SD1 : PARTIE « PC » (18 points)

A Liste chaînée (6 points)

Une équipe est constituée de plusieurs membres. On parlera d’équipiers.

Les équipiers sont placés dans une liste. Ils y apparaitront du **plus ancien au plus récent**.

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

Choix d’implémentation :

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

Un équipier sera identifié par son prénom (String).

La liste est implémentée via pointeurs. Elle est **doublement chaînée avec sentinelles**.

Afin d’éviter de devoir tester de nombreux cas particuliers, 2 nœuds appelés « sentinelle » sont placés en tête et en fin de liste. Les éléments contenus dans ces nœuds n’appartiennent pas à la liste.

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

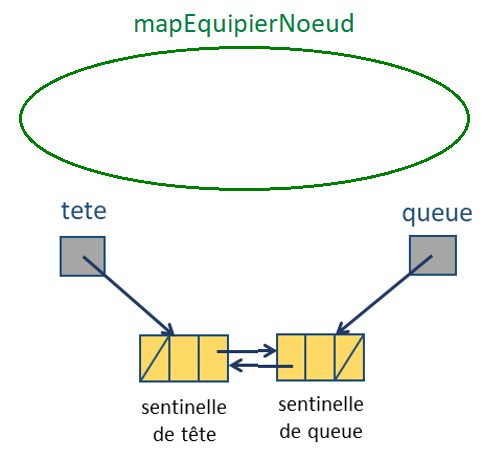
Vous allez compléter la classe *Equipe*.

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

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

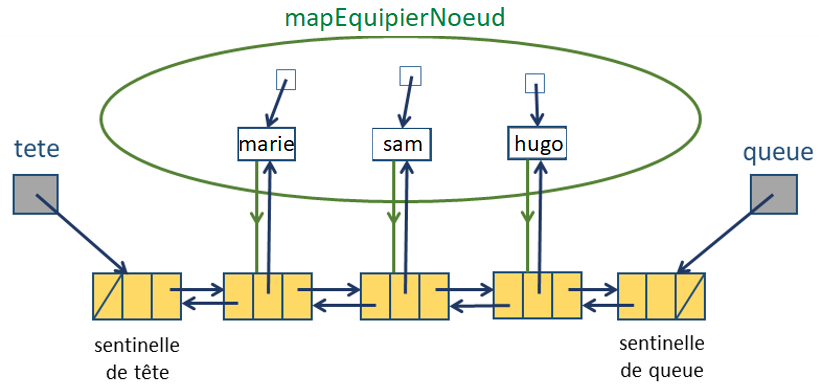
Une équipe possède comme attributs une sentinelle de tête, une sentinelle de queue ainsi qu'un *map*.

Voici une équipe après appel du constructeur :



La méthode ajouterEnFin(String equipier) ajoute l’équipier, s’il n’est pas encore présent, en fin de liste.

Voici le schéma de l’équipe après ajout de d’abord Marie, ensuite de Sam et ensuite de Hugo :



La méthode tronquerApres(String equipier) supprime tous les équipiers de la liste qui se trouve après l’équipier passé en paramètre.

Exemple :

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

Vous allez complétez la classe *Equipe.*

Respectez la *JavaDoc* et l’implémentation choisie.

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

Pour vos tests, utilisez la classe *TestEquipe*.

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

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

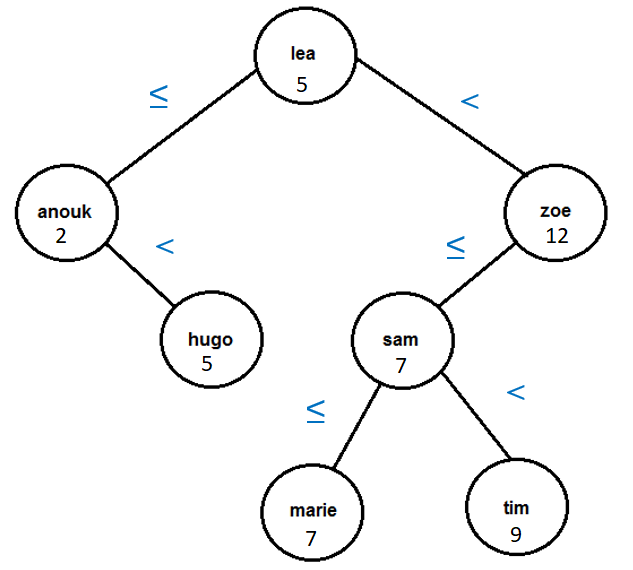
Plus la priorité est grande, plus elle est importante.

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

Hugo et Léa ont la même priorité. Hugo a été inséré après Léa. Léa sera servie avant Hugo

Sam et Marie ont la même priorité. Marie a été insérée après Sam. Sam sera servi avant Marie.

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

****

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

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

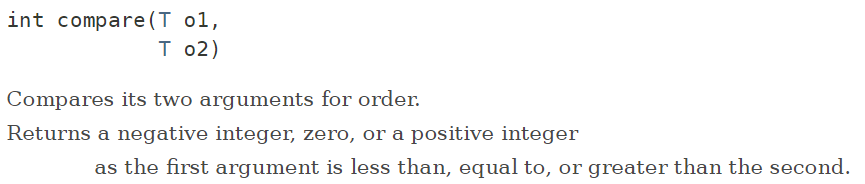
Elle propose 2 constructeurs qui reçoivent un « comparateur » en paramètre.

Un des constructeurs sert pour les tests.

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 (2) [ hugo (5) ] ] **lea (5**) [ [ [ marie (7) ] sam (7) [ tim (9) ] ] zoe (12) [ ] ] ]

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. Les éléments seront parcourus du plus prioritaire au moins prioritaire.

**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 Guignol (7 points)

C’est bientôt le retour de guignol et son traditionnel spectacle des enfants !

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

Chaque enfant aura la possibilité de choisir ses places.

Il peut en réserver 4 maximum.

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 *Guignol* et *GestionGuignol*.

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 *Guignol* possède 3 attributs :

static int *MAX* = 4 : le nombre maximum de places que peut réserver un enfant.

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.

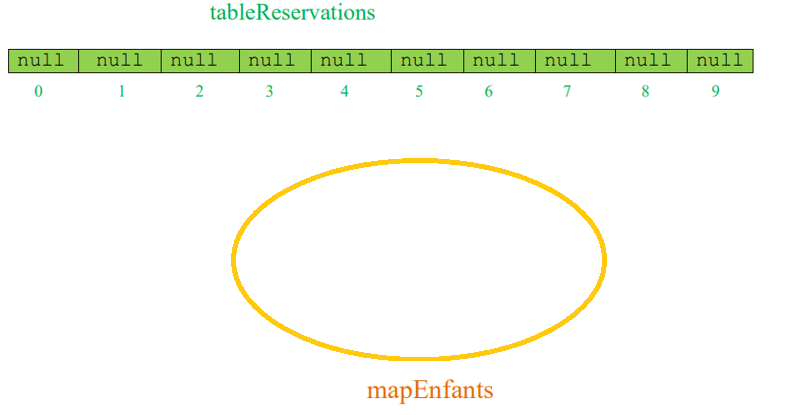
HashMap<String, ArrayList<Integer>> mapEnfants : ce *map* associe à chaque enfant la liste des places qu’il a réservées.

Tous les enfants qui réservent sont placés dans ce *map*.

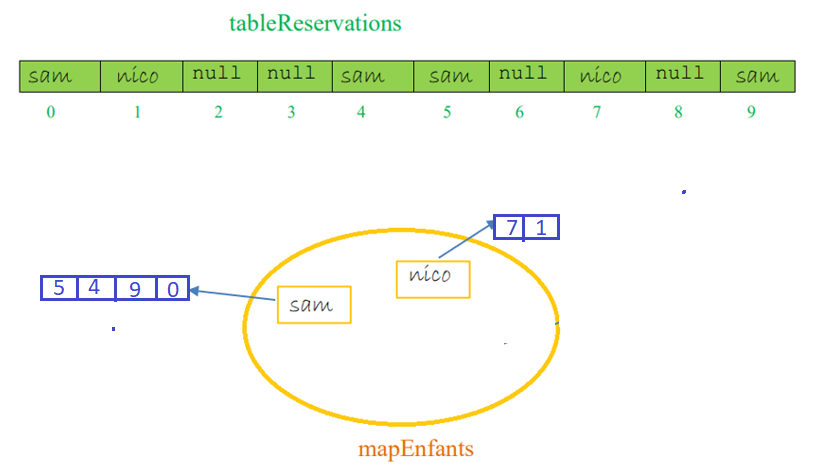
Le constructeur reçoit en paramètre le nombre total de places.

Voici un exemple : il y a 10 places.

Après appel du constructeur :



Après quelques réservations :



Sam a réservé 4 places : les places 0, 4, 5 et 9. Il ne peut plus réserver de places.

Nico en a réservé 2 : les places 1 et 7. Il peut encore en réserver 2.

Complétez la classe *Guignol* 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 *GestionGuignol* va vous servir pour tester la classe *Guignol*.

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 *InputGuignol.txt* avec déjà quelques réservations. En utilisant ce fichier, le « guignol » correspondra à l’exemple ci-dessus.