Chap 1 : Piles

Table des matières

1 Piles - Le cours **1**
1.1 Introduction : les structures de données 1
1.2 Définition d'une pile 2
1.3 Mise en oeuvre d'une pile avec un tableau 2
1.4 Opérations EMPILER et DEPILER 3
1.5 Utilisation d'une pile par l'unité centrale d'un ordinateur 3

2 TD sur les piles **4**
2.1 Implémentation par un tableau 4
2.2 Définition d'une pile à l'aide des méthodes de la classe list 4
2.3 Évaluation d'une expression arithmétique 5
2.4 Exercices supplémentaires 6
2.5 Ouverture : la structure **File** 6

1 Piles - Le cours

1.1 Introduction : les structures de données

Certaines données informatiques peuvent contenir d'autres données plus simples. On les appelle des **conteneurs** et les données contenues en sont les **éléments**.

C'est le cas par exemple des chaines de caractères, "abracadaba" par exemple, ou encore des listes dans le langage Python. Un conteneur peut être vu comme une *collection* de données. Ce qui le distingue d'un ensemble mathématique est que :

- il peut renfermer le même élément plusieurs fois : le caractère "a" dans "abracadabra" par exemple ;
- sa taille peut augmenter ou diminuer au cours de l'exécution du programme. De plus, ses éléments peuvent être modifiés. On dit que le conteneur est **dynamique** ($\neq$ statique).

Lorsqu'on fait de l'algorithmique, on s'intéresse à développer des conteneurs dynamiques spéciaux qui permettent de réaliser certaines opérations. Dans la plupart des cas, ces opérations sont :

- RECHERCHER(S, x) : chercher si x est un élément de S.
- INSERER(S, x) : insérer l'élement x dans S.
- SUPPRIMER(S, x) : supprime l'élément x de S.
- MINIMUM(S) : renvoie le plus petit élément de S dans le cas on on peut définir une relation d'ordre total sur ces éléments.
- MAXIMUM(S) : renvoie le plus grand élément de S, toujours dans le cas où un ordre total existe.

Il faut alors organiser les éléments du conteneur pour **maximiser** l'efficacité de ces opérations. Par maximisation de l'efficacité, on entend :

- La recherche du moindre coût temporel : recherche de rapidité.
- la recherche du moindre espace en mémoire : préserver la mémoire de l'ordinateur.

Les deux efficacités sont parfois contradictoires et il faut faire des compromis !

Définition (Structure de données)

On appelle **structure de données**, un conteneur dynamique S utilisé en algorithmique pour la conception de programmes. Elle est destinée à organiser des données informatiques dans le but de maximiser l'efficacité de certaines opérations.

Les structures de données que l'on rencontre le plus souvent en algorithmique sont les **piles**, les **files**, les **listes chainées**, les **tas**, les **arbres** et les **graphes**. Conformément au programme d'IPT, nous étudierons seulement la structure d'une pile.

1.2 Définition d'une pile

Une **pile** (**stack** pour les anglo-saxons) est une structure de données qui met en oeuvre le principe : **dernier entré, premier sorti** ou **LIFO** (Last - In - First - Out).

L'image qu'on peut en donner est celle d'une pile d'assiettes : la dernière assiette placée dans la pile est au sommet de celle-ci. Ce sera bien sûr aussi la première assiette que l'on enlèvera de cette pile.

Cette structure permet de "stocker" provisoirement des éléments, en attendant de les utiliser plus tard. Les seules opérations dont on a besoin avec cette structure sont INSERER et SUPPRIMER qu'on appelle ici EMPILER (**push** en anglais) et DEPILER (**pop** en anglais).

1.3 Mise en oeuvre d'une pile avec un tableau

Il y a beaucoup de façons de concevoir une pile. La plus simple consiste à utiliser un conteneur de type **tableau**. Il s'agit d'une structure qui contient des éléments de même nature : que des entiers, que des réels ou encore que des chaines de caractères, etc ...

Un **tableau** T de n éléments est noté T[0...$n-1$]. Le numéro $i \in [\![0, n-1]\!]$ de chaque élément est son **indice** : il représente la place de l'élément dans le tableau T. L'élément d'indice i sera noté T[i]. Vous pouvez voir un tableau comme une suite de "cases" contenant les différents éléments :

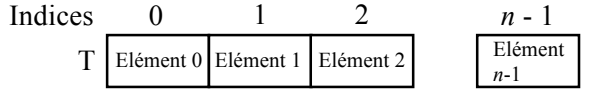

Figure 1 Représentation d'un tableau T[0 ... n - 1] de n éléments.

Tous les langages de programmation permettent de réaliser des tableaux. Pour le langage Python, le plus simple est d'utiliser des **listes**.

On peut réaliser une pile d'entiers contenant au plus n éléments avec un tableau P[0...n] pouvant contenir $n+1$ entiers :

- La première case (de rang 0) contient l'indice du *prochain élément à insérer* dans la pile.
- les n "cases" suivantes, d'indices 1 à n, contiennent les éléments à insérer dans la pile. Le dernier élément inséré est appelé **sommet** de la pile.

Si P[0] == 1 la pile est **vide**. À chaque fois qu'on insère un élément, on augmente P[0] d'une unité. Lorsque P[0] == $n+1$: la pile est **pleine**.

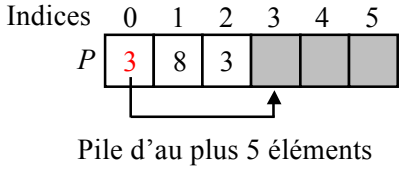

Figure 2 Réalisation d'une pile P à l'aide d'un tableau. Les éléments de la pile apparaissent uniquement aux positions blanches. L'élément au sommet de la pile est 3. P[0] contient l'indice du prochain élément à insérer.

1.4 Opérations EMPILER et DEPILER

On les rédige en pseudo-code avec quelques emprunts à Python pour le rendre plus lisible, à savoir :

- les deux points " **:** " et l'indentation pour marquer le corps d'une fonction, d'une boucle ou d'une structure conditionnelle.
- Les opérateurs logiques : == (identique à), != (différent de), >, >=, < et <=.
- l'opérateur d'affectation =
- La numérotation des lignes mais uniquement à l'intérieur d'une fonction ou d'un programme, ainsi que les commentaires qui commencent par un #.

Lorsqu'on rédige les opérations EMPILER et DEPILER, il faut gérer le fait qu'on ne peut rien retirer à une pile vide et rien ajouter à un pile pleine. Si on tente de dépiler une pile vide, on dit qu'elle **déborde négativement**, ce qui est une erreur. De même, si on tente d'empiler une pile pleine, on dit qu'elle **déborde**, ce qui est aussi une erreur.

L'opération **empiler** insère l'élément x au sommet de la pile P. On la rédige sous la forme d'une **fonction** :

fonction EMPILER(P, x) :

```
1   si P[0] == n + 1 faire :
2     erreur " débordement", arrêter programme
3   sinon :
4     i = P[0]      # i <- indice du prochain élément à insérer
5     P[i] = x      # Insertion de x à sa place
6     P[0] = i + 1
```

L'opération **dépiler** retire le dernier élément entré dans la pile et renvoie cet élément :

fonction DEPILER(P) :

```
1   si P[0] == 1 faire :
2     erreur " débordement négatif ", arrêter programme
3   sinon :
4     P[0] = P[0] − 1
5     i = P[0]
6     retourner P[i]
```

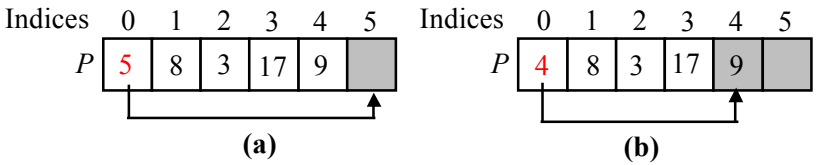

Figure 3 **(a)** État de la pile P de la Figure 2 après les appels EMPILER(P, 17) et EMPILER(P, 9). **(b)** État de la pile P après que l'appel DEPILER(P) ait retourné 9. Bien que 9 apparaisse encore dans le tableau, il ne fait plus partie de la pile.

1.5 Utilisation d'une pile par l'unité centrale d'un ordinateur

L'unité centrale (microprocesseur) d'un ordinateur utilise une pile pour gérer les **appels de fonction** :

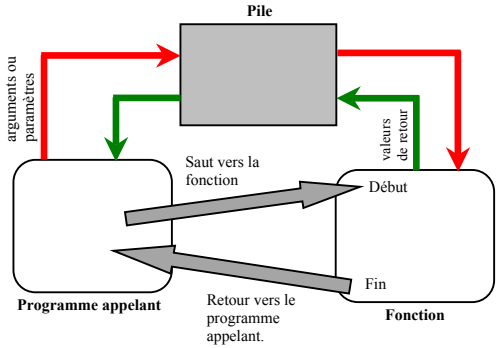

Figure 4 : Mécanisme de "communication" entre un programme et une fonction appelée.

- Pour passer un paramètre à la fonction : une **copie** de celui-ci est empilé sur la pile.
- La fonction récupère cette copie en la dépilant.
- De même, si la fonction retourne une valeur, elle le fait en la recopiant sur la pile. Le programme n'a plus qu'à récupérer cette valeur en la dépilant.

2 TD sur les piles

2.1 Implémentation par un tableau

Dans cette section, nous souhaitons définir une pile d'entiers à l'aide d'un tableau P[0...n], comme dans le cours. Comme nous utilisons Python, ce tableau **P** sera défini comme une liste de $n+1$ entiers, initialisée avec des 0 partout [1].

1. Écrire un programme qui génère cette liste :

$$P = [0, 0, ..., 0, 0]$$

   Vous pourrez prendre $n = 10$ par exemple.
2. Écrire les fonctions `EMPILER(P, x)` et `DEPILER(P)` comme dans le cours. En Python, vous pouvez produire un arrêt brutal du programme grâce à l'instruction **raise**(IndexError). Cette instruction met immédiatement fin au programme en générant une erreur de type IndexError (erreur d'indice) [2].
3. Écrire une fonction `PILE_VIDE(P)` qui retourne **True** si la pile est vide et **False** sinon.

2.2 Définition d'une pile à l'aide des méthodes de la classe list

Nous allons maintenant générer une pile dont la taille maximale n'est pas définie : on peut y empiler autant d'entiers que l'on veut (dans la limite de la mémoire de l'ordinateur).

1. On définit toujours notre pile **P** comme une liste d'entiers. Initialement, elle est vide et : P = []. Ré-écrire complètement les fonctions `EMPILER(P, x)`, `DEPILER(P)` et `PILE_VIDE()` uniquement à l'aide des méthodes **append** et **pop** des objets de la classe `list`. On n'y mettra pas d'instructions de sortie brutale de programme.

1. En informatique, un tableau ne peut contenir que des éléments du même type (que des entiers ou que des chaines de caractères par exemple). Les listes Python sont une structure un peu différente puisqu'elles peuvent contenir des éléments de type différents.
2. Il existe différents type d'erreurs, comme NameError, TypeError, SyntaxError, AssertionError, etc...

2. Que se passe-t-il quand on essaye de dépiler un pile vide ?

   Pour la suite, vous allez garder ces deux dernières fonctions `EMPILER(P, x)` et `DEPILER(P)`. Comme le paramètre `x` n'a pas besoin d'être un entier, nous allons en profiter pour écrire un *détecteur de parenthésage correct d'une expression.*

   Dans une expression arithmétique, il doit y avoir autant de parenthèses ouvrantes que fermantes, par exemple : ($(3 + 5) \times 2$) $+ 1$ est correcte, tandis que $(2 + 3) \times 4) - 1$ est fausse.

3. Écrire une fonction `analyse_parenthese(ch)` qui analyse une chaîne de caractères `ch` représentant une expression arithmétique, caractère par caractère et qui détecte si le parenthésage est correct. Vous utiliserez une pile `P` en variable locale et uniquement les fonction `EMPILER`, `DEPILER` et `PILE_VIDE`. La fonction affichera un message indiquant si le parenthésage est correct ou non.

2.3 Évaluation d'une expression arithmétique

Une des tâches fondamentales d'un interpréteur comme celui de Python (ou d'un compilateur dans un autre langage de programmation) est d'attribuer une valeur à une expression arithmétique. Une expression arithmétique est formé des 10 chiffres 0, 1, ..., 9, des signes +, -, $\times$ et / et des parenthèses ouvrante (et fermante). Par exemple :

$$(3 + (4 \times 5))/2$$

est une expression arithmétique dont la valeur est 11,5.

En général, l'utilisateur tape cette expression à l'aide de son clavier, ce qui génère une chaîne de caractères ch = "(3 + (4 * 5))/2". Le problème est de traduire cette chaîne en un nombre, tout en gérant les priorités des opérations et les parenthèses : l'interpréteur doit se livrer pour cela à une **analyse syntaxique**.

Une même expression arithmétique a au moins trois représentations naturelles :

- La représentation **infixée**. C'est celle qu'on utilise tout le temps : $(3 + 2) \times 5$
- La représentation **préfixée** où on place tous les opérateurs avant les opérandes : $2 + 5$ s'écrit $+ \ 2 \ 5$ et $(3 + 2) \times 5$ s'écrit $\times + 3 \ 2 \ 5$. Dans ce cas, les parenthèses sont superflues : si on avait voulu dire : $3 + (2 \times 5)$, on aurait écrit : $+ \ 3 \times 2 \ 5$.
- On peut également placer les opérateurs après les opérandes : c'est la représentation **postfixée**. Dans ce cas $2 + 5$ s'écrirait : $2 \ 5 +$ et si on partait de $(3 + 2) \times 5$ on aurait : $3 \ 2 + 5 \times$. Aucun besoin de parenthèses là non plus.

1. Donner les deux autres formes des expressions suivantes et indiquer à chaque fois leurs valeurs : a) $2 + (\ 3 - (5 + 1) \times 2 \ )$ b) $2 \ 3 \ 4 \ 5 + - \ 6 \times /$

   L'évaluation d'une expression **postfixée** peut se faire de façon très simple, en une seule lecture de la gauche vers la droite à l'aide d'une pile qui stocke les résultats intermédiaires. Sur l'exemple suivant, le caractère | a été rajouté pour indiquer la limite de la partie déjà traitée de l'expression.

$\vert \ 2 \ 3 \ 4 + - \ 5 \times$	pile **P** : []
$2 \ \vert \ 3 \ 4 + - \ 5 \times$	pile **P** : [2]
$2 \ 3 \ \vert \ 4 + - \ 5 \times$	pile **P** : [2 , 3]
$2 \ 3 \ 4 \ \vert \ + - \ 5 \times$	pile **P** : [2 , 3 , 4]
$2 \ 3 \ 4 + \vert \ - \ 5 \times$	pile **P** : [2 , 7]
$2 \ 3 \ 4 + - \ \vert \ 5 \times$	pile **P** : [-5]
$2 \ 3 \ 4 + - \ 5 \ \vert \times$	pile **P** : [-5 , 5]
$2 \ 3 \ 4 + - \ 5 \times \ \vert$	pile **P** : [-25]

Résultat : -25

2. Partons de la chaîne postfixée : ch = "123+*4−" (on appelle cela une chaîne de *tokens*).

   Écrire une fonction `calc_exp_arith(ch)` qui prend en paramètre la chaîne `ch` et qui renvoie la valeur numérique de l'expression arithmétique.

   Quelques pistes : outre les fonctions `EMPILER` et `DEPILER` et `PILE_VIDE` déjà écrites, vous pourrez utiliser :

   - Une boucle **for** pour parcourir toute la chaîne **ch**.
   - Autant de structures conditionnelles **if** ... **elif** ... **else** que vous voulez.
   - Le mot clé **in**. Par exemple, l'expression (s **in** ['a', 'b', 'c']) vaut **True** si **s** est élément de la liste et elle vaut **False** sinon [3].

2.4 Exercices supplémentaires

1. Écrire une fonction `melange(P1, P2)` qui prend en argument deux piles et qui mélange leurs éléments dans une troisième pile `P3` de la façon suivante : tant qu'une pile au moins n'est pas vide on retire aléatoirement un élément du sommet d'une des deux piles et on l'empile sur `P3`. La fonction retourne `P3`. Remarque : à l'issue du mélange, les deux piles `P1` et `P2` sont vides.
2. Écrire une fonction `couper(P)` qui prend une pile et qui retire de son sommet k éléments qui sont replacés dans une autre pile `autre_P`. k est un nombre tiré au hasard et si $k \geqslant$ au nombre d'élément de `P`, on retire simplement tous les éléments de `P`. Par exemple, si `P = [1, 2, 3, 4, 5]` et si `k == 2`, alors `autre_P = [5, 4]`. La fonction renvoie `autre_P`.
3. **Tour de magie de Gilbreath**. Construire une liste de n cartes, par exemple [9, 10, "reine", "roi", "as"]. Empiler K fois cette liste de n cartes dans une pile `P`. Couper alors cette pile avec la fonction `couper`, puis mélanger les deux piles obtenue avec la fonction `melange`. On observe alors que la pile finale est constituée de K blocs contenant tous les n cartes de la liste initiale (même si ces dernières peuvent apparaître dans un ordre différent au sein de chaque bloc).

[3] Cela marche aussi avec les chaînes de caractères : ('c' **in** "coucou !") vaut **True** par exemple.

2.5 Ouverture : la structure File

Une structure de **File** fonctionne sur le principe **premier entré – premier sorti** (comme les files d'attentes à un guichet). Une façon d'implémenter une File contenant au plus n éléments est de créer `F = [queue, tete, [0, 0, ..., 0] ]` où la sous-liste `F[2]` de `F` est initialisée avec n zéros. Les éléments `tete` et `queue` sont deux entiers. Le fonctionnement est le suivant :

- À la création de la File `F` : `tete == 0` et `queue == 0`
- Quand on insère un élément x dans la File `F`, on le met dans `F[2][queue]`, puis `queue` est augmenté d'une unité.
- Si on retire un élément de la File `F`, on le retire de `F[2][tete]`, puis on augmente `tete` d'une unité. La fonction qui gère cela doit retourner la valeur retirée.
- Si `tete` ou `queue` atteignent la fin de `F[2]`, ils doivent pouvoir revenir à 0 : la gestion des indices `tete` et `queue` se fait donc de *manière circulaire*.
- La File `F` est vide lorsque `tete == queue` et elle est pleine lorsque `queue == tete - 1` ou `queue == n-1` si `tete == 0`.

Écrire les fonctions `ENFILER(F, x)` qui insère l'élément `x` dans la file `F` si celle-ci n'est pas pleine (sinon on peut provoquer l'arrêt du programme), `DEFILER(F)` qui retire l'élément correct de la File `F` et renvoie cet élément (si la file est déjà vide, on provoque un arrêt du programme).