Les files de priorité

Exercices obligatoires

A Implémentation de l'interface *FileDePriorite*

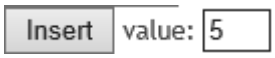
Vous allez implémenter l'interface *FileDePrioriteDEntiers* en utilisant la technique du tas. Dans cette file, les objets sont des entiers. Plus un entier est grand, plus sa priorité est grande.

A1 Commencez par dessiner *sur papier* un tas en partant de l'arbre vide puis en réalisant les opérations suivantes :
Nb : ici, on ne s'intéresse pas encore à l'implémentation d'un tas via une table. Dessinez l'arbre comme on a l'habitude de le dessiner.

Vérifiez, après chaque opération, l'arbre obtenu en utilisant l'animation :

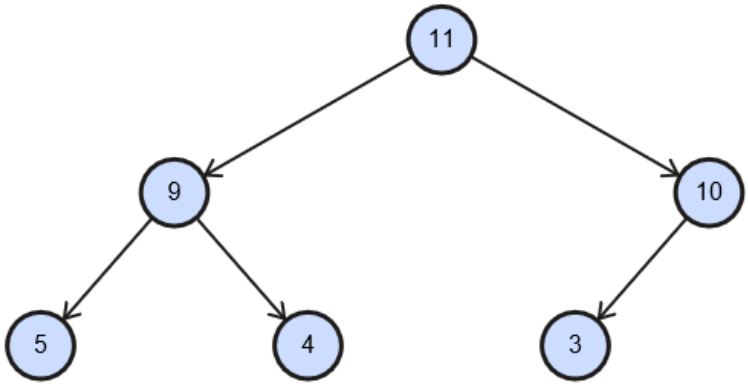
http://btv.melezinek.cz/binary-heap.html

a)
- insere 5
- insere 9
- insere 3
- insere 12
- insere 11
- insere 15
- insere 10
- insere 4



b)
- supprime max
- supprime max

Voici l'arbre à obtenir :



A2 Complétez la classe *FileDePrioriteDEntiersImpl*.

Il vous reste à compléter les méthodes `pushUp()` et `supprimeMax()`.
Si vous avez du mal à démarrer, nous vous conseillons d'essayer de comprendre les méthodes `pushDown()` et `insere()` qui sont données. La méthode `pushUp()` est plus simple ☺.
Testez cette classe via la classe *TestFileDePrioriteDEntiersImpl*.
Cette classe de tests suit le scénario de l'ex A1.

B Application Brasserie

B1 Application Brasserie avec file d'attente

Dans un premier temps, vous implémenterez une version avec file d'attente.
Il s'agit d'une ancienne question d'examen (juin 2021Bleu).

Vous passerez ensuite à la version avec file de priorité.

A l'examen, pour une des questions, vous ne recevrez pas de classe de tests.
Il vous faudra tester vos méthodes via une classe de gestion.
Ce n'est pas évident de penser à tout.
Mettez-vous dans cette situation.
N'utilisez la classe de tests *TestSessionDeVente* que plus tard, lorsque vous pensez que l'application est au point.

Vous allez implémenter une application de vente de casiers de bière pour une brasserie.
Celle-ci s'inspire du processus de vente d'une brasserie belge mondialement connue.
L'abbaye de Westvleteren n'a aucune vue commerciale. Sa production de bière est très limitée.
Elle veut que ses clients soient des consommateurs directs (pas de revente en magasin ou autre).
Plusieurs sessions de vente sont organisées par an.
Le nombre de casiers de bière à vendre lors d'une session dépendra de la production de la brasserie.
Pour satisfaire le plus de monde possible, le nombre de casiers que peut acheter un client au cours d'une session de vente est limité à 3.
Les commandes d'une session de vente se font via un magasin en ligne qui ne sera ouvert qu'à un moment bien précis.
Au moment de la vente, le client est placé dans une file d'attente.
Lorsque son tour arrive, il voit le nombre de casiers restants. Il est invité à entrer un nombre de casiers. (Si le client avait déjà fait une commande, ce nombre sera ajouté au nombre de casiers déjà commandés.)
Le magasin en ligne ferme lorsque tous les casiers sont vendus.
Les clients viendront chercher leurs casiers sur place après fermeture du magasin en ligne.

Implémentation choisie :

Vous n'allez pas introduire de classe *Client.*
Le client sera représenté par son nom (*String*).

La classe *Commande* vous est donnée. Une commande retient le client qui a passé cette commande et le nombre de casiers qu'il a demandé. Il est possible de modifier le nombre de casiers.

Vous allez complétez la classe *SessionDeVente*.

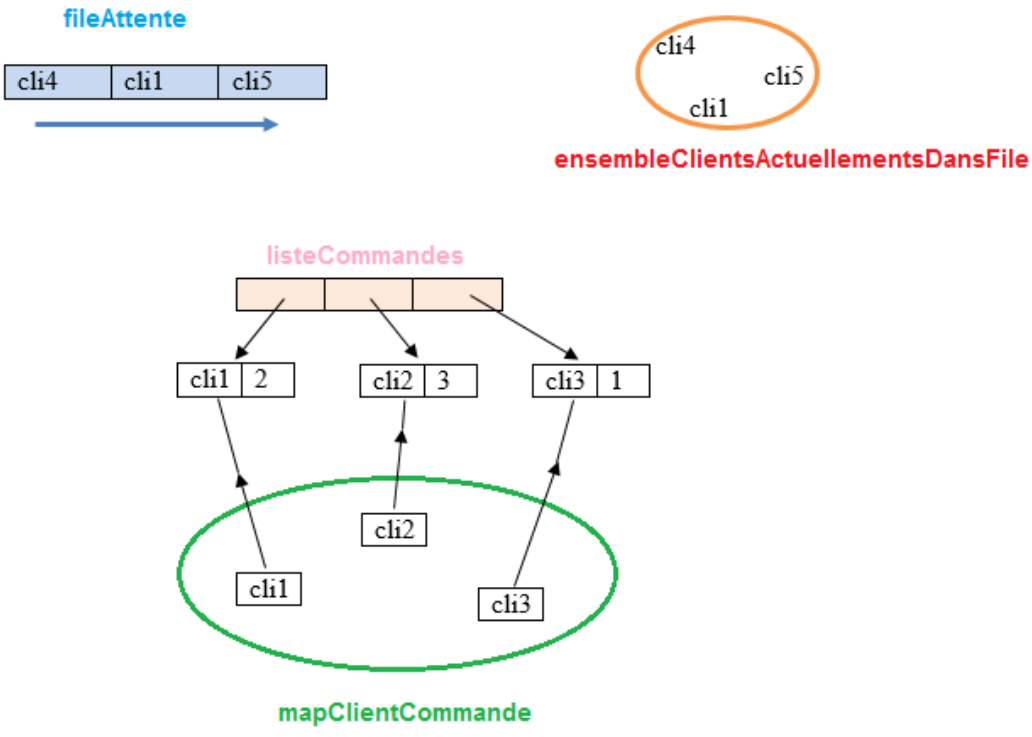
Pour gérer la file d'attente, on a fait le choix d'une file (*ArrayDeque<String>*).

Un client peut prendre place dans cette file à condition qu'il ne s'y trouve pas déjà.
On décide d'ajouter l'ensemble (*HashSet<String>*) des clients présents dans la file d'attente.
La vérification d'existence est moins coûteuse dans un ensemble que dans une file !

On va retenir toutes les commandes dans une liste (*ArrayList<Commande>*).
Un *map* (*HashMap<String,Commande>*) permet de retrouver facilement la commande d'un client.

Ne perdez pas de vue que la liste des commandes et le map référencient des mêmes commandes.

Exemple :
Dans cet exemple, on remarque que cli1, cli2 et cli3 ont réservés respectivement 2, 3 et 1 casier.
Actuellement dans la file d'attente on retrouve cli5, cli1 et cli4. Le prochain client qui sera traité sera cli4. Cli1 a pu (re)prendre place dans la file d'attente car il n'a pas encore commandé les 3 casiers maximum auquel il a droit.



Le constructeur de cette classe et les attributs sont donnés.

En plus de ces structures de données, des variables et constantes permettant une implémentation efficace figurent parmi les attributs proposés.
Prenez bien connaissance de tous les attributs de la classe avant de vous lancer dans la programmation !

Complétez la classe *SessionDeVente* en respectant bien la *JavaDoc* et les choix d'implémentation imposés ci-dessus.
Si c'est plus facile pour vous, vous pouvez introduire d'autres attributs et des méthodes (*private*)

La classe *GestionSessionDeVente* va vous servir pour tester la classe *SessionDeVente*.
Vous pouvez la modifier.

B2 Application Brasserie avec file de priorité

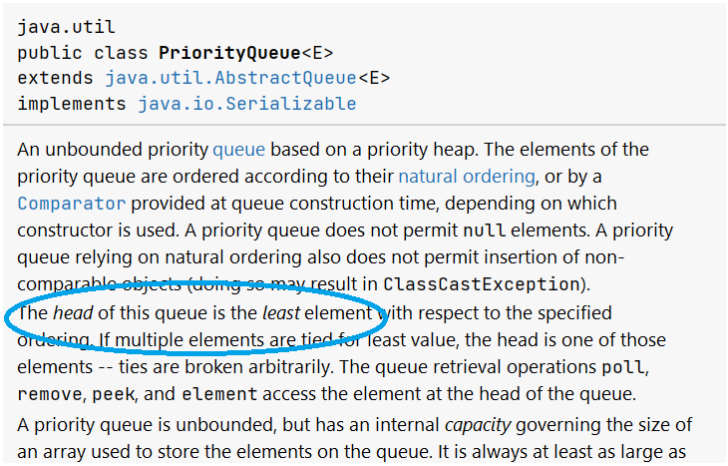
Lors de chaque session de ventes, il y a beaucoup de déçus. Seuls les plus rapides réussissent à commander.
Pour éviter que ce soit toujours les mêmes qui réussissent à acheter des bières, un système de priorité a été mis en place.
Un client qui n'a plus réussi à commander depuis longtemps devient prioritaire.
Toute personne qui désire commander devra s'être enregistrée préalablement.
Lors de cette inscription, le client reçoit la priorité 3.
Lors d'une session de vente, à chaque commande aboutie, il perd 1 point de priorité.
En revanche le client qui s'est trouvé dans la file d'attente, mais qui n'a pu commander augmente sa priorité de 1.
Donc, un client, lorsqu'il accède à la file d'attente, n'y sera plus placé en fin de file, mais en fonction de sa priorité.

Commencez par écrire une classe *Client*.
Pour cet exercice, cette classe sera très simple : un client possède un nom (String) et une priorité.

Ecrivez une classe *ClientEnAttente* qui hérite de la classe *Client*.
Dans la file de priorité, si deux clients ont même priorité, rien ne garantit que ce soit le premier arrivé qui sera le premier servi. Un client en attente est un client qui a reçu un numéro d'ordre d'arrivée. Le premier client reçoit le numéro 1, le deuxième, le numéro 2 et ainsi de suite. Pour gérer ce numéro, pensez à introduire une **variable de classe** (attribut `static`). Appelez cette variable de classe `numeroSuivant`. Cette variable est initialisée à 1 et est incrémentée à chaque création d'un client en attente.

Ecrivez une classe *ComparateurClientEnAttente* qui implémente l'interface *Comparator<ClientEnAttente>*.
2 clients doivent être comparés selon leur priorité. L'ordre d'arrivée dans la file doit départager 2 clients de même priorité.

Il faut penser « bizarrement » lorsqu'on écrit la méthode `compare()`. On ne veut pas que ce soit le client le moins prioritaire qui soit servi prioritairement.



```
java.util
public class PriorityQueue<E>
extends java.util.AbstractQueue<E>
implements java.io.Serializable
```

An unbounded priority queue based on a priority heap. The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used. A priority queue does not permit null elements. A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so may result in ClassCastException).
The *head* of this queue is the *least* element with respect to the specified ordering. If multiple elements are tied for least value, the head is one of those elements -- ties are broken arbitrarily. The queue retrieval operations poll, remove, peek, and element access the element at the head of the queue.
A priority queue is unbounded, but has an internal *capacity* governing the size of an array used to store the elements on the queue. It is always at least as large as

Ecrivez une classe *CommandeAvePriorite*.
L'attribut `client` est un objet de la classe *Client*.

Ecrivez une classe *SessionDeVenteAvecPriorite*.

`ensembleClientsActuellementDansFile`, `mapClientCommande` contiennent des clients. Ce sont des objets de la classe *Client*.

`fileAttente` est une file de priorité. Elle contient des objets de la classe *ClientEnAttente*.
Utilisez un objet de la classe *PriorityQueue<ClientEnAttente>*.

Les méthodes de cette classe sont les mêmes que la classe *SessionDeVente.*

MAIS :

Il faut penser à gérer la priorité :
La méthode `passerNouvelleCommande()` doit diminuer de 1 la priorité du client à chaque commande aboutie.

Il faut ajouter la méthode `cloturerSession()`. Tous les clients qui sont encore dans la file d'attente et qui n'ont rien commandé verront leur priorité augmenter de 1.

La classe *GestionSessionDeVenteAvecPriorite* est une classe adaptée pour cette version.
Dans un premier temps, elle demande d'encoder les clients préalablement enregistrés.
Lors de cet encodage, on demande pour chaque client, son nom et sa priorité. Mais on demande également un login.
Un client non enregistré ne peut accéder à la file d'attente ! L'option 2 (`mettre un client dans la file d'attente`) s'occupe de vérifier si le login existe.
La classe donnée fait tout cela ! Vous ne devez pas y apporter de modifications.

Suggestion : utilisez la classe *MonScanner* !

Attention :
Ni l'itérateur, ni le `toString()` de la classe *PriorityQueue* donne les éléments selon l'ordre

de priorité.
L'affichage du contenu de cette file ne permet pas de bien tester votre application.

Exercice défi

❤️
B2
Lorsqu'on demande d'afficher quelques informations sur l'état des ventes, on voudrait que les clients en attente apparaissent dans l'ordre où ils seront servis.

Réfléchissez ! Parlez-en avec un professeur.
Plusieurs solutions sont envisageables. Certaines plus coûteuses que d'autres.

Exercice supplémentaire

C *Heap Sort*

Le *Heap Sort* est un algorithme de tri qui est basé sur le principe du tas.
Cet algorithme est de complexité asymptotiquement optimale, c'est-à-dire que l'on démontre qu'aucun algorithme de tri par comparaison ne peut avoir de complexité asymptotiquement meilleure. Il est en O(n*log(n)) même dans le pire des cas.
Un autre avantage de ce tri est qu'il ne demande pas de mémoire annexe.
(Il peut être cependant plus lent que le *Quick Sort*.)

Le *Heap Sort* se déroule en 2 étapes :

Lors de la première étape, la table à trier va être construite en tas (`buildHeap`()).

Lors de la deuxième étape, la table va être triée en commençant par la fin.
La méthode `supprimeMax()` va être appelée n-1 fois.
Ceci vient de la constatation que si on supprime le maximum, la dernière case du tableau devient libre. On peut donc y stocker le maximum.
En continuant de la sorte, le tableau sera trié.

C'est toujours la méthode `pushDown()` qui est appelée aussi bien pour le `buildHeap()` que pour la méthode `supprimeMax()`.

Le `buildHeap()` commence au « dernier » nœud interne et appelle la méthode `pushDown()`. Il remonte ainsi jusqu'à la racine. A la fin du `buildHeap()` la table de départ est devenue un tas !

Découvrez cet algorithme de tri via l'animation :

`https://www.cs.usfca.edu/~galles/visualization/HeapSort.html`

Vous trouverez également sur moodle une proposition de code de cet algorithme en consultant la classe *HeapSort*.
Considérons le tableau formé des éléments suivants dans cet ordre :

57 85 44 21 23 52 17 7 95 64 87
On vous demande de trier « sur papier » ce tableau en utilisant le *Heap Sort* en indiquant l'état du tableau après chaque `pushDown()`.
Pour vous aider, veuillez dessiner l'arbre de départ, puis, pour chaque `pushDown()` l'évolution de cet arbre (ce qui nécessitera parfois plusieurs dessins d'arbre) au cours du `pushDown()` suivi de l'état du tableau après le `pushDown()` que vous reporterez dans le tableau suivant :
Vérifiez de temps à autre votre tableau avec la solution qui se trouve dans le document CSol sur moodle !
Le `main()` de la classe *HeapSort* trie cette table et un affichage de celle-ci est prévu après chaque `pushDown()`.

57 85 44 21 23 52 17 7 95 64 87	tableau initial
.. ..	construction du tas
.. ..	
.. ..	
.. ..	
.. ..	
.. .. 95	tri
.. .. 87 95	
.. .. 85 87 95	
.. .. 64 85 87 95	
.. .. 57 64 85 87 95	
.. .. 52 57 64 85 87 95	
.. .. 44 52 57 64 85 87 95	
.. .. .. 23 44 52 57 64 85 87 95	
.. .. 21 23 44 52 57 64 85 87 95	
7 17 21 23 44 52 57 64 85 87 95	