Ch 19. Les piles et les files.

1 Structure de piles et de files

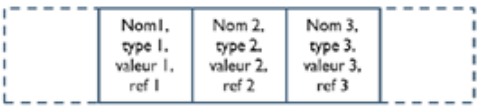
Définition :

Les piles et les files sont des listes particulières : on accède aux éléments par les extrémités, c'est-à-dire au début ou à la fin. Elles sont utilisées par exemple pour des programmes qui doivent traiter des données qui arrivent au fur et à mesure. On distingue :

- les piles (ou « stacks ») : le premier empilé est le dernier à être dépilé, « LIFO » (Last In first Out) ;
- les files : le premier entré est le premier à sortir, « FIFO » (First In First Out).

1.a Structure des données

Les données sont stockées de façon structurée dans la mémoire de l'ordinateur. Les objets de types « simples » (non mutables) sont stockés dans des variables ayant les caractéristiques suivantes :
un identificateur (nom), un type, une valeur, une référence et des opérations associées.
La référence constitue l'adresse de l'objet dans la mémoire de l'ordinateur. Ainsi, si on représentait la mémoire de façon linéaire (ce qui est le cas sur un disque dur non SSD), on pourrait procéder ainsi :



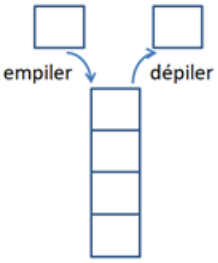
Suivant la valeur, le cadre peut s'étendre sur un grand nombre de bits.
Pour les types Python non mutables (int, float, str, tuple) il est possible de considérer que les données sont juxtaposées car il est possible de connaître la taille de chaque variable lors de l'affectation.
Pour les types mutables (comme les listes), on ne sait pas à l'avance combien d'emplacements mémoire vont être nécessaires car on peut ajouter des données à une même liste de façon indéfinie. Ce sont des structures de taille variable (dynamique) et linéaire.
Dans beaucoup de langages de programmation, les listes sont de type array (tableau) des structures de taille fixe (statique) :

- un nombre de cases fixées à l'avance par l'utilisateur ;
- des cases qui sont toutes du même type ;
- un accès aux cases se faisant théoriquement à coût constant (O(1)).

En Python, ce n'est pas le cas ce qui rend très aisé la gestion des piles et des files.

1.b Importance des piles en informatique

Les piles sont les objets les plus fondamentaux en informatique.
L'insertion et la suppression se font toujours du même côté, le fond n'est jamais accessible, seul le sommet est accessible.



- mémorisation des pages visitées dans un navigateur web, l'adresse de chaque nouvelle page visitée est empilée et l'utilisateur dés-empile l'adresse de la page précédente en cliquant sur précédente ;
- gestion des environnements d'exécution des fonctions en particulier récursives ;
- gestion des expressions arithmétiques de la forme $5\times(3-6)+(1+7)\times 12$ sans parenthèse (notation polonaise inversée) ;
- retour en arrière dans des logiciels de création de document ou des logiciels de jeux avec `Ctrl z` ;
- pour construire le parcours d'un graphe **en profondeur**.

1.c Importance des files en informatique

L'insertion d'un élément se fait par le sommet et la suppression se fait par le fond.

- une file d'attente ;
- les mémoires tampons (buffers) ;
- les systèmes d'exploitation multitâches qui répartissent le temps-machine entre différentes tâches ;
- les serveurs d'impression qui traitent les requêtes dans l'ordre d'arrivée ;
- pour construire le parcours d'un graphe **en largeur**.

1.d Implémentation des piles

L'implémenter des piles se fait généralement à partir de listes (parfois des tableaux et dans ce cas là, la taille est imposée).
Chacun des éléments de la pile peut être un objet de type différent. Le coût des opérations suivantes est constant.
Il y a quatre opérateurs définis pour les piles à partir desquels toutes les manipulations sont possibles.

```
# opération 1 : créer une pile vide
pile=[] # cout constant

# opération 2 : vérifier que la pile est vide
len(pile)==0 # cout constant

# opération 3 : empiler un élément v dans la pile (v est forcément ajouté en dernière place)
pile.append(v) # cout constant

# opération 4 : dépiler une pile (c'est forcément le dernier élément qui est enlevé)
sommet=pile.pop() # cout constant
```

1.e Implémentation des files avec le module deque

Les deques (se prononce « dèque », de l'anglais *deque* pour *double-ended queue*, littéralement « file à deux bouts ») sont accessibles grâce à la bibliothèque `collections`. Comme son nom l'indique, elle optimise l'ajout et le retrait d'élément par les 2 bouts.
Chacun des éléments de la file peut être un objet de type différent.
Il y a quatre opérateurs à coût constant définis pour les files à partir desquels toutes les manipulations sont possibles.

```
from collections import deque

# opération 1 : créer une file vide
file=deque()

# opération 2 : vérifier que la file est vide
len(file)==0

# opération 3 : enfiler un élément v dans la file (v est ajouté a droite)
file.append(v) # on pourrait enfiler v a gauche file.appendleft(v)

# opération 4 : défiler une file (c'est le premier élément qui est enlevé, celui a gauche)
sortie=file.popleft() # on pourrait défiler a droite sortie=file.pop()

# opérations utiles liées a deque
# vider une file
file.clear()
# copier une file
F=file.copy()
```

Les piles sont des listes qui peuvent être de longueur "illimitée" même avec le module `deque`. Pour les files, il est possible d'imposer une longueur en fixant `maxlen`. Dans ce cas là, si la file est pleine par ajout d'éléments alors elle se vide à l'autre extrémité.

```
from collections import deque

file=deque(maxlen=6)
for i in range(5):
    file.append(i)
print (file)
for j in ['pts1','pts2','pt','ptstar']:
    file.append(j)
print (file)
```

Python shell

```
deque([0, 1, 2, 3, 4], maxlen=6) # le 6e élément n'est pas défini
deque([3, 4, 'pts1', 'pts2', 'pt', 'ptstar'], maxlen=6)
```

2 Exemples d'utilisation des deques

2.a Utilisation d'une pile pour la conversion de base

On a les 2 fonctions suivantes pour obtenir la conversion d'un entier vers une liste de bits et inversement :

```
def convToBinary(n):
    q=n
    L=[]
    while q!=0:
        L.append(q%2)
        q//=2
    return L

def convFromBinary(L):
    n=0
    while not len(L)==0:
        n*=2
        n+=L.pop()
    return n
```

Le fonctionnement est bon et à coût minimum, mais la liste des chiffres n'est pas dans un ordre de lecture satisfaisant (lecture de droite à gauche au lieu de gauche à droite).
Proposer des modifications utilisant une deque à la place d'une liste afin de régler ce souci sans détériorer la complexité.

2.b Utilisation d'une file pour faire une permutation circulaire

On souhaite utiliser une file pour faire une permutation circulaire d'un cran. La fonction `perm_circ(f)`, doit renvoyer f qui a subie une permutation circulaire (si `[1, 2, 3, 4]` est mis en entrée, on souhaite voir ressortir `deque([4, 1, 2, 3])`).

2.c La parenthèse inattendue

Dans cet exercice, on souhaite savoir si une chaîne de caractères est bien parenthésée ou non. Une chaîne bien parenthésée est une chaîne vide ou bien la concaténation de chaînes bien parenthésées.
Exemples : Chaînes bien parenthésées :

— `"()","()()","(())"` et `"(()())"`.

Chaînes mal parenthésées :

— `")(","((","(()"` et `"())"`.

Question 1. Écrire la fonction `parentheses(chaine:str)->bool` prenant en argument une chaîne de caractères constituée uniquement de parenthèses et affichant les couples d'indice parenthèse ouvrante, parenthèse fermante. Cette fonction renvoie `True` si la chaîne est bien parenthésée et `False` sinon.

Python shell

```
>>> parentheses('(())()')
(1, 2)
(0, 3)
(4, 5)
```

Références :

T. Kovaltchouk, *Informatique Commune PCSI*, Reims
UPSTI, *Informatique Commune*

Plan du cours