Structures de données usuelles

Tableaux

Un tableau est une structure de longueur fixe (cf par exemple `numpy.array` en PYTHON)

La modification d'une valeur $a[i] = ...$ se fait en temps constant $O(1)$.

En machine, un tableau $a = (a_0, ..., a_{n-1})$ de longueur n est stocké par n cases *mémoires consécutives*. Ainsi, l'ordinateur peut avoir accès facilement à l'adresse de la case $a[i]$: il suffit d'ajouter i à l'adresse de la case contenant a_0 (laquelle est stockée dans la case mémoire du tableau a, ainsi que la longueur de a).

Piles et files d'attente

1. Piles (stucture naturelle des listes PYTHON munies des "méthodes" `append` et `pop`)

a) *Principe* : LIFO (*Last in, first out*)

L'élément extrait de la pile est parmi les éléments de la pile celui qui a été ajouté en dernier.

Complexité souhaitée : Les opérations d'ajout et de suppression doivent être en $O(1)$.

b) La structure naturelle de liste en PYTHON, munie des opérateurs `append()` et `pop()`, correspond précisement à la structure de pile : ajouts et suppressions se font *en fin de liste* en temps $O(1)$.

Remarque : En PYTHON, les listes (= les piles) sont en fait stockées par des tableaux de taille supérieure ou égale à la liste (la liste est alors définie d'une part par le tableau et d'autre part par le nombre d'éléments n de la liste, lequel varie au cours des opérations alors que la longueur du tableau est fixe) :

$L =$

x_0	x_1	x_2	...	x_{n-2}	x_{n-1}	*	*	*	*

L'instruction `L.pop()` est de complexité $O(1)$: il suffit de décrémenter n.

En revanche, il se peut qu'à force d'ajouts, la capacité initiale du tableau soit dépassée. Dans ce cas, on recopie le premier tableau dans un tableau plus grand (par exemple, deux fois plus grand). C'est pourquoi l'instruction `L.append(x)` est en $O(1)$ sauf lorsqu'un redimensionnement du tableau est nécessaire (coût amorti en $O(1)$).

Attention : Les fonctions `L.insert(i,x)` et `L.pop(i)` sont de complexité $O(n)$. Par exemple, l'insertion d'un élément en début de liste nécessite de décaler d'une place toutes les valeurs de la liste. Ces opérations ne correspondent pas à la structure de pile.

b) *Complément culturel* : Implémentation d'une pile à l'aide d'un tableau A de longueur N :

- $A[0]$ est la taille n de la pile (avec $n < N$)
- les éléments de la pile sont $A[1]$, $A[2]$, ... , $A[n]$ (où $A[n]$ est la tête de pile)

```
def append(A,x) :      # il faut avoir n + 1 < N
```

```
    n = A[0] ; assert ( n<len(A) , "dépassement de tableau" )
    A[n] = x ; A[0] = A[0]+1
def pop(A) :     # il faut avoir n > 0
    n = A[0] ; assert ( n>0 , "tableau vide" )
    A[0] = A[0]-1 ; return (A[n])
```

2. Files (files d'attente)

a) *Principe* : FIFO (*First in, first out*)

L'élément extrait de la file est parmi les éléments de la file celui qui a été ajouté en premier.

Complexité souhaitée : Les opérations d'ajout et de suppression doivent être en $O(1)$.

b) *Implémentation par une file à deux bouts* : On considère un tableau a de longueur n strictement plus grande que celle de la file, et deux entiers d et f définissant les indices de début (inclus) et de fin (exclu). Autrement dit, lorsque $0 \leq d \leq f < n$, la file est définie par $(a_d, a_{d+1}, ..., a_{f-1})$

Lorsque $0 \leq f < d < n$, on vient qu'il s'agit de la file $(a_d, a_{d+1}, ..., a_{n-1}, a_0, a_1, a_2, ..., a_{f-1})$.


```
def CreerFile(n) :  return [[0]*n,0,0]
def FileNonVide(F) :            ###    la file est codée par F = [a,d,f]
    return F[1] != F[2]         ###    avec 0 ≤ d < n et 0 ≤ f < n
def enfiler(x,F) :                  # variante :
    n = len(F[0])                   #  [a,d,f] = F ; n = len(a)
    F[0][F[2]] = x                  #  a[f] = x        # ceci modifie F
    F[2] = (F[2] + 1) % n           #  F[2] = (f+1)%n     # ne pas utiliser f ici
def defiler(F) :                    # variante :
    n = len(F[0])                       #  [a,d,f] = F ; n = len(a)
    x = F[0][F[1]]                      #  x = a[d]
    F[1] = (F[1] + 1) % n               #  F[1] = (d+1)%n     # ne pas utiliser d ici
    return x
```

Remarque : Si le nombre d'éléments de la file dépasse la capacité du tableau a, il faut alors recopier la file dans un tableau plus grand. On obtient une complexité amortie $O(1)$.

c) *Implémentation d'une file par deux piles* : on utilise une pile qui gère les ajouts et une pile qui gère les suppressions. Autrement dit, une file $(a_0, a_1, ..., a_{n-1})$ peut par exemple être codée par la liste $[p, q]$, où p et q sont les deux piles $q = (a_k, a_{k-1}, ..., a_1, a_0)$ et $p = (a_{k+1}, a_{k+2}, ..., a_{n-1})$.

Lorsque la seconde pile q est vide, on vide la première p dans q (en l'inversant l'ordre).

Complexité : Les opérations d'ajout et de suppression se font en temps amorti $O(1)$.

```
def CreerFile(n) :  return [[],[]]

def FileNonVide(F) :
     return F[0] or F[1]      # la valeur booléenne d'une liste non vide est True

def enfiler(s,F) :
     F[0].append(s)          # variante : p = F[0] ; p.append(s)

def defiler(F) :   # on suppose que F n'est pas vide
     if F[1] == [] :
          while F[0] :  F[1].append(F[0].pop())
     return F[1].pop()
```

d) *Remarque* : On pourrait implémenter en PYTHON à l'aide d'une liste L, avec les instructions :

```
L.append(y) ; x = L.pop(0)
```

mais l'instruction `L.pop(0)` a un coût trop élevé, en $O(n)$, où n est la taille de la liste L.

Files de priorité

Une file de priorité est une structure permettant de gérer un ensemble de couples (x, c), où c est la clé de l'objet x indiquant son degré de priorité, sur laquelle on peut effectuer trois opérations :

- insérer un élément ;
- extraire l'élément ayant la plus petite clé (plus la clé est petite, plus la priorité est grande)
- tester si la file de priorité est ou non vide.

Ainsi, elle permet d'implémenter efficacement des planificateurs de tâches, où un accès rapide aux tâches prioritaires est souhaité. On ajoute parfois à cette liste l'opération

- modifier la clé d'un élément (c'est-à-dire à dire modifier la priorité d'un élément), cette dernière opération étant par exemple utilisée dans la version améliorée de l'algorithme de Dijkstra.

a) **Implémentation d'une file de priorité par une file** : la suppression est en $O(1)$, mais l'inconvénient de cette structure vient du fait que les éléments ne sont pas ajoutés dans l'ordre de priorité. Autrement dit, le coût de l'ajout est en $O(n)$. Il en est de même de la modification de la clé d'un élément.

b) **Implémentation d'une file de priorité par un tas binaire** : L'ajout, la suppression et la modification d'une priorité se font par percolations et ont un coût $O(\log n)$:

Structure de tas binaires pour optimiser les files de priorité

En informatique, un tas (*heap* en anglais) est une structure de données arborescente qui permet de retrouver directement l'élément qu'on veut traiter en priorité (cf notion de file de priorité).

Un tas est un arbre binaire étiqueté et presque complet à gauche. Un arbre binaire est dit presque complet à gauche si tous ses niveaux sont remplis, sauf éventuellement le dernier, qui doit être rempli sur la gauche. Ainsi, il existe au plus un sommet admettant un unique fils.

Autrement dit, on considère un ensemble E dynamique et une fonction $k : E \to \mathbb{N}$ donnant le niveau de priorité : $k(x)$ est appelée clé (key) de x, et ici, plus $k(x)$ est petit, plus x est prioritaire.

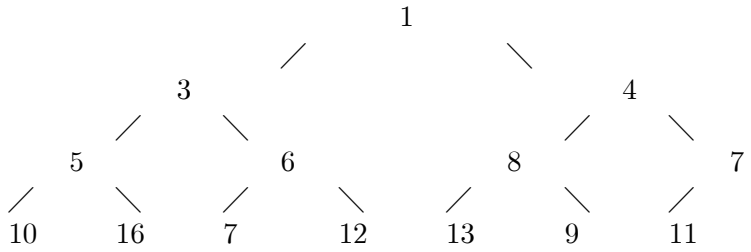
Dans un tas, les clés sont ordonnées le long des branches : un nœud parent a une plus haute priorité que ses fils (ou son fils). Autrement dit, $k(x) \leq k(y)$ si x est le parent de y.

Un tas est codé informatique par une liste $L = [a_0, ..., a_{n-1}]$ qui vérifie les propriétés suivantes :

$$\forall i, \quad \begin{cases} k(a_i) \leq k(a_{2i+1}) \text{ lorsque } 2i+1 < n \\ k(a_i) \leq k(a_{2i+2}) \text{ lorsque } 2i+2 < n \end{cases}$$

En particulier, la racine du tas (élément d'indice 0) est l'élément de clé minimale.

Exemple d'arbre binaire représentant un tas : **on a ici étiqueté les sommets avec la clé de priorité** :



Il correspond au tableau des clés :

1	3	4	5	6	8	7	10	16	7	12	13	9	11

Les opérations usuelles sur un tas sont les suivantes :

- *suppression de la racine* (et reconfigutation du tas)

- *ajout d'un nouvel élément* (et reconfiguration du tas)

Une opération supplémentaire possible (qui est nécessaire dans l'algorithme de Dijkstra) est :

- *modification de la priorité d'un élément* (et reconfigutation du tas).

e) Le module `collection` en PYTHON implémente la notion de file :

```
from collections import deque

L = deque([1,2,3]) ; L.append(4) ; L.appendleft(0)

print(L)   ###  renvoie  deque([0,1,2,3,4])
```

Dictionnaires (= tables d'association) et table de hachage

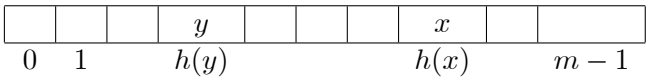
1) Un dictionnaire est une collection d'éléments de la forme (*clé,valeur*), où *clé* est un objet non mutable, (entier, flottant, chaînes de caractère) caractérisant l'élément conrrespondant (à chaque clé correspond une unique valeur ; il se peut d'ailleurs que la valeur soit constante). Une liste peut être représentée par un dictionnaire en prenant comme clés les entiers de 0 à $n-1$, mais dans ce cas, les cases mémoires ne sont plus consécutives en machine.

```
dico = {'a' :  421 , 'b' :  17 , 'c' :  256}
```

Une fois défini, on peut récupérer les éléments du dictionnaire : `dico['a']` vaut `421`.

L'implémentation se fait en utilisant une table de hachage.

2) a) Principe des tables de hachage : L'idée est de placer les n éléments à stocker dans un tableau de longueur m de sorte que les opérations élémentaires (recherche, ajout et suppression) se fassent en $O(1)$. L'idée fondamentale est que la position d'un élément x dans le tableau est définie comme une fonction de la clé. Autrement dit, on dispose d'une fonction de hachage qui à chaque clé k associe un entier $h(x) \in [\![m]\!]$. Pour limiter les collisions, il faut que m soit assez grand et que la fonction de hachage soit bien distribuée.



La position $h(x)$ dans la table est fonction de x.

b) Implémentation classique par listes : On considère un tableau H de longueur m de sorte que pour tout $0 \leq i < m$, $H[i]$ est la liste des éléments x dont la clé $h(x)$ vaut i.

La recherche, l'ajout et la suppression se font (en moyenne) en temps $O(L)$, où L est la longueur (moyenne) des listes $H[i]$ de la table. On dit que la complexité est en temps amorti $O(L)$.

Si la fonction de hachage est bien distribuée, le coût est en $O(1+\tau)$ où $\tau = \dfrac{n}{m}$ est le taux de remplissage de la table. En particulier, si m et n sont du même ordre de grandeur, les opérations se font en moyenne à *temps amorti constant*. Mais l'obtention d'une *bonne* table de hachage (avec une distribution presque uniforme) est souvent délicate en pratique, voire utopique.

Exemple : Pour un ensemble de n étudiants du lycée, on peut choisir pour h l'application $s \longmapsto s \bmod m$, où s est le numéro de sécurité sociale (qui correspond de fait à une clé). Pour déterminer si un étudiant est dans la table, on calcule $i = h(s) = s \bmod m$, et on parcourt la liste $H[i]$. En prenant m du même ordre de grandeur que n, on peut espérer que chaque $H[i]$ ne contienne qu'un petit nombre d'étudiants.

Ensembles et parties d'un ensemble

1) Ensembles. La structure dépend des objectifs souhaitées.

- Objectif souhaité : Tester rapidement l'appartenance d'un élément.

Implémentation conseillée : Liste des éléments triés selon un ordre judicieusement choisi.

Complexité :

- Le tri initial se fait en $O(n \ln n)$, qui est le coût optimal d'une liste de longueur n.

- La recherche (test d'appartenance) se fait ensuite par *dichotomie* en $O(\ln n)$.

Remarques :

- Il faut munir l'ensemble d'une relation d'ordre total.
- Les opérations d'ajout et de suppression d'un élément ne sont pas adaptées à cette structure (complexité en $O(n)$).
- En revanche, la fusion de deux parties se fait en temps linéaire.

Exemple : Un ensemble de points du plan est souvent muni de l'ordre lexicographique.

Exemple : On associe à une partie A de $[\![0, n-1]\!]$ sa fonction caractéristique $X = (x_i)_{0 \leq i < n} \in \{0,1\}^n$, c'est-à-dire $x_i = 1$ si $i \in A$ et $x_i = 0$ sinon.
On définit alors un ordre total sur A en considérant l'ordre lexicographique sur $\{0,1\}^n$.

- Objectif souhaité : On veut une structure d'ensemble dynamique

Autrement dit, on souhaite *que les tests d'appartenance, d'ajout et de suppression soient rapides,*

Implémentation conseillée : utilisation d'un dictionnaire : recherche, ajout et suppression se font en temps amorti $O(1)$. Les clés sont les éléments (et les valeurs sont arbitraires).

2) Parties d'un ensemble

De façon générale, une partie $A \subset E$ peut être considéré comme un ensemble, et ainsi être codé par exemple par la liste croissante de ses éléments (en supposant E muni d'une relation d'ordre).

Lorsque les éléments de E sont numérotés de 0 à $(n-1)$, une partie A d'un ensemble E dont les éléments sont numérotés de 0 à $(n-1)$ peut être codée par sa fonction caractéristique, c'est-à-dire un tableau a de longueur n telle que $a[i] = 1$ si $i \in A$ et $a[i] = 0$ si $i \notin A$.

Complexité :

- Pour la liste triée : recherche, ajout et suppression en temps $O(\log n)$.
- Pour la fonction caractéristique : recherche, ajout et suppression en temps $O(1)$

Remarque : Si $\operatorname{card} A$ est petit devant n, il vaut donc mieux utiliser des listes.

Drapeaux (*flag*), tableaux de marquage et de comptage

On peut être amené dans plusieurs situations à utiliser des variables ou des tableaux auxiliaires.
L'objectif en général est de mémoriser de l'information.

- Un drapeau est une valeur booléenne indiquant le résultat d'une opération ou le statut d'un objet.

Exemple :

```
01.     def sorted(L) :
```

```
02.         n= len(L) ; flag = True
03.         while flag :
04.             flag = False
05.             for i in range(n) :
06.                 for j in range(i) :
07.                     if A[i]<A[j] :  A[i],A[j] = A[j],A[i]
08.                     flag = True
09.         return A
```

- Un tableau de marquage peut être vu à ce titre comme une famille de drapeaux.

Exemple : Parcours (hamiltonien) d'un graphe pour indiquer les sommets déjà visités.

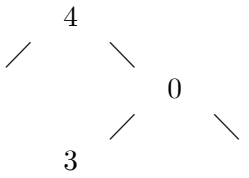
- Un tableau de comptage peut servir à compter les occurrences dans une liste.

Exemple : Dans le parcours eulérien d'un graphe (on veut passer une fois par chaque arête), on peut mémoriser pour chaque sommet le nombre d'arêtes non encore sélectionnées.

Arbres (*complément culturel*)

Un **arbre** (enraciné et numéroté) peut être considéré comme un cas particulier de graphe et être codé par **la liste des listes d'adjacence** (ici, la liste des fils de chaque sommet). Mais un arbre peut aussi être codé par un seul tableau (de longueur n, où n est le nombre de sommets), **le tableau des pères**, associant à chaque sommet son père (son unique précédesseur le long de l'unique chemin menant de la racine). On convient alors que le père de la racine est la racine elle-même. En temps linéaire $O(n)$, on peut passer des listes d'adjacence au tableau des pères et réciproquement

Par exemple,



est codé par le tableau des pères

4	4	0	0	4
0	1	2	3	4

Remarque : On peut en fait aussi coder un arbre par une liste de listes imbriquées.

L'arbre de l'exemple précédent ainsi peut se coder par `[4,[1,[0,[3,2]]]]`.

Un arbre a est alors soit un entier x (feuille) soit une liste $[x, L]$, où x est la racine et L la liste des sous-arbres dont ses fils sont les racines. On utilise alors le test `(type(a) == int)` pour savoir si a est réduit à une feuille.