INF2010
Structures de données et algorithmes
TP04 : Monceaux

Hiver 2020

Objectifs du TP:

- Implémenter les différentes techniques de construction d'un monceau
- Implémenter une méthode pour retirer la racine d'un monceau
- Implémenter une fonction pour afficher le monceau sous forme d'arbre.
- Tester votre implémentation

Indication : Un monceau est un arbre binaire complet ordonné en tas, c'est-à-dire un arbre binaire complet obéissant à la définition récursive voulant que tout nœud dans l'arbre possède une clé où :

- La valeur de la clé est $\leq$ à l'ensemble des clés de ses enfants dans le cas d'un *min-heap*.
- La valeur de la clé est $\geq$ à l'ensemble des clés de ses enfants dans le cas d'un *max-heap*.

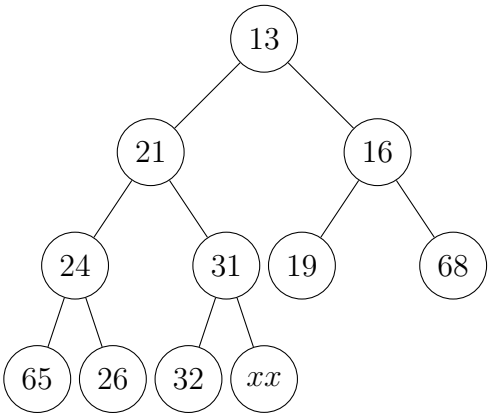
La construction d'un monceau peut s'effectuer d'au moins deux manières :

1. En insérant les éléments un à un dans le monceau.
2. En créant un arbre binaire complet contenant tous les éléments et en ordonnant progressivement les éléments de l'arbre jusqu'à l'obtention d'un monceau.

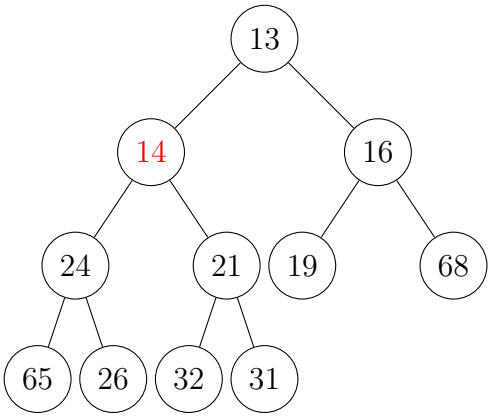
1 Première partie : Monceaux

1.1 Construction d'un monceau :

Complétez la méthode *offer*(*AnyType x*) qui permet d'insérer l'élément x dans le monceau. Tout le travail consiste à s'assurer que la propriété d'ordre d'un monceau est conservée.
Les figures (a) et (b) montre que pour insérer 14, nous créons un case candidate pour accueillir le nouvel elemente. Insérer 14 dans la case violerait l'order de monceau , donc 31 est glissé dans la case jusqu'à ce que l'emplacement correct pour 14 soit trouvé .
Remarque : notez que les éléments doivent être stockés à partir de la case 1 et non pas 0 dans le tableau interne .



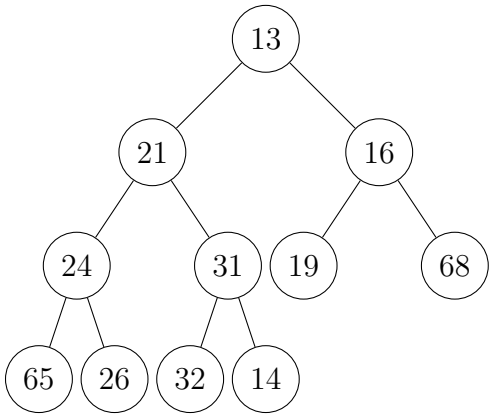
(a) Tentative d'insertion 14



(b) Insertion valeur 14 aprés deux étapes

1.2 Construction d'un monceau depuis un tableau :

Le constructeur *BinaryHeap*(*AnyType*[] *items, boolean min*) prend en entrée un tableau et un booléen indiquant s'il s'agit d'un ***min-heap*** ou d'un ***max-heap***. Le constructeur copie les données dans un tableau interne, puis manipule le tableau interne pour obtenir le type de monceau désiré en faisant appel à *buildMinHeap*() ou *buildMaxHeap*() selon le cas.



(a) monceau

	13	21	16	24	31	19	68	65	26	32	14		

(b) Tableau qui contient les éléments du monceau

1.2.1 Question :

Complétez *buildMinHeap*() et *buildMaxHeap*() au moyen des fonctions *percolateDownMinHeap*() ,*percolateDownMaxHeap*() et *leftChild*() .

1.3 Retrait de la racine d'un monceau :

Complétez la méthode *poll*() qui permet de retirer l'élément en tête du monceau.

2 Seconde partie : Manipulation des monceaux

2.1 Implantation d'un itérateur *fail-fast*

Les monceaux sont souvent utilisés pour implémenter des files de priorités. D'ailleurs, le monceau binaire de Java se nomme *PriorityQueue* et hérite de la classe abstraite *AbstractQueue*. Afin de fournir une implantation compatible avec les standards de Java, la classe *BinaryHeap* hérite aussi d'AbstractQueue. En plus des méthodes *peek*(), *poll*() et *offer*() , AbstractQueue impose l'implantation de la méthode *iterator*() .
Pour ce TP, on vous impose une contrainte supplémentaire : votre itérateur doit être fail-fast. C'est-à-dire qu'il doit soulever une exception de type *ConcurrentModificationException* s'il détecte une modification au monceau (insertion ou délétion) en cours d'itération. La vérification doit se faire au moment de l'appel à la méthode *next*().

2.1.1 Questions :

Compléter le *HeapIterator*. Utilisez la variable de classe *modifications* afin de détecter s'il y a des des modifications en cours d'itération. L'itération peut se faire dans un ordre arbitraire.

2.2 Tri par monceau

La class BinaryHeap offre une fonction statique de tri *heapSort*() qui reçoit un tableau et le trie suivant la technique du tri par monceau. On vous demande de la compléter en faisant appel à *percolateDownMaxHeap*() . Inversement, complétez *heapSortReverse*() en faisant appel à *percolateDownMinHeap*().

2.3 Affichage du monceau en arbre

Complétez le code de *nonRecursivePrintFancyTree*() permettant d'afficher le monceau sous la forme d'un arbre en n'effectuant aucun appel récursif. Référez-vous au document Affichage.txt pour voir le résultat attendu.

3 Troisième partie : Test de votre implantation

Complétez le fichier *Main.java* afin de vérifier le bon fonctionnement de votre structure de données. Par exemple, assurez-vous que *poll*() retourne toujours l'élément minimal ou maximal, vérifiez que votre itérateur détecte bien tous les types de modifications concurrentes, etc. Notez que puisque votre structure implémente les mêmes méthodes que la *PriorityQueue* de Java, il est aisé d'effectuer les mêmes suites d'opérations sur votre Binary-Heap et sur une *PriorityQueue* et de vérifier la concordance entre les deux structures.

4 Instructions pour la remise

Le travail doit être remis via Moodle :

- 24 mars avant 23h59 pour le groupe 01
- 17 mars avant 23h59 pour le groupe 02

Veuillez envoyer dans une archive de type $*$.zip qui portera le nom inf2010_lab4_MatriculeX_ MatriculeY (MatriculeX $<$ MatriculeY) :

- Vos fichiers .java

Les travaux en retard seront pénalisés de 20 % par jour de retard. Aucun travail ne sera accepté après 4 jours de retard.

4.0.1 Avertissement :

Pour ceux qui voudraient déposer leur laboratoire sur GitHub, assurez-vous que vos répertoires soient en mode **privée** afin d'éviter la copie et l'utilisation non autorisée de vos travaux.

4.1 Barème de correction

9 pts: Première partie

- **3 pts:** 1.1
- **4 pts:** 1.2
- **2 pts:** 1.3

7 pts: Seconde partie

- **3 pts:** 2.1
- **2 pts:** 2.2
- **2 pts:** 2.3

4 pts: Troisième partie

- **4 pts:** 3.1