TP Python – Structures de données et algorithmique 3

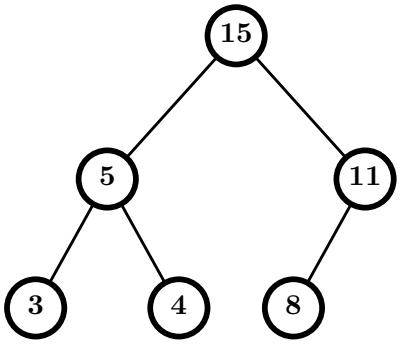
Objectifs :
- Implémentation par un **tas** de l'ADT **file de priorité**
- Implémentation d'un algorithme de **tri optimal** et **en place**

Remarque préliminaire : Tous les langages de programmation désignent par le terme « tableau » (*array*) un espace de mémoire contigu dont chaque case peut être indexé en temps constant. Sauf Python, dont les concepteurs ont choisi le terme « list ». Les tableaux mentionnés dans ce TP sont donc à interpréter comme des « listes » en Python.

Le but de ce TP est d'implémenter un tri de complexité temporelle en pire cas optimale, le **tri par tas** (*heapsort*). Un *tas* (*heap*) est l'une des implémentations possibles d'une *file de priorité* sous la forme d'un arbre binaire qui maintient toujours l'élément le plus petit (*min-heap*) ou le plus grand (*max-heap*) à sa racine.

Pour implémenter le tas, on va utiliser un codage de l'arbre binaire dans un tableau : pour un tableau indexé de 1 à n, on trouvera les enfants droit et gauche du nœud d'index i aux index $2 * i$ et $2 * i + 1$ ($2 * i + 1$ et $2 * i + 2$ pour un tableau indexé à partir de 0), et le parent d'un nœud d'index i sera donc situé à la case d'index $\frac{i}{2}$ ($\frac{i-1}{2}$ si l'index commence à 0).

Il est possible d'utiliser une telle implémentation d'un tas lorsqu'on connaît à l'avance le nombre (maximal) d'éléments à insérer dans le tas (ce qui est le cas pour le heapsort). Elle permet alors de réaliser le tri « en place », directement dans le tableau, sans avoir à gérer de référence vers les enfants (ou le parent) d'un nœud. Pour un tri par ordre *croissant*, on utilisera alors une *max-heap*, i.e. un tas dont le plus grand élément est à la racine, pour pouvoir les retirer un à un et les placer à la fin du tableau.



Nœud	15	5	11	3	4	8
Indice	0	1	2	3	4	5

FIGURE 1 – Un tas dans un tableau.

En plus de respecter la propriété de tas, i.e. que les éléments d'un nœud soient toujours plus grand (ou plus petit) que leurs enfants, ces tas doivent être des arbres binaires « parfaits », i.e. tous les niveaux exceptés le dernier doivent être totalement remplis (cf. figure 1), et si le dernier ne l'est pas, il doit être

rempli de gauche à droite. Il ne peut donc pas y avoir de case du tableau non utilisée entre deux cases utilisées, ou encore toutes les cases vides sont le plus à droite possible.

Pour pouvoir réaliser le heapsort en place, on considérera que le tas est situé dans le début du tableau, de l'index 0 jusqu'à l'index *lim* inclus, et que la suite du tableau constitue les éléments (non triés) à insérer dans le tas. À l'initialisation, on peut considérer que $lim = 0$ (pour un tableau indexé à partir de 0) et que le premier élément du tableau constitue un tas contenant un seul élément.

Insertion

Pour ajouter l'élément suivant d'index $lim + 1$ au tas, on va le faire remonter jusqu'à sa place. On compare l'élément à ajouter avec son parent : s'il est inférieur, on s'arrête ; sinon on les échange et on recommence récursivement avec le parent tant qu'on n'a pas atteint la racine (cf. figure 2).

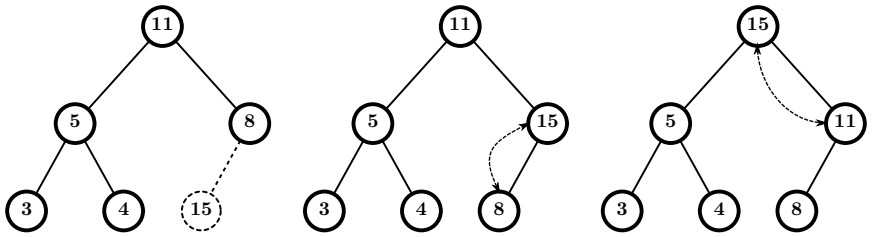


FIGURE 2 – Insertion.

On va ainsi insérer les éléments du tableau un à un dans le tas situé dans la première partie du tableau. Une fois tous les éléments insérés ($lim = n - 1$), on va retirer les éléments un à un du tas et les placer à la fin du tableau.

Suppression

On échange la racine du tas (premier élément du tableau) et le dernier élément du tas (d'index *lim*) ; l'ancienne racine sera donc correctement placée dans la fin (triée) du tableau. Puis on va faire descendre cet élément à sa place dans le tas. On compare l'élément avec ses deux enfants : s'il est supérieur, on s'arrête ; sinon on l'échange avec le plus grand de ses enfants et on recommence récursivement avec cet enfant (cf. figure 3).

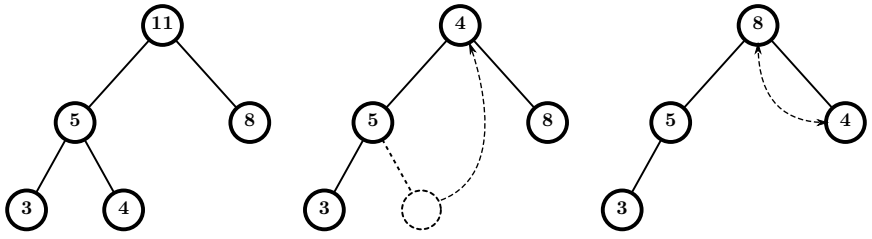


FIGURE 3 – Suppression.

1. Écrire la fonction `insert(l, i, cmp)` qui insère l'élément d'index `i` de la liste `l` dans le début du tableau considéré comme un tas, en utilisant la fonction de comparaison `cmp`.

   Cette fonction sera initialement passée en paramètre à la fonction de tri pour indiquer son ordre (croissant ou décroissant), puis la fonction de tri la transmettra à son tour à la fonction `insert`. Pour simplifier le raisonnement, on pourra considérer que `cmp` correspond à l'opérateur $<$ (ou plutôt à la fonction `operator.lt`), ce qui correspond à un tri dans l'ordre croissant. Le tas doit alors être une max-heap pour toujours insérer le *plus grand* élément en fin de tableau lors de la phase de suppression. On cherchera donc à faire remonter le plus grand élément vers la racine dans la fonction `insert`.

2. Écrire la fonction `heapify(l, cmp)` qui réalise l'insertion de tous les éléments de la liste `l` dans le tas.

3. Écrire la fonction `delete(l, lim, cmp)` qui supprime la racine du tas `l` occupant les cases indexées de 0 à `lim`. Le tas occupera donc une case de moins, i.e. jusqu'à `lim-1`.

4. Écrire la fonction `heapsort(l, cmp=operator.lt)` qui trie en place la liste `l` de taille `n`. On commencera par transformer la liste en tas puis on retirera ses éléments un à un en les plaçant à la fin du tableau et en maintenant la limite du tas à chaque étape.

5. Tester votre fonction de tri dans l'ordre croissant et décroissant à l'aide d'une liste aléatoire dont la taille sera passée en paramètre sur la ligne de commande :

   ```
   > python3 heapsort.py 15
   [138, 132, 38, 143, 52, 119, 135, 65, 119, 39, 7, 82, 33, 40, 90]
   [7, 33, 38, 39, 40, 52, 65, 82, 90, 119, 119, 132, 135, 138, 143]
   [143, 138, 135, 132, 119, 119, 90, 82, 65, 52, 40, 39, 38, 33, 7]
   ```

6. Calculer la complexité temporelle de l'algorithme. Le heapsort est-il stable ? Quelle est sa complexité sur un tableau déjà trié ?