Structure de données avancées et algorithmes

Ce document traite des structures de données classiques en algorithmique : listes, files, piles, arbres, graphes et de leur utilisation avec le langage Python.

Le parti pris est ici de décrire les structures de données à partir des opérations qui permettent de les manipuler.

Listes

Propriétés des listes

Une liste est une séquence d'objets ordonnés. Les opérations sont les suivantes :

- la liste est-elle vide ?
- quel est le premier élément de la liste ?
- quelle est la liste obtenue si on prive la liste de départ du premier élément ?

Ces 3 opérations sont réalisées en temps constant. Dans les listes chaînées, on peut généralement, à l'aide d'un pointeur, parcourir la liste en passant d'une case à la suivante. Accéder à la case numéro *i* est généralement réalisé en temps linéaire en *i*.

Les listes chaînées sont un exemple classique de manipulation de pointeurs en C. Les listes sont aussi utilisées comme base à d'autre structures de données.

Listes en Python

Les listes Python ne sont pas identiques aux listes décrite ci-dessus. Cependant, elles donneront entièrement satisfaction dans de nombreux cas :

- `len(s)` permet de connaître la longueur de la liste, donc de savoir si elle est vide
- `s[0]` permet de connaître le premier élément de la liste
- `s[-1]` permet de connaître le dernier élément de la liste

La notion de *queue* de liste n'est pas disponible simplement. On n'a pas en Python de “vue” sur une liste permettant d'en désigner une partir seulement (l'extension NumPy permet en revanche de le faire avec des tableaux de nombres).

Il est néanmoins possible de :

- recopier la queue de la liste : `q=s[1:]` mais ceci est réalisé en temps linéaire en la longueur de la liste, et non en temps constant

- transformer la liste en queue de liste, si on considère que la tête de liste est l'élément final : `s.pop()`. Ceci est fait en temps constant.

On peut de plus en Python accéder à une valeur par son numéro d'ordre et ceci en temps constant : `s[10]` est la onzième valeur de la liste.

Piles

Une pile est à comparer à une pile d'assiettes par exemple. Les concepts associés aux piles sont : LIFO (Last in, first out), Premier arrivé, dernier servi.

Opérations sur les piles

- La pile est-elle vide ?
- Ajouter un élément sur la pile
- Consulter le sommet de la pile
- Retirer l'élément au sommet de la pile

Piles en Python

Les listes Python permettent de simuler les piles :

- `len(p)` permet de savoir si la pile est vide
- `p.append(v)` ajoute `v` en sommet de pile
- `p[-1]` est l'élément au sommet de la pile
- `p.pop()` retire l'élément au sommet de la pile

Files

Une file est à comparer à une file d'attente de cinéma par exemple. Les concepts associés aux files sont : FIFO (First in, first out), Premier arrivé, premier servi.

Opérations sur les files

- La file est-elle vide ?
- Enfiler un élément
- Défiler un élément (et récupérer sa valeur). Cet élément sera le premier a avoir été enfilé.

Ces opérations doivent être faites en temps constant.

Files en Python

Les listes Python ne simulent pas correctement des files. En effet, les objets ne peuvent être enfilés

en temps constant qu'en fin de liste. Ils doivent donc être défilés en début de liste. Or l'insertion en début de liste ne se fait pas en temps constant (car Python décale les autres valeurs).

Il existe un type supplémentaire : `deque`, dans le module `collections` permettant de simuler les files (queue en anglais) :

- `from collections import deque` pour utiliser le module
- `d=deque()` : création d'une file vide
- `d=deque( (4,5,6) )` : création d'une file à partir d'un tuple
- `len(d)` permet de savoir si la file est vide
- `d.pop()` renoie l'élément de tête de file
- `d.appendleft(v)` ajoute un élément un fin de file

Début et fin sont interchangeables car il existe aussi les fonctions suivante qui s'exécutent en temps constant :

- `d.popleft()`
- `d.append(v)`

Le type de `dequeue` permet en outre de construire des files d'une certaines longueur maximum si bien que l'ajout en fin de file de nouveaux éléments *ejecte* les éléments de début de liste (ou l'inverse).

Arbres

Operations sur les arbres

- L'arbre est-il vide ?
- Quelle est la valeur à la racine de l'arbre ?
- Création d'un arbre à partir d'une racine et d'une liste de sous-arbres
- Parcourir la liste des fils de la racine (en temps proportionnel au nombre de fils)

Les arbres binaires sont des arbres dans lesquels chaque noeud a au plus deux fils. On accède alors en temps constant au fils gauche ou au fils droit.

Les arbres avec Python

arbres.py

Python ne dispose pas en standard de type permettant de représenter les arbres. Nous mettons à disposition le type `Arbre` qui permet de réaliser les opérations ci-dessus :

- un objet de type `Arbre` n'est pas vide. Il contient au moins une racine
- `a.racine` est le label de la racine de l'arbre (on peut le changer en écrivant : `a.racine=XXX`)
- `a=Arbre(v,fils=l)` crée l'arbre ayant une racine `v` qui a pour fils les arbres de la liste `l`
- un `Arbre` est itérable et on peut donc écrire : `for f in a : print(f.racine)` pour

afficher les fils de la racine a
- `len(a)` donne le nombre de fils de a
- `a[0]` donne une référence vers le premier fils de a

D'autres fonctionnalités sont disponibles, comme la création d'un arbre à partir de tuples imbriquées : *ar=(racine, fils1, fil2, ...)* où *fils1* et *fils2* peuvent eux même être des tuples sur le même modèle :

```
# La racine est 5. Elle a 3 fils : 3, 4 et 6. 6 a deux fils : 7 et 8
a=Arbre( (5,3,4,(6,7,8)) )
# Affichage sous forme de listes imbriquées lisibles
print(a)
# Affichage "évaluable"
a
```

On peut aussi remplacer un sous arbre particulier par un autre :

```
a=Arbre( (5,3,4,(6,7,8)) )
a.remplace(1,Arbre((7,8,9)))
print(a)
a.ajoute(Arbre(10))
print(a)
```

On peut accéder à un sous arbre par son numéro d'ordre :

```
a=Arbre( (5,3,4,(6,7,8)) )
b=a[2]
print(b)
```

On peut modifier un fils ou l'ajouter par son numéro d'ordre (si on ajoute le fils numéro 3 à un noeud qui n'a qu'un fils, la liste des fils est complétée par des None, et aucune exception n'est levée) :

```
b=Arbre((10,11,12))
a=Arbre(5)
a[1]=b
a[0]=3
a[1][2]=13
a[1][1]=42
```

Si la machine utilisée dispose des programmes `dot` (Graphviz) et `display` (ImageMagick), on peut affiche les arbres de manière graphique :

```
a=Arbre( (5,3,(4,2,1),(6,(7,10,11),8),2) )
a.view()
```

La classe `ArbreBinaire` du même module dérive de `Arbre` et permet de représenter plus spécifiquement les arbres binaires. La création des arbres se fait de la même manière (ou en passant un arbres quelconque en paramètres), mais les éventuels fils supplémentaires sont supprimés. On a de plus accès aux deux champs `a.fg` et `a.fd` pour lire ou mettre à jour le fils gauche et le fils droit :

```
a=ArbreBinaire(1)
a.fg=5
a.fd=6
a.fg.fg=7
a.fg.fd=9
a.view()
```

Certaines manipulations peuvent mener à la fabrication d'arbres incorrects. L'affichage permettra alors de repérer les problèmes :

```
a=Arbre((1,2,(3,4,5)))
a.remplace(0,a[1][1])
a.view()
```

Graphes