Chapitre 8

Les piles

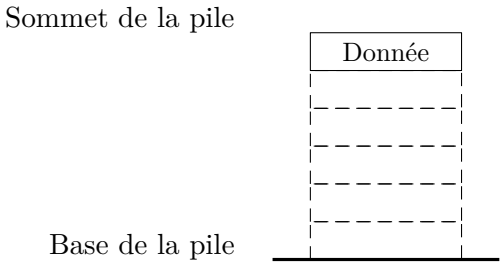
8.1 Les piles

8.1.1 Introduction

• Une pile est une structure de donnée séquentielle.

• Elle est semblable à une pile d'assiette, où les seules opérations possibles consistent à :

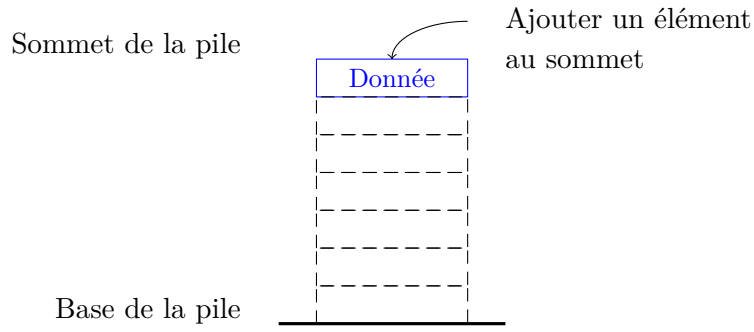
1. ajouter une assiette (une donnée) au sommet de la pile,
2. retirer l'assiette (la donnée) du sommet de la pile.



• Une pile est une structure de donnée séquentielle.

• Elle est semblable à une pile d'assiette, où les seules opérations possibles consistent à :

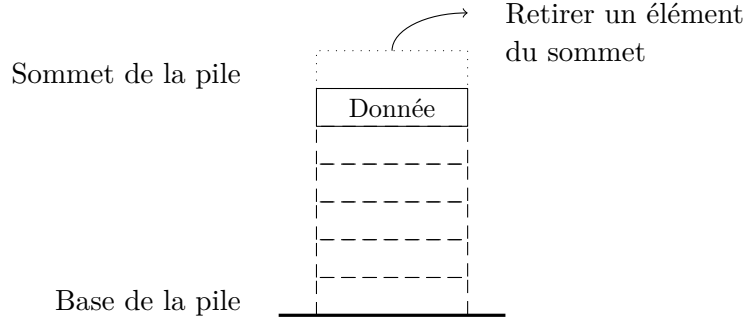
1. ajouter une assiette (une donnée) au sommet de la pile,
2. retirer l'assiette (la donnée) du sommet de la pile.



• Une pile est une structure de donnée séquentielle.

• Elle est semblable à une pile d'assiette, où les seules opérations possibles consistent à :

1. ajouter une assiette (une donnée) au sommet de la pile,
2. retirer l'assiette (la donnée) du sommet de la pile.



8.1.2 Primitives des piles

• Une pile informatique (stack en anglais) est une structure de donnée séquentielle basée sur le principe "Dernier entré, premier sorti", en anglais : "Last In First Out", abrégé en **LIFO**.

• Les opérations caractéristiques d'une pile (ou *primitives*) sont :

1. Empiler un élément (en anglais *push*),
2. Dépiler un élément (en anglais *pop*),
3. La pile est-elle vide ?

Auxquelles on peut ajouter :

4. Lire l'élément au sommet de la pile (en anglais *peek*),
5. Obtenir la taille (= nombre d'éléments) de la pile

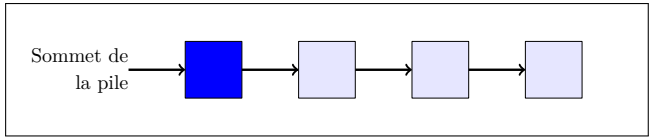
• Les 3 premières primitives doivent se faire en temps constant $O(1)$. La taille peut s'obtenir en temps constant ou linéaire selon l'implémentation.

• C'est une structure de donnée plus simple/restrictive qu'un tableau.

• Les microprocesseurs disposent nativement d'une pile (de capacité limitée) parmi leurs registres. L'éxécution des programmes utilise aussi souvent une pile dans la mémoire centrale. N'avez vous jamais rencontré un message d'erreur "Stack Overflow" ?

8.1.3 Implémentation par liste chainée

• Pour implémenter une pile, les données en mémoire n'ont pas à être contigües. Exemple d'implémentation :



• L'objet pile pointe vers l'adresse du sommet. Ce dernier contient la donnée stockée et l'adresse mémoire de l'élément suivant, et ainsi de suite.

• Pour y accéder en $O(1)$, la taille doit aussi être stockée (encapsulée) dans l'objet pile. Autrement on ne l'obtient qu'en temps linéaire $O(taille)$.

• Une pile est moins coûteuse à implémenter qu'un tableau, car il n'est plus nécessaire de réserver de l'espace mémoire contigu. A chaque empilement il suffit de réserver de l'espace mémoire pour le seul couple `(donnée, adresse)` n'importe où dans la mémoire disponible.

• En contrepartie certaines opérations possibles avec un tableau ne le sont plus avec une seule pile : par exemple ordonner une pile de données numériques.

• Une pile est suffisante par exemple : dans un navigateur web pour accéder aux pages précédentes. Pour l'opération "undo" (annuler), etc...

8.2 Implémentation en `python`

8.2.1 Implémentation d'une pile à capacité illimitée

• Pour simuler le fonctionnement d'une pile en `python`, on peut utiliser une liste.

• Commençons par une pile à capacité illimitée (comme l'est une liste en `python`) ; c'est particulièrement simple puisque nativement les listes en `python` disposent des méthodes `append()` et `pop()`.

• La fonction de création d'une liste retourne la liste vide :

```
def creer_pile():
    return []
```

• Empilement et dépilement s'obtiennent grâce aux méthodes `append()` et `pop()` de la classe `<list>`.

```
def empiler(p,e):
    p.append(e)
```

```
def depiler(p):
    assert len(p)>0    # erreur si la pile est vide
    return p.pop()
```

• Remarquer l'utilisation de l'instruction "`assert` *condition*" qui retourne une erreur si *condition* n'est pas vérifiée.

• La taille de la pile s'obtient grâce à la fonction `len()` qui retourne la longueur de la liste sous-jacente, en $O(1)$:

```
def taille(p):
    return len(p)
```

• Pour tester si une pile est vide :

```
def est_vide(p):
    if len(p)==0:
        return True
    else:
        return False
```

• Lecture du dernier élément :

```
def top(p):
    return p[-1]
```

8.2.2 Implémentation d'une pile à capacité limitée

• La fonction `creer_pile(c)` crée une liste vide de longueur `c+1`.

```
def creer_pile(c):    # c est la capacité de la pile
    pile = (c+1) * [None]
    pile[0] = 0
    return pile
```

Le mot-clef `None` réserve un espace vide. Le premier élément `pile[0]` contient le nombre d'élément de la pile, initialement 0.

• Avec cet implémentation il est alors facile de retourner la taille de la pile :

```
def taille(p):
    return p[0]
```

• Pour empiler un élément il faut prendre garde que la taille de la pile n'excède pas sa capacité et incrémenter le compteur de la taille `p[0]`. :

```
def empiler(p,e):
    taille = p[0]
    assert taille < len(p)-1
    p[taille+1] = e
    p[0] += 1
```

- De même pour le dépilement :

```
def depiler(p):
    taille = p[0]
    assert taille > 0    # erreur si pile vide
    e = p[taille]
    p[taille] = None    # l'élément est supprimé
    p[0] -= 1    # Décrémentation de la taille
    return e
```

- Pour tester si la pile est vide :

```
def est_vide(p):
    if p[0]==0:    return True
    else:    return False
```

- Pour lire le dernier élément ; s'assurer que la pile est non vide.

```
def top(p):
    taille = p[0]
    assert taille > 0
    return p[taille]
```

8.3 Exemples d'application

8.3.1 Calcul en notation polonaise

- En algèbre, la notation polonaise, ou préfixée, permet l'écriture d'expression algébrique sans utiliser de parenthèses :

$$(1+2)\times(3-(4\times5)) \qquad \text{s'écrit} \qquad \times + 1\ 2\ -\ 3\ \times\ 4\ 5$$

- Chaque opération $\times$, $\div$, $+$, $-$, étant binaire (elle prend 2 opérandes), l'écriture est sans ambiguité.

- L'évaluation d'une expression de longueur N en notation polonaise nécessite l'usage d'une pile de capacité limitée à N données.

Algorithme permettant d'évaluer une expression en notation polonaise :

1. L'expression sera contenue dans une liste, que l'on parcourt de droite à gauche,
2. dès que l'on rencontre un nombre (on se limite aux entiers) on l'empile,
3. dès que l'on rencontre une opération on dépile ses deux opérandes de la pile, on effectue l'opération,
4. puis on empile le résultat.

Nécessite de stocker au plus N données pour une expression de longueur N :
Exemple : $(((1+2)\times3)-4)\times5)$ se notera : $\times - \times + 1\ 2\ 3\ 4\ 5$.

```
def polonaise(liste):
    pile = creer_pile(len(liste))    # Création d'une pile
    for x in reversed(liste):    # Parcours de droite à gauche *
        if isinstance(x,int):        # x de type entier ? **
            empiler(pile,x)         # si oui on l'empile
        else:                       # Sinon :
            a = depiler(pile)
            b = depiler(pile)
            if x == '+':            # Addition
                empiler(pile,a+b)
            if x == '*':            # Multiplication
                empiler(pile,a*b)
            if x == '-':            # Soustraction
                empiler(pile,a-b)
            if x == '/':            # Division
                empiler(pile,a/b)
    return depiler(pile)        # résultat au sommet de la pile
```

* `reversed(list)` retourne un itérateur sur la liste '`list` inversée' .

** `isinstance(variable,type)` teste si `variable` est de type `<type>`

8.3.2 Parenthésage

- Nous allons appliquer une structure de pile pour créer une fonction permettant de déterminer si une chaîne de caractère passée en argument est ou non bien parenthésée et retourne les positions des parenthèses (plus tard en TD).

- Avec les parenthèses `(` et `)` un mot est bien parenthésé si :
  1. Il ne contient aucune parenthèse `'('` ou `')'`, ou
  2. Il est de la forme `'(mot)'` avec `mot` un mot bien parenthésé, ou
  3. Il est la concaténation de 2 mots bien parenthésés.

Par exemple, si l'on omet tous les caractères autres que les parenthèses, les mots suivants sont bien parenthésés :

`()` ; `(()())` ; `(()())(())()`

et les mots suivants ne sont pas bien parenthésés :

`)(` ; `((())` ; `()()())`

Une condition nécessaire pour qu'un mot soit bien parenthésé est que le nombre de

parenthèses ouvrantes `'('` soit égal au nombre de parenthèses fermantes `')'`, mais c'est une condition non-suffisante.

- Pour le résoudre à l'aide d'une pile :
  1. On parcourt le mot de gauche à droite,
  2. Dès que l'on rencontre un caractère `'('` on l'empile sur la pile.
  3. Dès que l'on rencontre un caractère `')'`, deux cas se présentent :
     (a) Soit la pile est vide, et dans ce cas le mot n'est pas bien parenthésé.
     (b) Soit la pile est non-vide, et dans ce cas on la dépile.
  4. A la fin si la pile est vide le mot est bien parenthésé, sinon il ne l'est pas.
- On pourra utiliser une pile à capacité limitée, de capacité égale à la longueur du mot.

```
def parenthesage(mot):
    pile = creer_pile(len(mot))    # création de la pile
    for char in mot:                # Parcours du mot
        if char == '(':             # Si parenthèse ouvrante
            empiler(pile,1)         # on empile (ce qu'on veut)
        elif char == ')':           # Si parenthèse fermante
            if est_vide(pile):      # Et si pile vide
                return False        # Alors mal parenthésé
            else:
                depiler(pile)       # Sinon dépilement
    return est_vide(pile)    # retourne True si la pile est vide
```

- **Exercice 1.** Ici à la place d'une pile on pourrait aussi utiliser un compteur entier, puisque la donnée empilée n'a aucune importance. Le faire.
- **Exercice 2.** Adapter le programme aux 3 parenthésages : `'()'`, `'[]'`, `'{}'`.
- **Exercice 3.*** Démontrer qu'un mot est bien parenthésé si et seulement si il contient autant de `'('` que de `')'` et tout préfixe contient au moins autant de `'('` que de `')'`. On pourra procéder par récurrence forte sur la longueur du mot. En déduire la correction du programme.