UNIVERSITÉ DE CARTHAGE

IPEIN

INSTITUT PRÉPARATOIRE AUX ETUDES D'INGÉNIEURS DE NABEUL

DÉPARTEMENT MATH ET INFORMATIQUE

INFORMATIQUE : PROGRAMMATION PYTHON

Chapitre 1.
Structures de données avancées
Pile et File

Enseignant : Anis SAIED
anis_saied@hotmail.com

Tous les cours / TPs sont disponibles sur le cahier numérique de la classe :
http://cahier-de-prepa.fr/info-ipein

Dernière mise à jour 2 octobre 2016

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, "Bonjour" 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 "n" dans "Bonne journée" 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 (≠ statique).

Dans la plupart des cas, les opérations à réaliser sont :

- CREER(S) :créer la séquence S.
- 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.

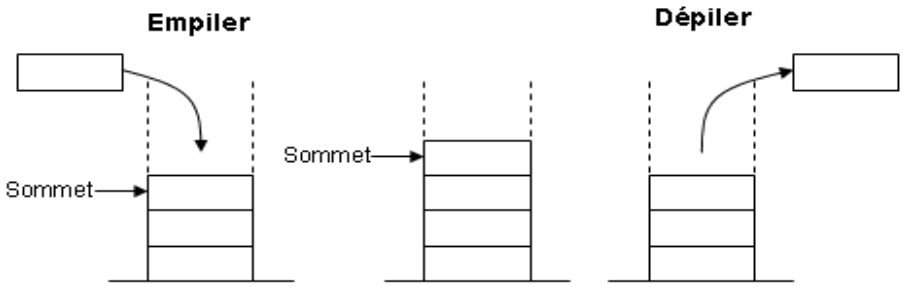
Les structures de données que l'on rencontre le plus souvent en algorithmique sont les **piles**, les **files**, les **listes chainées**, les **tas**, les **arbres** et les **graphes**. Conformément au programme des prépas, nous étudierons seulement les structures des *piles* et *files*.

1 Les Piles

Une **pile** (**stack** pour les anglo-saxons) est une structure de données qui met en oeuvre le principe : **dernier entré, premier sorti** ou **LIFO** (**L**ast - **I**n - **F**irst - **O**ut).

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). Elle permet d'arranger et de récupérer les données les plus récentes.



Autres exemples :

- Pile d'appels d'une fonction récursive,
- Les deux boutons « page suivante » et « page précédente» de votre navigateur chacun utilise une pile.

1.1 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 $T[0 \ldots n-1]$. 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. 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 $P[0 \ldots n]$ 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**.
Exemple :

1.2 Opérations EMPILER et DEPILER

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

Indices	0	1	2	3	4	5
Elements	3	8	3			

TABLE 1 – Réalisation d'une pile P d'au plus 5 éléments 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.

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 :

```
def EMPILER(P, x) :
        if P[0] == n + 1 :
                print("Débordement")
                raise (IndexError) #met immédiatement fin au programme en générant
                #une erreur de type IndexError (erreur d'indice)
        else :
                i = P[0] # i ⟵ indice du prochain élément à insérer
                P[i] = x # Insertion de x à sa place
                P[0] = i + 1
```

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

```
def DEPILER(P) :
        if P[0] == 1 :
                print("Débordement négatif")
                raise (IndexError)
        else:
                P[0] = P[0] - 1
                i = P[0]
        return P[i]
```

Indices	0	1	2	3	4	5
Elements	5	8	3	17	9	

TABLE 2 – État de la pile P après les appels EMPILER(P, 17) et EMPILER(P, 9).

Indices	0	1	2	3	4	5
Elements	4	8	3	17	9	

TABLE 3 – 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.

1.3 TD sur les piles

1.3.1 Définition d'une pile à l'aide d'un tableau

Dans cette section, nous souhaitons définir une pile d'entiers à l'aide d'un tableau $P[0 \dots n]$, 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. 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.

1. Écrire un programme qui génère cette liste : $P = [0, 0, \ldots, 0, 0]$. Vous pourrez prendre $n = 10$ par exemple.
2. Écrire une fonction PILE_VIDE(P) qui retourne $True$ si la pile est vide et $False$ sinon.
3. 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].

1.3.2 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.
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) × 2) + 1 est correcte, tandis que (2 + 3) × 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.

1.3.3 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 $0, 1, \ldots, 9$, des signes +, -, × et / et des parenthèses ouvrante (et fermante). Par exemple : (3 + (4 × 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) × 5
- La représentation **préfixée** où on place tous les opérateurs avant les opérandes : 2 + 5 s'écrit + 2 5 et (3 + 2) × 5 s'écrit × + 3 2 5. Dans ce cas, les parenthèses sont superflues :
  si on avait voulu dire : 3 + (2 × 5), on aurait écrit : + 3 × 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) × 5 on aurait : 3 2 + 5 ×. Aucun besoin de parenthèses là non plus.

2. Il existe différents type d'erreurs, comme NameError, TypeError, SyntaxError, AssertionError, etc...

1. Donner les deux autres formes des expressions suivantes et indiquer à chaque fois leurs valeurs :
   a) 2 + (3 - (5 + 1) × 2)
   b) 2 3 4 5 + - 6 × /
   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.
   | 2 3 4 + - 5 × pile P : []
   2 | 3 4 + - 5 × pile P : [2]
   2 3 | 4 + - 5 × pile P : [2,3]
   2 3 4 | + - 5 × pile P : [2,3,4]
   2 3 4 + | - 5 × pile P : [2,7]
   2 3 4 + - | 5 × pile P : [−5]
   2 3 4 + - 5 | × pile P : [−5,5]
   2 3 4 + - 5 × | 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.

1.3.4 Conversion d'un entier (base 10/base 2)

Écrire une fonction `conversion(nb10)` qui retourne la représentation en base 2 du nombre nb10 représenté en base 10 à l'aide de piles.

2 Les Files

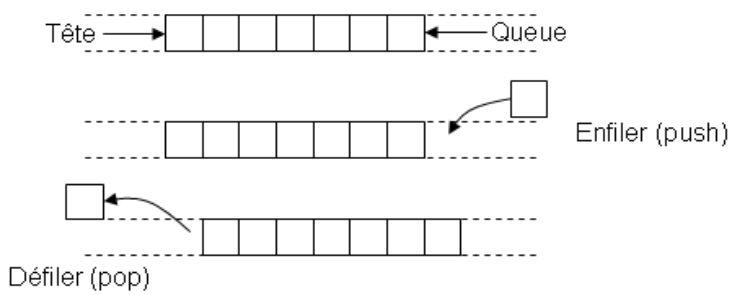
Une **file** est un conteneur dans lequel on peut ajouter des objets, et retirer à tout moment le premier objet ajouté parmi ceux restants. Il correspond à l'idée qu'on se fait d'une file d'attente, où les personnes qui arrivent se placent en **queue** de file, et attendent d'arriver en **tête** avant d'en sortir.

On dit qu'il s'agit d'une structure **FIFO** (**F**irst-**I**n **F**irst-**O**ut), c'est-à-dire que le premier élément qui a été ajouté dans la liste sera aussi le premier qui en sortira.

La première valeur de la file est la tête de la file et la dernière est la queue de la file.

Exemples : En Informatique, le files sont utilsées pour
- Mémoriser les données en attente de traitement.
- Ordonnancer les tâches d'impression,
- Ordonnancer les processus au niveau du système d'exploitation.

Les opérations ordinaires avec les files sont :
- Enfiler (enqueue) : Ajouter en fin de la file.
- défiler (dequeue) : Supprimer le premier élément (sommet) de la file.



2.1 Mise en oeuvre d'une file avec une liste

On peut utiliser les listes pour réaliser une file. Cependant, les listes ne sont pas très efficaces dans ce cas précis ((first in first out) : alors que les ajouts et suppressions en fin de liste sont rapides, les opérations d'insertions ou de retraits en début de liste sont lentes (car tous les autres éléments doivent être décalés d'une position).

2.2 Indications pour la réalisation en Python

Le type `'list'` intègre déjà toutes les méthodes pour réaliser ces deux structures de données :

- `list.append(x)` qui ajoute un élément à la fin de la liste.
- `list.insert(i, x)` qui insère un élément à la position indiquée. Le premier argument est la position de l'élément courant avant lequel l'insertion doit s'effectuer, donc `a.insert(0, x)` insère l'élément en tête de la liste, et `a.insert(len(a), x)` est équivalent à `a.append(x)`.
- `list.remove(x)` qui supprime de la liste le premier élément dont la valeur est x. Une exception est levée s'il existe aucun élément avec cette valeur.
- `list.pop([i])`, qui enlève de la liste l'élément situé à la position indiquée, et le retourne. Si aucune position n'est indiqué, `a.pop()` enlève et retourne le dernier élément de la liste

2.3 TD sur les files

2.3.1 Application 1

Programmer le module `file.py` contenant les fonctions suivantes :

- `init()` : créer une file `f` vide (initialisation de la file).
- `vide(f)` : tester si une file `f` est vide.
- `enfiler(f,x)` : enfiler x à la fin de la file `f`.
- `defiler(f)` : retourner le sommet de la file `f`.
- `sentinelle(f)` : retourner la sentinelle (dernier élément) de la file `f`.
- `taille(f)` : donner le nombre d'éléments de la file `f`.
- `afficher(f)` : afficher la file `f`.

Puis effectuer les appels nécessaires pour créer une file de taches en prenant en considération les informations ainsi fournis :

Nom	Instant d'arrivée	Durée
T1	2	10
T2	0	12
T3	1	5
T4	-1	8

Les éléments de la file seront des tuples de la forme **(Nom,Durée)**.

Reprendre le module file.py en essayant de fixer une taille maximale de la file par l'introduction d'une constante `NMAX=40`. Prévoir les traitements nécessaires au bon fonctionnement du module en cas de saturation de la file.

2.3.2 Application 2

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 $F = [queue, tete, [0,0,\ldots,0]]$ 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).