SD1 : PARTIE « PC » (18 points)

A Liste chaînée (6 points)

Chaque série d’étudiants possède un délégué. Le délégué est l’étudiant qui représente sa série lors des conseils de département.

Les étudiants sont placés dans une liste.

Cette liste ne sera jamais vide, car elle contient toujours au minimum le délégué.

Dans la liste, on trouve **en tête le délégué** et ensuite les autres étudiants.

Ces autres étudiants apparaitront du **plus récent (dernier ajouté) au plus ancien**.

On ne peut pas retrouver plusieurs fois un même étudiant dans cette liste.

Choix d’implémentation :

Dans le cadre de l’examen, nous n’allons pas introduire de classe *Etudiant*.

Un étudiant sera identifié par son prénom (String).

Vous allez compléter la classe *SerieEtudiants*.

Une série d’étudiants possède comme attributs un nœud de tête, un nœud de queue ainsi qu'un *map*.

La liste est implémentée via pointeurs. Elle est doublement chaînée.

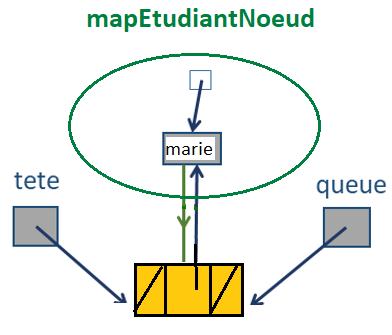
La classe *SerieEtudiants* contient une classe interne *Nœud*.

Chaque nœud possède un « étudiant » et référencie le nœud précédent et le nœud suivant.

Pour éviter des nombreux parcours de la liste à la recherche d’un étudiant, on utilise un *map*. La clé est l’étudiant recherché, la valeur associée est le nœud contenant cet étudiant. L’utilisation d’un *map* permet la recherche d’un nœud en O(1).

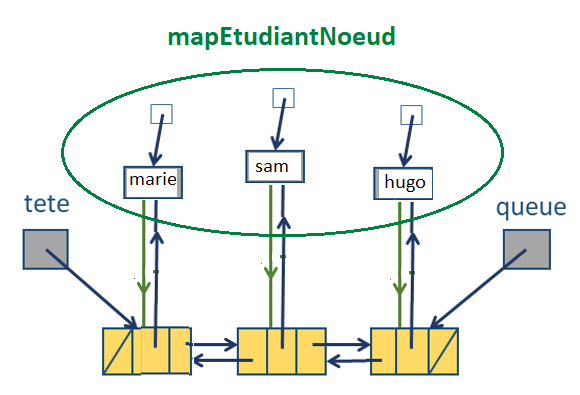
Le constructeur reçoit en paramètre le délégué.

Voici le schéma d’une série d’étudiants après appel du constructeur avec comme paramètre « Marie » (qui est la déléguée) :



La méthode insererApresDelegue(String etudiant) ajoute l’étudiant directement après son délégué.

Voici le schéma de la série d’étudiants après insertion de d’abord Hugo et ensuite de Sam :



La méthode tronquerAPartir(String etudiant) supprime tous les étudiants de la liste à partir de l’étudiant passé en paramètre.

Exemple :

Si on demande de tronquer la liste  *marie ⮀ sam ⮀ hugo ⮀ pierre ⮀ lea* à partir de *pierre*, la liste devient *marie ⮀ sam ⮀ hugo*.

Vous allez complétez la classe *SerieEtudiants.*

Respectez la *JavaDoc* et l’implémentation choisie.

Vous ne pouvez pas introduire d’autres méthodes que celles présentes.

Pour vos tests, utilisez la classe *TestSerieEtudiants*.

B File de priorité via ABR (5 points)

Pour l’examen, nous vous avons fourni une interface *FileDePriorite* réduite.

Cette interface étend l’interface *Iterable*.

Vous allez compléter la classe *FileDePrioriteImpl* qui implémente l’interface *FileDePriorite* en utilisant un arbre binaire de recherche (ABR).

On fait le choix que la descendance **gauche** d’un nœud ne contiendra que des éléments **de priorité supérieure** à l’élément de ce nœud et la descendance **droite** d’un nœud ne contiendra que des éléments **de priorité inférieure ou égale**.

Voici un exemple d’une file de priorité qui contient des personnes qui ont reçu chacune une priorité :

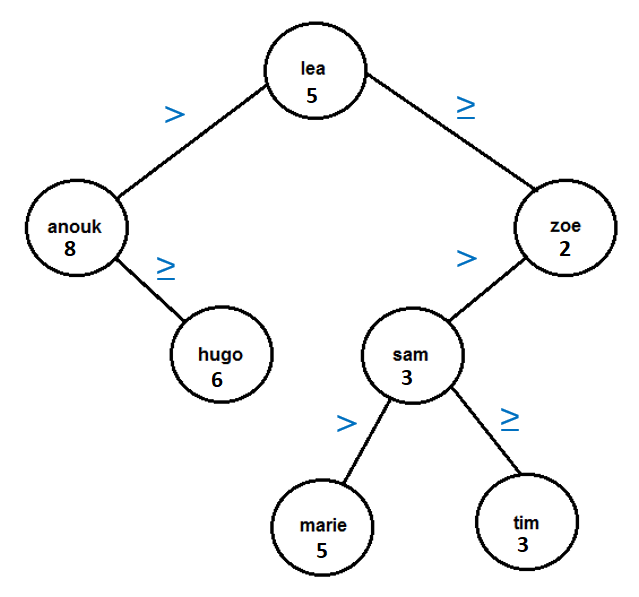
Plus la priorité est grande, plus elle est importante.

C’est Anouk qui a reçu la plus grande priorité. Ce sera la première servie.

Marie et Léa ont la même priorité. Marie a été insérée après Léa.

Sam et Tim ont la même priorité. Tim a été inséré après Sam.

Les personnes seront servies dans cet ordre : Anouk Hugo Léa Marie Sam Tim Zoé.

****

La classe *FileDePrioriteImpl* contient 2 classes internes : *Nœud* et *Iterateur*

Elle possède comme attribut le nœud racine et un « comparateur ».

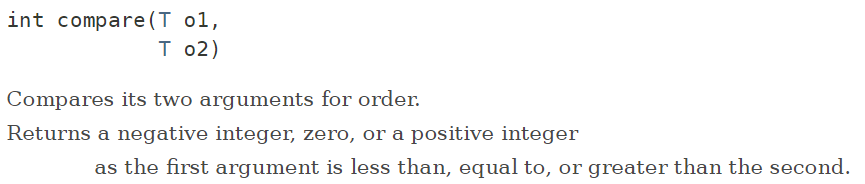
Elle propose 2 constructeurs qui reçoivent un « comparateur » en paramètre.

Un des constructeurs sert pour les tests. (Il est possible de tester chaque méthode séparément grâce à la classe *TestFileDePrioriteImpl*.)

Le comparateur est un objet d’une classe qui implémente l’interface *Comparator*.

Pour pouvoir comparer les priorités de 2 éléments, vous allez utiliser la méthode compare()de cette interface.

Voici un extrait de la *JavaDoc* de cette méthode :



Voici un extrait de code possible :

if (comparateur.compare(noeud.element,e)<0) …

La méthode toString()vous est donnée.

Pour l’arbre ci-dessus cette méthode renvoie :

[ [ [ ] anouk (8) [ hugo (6) ] ] **lea (5**) [ [ [ marie (5) ] sam (3) [ tim (3) ] ] zoe (2) [ ] ] ]

La classe interne *Iterateur* possède comme attribut une file.

Le constructeur commence par remplir la file en y plaçant tous les éléments de l’arbre.

Les éléments doivent être placés dans la file de sorte que le parcours de l’arbre respecte l’ordre des priorités.

**Dans la classe *FileDePrioriteImpl*, vous allez compléter les méthodes insere()et max()ainsi que la méthode remplirFile() de la classe interne *Iterateur*.**

**Vous écrirez ces 3 méthodes de façon récursive.**

Respectez bien la *JavaDoc* (reprise dans l’interface) et les choix d’implémentation imposés.

Pour programmer des méthodes de façon récursive, il vous est permis, bien sûr, d’introduire d’autres méthodes (*private*).

La classe *TestFileDePrioriteImpl* permet de tester vos méthodes avec l’arbre donné en exemple ainsi que l’arbre vide.

C API JAVA : Application Cirque (7 points)

Le cirque est de retour au village pour un spectacle unique.

Le responsable des réservations désire informatiser l’attribution des différentes places pour ce spectacle.

Seuls les enfants du village peuvent réserver des places.

Chaque enfant aura la possibilité de choisir ses places.

Il peut les réserver en une ou plusieurs fois.

Voici l’implémentation qui a été choisie :

Dans le cadre de l’examen, nous n’allons pas introduire d’autres classes que les classes *Cirque* et *GestionCirque*.

L’enfant qui demande des places sera reconnu par son prénom (*String*).

La place sera représentée par son numéro (*int*). La numérotation commence à 0.

La classe *Cirque* possède 2 attributs :

String[] tableReservations : une table pour retenir les réservations. Chaque case correspond à une place. Au départ cette table ne contiendra que des null. On y placera, au fur et à mesure, les prénoms des enfants qui réservent.

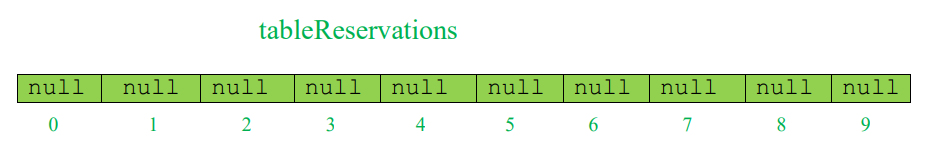
HashMap<String, HashSet<Integer>> mapEnfants : ce *map* associe à chaque enfant l’ensemble des places qu’il a réservées.

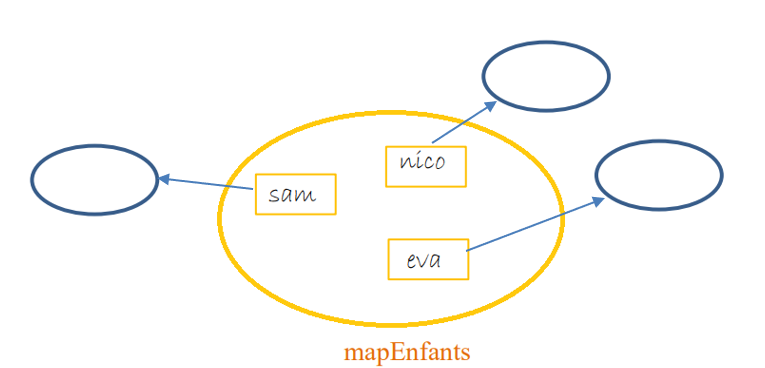
Tous les enfants du village sont placés dans ce *map*.

Le constructeur reçoit en paramètre le nombre total de places et une table avec les prénoms des enfants du village.

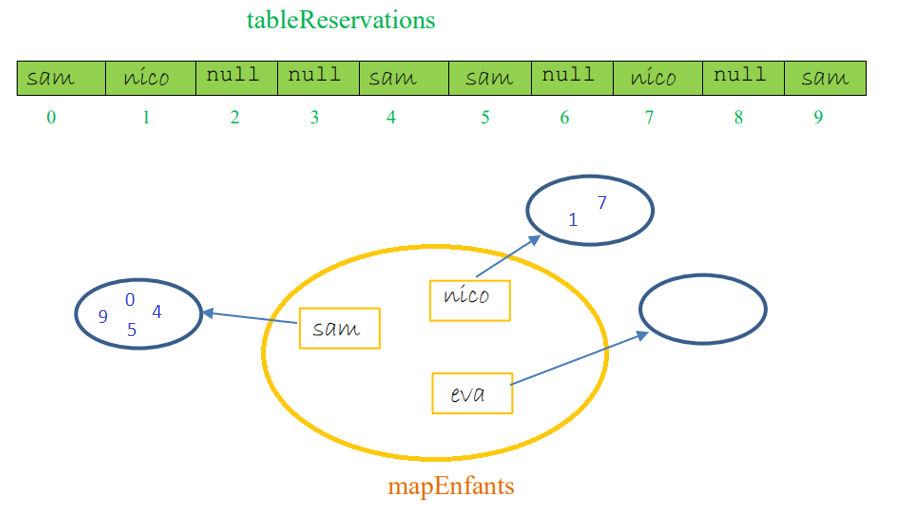
Voici un exemple : il y a 10 places et 3 enfants : Eva, Sam et Nico

Après appel du constructeur :





Après quelques réservations :



Eva n’a réservé aucune place.

Sam a réservé 4 places : les places 0, 4, 5 et 9.

Nico en a réservé 2 : les places 1 et 7.

Complétez la classe *Cirque* en respectant bien la *JavaDoc* et les choix d’implémentation imposés.

Si c’est plus facile pour vous, vous pouvez introduire d’autres attributs et des méthodes (*private*).

La classe *GestionCirque* va vous servir pour tester la classe *Cirque*.

Le point 2 du menu permet d’afficher la table des réservations et le *map*.

Vous pouvez modifier cette classe.

Mais ne perdez pas de temps à l’améliorer. Cette classe ne sera pas évaluée.

Lors de vos tests, pour éviter de devoir chaque fois encoder toute une séquence de commandes identiques, on vous conseille de les encoder dans un fichier. La classe *MonScanner* permet de passer de l’encodage via fichier à l’encodage manuel. On vous a fourni le fichier *InputCirque.txt* avec déjà quelques commandes. En utilisant ce fichier, le « cirque » correspondra à l’exemple ci-dessus.