Piles et Files d'attentes

1 Motivation : problèmes liés à la notion de pile ou de file d'attente

(i) Un programme informatique dédié à l'édition, que sa nature soit de composer du code informatique, un texte littéraire, une pièce de musique nécessite souvent de pouvoir annuler des modifications et de revenir à un état précédent du document édité

(ii) Une imprimante connectée en réseau reçoit de plusieurs ordinateurs des tâches d'impression, et chaque document à imprimer se compose de plusieurs pages à imprimer. Comment faire pour que les documents soient imprimés dans l'ordre où ils ont été transmis à l'imprimante ?

(iii) Un programme structuré est composé de nombreux blocs parmi lesquels des fonctions qui s'appellent les unes les autres (et même, on le verra des fonctions qui peuvent s'appeler elles-mêmes), par exemple le programme principal fait appel à une première fonction, qui à son tour en invoque une seconde, et la seconde une troisième. Lorsque la dernière fonction a terminé son travail et donné sa réponse, l'exécution reprend dans le bloc depuis lequel cette fonction a été invoquée, à la suite de l'instruction réalisant cet appel... Comme il peut y avoir d'assez nombreux appels imbriqués, comment gérer l'exécution du code dans ces conditions ?

(iv) Vous cherchez à résoudre un puzzle, comme par exemple un sudoku. Il y a des éléments qui sont certains et que l'on peut compléter sans risque d'erreur, mais il arrive que des positions restent à compléter, et plusieurs possibilités s'offrent à nous, et aucun raisonnement simple ne permet d'affirmer en toute certitude que la bonne valeur est celle-ci ou celle-là. Une solution simple consiste à essayer une par une les valeurs possibles, et à garder trace de la position du puzzle avant complétion et de la valeur proposée pour, en cas d'échec, revenir en arrière (backtracking) et en tester une autre.

2 Définitions

Définition 2.1 *Une* pile *(*stack *pour les anglo-saxons) est une structure permettant de contenir un certain nombre de données, mais qui ne permet l'ajout et le retrait de données que de manière assez stricte : les données retirées le sont dans l'ordre inverse des ajouts. Les anglo-saxons parlent du principe LIFO (last-in-first-out).*

Deux fonctions essentielles régissent l'utilisation d'une pile :

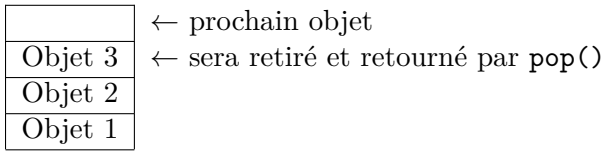
- `push(object)` *qui prend un objet en argument et le place sur la pile*
- `pop()` *qui retourne le dernier objet empilé et le retire de la pile*

Les fonctions suivantes sont moins importantes, mais peuvent également être implémentées :

- `isEmpty()` *qui retourne le booléen* `True` *ou* `False` *selon bien entendu que la pile soit ou non vide*
- `peek()` *qui, sans retirer le dernier objet empilé, retourne celui-ci*
- `size()` *qui retourne le nombre d'éléments présents sur la pile.*

Bien entendu, une tentative d'extraction d'un objet d'une pile vide se soldera par une erreur (qui sauf traitement de celle-ci conduira à faire planter le programme...)

Une représentation commode d'une pile pourrait être la suivante :



Comme un empilement de livres, on retire aisément le dernier ajouté, tout en haut de la pile, et on en ajoute un par-dessus le dernier tout aussi facilement.

Définition 2.2 *Une* file d'attente *(*queue *pour les anglo-saxons) comme une pile est une structure permettant également de contenir un certain nombre de données mais dont les données sont cette fois-ci retirées dans le même ordre que les ajouts.*

Comme son nom l'indique, une file d'attente *modélise par exemple l'ensemble des élèves de Kju attendant leur tour au self (on oublie bien sûr les VIP qui passent sur le côté !) et les anglos-saxons parlent cette fois du principe FIFO (first-in-first-out).*

Les fonctions les plus importantes qui régissent le fonctionnement d'une file d'attente sont :

- `enqueue(object)` *qui ajoute un objet à la file d'attente*
- `dequeue()` *qui retourne et retire un objet de la file d'attente : le plus anciennement ajouté parmi ceux restants.*

et comme pour les piles, on implémente souvent aussi :

- `isEmpty()` *qui retourne* `True` *ou* `False` *selon que la file d'attente est ou n'est pas vide*
- `size()` *qui retourne le nombre d'objets contenus dans la file d'attente.*

Exercice 1 *Reprendre les quatre exemples décrits en préambule pour décider laquelle de ces deux structures, piles ou file d'attente, répond au problème posé.*

3 Implémentation : classes squelettes. Un peu de POO

L'utilisateur d'une implémentation d'une pile ou d'une file d'attente n'a pas à savoir comment cette structure est implémentée. Il lui faut juste savoir quelles sont les fonctions accessibles et la syntaxe, mais ce qui se passe en coulisses est sans importance pour l'utilisateur. L'utilisation de classes est assez appropriée à ce genre de problème, et en voici le squelette pour les piles :

```
class stack:
    """ Une pile """
    def __init__(self):
        # le code : ici on initialise la ou les variables dont on a besoin pour l'objet créé.

    def push(self, item):
        """ empile l'objet donné en paramètre """
        # le code ici

    def pop(self):
        """ dépile et retourne le dernier objet ajouté """
        # le code ici

    def peek(self):
        """ retourne le dernier objet ajouté (sans le retirer)"""
        # le code ici

    def isEmpty(self):
        """ comme son nom l'indique... """
        # le code ici

    def size(self):
        """ la taille, en nombre d'objets empilés """
        # le code ici
```

Bien sûr, le code présenté est tel quel non seulement incomplet, mais erroné, les fonctions étant vides (ce qui est interdit...)

Une fois le code correctement complété, et exécuté dans le shell courant afin de rendre accessible cet objet, on l'utilise ainsi : `p = stack()` crée et initialise une pile, que la variable `p` permettra de référencer.

Pour ajouter par exemple l'entier 1 à la pile, on écrira : `p.push(1)`

Pour dépiler le dernier objet, on écrira bien sûr `p.pop()`

3.1 Quelques éléments quant à la syntaxe des classes

La méthode `__init__()` n'est jamais appelée directement, mais elle l'est automatiquement à chaque fois qu'un nouvel objet est créé. C'est alors le moment idéal pour initialiser des variables. Par exemple, on pourrait

décider que les valeurs que contiendra la pile formeront un tableau (enfin, une liste en python...) qu'on nommera `items`, et pour ce faire on écrirait :

```
def __init__(self):
    self.items = []
```

self fait référence à l'objet qui est défini (et dans les autres méthodes, à l'objet duquel on invoque une méthode). Une fois la pile `p` créée par la commande `p = stack()`, alors sa variable d'instance `items` est directement accessible (mais ce n'est a priori pas souhaitable d'y accéder ainsi) par `p.items`

Au sein de chacune des autres méthodes `push, pop, peek`..., on accède à la variable `items` de l'objet concerné en écrivant : `self.items`.

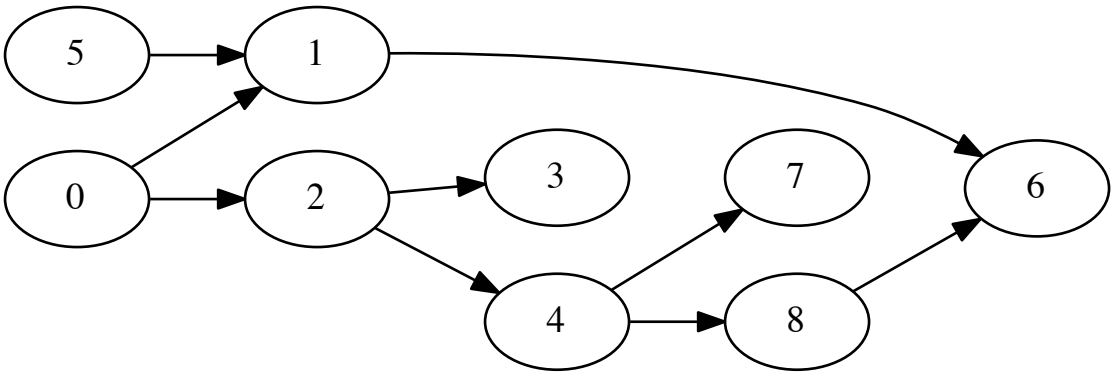
Enfin, la présence un peu surprenante au début de *self* dans les définitions de chacune des méthodes de la classe `stack` est ce qui permet de traduire un appel tel que `p.push(1)` en `stack.push(p, 1)` (les deux commandes sont équivalentes, mais la première est tout de même plus courte et lisible !)

Remarque 3.1 Certains langages utilisant la programmation orientée objets permettent l'*encapsulation* d'attributs (variables) et de méthodes (fonctions) en ne permettant leur accès que depuis l'objet lui-même (les méthodes qui en font partie sont donc les seules à pouvoir lire et écrire des valeurs pour ces attributs, ou à faire appel à ces méthodes dites privées) voire ses descendants (héritage).

Rien de tel n'existe avec python, mais la coutume est de préfixer d'un caractère de soulignement `_` le nom d'une variable ou d'une méthode que l'on considère comme privée. On peut rendre les choses un peu plus corsées en préfixant le nom d'une variable ou d'une méthode de deux caractères de soulignement : `__` qui conduit à ce que le nom depuis l'extérieur soit modifié en `_nomClasse__nomVar` mais dans les faits, même ainsi, les variables restent accessibles hors de la définition de la classe.

4 Application au parcours de graphes

Un graphe est la donnée d'un ensemble de noeuds connectés les uns aux autres par des arêtes, lesquelles peuvent être, ou non, orientées. D'un graphe dont les arêtes sont orientées, on parle de graphe orienté, et bien sûr de graphe non orienté si les arêtes ne sont pas orientées. L'exemple suivant est celui d'un graphe orienté :



Un algorithme de parcours de graphe cherche à déterminer à partir d'un noeud donné l'ensemble des noeuds que l'on peut atteindre en un certain nombre d'arêtes. Par exemple, dans l'exemple ci-dessus, depuis le noeud portant le numéro 0, les noeuds atteignables sont les noeuds $1, 2, 3, 4, 6, 7$ et 8, mais pas le noeud 5, duquel on ne peut atteindre que les noeuds $5, 1$ et 6.

L'utilisation d'une pile ou d'une file d'attente conduit à deux algorithmes classiques de parcours de graphe.

4.1 Avec une file d'attente : parcours en largeur

N.B. : l'algorithme décrit ici est souvent cité dans la littérature anglo-saxone sous le sigle BFS (breadth first search).

Etude d'un exemple : supposons que l'on souhaite parcourir le graphe précédent à partir du noeud numéroté 0.

On initialise une file d'attente, et on empile l'étiquette 0 du noeud, et on marque également le noeud 0 comme visité pour éviter d'y repasser plus tard (la question peut se poser en présence de cycle afin d'éviter que l'algorithme ne tourne en rond !)

Commence une boucle conditionnelle : tant que la file d'attente n'est pas vide, on extrait un élément, on observe quels sont les noeuds qu'on peut atteindre depuis celui-ci et qui n'ont pas été visités : on les marque comme visités et on les ajoute à la file d'attente.

Dans le cas précédent : après avoir extrait 0, on ajoute les étiquettes 1 et 2 qui sont les numéros des noeuds que l'on peut atteindre depuis le noeud 0, et on marque comme visités ces deux noeuds.

La file prend donc la forme : | 2 | 1 | (on ajoute à gauche et on retire à droite)

On extrait un élément, qui est alors 1, et on ajoute l'étiquette 6 (que l'on marque comme visité) et la file devient : | 6 | 2 |

Puis on extrait 2 et on rajoute 3 et 4 (et on marque les noeuds correspondants comme visités)

La file est maintenant constituée de : | 4 | 3 | 6 |

On extrait 6, puis 3 puis 4 et alors seulement on rajoute 7 et 8 et la file est alors | 8 | 7 |

On extrait enfin 7 et 8. A noter que le noeud 6 ayant été visité n'est pas rajouté à la file qui est alors vide, et le parcours s'achève.

Les noeuds ont ainsi été obtenus dans l'ordre suivant : $0, 1, 2, 6, 3, 4, 7, 8$.

Une propriété importante d'un parcours en largeur est que les noeuds sont obtenus selon la distance minimale au noeud initial : après le noeud initial sont obtenus les noeuds voisins de celui-ci, puis tous les noeuds qu'on peut atteindre par un chemin de longueur 2 et ainsi de suite.

Exemple de code (les graphes que l'on considère ont des noeuds numérotés à partir de 0 et sont représentés par des listes de listes, ainsi $G[i]$ est la liste des noeuds voisins du noeud i.

Le graphe présenté précédemment est donc donné par la liste :

```
[[1, 2], [6], [3, 4], [], [7, 8], [1], [], [], [6]]
```

```
def bfs(G, node):
    L = []
    visited = {node}
    Q = queue()
    Q.enqueue(node)
    while not Q.isEmpty():
        n = Q.dequeue()
        L.append(n)
        for w in G[n]:
            if w not in visited:
                Q.enqueue(w)
                visited.add(w)
    return L
```

4.2 Parcours en profondeur

L'idée d'un parcours en profondeur de graphe est, à partir d'un noeud initial, de se déplacer en suivant les arêtes du graphe jusqu'à ou bien arriver à une impasse, ou bien devoir repasser sur un noeud déjà visité. Parvenant à une telle situation, on revient en arrière jusqu'à pouvoir atteindre un nouveau noeud à partir d'un noeud déjà

visité et d'un arête non traversée jusque-là.

Il n'y a pas unicité d'un parcours en profondeur d'un graphe, à moins de s'imposer par exemple de se diriger vers le noeud voisin d'un noeud donné qui a la plus petite ou la plus grande étiquette...

Pour le graphe précédent, les parcours suivants répondent à la définition d'un parcours en profondeur :

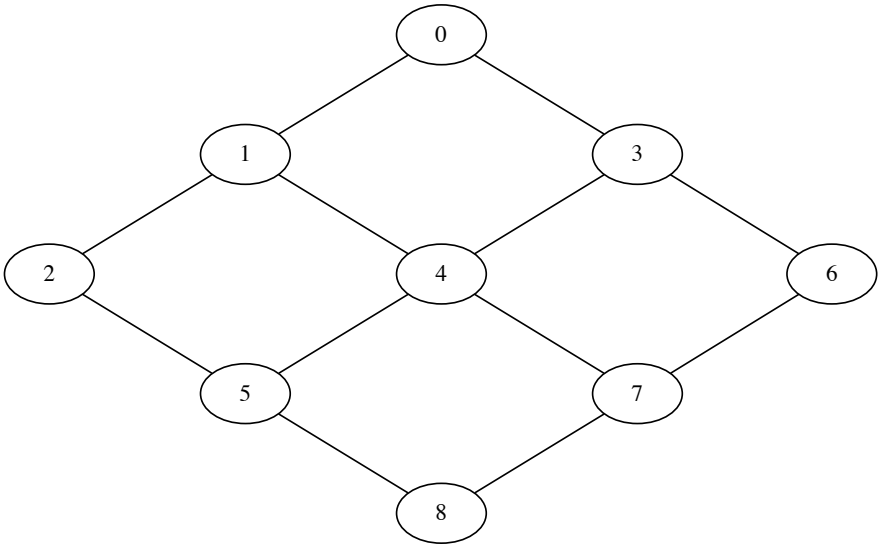
- 0, 1, 6, 2, 3, 4, 7, 8
- 0, 2, 4, 8, 6, 7, 3, 1
- 0, 2, 3, 4, 8, 6, 7, 1

Comme le retour en arrière n'est autre que le backtracking qu'on évoquait plus tôt, on se doute qu'une pile est judicieuse pour le permettre, mais il faut faire un peu attention... Si on modifie le code précédent pour remplacer la file d'attente par une pile, on obtient ceci :

```
def stackTraversal(G, node):
    L = []
    visited = {node}
    S = stack()
    S.push(node)
    while not S.isEmpty():
        n = S.pop()
        L.append(n)
        for w in G[n]:
            if w not in visited:
                S.push(w)
                visited.add(w)
    return L
```

Quel est l'ordre des sommets obtenu pour le graphe précédent ?

Et pour le graphe suivant, en partant toujours du sommet 0 ?



(On supposera que le graphe est donné par la liste :
`[[1, 3], [0, 2, 4], [1, 5], [0, 4, 6], [1, 3, 5, 7], [2, 4, 8], [3, 7], [4, 6, 8], [5, 7]]`)

Quelles modifications effectuer pour obtenir un vrai parcours en profondeur de ce graphe ?