Les files à priorité

par Jean-Philippe Collette

Date de publication : 7 juillet 2011

Dernière mise à jour : 7 juillet 2011

Cet article vous présente les files à priorité, ainsi que différentes implémentations, dont une réalisée en Java. Aucune connaissance préalable n'est requise. Bonne lecture !

I - Théorie ... 3
I-A - Les files à priorité ... 3
I-A-1 - Type de donnée abstrait ... 3
I-A-2 - Exemples d'utilisation ... 4
I-B - Les implémentations inefficaces ... 4
I-B-1 - Implémentation avec une liste liée ... 4
I-B-1-a - Présentation ... 4
I-B-1-b - Pseudo-code ... 5
I-B-2 - Implémentation avec un tableau ... 6
I-B-2-a - Présentation ... 6
I-B-2-b - Pseudo-code ... 7
I-C - Implémentation avec un tas ... 8
I-C-1 - Définition d'un tas ... 8
I-C-2 - Implémentation avec un tas ... 9
I-C-3 - Implémentation du tas dans un tableau ... 10
I-C-4 - Pseudo-code ... 11
I-D - Synthèse des temps d'exécution ... 12
I-E - Conclusion ... 13
II - Implémentation en Java ... 14
II-A - Introduction ... 14
II-B - Les comparaisons ... 14
II-C - Implémentation avec un tas ... 15
II-C-1 - Les constructeurs et variables membres ... 16
II-C-2 - La méthode add ... 17
II-C-3 - La méthode peek ... 17
II-C-4 - La méthode poll ... 18
III - Annexes ... 19
III-A - Implémentation avec une liste liée ... 19
III-A-1 - Les constructeurs et variables membres ... 19
III-A-2 - La méthode add ... 20
III-A-3 - La méthode peek ... 20
III-A-4 - La méthode poll ... 20
III-B - Implémentation non optimisée avec un tableau ... 21
III-B-1 - Les constructeurs et variables membres ... 21
III-B-2 - La méthode add ... 21
III-B-3 - La méthode peek ... 22
III-B-4 - La méthode poll ... 22
III-C - Implémentation optimisée avec un tableau ... 22
III-C-1 - Les constructeurs et variables membres ... 22
III-C-2 - La méthode add ... 22
III-C-3 - La méthode peek ... 23
III-C-4 - La méthode poll ... 23

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I - Théorie

I-A - Les files à priorité

Les files à priorités sont des structure de données permettant de stocker des éléments et de retrouver efficacement celui qui a la plus haute priorité.

Par exemple, on pourrait imaginer les urgences d'un hôpital : chaque nouveau patient serait ajouté à une telle file, et chaque fois qu'un médecin serait libre, il s'occuperait du patient avec l'état le plus critique. Le tri des patients se ferait sur des critères, comme l'état de conscience, s'ils respirent, ou s'ils saignent. On pourrait aussi vouloir mettre en avant les personnes plus jeunes, les femmes enceintes, etc.

Ce petit exemple nous conduit a une propriété fondamentale que les éléments à placer dans une file à priorité doivent avoir : il doit y avoir une relation d'ordre, si p est prioritaire par rapport à q, on a $p > q$.

Plus formellement, pour qu'un ensemble d'éléments e_i soit utilisable dans des files à priorités, on doit définir un ordre sur les éléments, une relation $\leq$ qui respecte les propriétés suivantes :

- propriété de totalité : pour deux clés e_1 et e_2, on a $e_1 \leq e_2$ ou $e_2 \leq e_1$ (cela implique la réflexivité ; $e \leq e$) ;
- propriété d'anti-symétrie : si $e_1 \leq e_2$ et $e_2 \leq e_1$, alors $e_1 = e_2$;
- propriété de transitivité : si $e_1 \leq e_2$ et $e_2 \leq e_3$, alors $e_1 \leq e_3$. Cela implique l'existence d'un minimum, un e_{min} tel que $e_{\text{min}} \leq e_i \quad \forall i$.

I-A-1 - Type de donnée abstrait

Passons au FAQ **type de donnée abstrait** :

- **add(élément)** : permet d'ajouter un élément dans la file ;
- **remove(élément)** : supprime un élément de la file ;
- **clear()** : vide une file ;
- **peek()** : permet de récupérer l'élément qui a la plus haute priorité, mais sans le retirer de la file ;
- **poll()** : permet d'extraire l'élément de plus haute priorité ;
- **size()** : retourne le nombre d'éléments dans la file.

Dans la partie pratique, nous implémenterons l'interface suivante :

```
public interface PriorityQueue<E>
{
 public void add(E element);
 public void clear();
 public E peek() throws PriorityQueueException;
 public E poll() throws PriorityQueueException;
 public int size();
}
```

L'exception des méthodes *peek* et *poll* sera lancée lorsque la file sera vide. Notez qu'il s'agit d'un choix personnel. Ainsi, dans l'implémentation pré-existante en Java, ces méthodes renvoient *null* si la file est vide. Voici le code (très simple) de mon exception :

```
public class PriorityQueueException extends Exception
{
 public PriorityQueueException()         { super(); }
 public PriorityQueueException(String s) { super(s); }
}
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I-A-2 - Exemples d'utilisation

Les files à priorité s'utilisent dans n'importe quel contexte et dans n'importe quelle situation. En voici quelques-unes :

- pour implémenter d'un algorithme de tri : avec un comparateur d'entiers, un tri par tas est très facilement réalisable ;
- dans un correcteur orthographique, lorsqu'on détecte une faute, on pourrait avoir plusieurs possibilités de correction. Par exemple, si j'écris "mre", on pourrait avoir les mots candidats "mère", "maire" ou encore "mare". En utilisant **la distance de Damerau-Levenshtein** (qui, en gros, donne le nombre de modifications basiques à effectuer pour passer d'un mot à l'autre), on peut mettre en évidence les mots qui ont la plus petite distance par rapport à ce qui a été écrit, donc les meilleurs candidats. Ainsi, pour "mre", on proposera d'abord "mare" (une modification, ajout de "a"), "mère" (une modification, ajout de "è") et enfin "maire" (deux modifications, ajout de "a" et "i"). La distance définit la prorité, mais en sens inverse ; plus la distance est courte, plus la priorité est grande ;
- dans **l'algorithme de Dijkstra**, il est nécessaire, pour un ensemble de noeuds, de trouver celui qui minimise la distance par rapport au noeud d'origine. Naïvement, pour N noeuds, à l'itération i, il faudra parcourir N - i noeuds, avec i \in [0 \dots N - 1], la complexité est quadratique (\mathcal{O}(N^2)). Avec une file à priorité basée sur un tas, on peut prouver que cette opération se fera en \mathcal{O}(N \log_2 N) ;
- etc.

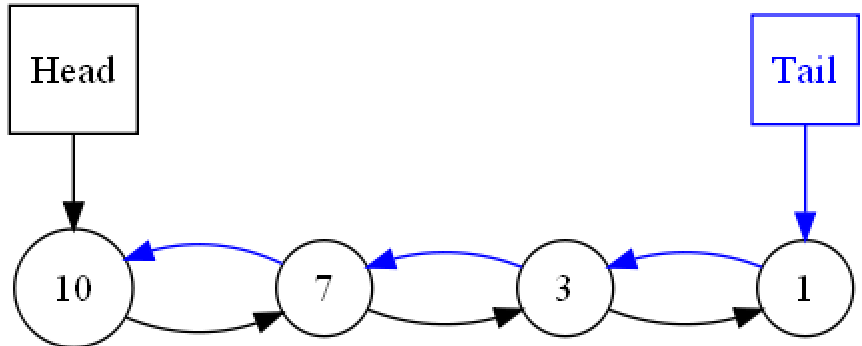
I-B - Les implémentations inefficaces

Nous allons tout d'abord voir deux implémentations intuitives et pourquoi elles ne sont pas du tout recommandables. Une implémentation efficace sera abordée par après.

I-B-1 - Implémentation avec une liste liée

I-B-1-a - Présentation

La première chose qui viendrait à l'esprit, lorsqu'on lit "file à priorité", c'est d'utiliser... une file ! L'élément avec la plus grande priorité serait la tête de file, les noeuds suivants ayant des priorités décroissantes. Par exemple, supposons que l'on a une file avec les entiers 3, 10, 7 et 1.
Une file peut s'implémenter avec une liste liée ou un tableau. Nous allons voir les voir tous les deux, en commençant par les listes.



Pour obtenir l'élément de plus grande priorité, il suffit de suivre le pointeur de tête (ici *head*) : c'est une opération en temps constant (\mathcal{O}(1)). Pour obtenir cet élément et le supprimer, le temps est également constant, il suffit simplement de faire pointer *head* sur le deuxième élément.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Les choses se compliquent pour l'insertion d'un élément : il va falloir parcourir la liste pour le positionner correctement. Ainsi, si un veut insérer 0, il faudra passer par tous les noeuds, avant de se rendre compte qu'il sera le dernier élément de la liste. Le parcours se fait en temps linéaire (\mathcal{O}(N), avec N le nombre d'éléments).
Une optimisation serait d'utiliser un autre pointeur, ici *tail*, qui redirige vers le dernier élément de la liste. Il suffira alors de comparer l'élément à insérer aux premier et dernier éléments de la liste, pour savoir s'il vaut mieux commencer au début ou à la fin. Avec 0, le temps sera ainsi constant, mais par contre on retrouve un temps linéaire s'il faut insérer 5. La complexité descend à \mathcal{O}(\frac{N}{2}) (on parcourt au pire la moitié de la liste), ce qui donne quand même une complexité linéaire. Cette astuce est à double tranchant, et peut ne pas bien fonctionner si la liste n'est pas "équilibrée", c'est-à-dire avec autant de petites priorités que de grandes, comme la liste 10, 7, 3 et 1. Si on veut insérer 2 dans la liste 10, 1, 1, 1, 1, 1, on perd au change.

Une liste doublement liée n'est pas obligatoire, une liste simplement liée suffit, sauf si on a l'utilité d'accéder aux éléments à partir des plus petites priorités. Les éléments d'une liste doublement liée sont représentés en bleu sur le schéma ci-dessus.

I-B-1-b - Pseudo-code

Supposons qu'on possède un pointeur *head* vers le noeud de tête, et que chaque noeud possède trois références : *next* vers le prochain noeud, *prev* vers le noeud précédent, et *element* vers l'élément qui lui est associé. On va également garder un champ *nbrObject*, qui comptera le nombre d'éléments dans la file.
On a ainsi le pseudo-code suivant pour l'ajout d'un élément :

```
ADD(e)
DEBUT

 p <- NOUVEAU_NOEUD()
 p.element <- e

 SI head = nil // Si la file est vide.
  head <- p
  tail <- p
 SINON

  // Recheche de la position de e dans la file.
  n <- head

  TANT QUE n != nil ET n.element > e
   n <- n.next
  FIN TANT QUE

  // Mise à jour des références.
  SI n = nil ALORS // Si e est à la fin de la file.
   tail.next <- p
   p.prev <- tail
   tail <- p
  SINON
   SI n.prev != nil ALORS
    n.prev.next <- p
   SINON
    head <- p
   FIN SI

   p.next <- n
   p.prev <- n.prev
   n <- p
  FIN SI
 FIN SI
FIN
```

Pour accéder à l'élément de plus haute priorité, il suffit de suivre la référence *head*.

```
PEEK()
DEBUT
 SI head != nil ALORS
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  RENVOYER head.element;
 SINON
  RENVOYER nil
 FIN SI
FIN
```

Pour l'extraction de l'élément, il ne faut pas oublier de mettre à jour la référence *prev*, dans le cas où vous souhaitez utiliser une liste doublement liée.

```
POOL()
DEBUT
 tmp <- PEEK()

 SI head != nil ALORS
  SI head.next != nil ALORS
   head.next.prev <- nil
  SINON
   tail <- nil
  FIN SI

  head <- head.next
  nbrObject <- nbrObject - 1
 FIN SI

 RENVOYER tmp
FIN
```

I-B-2 - Implémentation avec un tableau

I-B-2-a - Présentation

Pour implémenter une file, on peut aussi utiliser un tableau. Le premier élément serait celui qui a la plus grande priorité, les indices suivantes contenant les éléments de priorités moindres.

10
7
3
1

Comme avec une liste, récupérer le premier élément se fait en temps constant. Le temps du retrait du premier élément dépend de la manière avec laquelle vous utilisez le tableau : si vous voulez que le premier élément soit toujours celui qui a la priorité la plus élevée, une fois que vous l'aurez retiré, il faudra décaler toutes les valeurs. Ainsi, on aura un temps linéaire.
On pourrait procéder autrement, et définir un indice qui indique le début du tableau. Ainsi, après avoir récupéré le premier élément, il suffira d'incrémenter l'indice.

Voici ce qu'il se passerait si on retirait l'élément de plus haute priorité, à gauche en décalant tout le tableau, à droite en jouant avec l'indice.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

7			
3		7	← Head
1		3	
		1	

Jusque là, on pourrait penser être bien parti, avec tous ces temps constants... malheureusement, lors de l'ajout d'un élément, on sera obligé de décaler tout le tableau, qu'on le veuille ou non, quelque soit la méthode utilisée. Si on ne le fait pas, on ne respecte plus l'invariant qui dit que les éléments sont ordonnées selon leur priorité de manière décroissante (ou croissante, c'est à vous de choisir).
On a donc pour l'insertion un temps en \mathcal{O}(N).

Comme pour les listes liées, on pourrait définir un indice qui indiquerait la fin de la file. Cependant, vu qu'on peut connaître le nombre d'éléments dans la file, et qu'il y a un accès direct aux éléments d'un tableau, on peut se passer de ce pointeur.

I-B-2-b - Pseudo-code

Passons au pseudo-code ; supposons qu'on possède un tableau nommé *array*, ainsi qu'une fonction *swap(a, b)* qui échange les valeurs de a et de b. Voici ce qu'il faudrait faire pour l'implémentation non optimisée :

```
ADD(e)
DEBUT
 i <-  0

 TANT QUE i < nbrObject ET array[i] > e
  i <- i + 1
 FIN TANT QUE

 tmp <- e

 TANT QUE i != nbrObject
  swap(array[i], tmp)
  i <- i + 1
 FIN TANT QUE

 array[nbrObject] <- tmp
 nbrObject <- nbrObject + 1
FIN
```

```
PEEK()
DEBUT
 SI nbrObject > 0
  RENVOYER array[0];
 SINON
  RENVOYER nil;
 FIN SI
FIN
```

```
POOL()
DEBUT
 tmp <- PEEK()

 i <- 1
 TANT QUE i < nbrObject
  array[i - 1] <- array[i]
  i <- i + 1
 FIN TANT QUE

 nbrObject <- nbrObject - 1
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
 RENVOYER tmp
FIN
```

Passons à présent à la version optimisée, où on utilise un indice *head* qui pointe vers le premier élément de la liste dans le tableau. La fonction d'ajout est la même, si ce n'est qu'il faut intégrer cet indice. C'est surtout la fonction *pool* qui change, en bien car elle devient nettement plus courte. Supposons que *size* est la taille du tableau, on a alors :

```
ADD(e)
DEBUT
 i <-  0

 TANT QUE i < nbrObject ET array[(head + i) % size] > e
  i <- i + 1
 FIN TANT QUE

 tmp <- e

 TANT QUE i != nbrObject
  swap(array[(head + i) % size], tmp)
  i <- i + 1
 FIN TANT QUE

 array[(head + nbrObject) % size] <- tmp
 nbrObject <- nbrObject + 1
FIN
```

```
PEEK()
DEBUT
 SI nbrObject > 0
  RENVOYER array[head];
 SINON
  RENVOYER nil;
 FIN SI
FIN
```

```
POOL()
DEBUT
 tmp <- PEEK()

 head <- (head + 1) % size
 nbrObject <- nbrObject - 1

 RENVOYER tmp
FIN
```

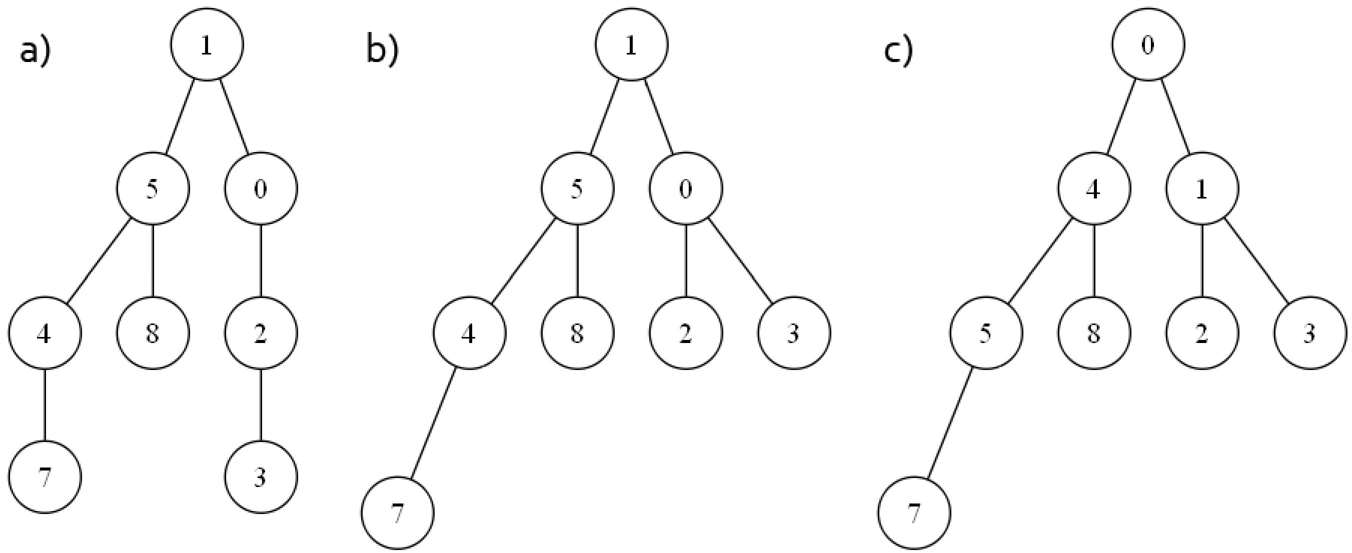
I-C - Implémentation avec un tas

Comme on a pu le constater, une file n'est pas spécialement le meilleur choix. On peut cependant trouver un compromis du point de vue des performances, grâce à une structure de FAQ **tas** FAQ **complet**

I-C-1 - Définition d'un tas

Un tas d'un arbre binaire (chaque noeud possède au maximum deux enfants) dont les étiquettes des noeuds sont des minorants (ou majorants) des sous-arbres. Cela signifie que les étiquettes de tous les noeuds des sous-arbres d'un noeud sont plus grandes (ou plus petites) que celle du noeud. On demande enfin que l'arbre soit FAQ **complet**, en d'autres mots, tous les niveaux de noeuds de l'arbre sont "pleins", sauf le dernier niveau.
Voici quelques exemples d'arbres possèdant une ou plusieurs de ces propriétés, qui sont plus explicites que la version texte.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

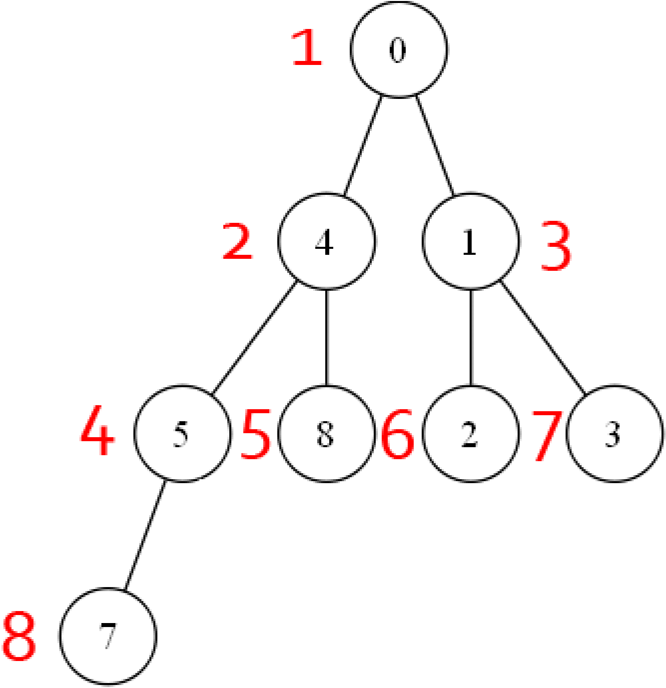


De gauche à droite : a) un arbre binaire b) un arbre binaire complet c) un tas complet

I-C-2 - Implémentation avec un tas

Nous allons donc utiliser un tas complet pour stocker les éléments. Que les étiquettes soient minorantes ou majorantes n'a pas d'importance, mais ici elles seront majorantes pour rester cohérent avec les précédentes implémentations.

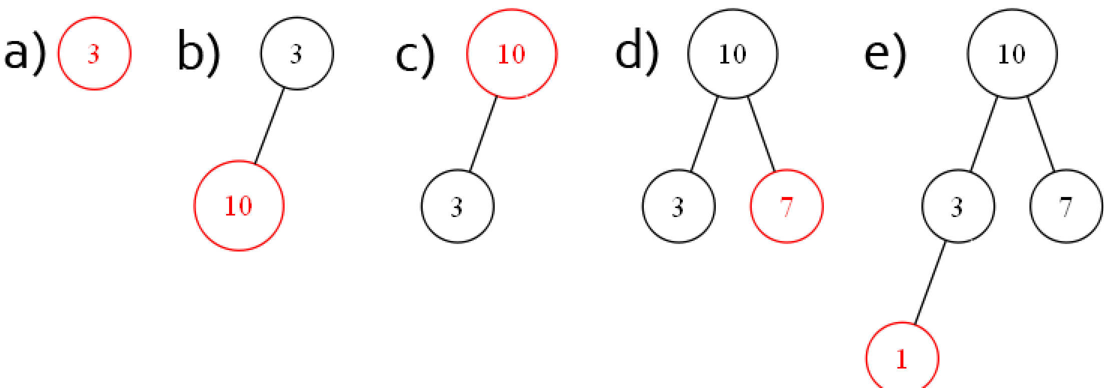
On va tout d'abord numéroter les noeuds, en partant de 1, de haut en bas et de gauche à droite. Ainsi, pour le tas des exemples, on aurait ceci :



C'est dans cet ordre que se construira le tas. Initialement, on a un tas vide ; le premier élément deviendra la racine, en 1.
Le second ira en 2, et on effectura un test avec son parent : si son parent est plus petit, on échange les valeurs des noeuds et on recommence le test sur le nouveau parent, sinon on s'arrête.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

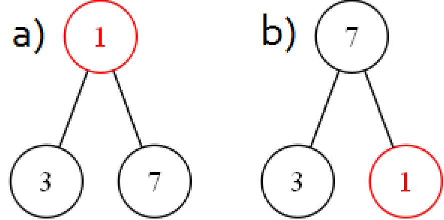
Pour reprendre notre exemple, on aura l'élément 3 en racine (a). Lorsqu'on ajoutera 10, il ira dans un premier temps en 2 (b). On effectue alors le test avec le parent : 10 est plus grand que 3, il prend sa place (c). Vu qu'après il n'y a plus de parent, on s'arrête.
Vient ensuite l'élément 7 : il sera placé en 3 (d). Vu qu'il est plus petit que son parent, on s'arrête. Il en est de même pour 1, qui sera placé en 4 (e) : 1 est plus petit que 3, inutile de les échanger.



C'était pour l'insertion. Le temps d'exécution dépendra du nombre de niveaux dans le tas ; dans le pire des cas, on partira d'une feuille (un arbre vide, tout en bas) et on remontra jusqu'à la racine. On a donc $\mathcal{O}(\text{Nombre de niveaux})$. Vu que l'arbre est complet, pour un nombre N de noeuds, il y aura $\lceil log_2(N) \rceil$ niveaux, donc l'insertion d'éléments dans le tas se fait en $\mathcal{O}(log_2(N))$.

Accéder à l'élément de plus haute priorité n'est pas compliqué, il suffit de retourner la racine du tas, un temps en $\mathcal{O}(1)$ est facile à obtenir. Par contre, les choses se corsent pour l'extraction. En effet, après avoir retiré le noeud racine, il faudra reformer le tas car, après cette décapitation, il est inutilisable.
L'idée est d'aller chercher l'élément le plus à droite dans le dernier niveau, en d'autres mots, celui qui a le plus grand numéro rouge sur les schémas, et de le mettre à la place de la racine. Ensuite, il faut le comparer à ses enfants : si la priorité d'un des enfants est plus grande, il faut les échanger pour rétablir la condition de tas. Cette opération est à répéter tant qu'on ne l'a pas, ou qu'on est au niveau des feuilles.

Reprenons l'exemple avec 3, 10, 7 et 1. On va donc retirer 10 et y placer l'élément 1 (a). Les deux enfants de 1 sont 7 et 3 qui sont tous les deux plus grands. On va prendre le maximum, soit 7, et l'échanger avec 1 (b), puis on reteste 1 avec ses enfants. Vu qu'il n'en a pas, l'algorithme se termine.

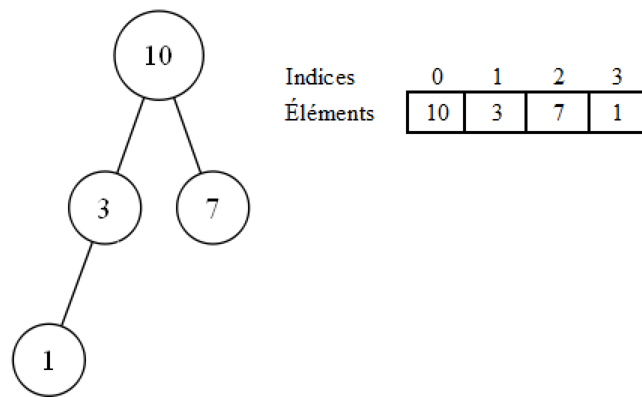


Comme pour l'ajout d'un élément, dans le pire des cas, on devra traverser tous les niveaux, depuis la racine jusqu'aux feuilles. La complexité est donc $\mathcal{O}(log_2(N))$.

I-C-3 - Implémentation du tas dans un tableau

Pour implémenter concrètement un tas, on pourrait utiliser une structure liée, avec des noeuds contenant des références vers d'autres noeuds. Il est également possible de le faire avec un tableau, une manière qui (à mon avis) est plus simple.
On va pour cela se baser sur la numérotation des noeuds de tout à l'heure, celle en rouge. On va en fait utiliser ces numéros comme indices pour les éléments dans un tableau (sans oublier de soustraire une unité, vu que généralement les indices d'un tableau commencent à 0). Pour reprendre l'exemple de tout à l'heure, on aurait ainsi :

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



On peut déduire des formules pour, à partir d'un indice, retrouver les enfants et parents d'un noeud ; pour un noeud d'indice i :

- son parent est donné par \lfloor \frac{i - 1}{2} \rfloor;
- son fils à gauche est donné par 2 \times i + 1 ;
- son fils à droite est donné par 2 \times i + 2.

Avec ces formules, on peut donc voyager dans le tableau tout en respectant la structure du tas.

I-C-4 - Pseudo-code

Supposons que les éléments sont stockés dans un tableau nommé *heap*. On peut alors définir l'opération d'insertion :

```
ADD(e)
DEBUT
 // Nouvel élément placé à la fin de la file.
 i <- nbrObject
 heap[i] <- e
 nbrObject <- nbrObject + 1

 p <- floor((i - 1) / 2)

 // Tant que la condition de tas n'est pas respectée, on monte dans les niveaux, en effectuant des
 échanges.
 TANT QUE p >= 0 ET e > array[p]
  swap(heap[i], heap[p])
  i <- p
  p <- floor((i - 1) / 2)
 FIN TANT QUE
FIN
```

L'élément de priorité maximale est le sommet du tas, soit l'indice 0.

```
PEEK()
DEBUT
 SI nbrObject > 0
  RENVOYER array[0];
 SINON
  RENVOYER nil;
 FIN SI
FIN
```

```
POLL()
DEBUT
 tmp <- PEEK()
 nbrObject <- nbrObject - 1

 heap[0] <- heap[nbrObject]
```

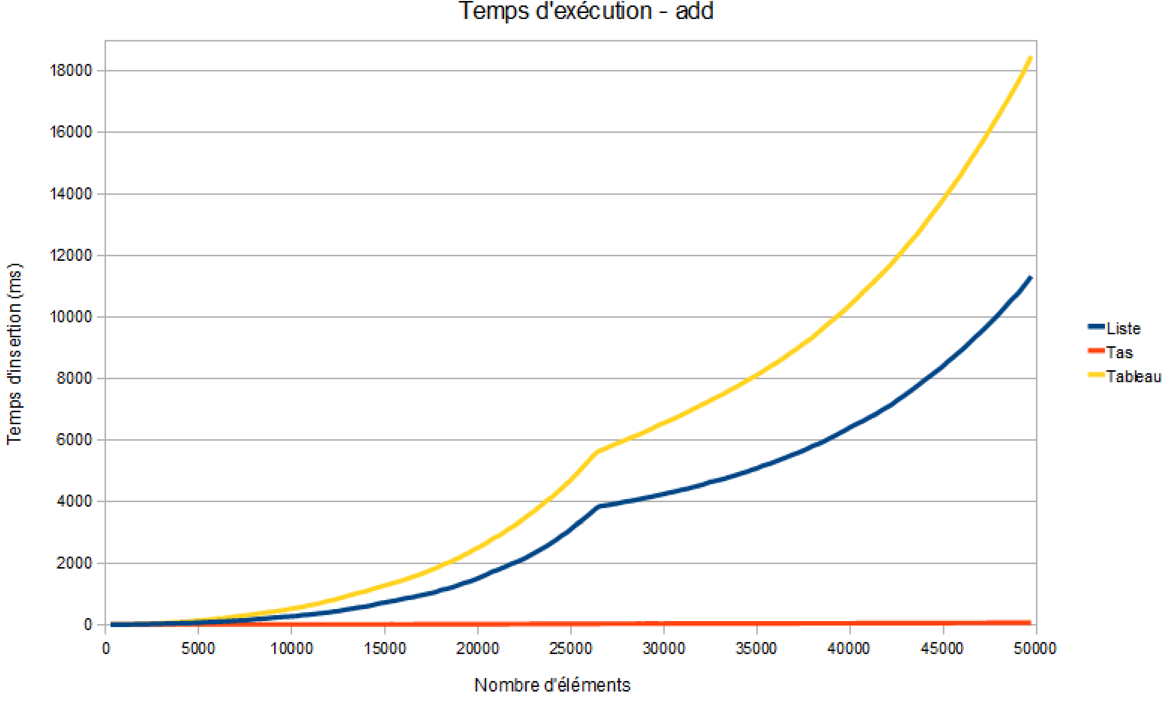

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
 i <- 0
 left <- 1
 right <- 2

 // Tant que la condition de tas n'est pas respectée, on descend.
 TANT QUE ( left <= nbrObject ET heap[left] > e ) OU ( right <= nbrObject ET heap[right] > e)

  SI right <= nbrObject ET heap[right] > heap[left]
   k <- right
  SINON
   k <- left
  FIN SI

  swap(heap[k], heap[i])

  i <- k
  left <- i * 2 + 1
  right <- i * 2 + 2
 FIN TANT QUE

 RETOURNER tmp
FIN
```

I-D - Synthèse des temps d'exécution

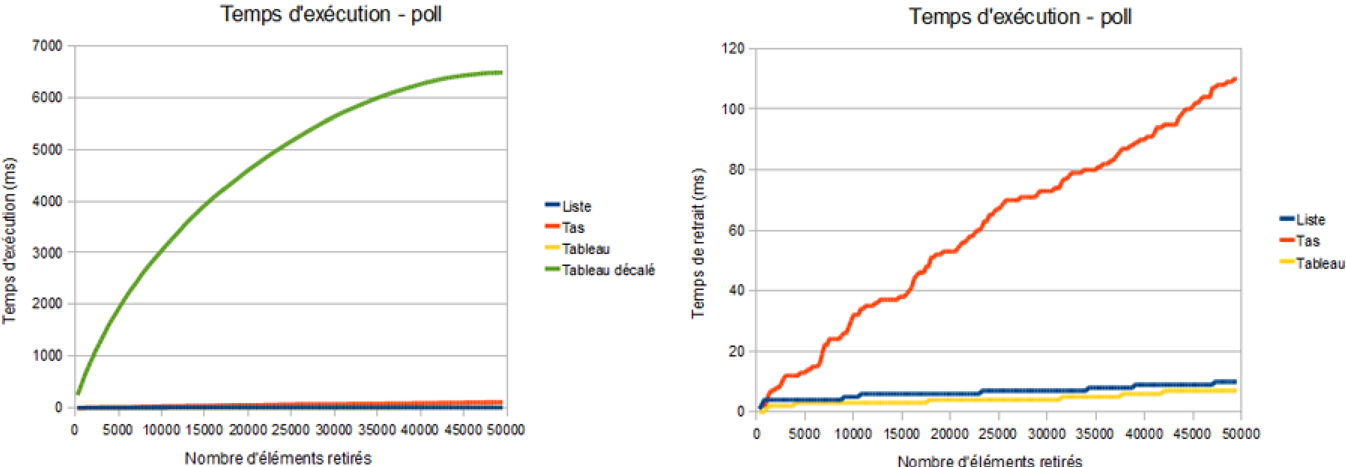
Implémentation	add	peek	pool
File	\mathcal{O}(N)	\mathcal{O}(1)	\mathcal{O}(N)
Tableau	\mathcal{O}(N)	\mathcal{O}(1)	\mathcal{O}(1)
Tas	\mathcal{O}(log_2(N))	\mathcal{O}(1)	\mathcal{O}(log_2(N))

Des tests reflètent très bien les caractéristiques de chaque implémentation. Voici les temps d'exécution accumulés que j'ai obtenu en insérant 50 000 entiers, choisis aléatoirement.

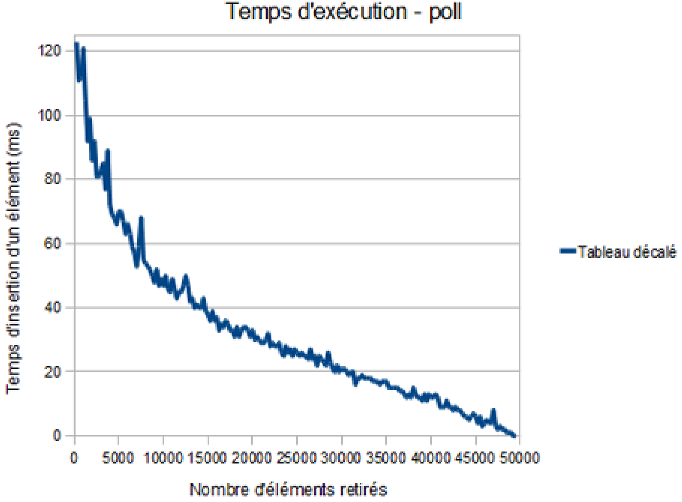


Il a fallu environ 11,5 secondes avec une liste, 18.7 secondes avec un tableau et... 67ms avec un tas !
Voici enfin ce qu'il se passe lorsqu'on retire les éléments. On y voit clairement l'intérêt d'optimiser l'implémentation avec un tableau, en utilisant des pointeurs au lieu de décaler les éléments. Le graphique de droite est le même que celui de gauche, l'implémentation avec un tableau décalé en moins, pour la lisibilité.

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



Pour retirer les 50 000 éléments, il aura fallu environ 6.5 secondes avec un tableau dont on décale les valeurs, environ 10ms avec une liste et un tableau avec des indices, et 111ms avec un tas.
On voit clairement l'impact du décalage dans le tableau : plus il y a d'éléments, plus le temps d'exécution augmente rapidement, au début du graphique. Cela se voit sur le graphique des temps d'insertion :



I-E - Conclusion

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II - Implémentation en Java

II-A - Introduction

Passons à présent à une implémentation concrète en Java.

Un excellent exercice serait de le faire par vous-même, et de corriger votre code par après. Le pseudo-code de la partie théorique devrait vous dépanner si vous bloquez.

Le but de cet article est de parler des files à priorité et non de Java. De ce fait, ne vous attendez pas à une implémentation optimale (par exemple, il n'y a pas de **synchronisation**). Mais nous allons quand même profiter de ses fonctionnalités, notamment pour coder une file à priorité que l'on pourra utiliser pour n'importe quel type d'éléments et avec n'importe quelle priorité.

Notez que Java possède déjà une **implémentation**, qui utilise un tas **comme on peut le voir dans le code source**.
Nous allons également utiliser un tas, vu tous les avantages, mais vour pouvez retrouver les autres implémentations dans les annexes, avec moins de commentaires.

II-B - Les comparaisons

Nous allons, avant de nous lancer dans l'implémentation, définir un comparateur : il permettra de comparer deux éléments d'une classe et de déterminer lequel est le plus grand (ou plus petit), ou s'ils sont égaux. Elle possèdera une méthode *compare* qui va renvoyer un entier : pour deux éléments a et b, *compare(a, b)* renverra

- -1 si a < b, ou
- 0 si a = b, ou
- 1 si a > b.

On redéfinit ainsi une relation <=. L'interface suivante sera à implémenter :

```
public interface Comparator<E>
{
 public int compare(E e1, E e2);
}
```

Par exemple, voici ce que l'on aurait si on voulait comparer des entiers :

```
public class IntegerComparator implements Comparator<Integer>
{
 public int compare(Integer e1, Integer e2)
 {
  int c = e1 - e2;

  if (c < 0)
   return -1;
  else if (c > 0)
   return 1;
  else
   return 0;
 }
}
```

Cela peut ressembler à du gaspillage, mais ce n'est qu'une mauvaise impression : l'exemple est trivial, et surtout ce petit travail en plus apportera deux gros avantages :

Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

1. on pourra comparer n'importe quel type d'éléments, en particulier des classes non-standards ;
2. on pourra changer très facilement l'ordre des priorités. Par exemple, dans le code plus haut, plus l'entier est grand, plus il est prioritaire. On pourrait imaginer le contraire, donc plus l'entier est petit et plus il est prioritaire : il suffit de ne pas retourner 1 mais -1, et inversément.

Ainsi, le code de la file à priorité ne changera pas d'un iota, seul le comparateur sera à implémenter.

II-C - Implémentation avec un tas

La première étape va être de définir une classe qui va servir de case dans le tableau. Il s'agit d'un simple conteneur générique, avec un mutateur et un accesseur, que nous allons appeler *Bucket*.
Cette classe définira également un mutateur qui modifie l'élément enregistré dans la case, et retourne l'ancien : *swapElement*. Elle permettra d'écrire moins de code par la suite.

```
public class Bucket<E>
{
 private E element;

 public Bucket()
 {
  this(null);
 }

 public Bucket(E element)
 {
  this.element = element;
 }

 public E getElement()
 {
  return this.element;
 }

 public void setElement(E element)
 {
  this.element = element;
 }

 public E swapElement(E element)
    {
        E tmp = this.element;
        this.element = element;
        return tmp;
    }
}
```

Ainsi, nous aurons dans les variables membres un tableau de ces cases. Cela génèrera un avertissement, mais il y en aurait eu plus si on avait utilisé un tableau d'**Object**. En effet, à chaque fois qu'il faudra renvoyer un élément, il faudra l'extraire du tableau sous forme d'Object et le caster dans la classe générique, ce que Java n'apprécie pas vraiment.

Nous aurons également besoin de voyager dans la structure de tas, ce que l'on peut faire facilement avec les formules de la partie théorique. On peut alors dédier trois méthodes pour ces calculs :

```
// Pour un index, calcule son parent dans le tas.
private int getParent(int index)
{
 return (int) Math.floor((index - 1) / 2);
}

// Pour un index, calcule son fils à gauche dans le tas.
private int getLeft(int index)
{
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.
http://jipe.developpez.com/articles/algo/file-priorite/

```
 return index * 2 + 1;
}

// Pour un index, calcule son fils à droite dans le tas.
private int getRight(int index)
{
 return getLeft(index) + 1;
}
```

Ensuite, il faudra aussi prévoir le cas où le tableau est plein. On va définir une méthode qui va doubler la taille du tableau lorsqu'il sera plein, et qui sera appelée lors de chaque insertion. Le code n'est pas compliqué, il s'agit d'une simple instanciation et d'une copie du tableau.

```
protected void resize()
{
 int l = array.length;

 if (this.nbrObject != l)
  return;

 head = 0; // Réinitialisation du pointeur de tête
 l *= 2;
 @SuppressWarnings("unchecked")
 Bucket<E>[] newheap = (Bucket<E>[]) new Bucket<?>[l];

 // Copie du tableau
 for (int i = 0; i < array.length; i++)
  newheap[i] = this.array[i];

 this.array = newheap;
}
```

On pourrait aussi gérer le cas où le tableau contient beaucoup de cases vides et peu de cases occupées ; on pourrait réduire sa taille, afin de ne pas gaspiller trop d'espace mémoire avec des cases inutilement réservées.

Enfin, nous aurons souvent besoin d'échanger deux éléments dans le tableau. Pour la clareté, on va définir une méthode *swap* qui va prendre deux indices, et procéder à l'échange. Cela évitera de répéter du code et de redéfinir une variable temporaire à chaque fois qu'il y en aura un.

```
/*
 * Echange les éléments situés aux indices a et b du tas. Les cases de ces
 * indices doivent être des instances de la classe Bucket.
 */
private void swap(int a, int b)
{
 E tmp = heap[a].getElement();
 heap[a].setElement(heap[b].getElement());
 heap[b].setElement(tmp);
}
```

II-C-1 - Les constructeurs et variables membres

Rentrons enfin dans le vif du sujet ! Comme variables membres, nous aurons donc besoin d'un tableau, ainsi que du nombres d'éléments insérés dans le tas. Il faudra également garder une référence vers une instance d'un comparateur.
Au niveau des constructeurs, il faudra prendre en paramètre un comparateur et éventuellement une taille de tableau. On définira également une constante, au cas où l'utilisateur n'aurait pas spécifié de taille de départ. De même, il faut vérifier que l'utilisateur n'entre pas n'importe quoi, comme des valeurs négatives, d'où l'appel à *Math.max*.

```
public class PriorityQueueHeap<E> implements PriorityQueue<E>
{
 private Bucket<E>[] heap;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  private Comparator<E> cmp;
  private static final int DEFAULTSIZE = 16;

  private int nbrObject = 0;

  public PriorityQueueHeap(Comparator<E> cmp)
  {
   this(cmp, DEFAULTSIZE);
  }

  @SuppressWarnings("unchecked")
  public PriorityQueueHeap(Comparator<E> cmp, int size)
  {
   this.cmp = cmp;
   this.heap = (Bucket<E>[]) new Bucket<?>[Math.max(DEFAULTSIZE, size)];
  }

  // peek, poll et add.
 }
```

II-C-2 - La méthode add

Pour ajouter un élément au tas, il faut le mettre au dernier niveau le plus à droite, autrement dit en dernière position dans le tableau. Ensuite, il faudra tester l'élément avec son parent, et éventuellement les échanger si la priorité de l'élément est plus grande que celle du parent. Cette opération est à répéter jusqu'à arriver à la racine, ou jusqu'à ce que l'élément ait une priorité inférieure à celle de son parent.

```
public void add(E element)
{
 // Faut-il redimensionner le tableau ou non ?
 resize();

 int index = nbrObject;
 this.heap[index] = new Bucket<E>(element);
 int parent = getParent(index);

 while ( parent >= 0 && cmp.compare(element, heap[parent].getElement()) > 0)
 {
  // Echange des deux éléments.
  swap(parent, index);

  // Déplacement vers le niveau supérieur.
  index = parent;
  parent = getParent(index);
 }
 nbrObject++;

}
```

II-C-3 - La méthode peek

Trouver l'élément de la plus haute priorité consiste à trouver la racine du tas. Or, c'est le premier élément... ce qui rend la méthode très simple :

```
public E peek() throws PriorityQueueException
{
 if (nbrObject > 0)
  return heap[0].getElement();
 else
  throw new PriorityQueueException("File vide.");
}
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

II-C-4 - La méthode poll

L'extraction de l'élément de plus haute priorité n'est pas compliqué, par contre le code est un peu plus lourd au niveau du rétablissement de la condition de tas.
Pour rappel, après avoir lu l'élément au sommet du tas, il faut le remplacer par l'élément le plus à droite du dernier niveau, autrement dit le dernier élément dans le tableau. Il faut ensuite effectuer un test triangulaire avec ses deux fils (s'ils existent), et effectuer des échanges si sa priorité est plus petite qu'une des deux autres.

Si le noeud courant n'a pas d'enfant, aucun échange n'est à effectuer. S'il n'y a qu'un seul enfant, il est obligatoirement à gauche, on les échange si l'enfant est plus grand.
Si le noeud courant possède deux enfants, on va échanger la valeur courante avec celle du plus grand enfant. Ainsi, on est sûr que le nouveau noeud courant respectera la condition de tas.

```
public E poll() throws PriorityQueueException
{
 E tmp = this.peek();
 nbrObject--;

 heap[0] = heap[nbrObject];
 int index = 0;
 int left = 1;
 int right = 2;


 while ((left <= nbrObject && cmp.compare(heap[index].getElement(), heap[left].getElement()) < 0)
  || (right <= nbrObject && cmp.compare(heap[index].getElement(), heap[right].getElement()) < 0))
 {
  int k;

  // k sera l'indice du plus grand fils.
  if (right <= nbrObject &&
   cmp.compare(heap[left].getElement(), heap[right].getElement()) < 0)
   k = right;
  else
   k = left;

  swap(k, index);
  index = k;
  left = getLeft(index);
  right = getRight(index);
 }

 return tmp;
}
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III - Annexes

III-A - Implémentation avec une liste liée

Pour commencer, on va définir une classe *ListNode* qui, comme son nom l'indique, va représenter un noeud dans une liste liée.

```
public class ListNode<E> {

    private E element;
    private ListNode<E> prev, next;

    public ListNode(E element) {
        this(element, null, null);
    }

    public ListNode(E element, ListNode<E> prev, ListNode<E> next) {
        this.element = element;
        this.prev = prev;
        this.next = next;
    }

    // Accesseurs et mutateurs.
    public E getElement() {
        return this.element;
    }

    public ListNode<E> getNext() {
        return next;
    }

    public ListNode<E> getPrev() {
        return prev;
    }

    public void setPrev(ListNode<E> prev) {
        this.prev = prev;
    }

    public void setNext(ListNode<E> next) {
        this.next = next;
    }

    public boolean hasNext() {
        return this.next != null;
    }

    public boolean hasPrev() {
        return this.prev != null;
    }
}
```

III-A-1 - Les constructeurs et variables membres

```
public class PriorityQueueList<E> implements PriorityQueue<E>
{
 private ListNode<E> head, tail;
 private int nbrObject;

 private Comparator<E> cmp;

 public PriorityQueueList(Comparator<E> cmp)
 {
  this.cmp = cmp;
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    this.clear();
  }

  // peek, poll et add.
}
```

III-A-2 - La méthode add

```
public void add(E element)
{
 nbrObject++;
 ListNode<E> newNode = new ListNode<E>(element);
 if (head == null)
 {
  this.head = newNode;
  this.tail = newNode;
  return;
 }

 ListNode<E> n = this.head;
 while (n != null && cmp.compare(element, n.getElement()) < 0)
  n = n.getNext();


 if (n == null)
 {
  tail.setNext(newNode);
  newNode.setPrev(tail);
  tail = newNode;
 }
 else
 {
  if (n.hasPrev())
   n.getPrev().setNext(newNode);
  else
   this.head = newNode;

  newNode.setNext(n);
  newNode.setPrev(n.getPrev());
  n.setPrev(newNode);
 }
}
```

III-A-3 - La méthode peek

```
public E peek()
{
 if (nbrObject > 0)
  return head.getElement();
 else
  return null;
}
```

III-A-4 - La méthode poll

```
public E poll()
{
 if (nbrObject == 0)
  return null;

 E tmp = this.peek();

 nbrObject--;

 ListNode<E> next = this.head.getNext();
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
    this.head = next;

    if (next != null)
     next.setPrev(null);

    return tmp;
  }
```

III-B - Implémentation non optimisée avec un tableau

III-B-1 - Les constructeurs et variables membres

```
public class PriorityQueueArray<E> implements PriorityQueue<E> {

    protected Bucket<E>[] array;
    protected Comparator<E> cmp;
    protected static final int DEFAULTSIZE = 16;

    protected int nbrObject = 0;

    public PriorityQueueArray(Comparator<E> cmp) {
        this(cmp, DEFAULTSIZE);
    }

    public PriorityQueueArray(Comparator<E> cmp, int size) {
        this.cmp = cmp;
        this.array = (Bucket<E>[]) new Bucket<?>[Math.max(1, size)];
    }
}
```

III-B-2 - La méthode add

```
public void add(E element) {
    // On vérifie s'il ne faut pas éventuellement redimensionner le tableau.
    resize();

    /*
     * Tant qu'on n'a pas trouvé un élément qui a une priorité inférieure à
     * celle de l'élément à ajouter, on descend dans le tableau.
     */
    int i = 0;
    while (i < nbrObject
            && cmp.compare(element, array[i].getElement()) <= 0)
        i++;

    array[nbrObject] = new Bucket<E>();
    E tmp = element;

    /*
     * Si on a trouvé un élément dont la priorité est inférieure à celle du
     * nouvel élément, alors on le place avant, et on décale tous les
     * éléments du tableau d'une position.
     */
    while (i != nbrObject) {
        tmp = array[i].swapElement(tmp);
        i++;
    }

    /*
     * Sinon on doit placer le nouvel élément à la fin, inutile de changer
     * quoi que ce soit.
     */
    array[nbrObject] = new Bucket<E>(tmp);

    nbrObject++;
}
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-B-3 - La méthode peek

```
public E peek() throws PriorityQueueException {
    if (nbrObject > 0)
        return array[0].getElement();
    else
        throw new PriorityQueueException("File vide.");
}
```

III-B-4 - La méthode poll

```
public E poll() throws PriorityQueueException {
    E tmp = this.peek();

    for (int i = 1; i < nbrObject; i++)
        array[i - 1].setElement(array[i].getElement());

    nbrObject--;

    return tmp;
}
```

III-C - Implémentation optimisée avec un tableau

III-C-1 - Les constructeurs et variables membres

Le code est le même que pour la version non optimisée, on peut alors en faire une extension. Il faut simplement ajouter un pointeur de tête, un entier *head*.

```
public class PriorityQueueArrayOptimized<E> extends PriorityQueueArray<E> {
    // Pointeur de tête du tableau.
    protected int head = 0;

    public PriorityQueueArrayOptimized(Comparator<E> cmp) {
        super(cmp);
    }

    public PriorityQueueArrayOptimized(Comparator<E> cmp, int size) {
       super(cmp, size);
    }
}
```

III-C-2 - La méthode add

```
public void add(E element)
{
 // On vérifie s'il ne faut pas éventuellement redimensionner le tableau.
 resize();

 /*
  * Tant qu'on n'a pas trouvé un élément qui a une priorité inférieure à
  * celle de l'élément à ajouter, on descend dans le tableau.
  */
 int i = 0;
 while (i < nbrObject && cmp.compare(element, array[(head + i) % array.length].getElement()) <= 0)
  i++;

 array[(head + nbrObject) % array.length] = new Bucket<E>();
 E tmp = element;

 /*
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  * Si on a trouvé un élément dont la priorité est inférieure à celle du
  * nouvel élément, alors on le place avant, et on décale tous les
  * éléments du tableau d'une position.
  */
 E tmp2;
 while (i != nbrObject)
 {
  tmp = array[(head + i) % array.length].swapElement(tmp);
        i++;
 }

 /*
  * Sinon on doit placer le nouvel élément à la fin, inutile de changer
  * quoi que ce soit.
  */
 array[(head + nbrObject) % array.length] = new Bucket<E>(tmp);

 nbrObject++;
}
```

III-C-3 - La méthode peek

```
public E peek() throws PriorityQueueException
{
 if (nbrObject > 0)
  return array[head].getElement();
 else
  throw new PriorityQueueException("File vide.");
}
```

III-C-4 - La méthode poll

```
public E poll() throws PriorityQueueException
{
 E tmp = this.peek();

 // Incrémentation du pointeur et diminution du nombre d'éléments.
 head = (head + 1) % array.length;
 nbrObject--;

 return tmp;
}
```


Les sources présentées sur cette pages sont libres de droits, et vous pouvez les utiliser à votre convenance. Par contre la page de présentation de ces sources constitue une oeuvre intellectuelle protégée par les droits d'auteurs. Copyright © 2011 - Jean-Philippe Collette. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.