SD1 : PARTIE « PC » (18 points)

A Liste chaînée (6 points)

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

Cette file n’accepte pas de doublons.

Elle propose les méthodes classiques sur les files : enfile() et defile().

Les suppressions sont autorisées : desister().

Choix d’implémentation :

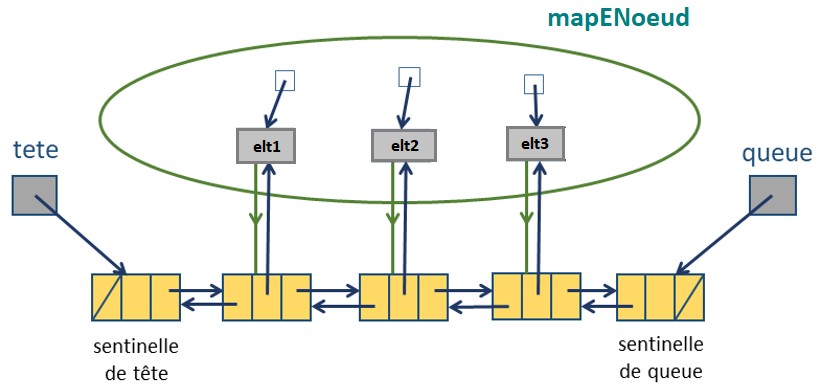
La file est implémentée via une liste doublement chaînée avec sentinelles.

Les ajouts se font en queue et les retraits en tête.

Dans l’exemple suivant, 3 éléments ont été ajoutés dans la file d’attente.

D’abord elt1, ensuite elt2 et enfin elt3.

C’est elt1 qui sera « le premier servi ».



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 file.

Pour éviter de devoir parcourir la liste à la recherche d’un élément, on utilise un *map*. La clé est l’élément, la valeur associée est le nœud contenant cet élément.

Vous allez compléter la classe *FileAttenteAvecDesistementImpl* qui implémente l’interface *FileAttenteAvecDesistement.*

Il vous reste 3 méthodes à implémenter : **enfile()**, **defile()** et **desister()**.

Respectez la *JavaDoc* (reprise dans l’interface) et l’implémentation choisie.

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

Pour vos tests, utilisez la classe *TestFileAttenteAvecDesistement*.

B ABR (5 points)

On va placer les prénoms (*String*) des enfants nés en 2024 dans un ABR.

Il s’agit d’un arbre de **recherche** avec les caractéristiques suivantes :

La descendance **gauche** d’un nœud ne contient que des prénoms **inférieurs (**selon l’ordre alphabétique) au prénom de ce nœud.

La descendance **droite** d’un nœud ne contient que des prénoms **supérieurs (**selon l’ordre alphabétique) au prénom de ce nœud.

La stratégie utilisée pour gérer les **doublons** est d’utiliser un **compteur**.

Chaque nœud contient donc un prénom et son nombre d’occurrences (>0) :

// classe interne

public class Noeud {

private String prenom;

private int nombreOccurrences;

private Noeud gauche;

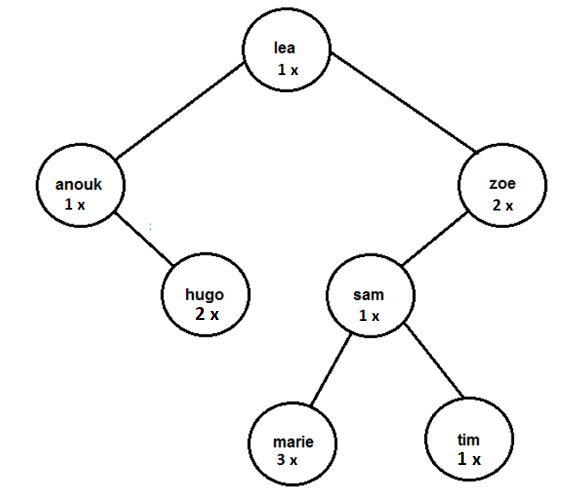
private Noeud droit;

}

Rappel :

La *classe String* implémente l’interface *Comparable*. Elle possède donc la méthode compareTo(). Cette méthode procède à une comparaison selon l’ordre alphabétique.

Voici un exemple d’un tel arbre :



Cet arbre contient les prénoms de 11 enfants. On compte 7 prénoms différents.

On vous demande **de compléter** la classe *ABRPrenoms*.

Cette classe contient 2 classes internes : *Nœud* et *Iterateur*

Cette classe possède comme attribut le nœud racine.

Respectez bien la *JavaDoc* et l’implémentation imposée.

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

La classe *TestABRPrenoms* permet de tester vos méthodes avec les arbres repris dans le document *ABRTestes*.

Voici les **3 méthodes** à implémenter :

La méthode **contientHomonymes()** vérifie si au moins 2 enfants portent le même nom.

Cette méthode doit être implémentée de façon récursive.

La méthode **nombreOccurrences(String prenom)** va renvoyer le nombre d’occurrences du prénom passé en paramètre. Cette méthode doit être implémentée de façon récursive.

La méthode Iterator<String> iterator() qui vous est donnée utilise le constructeur de la classe interne *Iterateur***.**

Le but de cet itérateur est de parcourir les prénoms des enfants selon l’**ordre alphabétique**.

L’itérateur renverra autant de fois un prénom que son nombre d’occurrences le spécifie.

La classe *Iterateur* possède un attribut : filePrenoms (*ArrayDeque<String>*).

La méthode hasNext() vérifie si cette file est vide.

La méthode next() « défile ».

Le constructeur de la classe va s’occuper de remplir cette file avec les prénoms de tous les enfants.

Cette file doit contenir autant de prénoms qu’il y a d’occurrences.

Le constructeur appelle la méthode récursive **remplirFile()** que vous devez implémenter.

C’est la méthode privée de la récursivité.

Pour l’arbre donné en exemple, après construction, la file de prénoms devrait donc contenir (dans cet ordre) :

anouk hugo hugo lea marie marie marie sam tim zoe zoe

C API JAVA : Application Coworking (7 points)

Un espace de coworking met à disposition des bureaux.

Il s’agit d’un espace de travail où on ne retrouve pas uniquement des bureaux. On y trouve un lieu d’accueil, une cuisine, des salles de détente, …

C’est l’occasion pour des petites sociétés de partager le quotidien de ses employés avec d’autres employés.

Pour pouvoir en bénéficier, une société doit contribuer financièrement.

La classe*Coworking* que vous allez **compléter** enregistre l’occupation des bureaux.

On veut savoir à tout moment quel bureau est occupé par quelle société et inversement.

Voici l’implémentation qui a été choisie :

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

La société sera représentée par un *String*.

Le bureau sera représenté par son numéro (*int*). La numérotation des bureaux commence à 0.

Le constructeur de la classe reçoit comme paramètres le nombre de bureaux et une table (*String*) avec les sociétés contributaires. Seules ces sociétés ont le droit d’occuper des bureaux.

Ex :

tableSocietes

|  |  |  |
| --- | --- | --- |
| « socX » | « soc99 » | « soci» |

Les occupations seront enregistrées dans une table de *String*.

Chaque case correspond à un bureau.

Au départ, cette table ne contiendra que des *null*.

On y placera, au fur et à mesure des occupations, les sociétés.

Chaque société contributaire a droit, dans la limite des disponibilités, à MAX bureaux. Ce nombre est le même pour chaque société.

Pour chaque société, on retient les bureaux qu’elle occupe.

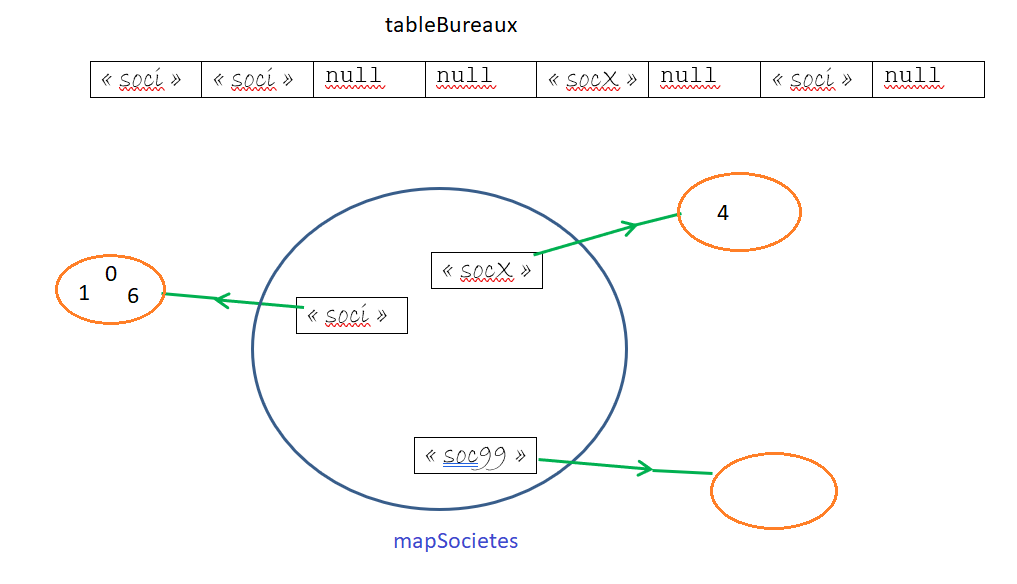
Les numéros des bureaux sont enregistrés dans un ensemble.

On associe donc à chaque société contributaire un ensemble.

C’est un *map* qui permet de retenir tout cela.

Pour ce *map*, on utilise un objet de la classe *HashMap.*

Exemple :



Dans cet exemple, Il y a 8 bureaux.

Les bureaux 0, 1 et 6 sont occupés par des employés de la société soci, le bureau 4 par un employé de la société socX.

Les autres bureaux sont libres.

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

Il vous reste à compléter **le constructeur** et les méthodes **attribuer()**, **liberer()** et **occupationsSociete()**

La classe *GestionCoworking* va vous servir pour tester la classe *Coworking*.

Le point 4 du menu permet d’afficher la table des bureaux 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 *InputCoworking.txt* qui permet la configuration d’un espace de coworking de 8 bureaux avec 3 sociétés contributaires : soci, soc99 et socX.