in range(n):
        if chaine[i] == "(":
            ouvrantes = ouvrantes + 1
        if chaine[i] == ")":
            if ouvrantes == 0:
                return False
            else:
                ouvrantes = ouvrantes - 1
    return ouvrantes == 0
```

FIGURE 2 – Réponse pour l'exercice 2.

Après avoir empilé les nombres on lit les opérateurs.

- La pile contient 32, 2, 12, 3, 7, 2.
- On dépile 2 puis 7, on calcule $7 - 2 = 5$ (attention à l'ordre), on empile 5.
- La pile contient 32, 2, 12, 3, 5.
- On dépile 5 puis 3, on calcule $3.5 = 15$, on empile 15.
- La pile contient 32, 2, 12, 15.
- On dépile 15 puis 12, on calcule $12 - 15 = -3$, on empile -3.
- La pile contient 32, 2, -3.
- On dépile -3 puis 2, on calcule $2.(-3) = -6$, on empile -6.
- La pile contient 32, -6.
- On dépile -6 puis 32, on calcule $32 - (-3) = 38$, on empile 38.
- La pile contient 38 qui est le résultat renvoyé.

FIGURE 3 – Réponse pour l'exercice 3.

```python
def npi(liste):
    pile = []
    for x in liste:
        if x == "+":
            b = depiler(pile)
            a = depiler(pile)
            empiler(pile, a+b)
        elif x == "-":
            b = depiler(pile)
            a = depiler(pile)
            empiler(pile, a-b)
        elif x == "*":
            b = depiler(pile)
            a = depiler(pile)
            empiler(pile, a*b)
        elif x == "/" :
            b = depiler(pile)
            a = depiler(pile)
            empiler(pile, a/b)
        else:
            empiler(pile, x)
    return pile
```

Le problème de la lecture directe d'une chaîne de caractères est le double sens du signe moins : il est à la fois une opération binaire et un changement de signe.

FIGURE 4 – Réponse pour l'exercice 4.