Devoir à la maison

L'objectif de ce devoir est d'étudier différentes implémentations possibles des files de priorités ainsi que de manipuler une des structures de données bien adpatées à cette implémentations : les tas.

1 Files de priorité

Une file de priorité est un objet contenant des éléments ayant chacun une priorité représentée par un nombre entier positif. On veut pouvoir effectuer les deux opérations suivantes de manière efficace :
– `push(x, f)` qui ajoute un nouvel élément de priorité `x` dans la file ;
– `pop(f)` qui renvoie et supprime de la file l'élément ayant la plus grande priorité.
Les files de priorité sont donc très proches des piles et des files (les opérations `push` et `pop` sont semblables) mais au lieu de ressortir les éléments dans un ordre dépendant de l'ordre dans lequel ils ont été insérés, on veut les ressortir dans un ordre qui dépend de leur priorité.
Remarque : En général, les éléments que l'on insère dans une file de priorités ont une valeur et une priorité. Cependant, d'un point de vue algorithmique la valeur de l'élément n'a aucune importance, seule compte la valeur de sa priorité. Dans tout ce devoir, on assimilera donc des éléments à leur priorité, c'est-à-dire que l'on insèrera directement des entiers dans la file, et que la fonction `pop` devra simplement renvoyer le plus grand entier contenu dans la file de priorités.

1.1 Avec des listes

Une première représentation des files de priorité consiste à placer les différents éléments dans une liste. Selon que l'on maintient les éléments triés ou non, on peut effectuer plus rapidement l'insertion ou la recherche.

1. En utilisant la définition des listes chaînées vue en cours (une classe `Liste` avec un champ `premier` et une classe `Noeud` avec deux champs `valeur` et `suivant`), écrivez les fonctions `push(x, f)` qui insère le nouvel élément en début de liste et `pop(f)`.

2. Ré-écrivez les fonctions `push(x, f)` et `pop(x)` en utilisant maintenant des listes ordonnées (les plus grands éléments en tête).
Remarque : La fonction `push` est plus compliquée parce qu'il faut insérer l'élément au bon endroit, mais la fonction `pop` est plus simple.

3. Quelles sont les complexités des fonctions `push` et `pop` dans chacun des deux cas ?

1.2 Avec des tas

Les tas sont une structure de données particulièrement adaptée pour représenter les files de priorité. Un tas est un arbre binaire vérifiant les deux propriétés suivantes (voir l'exemple représenté sur la figure 1.2) :
– pour tout nœud de l'arbre, la valeur du nœud est supérieure à la valeur de chacun de ses fils.
– à l'exception du dernier, tous les niveaux de l'arbre sont pleins, et le dernier niveau est rempli par la gauche.
La deuxième condition impose la forme du tas en fonction du nombre de nœuds qu'il contient : on remplit les niveaux un par à un par la gauche, quand un niveau est plein on descend sur le suivant. Cependant, à la différence d'un arbre binaire de recherche où la condition d'ordre sur les valeurs des nœuds impose une unique répartition possible des valeurs une fois la forme de l'arbre décidée, la contrainte sur les valeurs dans un tas est plus faible et l'on pourrait permuter certaines valeurs (par exemple, sur le tas de la figure 1.2, ce serait toujours un tas si l'on échangeait les valeurs 4 et 3).

4. Montrez par récurrence que la plus grande valeur contenue dans un tas se trouve nécessairement sur la racine.

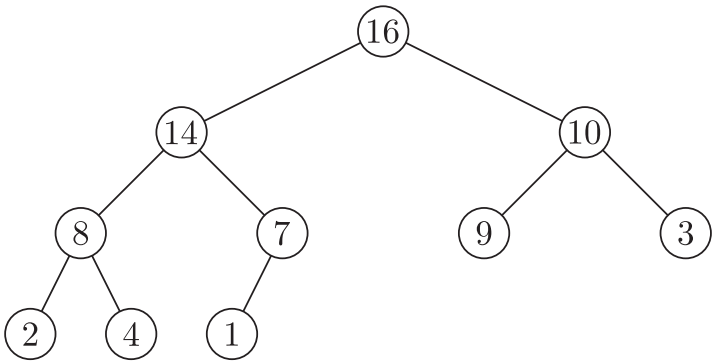

FIGURE 1 – Un exemple de tas.

On choisit de représenter les tas comme on a représenté les arbres binaires en cours : une classe `Arbre` avec un champ `racine` et une classe `Noeud` avec quatre champs `valeur`, `gauche`, `droit` et `parent`.

1.2.1 Insertion dans un tas

Pour insérer un nouvel élément dans un tas, on va dans un premier temps l'insérer à la première position disponible sur le dernier niveau de l'arbre (dans l'exemple de la figure 1.2 on insèrerait la nouvelle valeur à droite du 1, comme fils droit du nœud 7). On s'assure ainsi que l'on respecte la condition de remplissage des tas.
Après cette insertion, il est possible que l'arbre ne soit plus un tas si la valeur insérée est plus grande que la valeur de son père. Il va alors falloir corriger l'arbre en permutant des valeurs entre deux nœuds pour obtenir un tas.

5. Écrivez la fonction `inserer(x, d)` qui ajoute un nœud contenant la valeur `x` dans un tas `t` en connaissant le dernier nœud `d` qui avait été ajouté dans le tas et qui renvoie le nœud qu'elle vient d'ajouter (qui est maintenant le dernier nœud qui a été ajouté dans le tas).

Indications : Il faut remonter le long des ancêtres de `n` jusqu'à trouver un ancêtre dont `n` est dans le sous-arbre gauche, puis redescendre dans le sous-arbre droit de cet ancêtre en trouvant la feuille la plus à gauche de ce sous-arbre. Il est conseillé d'essayer de trouver à la main la position où inserer les prochains nœuds dans quelques tas pris comme exemples.

Attention : Il y a un cas particulier à traiter lorsque le dernier niveau de l'arbre est plein (il faut alors insérer le nouveau nœud tout à gauche du prochain niveau).

Si la valeur du nœud que l'on ajoute dans un tas est plus grande que la valeur de son père, la condition d'ordre sur les valeurs du tas n'est plus vérifiée. Pour la corriger, on va alors permuter les valeurs de certains nœuds en faisant remonter les plus grandes valeurs vers la racine jusqu'à vérifier la condition d'ordre des tas.
On considère la fonction suivante :

```
def corrige(n):
    if n.parent != None and n.valeur > n.parent.valeur:
        x = n.valeur
        n.valeur = n.parent.valeur
        n.parent.valeur = x
        corrige(n.parent)
```

6. Expliquez ce que fait la fonction `corrige(n)` si on l'appelle immédiatement après avoir inséré une nouvelle valeur dans un tas à l'aide de la fonction `inserer(x, n)` avec comme argument le nœud qui a été créé par cette fonction.