Leçon d'informatique : File de priorité

S. Benlhajlahsen - PCSI$_1$

Sommaire

Nous allons voir une nouvelle structure de données de la même famille : la **file de priorité**. En l'utilisant comme « sac » dans l'algorithme de parcours, nous allons aboutir à un parcours très intéressant.

I Définition

À retenir : Une file de priorité est une structure contenant plusieurs valeurs, et munie d'opérations **push** et **pop**. Elle ne mémorise pas l'ordre d'insertion des éléments, mais chaque élément est inséré avec une **priorité** et le pop fait ressortir l'élément prioritaire.

La figure 1 montre l'exemple d'une file de priorité où l'on a inséré quatre éléments (dans un ordre sans importance), puis sur laquelle on effectue deux pop, chacun sortant le plus petit élément.

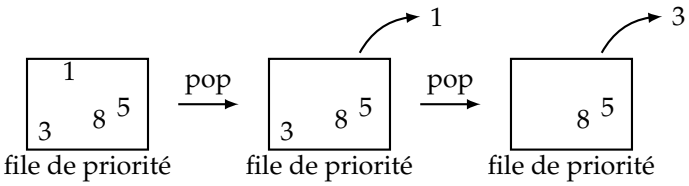

FIGURE 1 – Exemple de file de priorité où le critère d'extraction est le plus petit élément.

II Insertion dans une liste

Méthode `insert` : Python fournit une fonction `insert` qui permet d'insérer un élément à une position donnée dans une liste, avec une complexité en $O(n)$ car la liste est en fait recréée avec le nouvel élément à sa place.

II.A Exemple simple d'une liste de nombres

Exemple :

```
1 nombres = [8, 3, 5, 1]
2 valeur = 7
3 position = 2
4 nombres.insert(position, valeur)
```

Après exécution, `nombres` vaut `[8, 3, 7, 5, 1]` : la nouvelle valeur 7 a été insérée en position 2, autrement les éléments qui se trouvaient en positions 1 et avant n'ont pas bougé, et les éléments situés en position 2 et plus sont été repoussés d'un cran.

II.B Exemple avec une liste de couples

Dans le cadre d'une file de priorité, les éléments de la liste seront des couples `(valeur, priorité)` où `valeur` est quelconque, et `priorité` est un entier, et on veut que la liste soit triée dans l'ordre des priorités décroissantes.
Par exemple :

```
liste = [(12, 9), (37, 7), (19, 5), (77, 1)]
```

On veut insérer la valeur 25 avec la priorité 4, autrement dit l'élément `(25, 4)`. Pour cela, on écrit une fonction qui parcourt la liste à la recherche de la position d'insertion :

```
1 def position_insertion(liste, element):
2     for i in range(len(liste)):
3         if liste[i][1] < element[1]:
4             return i
5     return len(liste)
```

La boucle `for` est interrompue par le `return` ligne 4 si on trouve la bonne position, sinon le `return` ligne 5 couvre le cas où `element` a une priorité plus petite que celle de tous les éléments déjà dans la file de priorité. Dans ce cas, l'élément doit être ajouté **au bout** de la liste, et sa position doit être égale à la longueur de la liste.
Ainsi, `position_insertion(liste, (25, 4))` renvoie 3, que l'on peut utiliser avec `insert` :

```
element = (25, 4)
position = position_insertion(liste, element)
liste.insert(position, element)
```

III Implémentation naïve

On peut faire une implémentation simple de la file de priorité avec une liste maintenue triée par priorité décroissante, de sorte que le **pop** sur la file de priorité sera un simple **pop** sur la liste. Par contre, le **push** nécessitera un peu plus de travail pour déterminer où insérer un nouvel élément.

III.A Fonctions d'interface

Une file de priorité sera ici une liste où les éléments seront des couples `(valeur, priorité)`.

```
1 def nouvelle_file_prio():
2     return []
3
4 def file_prio_vide(file_prio):
5     return len(file_prio) == 0
6
7 def file_prio_pop(file_prio):
8     return file_prio.pop()
```

Pour le push, le gros du travail a été déjà fait : c'est la fonction `position_insertion` qui détermine où insérer l'élément, et il ne reste plus qu'à appeler `insert`.

```
1 def file_prio_push(file_prio, element):
2     position = position_insertion(file_prio, element)
3     file_prio.insert(position, element)
```

`file_prio_pop` hérite de la complexité de `pop`, donc $O(1)$. Par contre, `file_prio_push` hérite de la complexité de `insert`, donc $O(n)$. C'est le point faible de cette implémentation naïve.

III.B Exemple d'utilisation

Notez que la valeur d'un élément (le premier élément du couple) n'est pas utilisée dans la manipulation de la file de priorité. Cette valeur peut donc être de n'importe quel type (entier, flottant, chaîne de caractères, liste…)

```
1 fp = nouvelle_file_prio()
2 file_prio_push(fp, ("Alice", 3))
3 file_prio_push(fp, ("Bob", 2))
4 file_prio_push(fp, ("Cécile", 4))
5 file_prio_push(fp, ("Didier", 7))
6 x = file_prio_pop(fp)
```

Après exécution, `x` vaut `("Bob", 2)` et cet élément a été supprimé de la file de priorité.

IV Utilisation de `heapq` - Hors programme

Le module `heapq` (pour « *heap queue* ») fournit une structure de file basé sur les tas (*heap* en anglais). Cette méthode est beaucoup plus efficace (coût logarithmique).

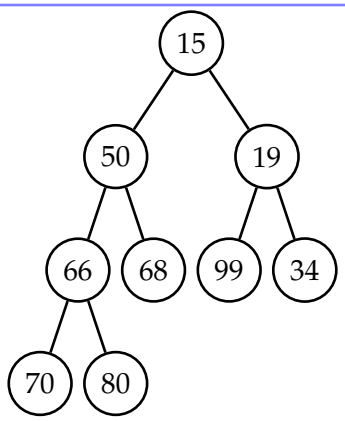

FIGURE 2

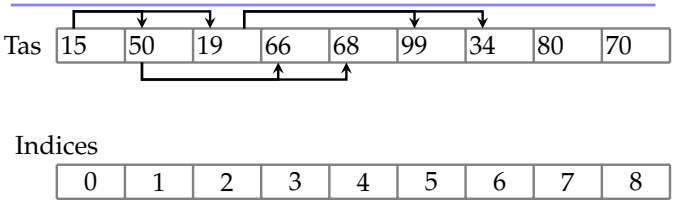

FIGURE 3

Tas : Un tas est une structure de donnée sous forme d'arbre binaire ordonnée. Si on prend l'exemple de la figure 2, l'élément prioritaire est le 15. Un élément est toujours prioritaire sur ceux de ses 2 fils (branches) : 50 est prioritaire sur 66 et 68 mais n'est pas prioritaire sur 34.
Cette implémentation utilise des tableaux (voir figure 3) tels que :

`tas[k] <= tas[2*k+1]` et `tas[k] <= tas[2*k+2]`

Méthode : Le module fournit plusieurs méthodes :

- `heappush` pour rajouter un élément dans la file de priorité.
- `heappop` pour faire sortir l'élément prioritaire.
- `heapify` qui transforme une liste en tas.

On prend la suite d'instructions ci-dessous :

```
1 import heapq
2
3 tas = [15, 50, 19, 66, 68, 99, 34, 80, 70]
4 x = heapq.heappop(tas)
5 print(tas)                          # renvoie [19, 50, 34, 66, 68, 99, 70, 80]
6 heapq.heappush(tas, 55)
7 print(tas)                      # renvoie [19, 50, 34, 55, 68, 99, 70, 80, 66]
```

- la ligne 4 fait sortir l'élément prioritaire qui est tout en haut.
- le premier `print` montre que le tas est réordonné pour conserver la priorité de gauche à droite.
- le second `print` montre que le tas est stable après un ajout.

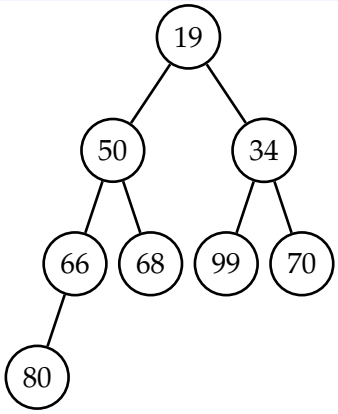

FIGURE 4

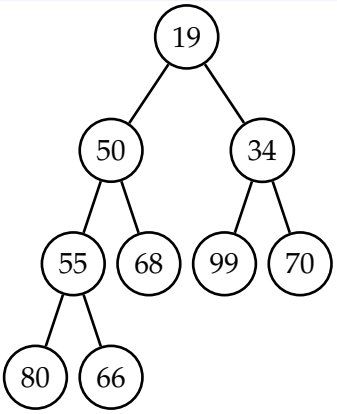

FIGURE 5

V Conclusion

Application des files de priorité Les files de priorité permettent d'implémenter les planificateurs de tâches où un accès rapide aux tâches d'importance maximale est souhaité. On la retrouve par exemple dans les ordonnanceurs des systèmes d'exploitation, notamment le noyau Linux.
La file de priorité sera aussi très importante dans la recherche du plus court chemin de l'algorithme de Dijkstra.