# 1 File de Priorités

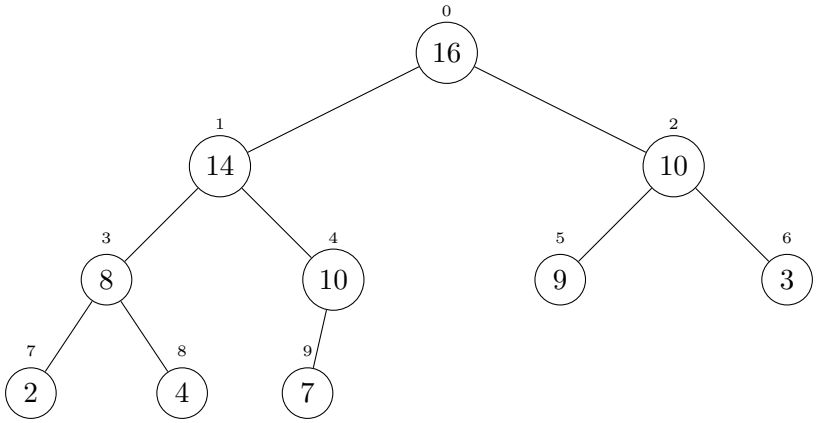


FIGURE1 – Exemple d'arbre représentant un tas

Une file de priorité est une structure de données abstraite dans laquelle on peut ajouter et retirer des éléments, et dans laquelle l'élément qui sort est toujours le plus grand élément de la file de priorité (celui qui a la plus haute priorité). Il existe plusieurs implémentations possibles, mais certaines sont plus efficaces que d'autres, comme le tas. Contrairement à une simple liste dont on rechercherait le maximum pour le sortir, l'utilisation d'un tas, permettrait d'ordonner correctement les éléments de la file de priorité afin d'éviter une recherche longue du maximum, et d'éventuels décalages à effectuer dans la structure.

Cette méthode consiste à gérer une structure d'ordre partiel grâce à un arbre. La file de priorité à **n** éléments est représentée par un arbre binaire contenant en chaque noeud un élément de la file de priorité (comme illustré dans la figure 1). L'arbre vérifie deux propriétés importantes : d'une part la valeur de chaque noeud est supérieure ou égale à celle de ses fils, d'autre part l'arbre est complet, propriété que nous allons décrire brièvement. Si l'on divise l'ensemble des nœuds en lignes suivant leur hauteur, on obtient en général dans un arbre binaire une ligne **0** composée simplement de la racine, puis une ligne **1** contenant au plus deux nœuds, et ainsi de suite (la ligne i contenant au plus $\mathbf{2^i}$ nœuds). Dans un arbre complet les lignes, exceptée peut-être la dernière, contiennent toutes un nombre maximal de nœuds (soit $\mathbf{2^i}$). De plus les feuilles de la dernière ligne se trouvent toutes à gauche, ainsi tous les nœuds internes sont binaires, excepté le plus à droite de l'avant dernière ligne qui peut ne pas avoir de fils droit. Les feuilles sont toutes sur la dernière et éventuellement l'avant dernière ligne.
On peut numéroter cet arbre en largeur d'abord, c'est à dire dans l'ordre donné par les niveaux de l'arbre. Les index sont représentés dessus des nœuds sur la figure 1. Dans cette numérotation on vérifie que tout noeud **i** a son père en position $\frac{i-1}{2}$, le fils gauche du noeud $i$ est $2i+1$, le fils droit $2i+2$. Formellement, on peut dire qu'un tas est un tableau **a** contenant $n$ entiers (ou des éléments d'un ensemble totalement ordonné) satisfaisant les conditions :

$$\begin{aligned} 1 \leq 2i+1 < n \quad &\Rightarrow \quad \mathbf{a}[2i+1] \leq \mathbf{a}[i] \\ 2 \leq 2i+2 < n \quad &\Rightarrow \quad \mathbf{a}[2i+2] \leq \mathbf{a}[i] \end{aligned}$$

Ceci permet d'implémenter cet arbre dans un tableau **a** où le numéro de chaque noeud donne l'indice de l'élément du tableau contenant sa valeur, comme indiqué dans le tableau ci-dessous :

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| **a[i]** | 16 | 14 | 10 | 8 | 10 | 9 | 3 | 2 | 4 | 7 |

Il vous est demandé d'implémenter cette version de file de priorité en respectant l'interface suivante :

```
  public interface FilePriorite<E> {
      void clear();
      Comparator<? super E> comparator()
      boolean isEmpty();
      boolean offer(E e);
      E peek()
      E poll();
      int size();
}
```

## 2 Implémentation pilotée par des tests

Les tests ont été écrits pour vous dans la classe `FilePrioriteTest`. Vous allez devoir implémenter la classe `FilePrioriteTas` qui contient les algorithmes d'utilisation de la file de priorité ainsi que des méthodes auxiliaires. Après chaque écriture de méthode, vous vérifierez que le test associé considère votre implémentation valide en faisant un clic droit sur la classe de test puis en choisissant "**Run As - JUnit test**" dans le menu contextuel.

**Q 1.**Classe `FilePrioriteTas`
Écrivez la classe `FilePrioriteTas` implémentant l'interface `FilePriorite`, avec les attributs nécessaires à l'implémentation de la structure proposée (tas dans un tableau), ainsi que les constructeurs (avec une taille par défaut, une taille spécifiée).

**Q 2.**Ajout d'un `Comparator`
Dans le cas où les éléments stockés ne sont pas comparables (au sens `Comparable<E>`, il faut fournir un `Comparator<E>` au constructeur du tas. Ajoutez un attribut pour garder trace de ce comparateur et faites une méthode privée qui émule la comparaison que les éléments soient comparables ou pas : `private int compare(E e1, E e2)`.
**N.B.** Cette méthode ne peut pas être statique car elle utilise le type générique `E`, et le comparateur.

**Q 3.**Méthode `toString`
Écrivez la méthode `String toString();` qui permet d'afficher la file de priorité sous la forme standard des collections Java : `[a, b, c, d, e, f, g, h]`.

**Q 4.** Méthode `size`
Écrivez la méthode `int size();` qui retourne le nombre d'élément dans la file de priorité.

**Q 5.** Méthode `isEmpty`
Écrivez la méthode `boolean isEmpty();` qui permet de tester si la file de priorité est vide ou non.

**Q 6.** Méthode `peek`
Écrivez la méthode `E peek()` qui permet de retourner l'élément maximum.

**Q 7.** Méthode `offer`
Écrivez la méthode `boolean offer(E e)` qui permet d'inserer un élément, si la file de priorité est pleine, l'insertion est impossible.

**Q 8.** Méthode `poll`
Écrivez la méthode `E poll()` qui eleve l'élément de priorité le plus élevé, vous retourne **null** quand la liste est vide.

**Q 9.** Méthode `clear`
Écrivez la méthode `void clear()` qui vide la file de priorité.

**Q 10.** Allocation dynamique
Modifiez votre structure de façon à doubler la taille du tableau si la file de priorité est pleine lors d'un ajout. De même, si le taux de remplissage passe sous les 50%, diviser la taille du tableau par 2.

**Q 11.** Méthode `toScreen` *(à faire en dernier)*
Écrivez une méthode `toScreen` qui affiche le tas sous forme arborescente. Le tas de la figure 1 donnerai, par exemple, l'affichage ci-dessous :

```
        __16__
       /      \
    _14_       10
   /    \     /  \
  8     10   9    7
 / \   /  \
2   4 7    6
```