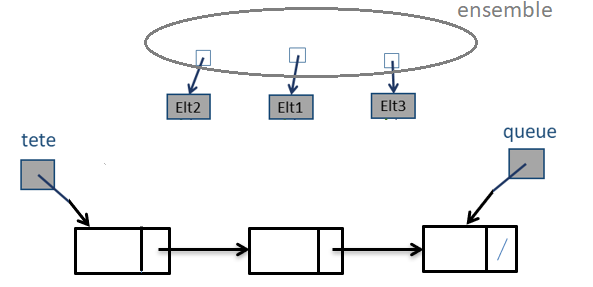
A *FileSD* (7 points)

Dans le cadre de l’examen, nous allons nous intéresser à l’implémentation d’une file dans laquelle on refuse des doublons (SD 🡪 sans doublon).

Nous avons rassemblé dans l’interface *FileSD*, toutes les méthodes intéressantes pour une telle file.

A1 Implémentation de l’interface *FileSD*

Voici l’implémentation choisie.



Cette implémentation utilise une liste simplement chaînée avec pointeur de tête et ponteur de queue ainsi qu’un ensemble.

L’utilisation d’un ensemblepermet d’**optimiser** certaines méthodes.

Complétez la classe *FileSDImpl.*

Testez-la avec la classe *TestFileSDImpl.*

A2 Coûts

Complétez « à la main » le document qui vous a été remis.

B API JAVA : Application Parc à conteneurs (7 points)

Il y a souvent de longues files pour accéder au parc à conteneurs.

Un système de réservation va être mis en place. Quelques périodes de la semaine ne seront accessibles qu’aux voitures qui se sont enregistrées préalablement. Le nombre de voitures pour chaque période est limité, ce qui devrait éviter des engorgements.

Une voiture ne peut réserver qu’un nombre limité de périodes par semaine. Ceci permet d’éviter que juste quelques voitures mobilisent toutes ces périodes.

L’application demandée dans le cadre de l’examen se concentre uniquement sur les réservations d’**une** semaine.



Voici l’implémentation qui a été choisie :

On n’introduit pas de classe *Voiture*. Seule sa plaque (*String*) est enregistrée.

Une période est reconnue via un numéro (*int*). La numérotation commence à 1.

La classe *ParcAConteneurs* est à compléter.

On n’introduit pas de classe *Periode*.

Les voitures autorisées pour une période seront placées dans un ensemble (*HashSet<String>*).

L’attribut **maxVoitures** retient le nombre maximum de voitures autorisées par période.

Tous ces ensembles sont placés dans la table **tablePeriodes**.

L’ensemble qui se trouve à l’indice 0 contient les voitures autorisées pour la période 1, l’ensemble qui se trouve à l’indice 1 contient les voitures autorisées pour la période 2 et ainsi de suite …

Le *map* (*HashMap<String,Integer>*) **mapVoitures** permet de connaître le nombre de périodes que chaque voiture a réservé.

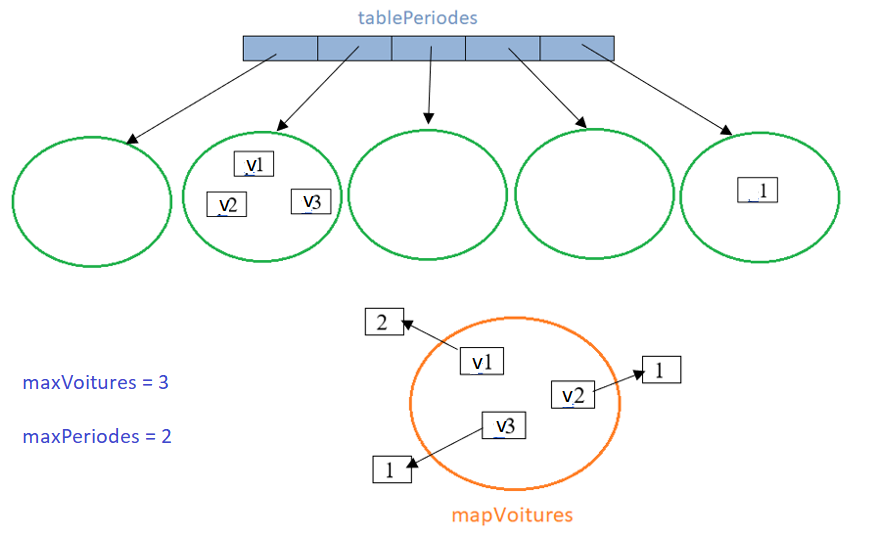
Seules les voitures qui ont au moins réservé une période y sont retenues.

(Attention : la méthode **get()** renvoie un *Integer* et non un *int.* Un *cast* sera peut-être nécessaire dans votre code)

L’attribut **maxPeriodes** retient le nombre maximum de périodes qu’une voiture peut réserver sur la semaine.

Le constructeur de la classe *ParcAConteneurs* reçoit comme paramètres le nombre de périodes, le nombre maximum de voitures autorisées sur une période et le nombre maximum de périodes qu’une voiture peut réserver.

Exemple :



Dans cet exemple, on remarque qu’il y a 5 périodes.

La voiture v1 a réservé la période 2 et la période 5.

Les voitures v2 et v3 ont réservés la période 2.

Si le nombre max de voitures par période est 3, la période 2 est pleine.

Si le nombre max de périodes par voiture est 2, la voiture v1 ne peut plus réserver.

Complétez la classe *ParcAConteneurs* en respectant bien la *JavaDoc* et les choix d’implémentation imposés ci-dessus.

Vous devrez compléter le constructeur et les méthodes enregistrerVoiture(), estAutorisee() et listePeriodesNonPleines().

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

La classe *GestionParcAConteneurs* va vous servir pour tester la classe *ParcAConteneurs*.

Vous pouvez la modifier.

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 commande identique, 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 *input.txt* avec déjà quelques commandes. En utilisant ce fichier, le « parcAConteneurs» correspondra à l’exemple ci-dessus.

C *ABRDEntiers* (6 points)

Chaque nœud de cet arbre binaire de recherche contient un entier.

Dans cet arbre, la descendance gauche d’un nœud ne contient que des entiers strictement plus petits que l’entier de ce nœud et sa descendance droite ne contient que des entiers égaux ou plus grands.

Pour les arbres, vous avez eu l’habitude d’écrire les méthodes de façon récursive.

Mais toutes ces méthodes peuvent s’écrire de façon itérative à condition de disposer d’un itérateur (via un *foreach*)

C1 Complétez « à la main » le document qui vous a été remis.

C2 Vous allez compléter la classe *ABRDEntiers*.

La classe interne *Iterateur* est presque complète.

Elle possède un attribut : une file d’entiers (*ArrayDeque<Integer>)*.

Le constructeur de la classe s’occupe de remplir cette file avec tous les entiers contenus dans l’arbre.

Le but de cet itérateur est de parcourir l’arbre en suivant l’**ordre décroissant** des entiers.

Il faut donc « enfiler » les objets dans la file de façon à respecter ce parcours. C’est la méthode remplirFile() qui se charge de remplir la file. A vous de mettre dans le bon ordre les instructions proposées !

Pour les méthodes nombrePositifs()et auMoins1Positif() , vous écrirez une version itérative et une version récursive.

On vous demande chaque fois de fournir une version la plus optimale possible.

Prenez bien en compte que l’arbre est un arbre de recherche et aussi que l’itérateur parcourt les entiers selon l’ordre décroissant.

Vous pouvez ajouter d’autres méthodes.

La classe *TestABRDEntiers* permet de tester les méthodes demandées.

Le document *ABRTestes* donne une visualisation des arbres testés.

C3 Comparaison auMoins1PositifVI()- auMoins1PositifVR()

Complétez « à la main » le document qui vous a été remis.