***Haute Ecole Léonard de Vinci***

Institut Paul Lambin Session de septembre 2021

Examen de SD

B. Lapière – I. Cambron – L. Lecharlier – A. Dupont

Année d’études : 1ère Informatique

Durée de l’examen : 2 h 30

1. **Liste chaînée (7 points)**

**C’est pour cette question que vous pouvez utiliser votre jokerCorona.**

**Si vous décidez de l’utiliser, ne soumettez pas la classe demandée. Toute classe soumise sera corrigée et cotée. C’est cette cote qui sera prise en compte, même si elle est moins bonne que celle de votre joker !**

Un tournoi de tennis de double mixte est organisé.

La classe *ListeDoubleMixte* va contenir la liste des joueurs de tennis inscrits au tournoi. On peut y découvrir les couples formés pour l’occasion.

On y retrouve une alternance d’hommes et de femmes. Pour chaque couple, on place d’abord l’homme et ensuite la femme.

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

adam ⮀ eve ⮀ william ⮀ kate ⮀ serge ⮀ jane

L’implémentation choisie est une liste doublement chaînée **avec sentinelles** + un map.

La clé sera le joueur recherché, la valeur associée est le nœud contenant ce joueur.

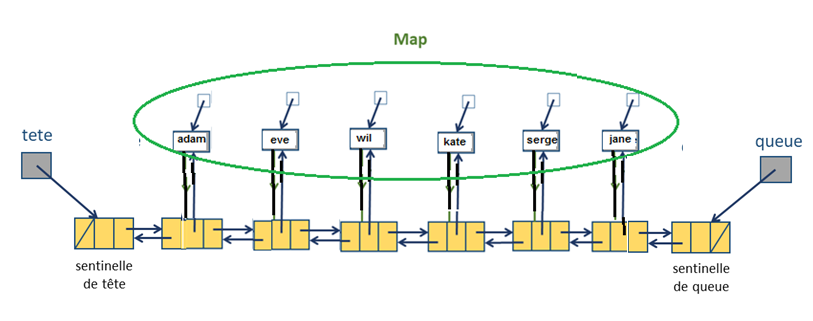
Ce map permet d’accéder directement au nœud contenant un joueur recherché.

Si la recherche échoue, cela signifie que le joueur n’est pas dans la liste.

Une amélioration bien connue de l’implémentation de liste est l’ajout de faux éléments.

L’ajout de ces faux éléments évite de devoir tester de nombreux cas particuliers.

Dans le cas d’une liste doublement chaînée, on ajoute 2 faux éléments. Ces faux éléments sont placés en tête et en queue de liste dans des nœuds appelés « **sentinelles**» ou encore « bidon ». Les éléments contenus dans ces nœuds n’appartiennent pas à la liste.



La classe *JoueurTennis* vous est fournie.

Chaque joueur possède un nom, un sexe, une nationalité et un nombre d’années d’expérience.

Il n’existe pas d’homonyme.

La classe *ListeDoubleMixte* contient une classe interne Nœud.

Chaque nœud contient un joueur, une référence vers le nœud précédent et une vers le nœud suivant.

La classe *ListeDoubleMixte* possède 3 attributs :

Le nœud sentinelle de tete, le nœud sentinelle de queue et le *map* dont l’utilité est donnée ci-dessus.

La classe propose un constructeur qui construit une liste vide.

Les méthodes taille() et estVide() vous sont données.

(La classe contient également un autre constructeur et 2 méthodes de parcours de la liste qui vont servir pour la classe de tests. Ne les modifiez pas.)

**On vous demande de compléter les méthodes auMoins1Belge(),nombreNationalitesDifferentes()** , **estPresent()et supprimerCouple().**

Respectez la *JavaDoc* et l’implémentation choisie.

Les méthodes demandées qui demandent un parcours peuvent être itératives ou récursives. On ne vous impose pas de contrainte.

Vous pouvez introduire d’autres méthodes que celles présentes.

Pour vos tests, vous pouvez utiliser la classe *TestDoubleMixte*.

1. **ABRNoms (5 points)**

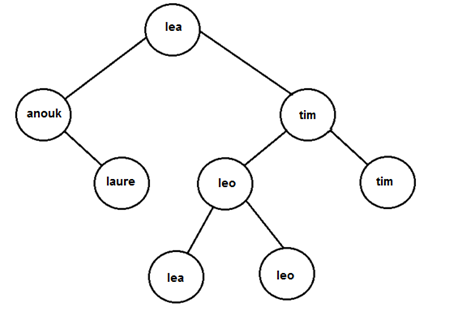
Pour l’examen, intéressons-nous à un arbre binaire de recherche(ABR) qui contient des noms (String).

On fait le choix que la descendance **gauche** d’un nœud ne contiendra que les noms « **inférieurs »** à l’élément de ce nœud et la descendance **droite** d’un nœud ne contiendra que les noms « **supérieurs ou égaux»**.

Voici un exemple :

(Les chaînes de caractères sont triées selon l’ordre alphabétique.)

On y a ajouté successivement : lea anouk laure tim leo lea leo tim

****

Vous allez compléter la classe *ABRNoms*.

Cette classe contient une classe interne *Nœud*.

Elle possède 1 attribut : le nœud racine.

Elle propose le constructeur qui construit l’arbre vide, ainsi que la méthode insere().

Elle possède déjà un itérateur pour lequel on a introduit une classe interne *Iterateur*.

**Vous allez compléter les méthodes nombreNomsCommencantPar(), ensembleHomonymes() et descendingIterator().**

Complétez ces méthodes en respectant bien la *JavaDoc* et les choix d’implémentation imposés.

Vous pouvez introduire d’autres méthodes que celles présentes ainsi qu’une classe interne pour l’itérateur demandé.

La classe *TestABRNoms* permet de tester vos méthodes avec l’arbre mis en exemple ci-dessus, l’arbre vide et un arbre qui contient 7 x leo

1. **API JAVA : Application Viticulture (8 points)**

Nous allons implémenter une application de vente des bouteilles de vin en ligne.

Le producteur aimerait faire connaître son vin au plus de monde possible et éviter qu’un même client en prenne beaucoup.

Dans le système à mettre en place, un client qui n’a encore rien commandé doit être prioritaire par rapport à un client qui a déjà passé commande.

Le client le moins prioritaire est celui qui a commandé au moins 3 bouteilles.

Si 2 clients ont la même priorité, c’est celui qui a accéder le premier au magasin en ligne qui sera le premier servi.

Lorsqu’un client accède au magasin en ligne, il sera placé en attente. Lorsque ce sera son tour il aura le choix entre demander une bouteille ou annuler toute sa commande.

Il n’y a pas de limite dans le nombre de bouteilles qu’un client pourra commander, mais il devra le faire en plusieurs fois.

Le nombre de bouteilles à vendre dépendra de la production.

Les clients viendront chercher leurs bouteilles sur place.

Implémentation choisie :

Dans le cadre de l’examen, nous n’allons 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 bouteilles qu’il a demandé. Il est possible d’augmenter de 1 le nombre de bouteilles.

Vous allez complétez la classe *VenteVin*.

Le constructeur de cette classe et les attributs sont donnés.

Pour gérer les priorités, on va utiliser 4 files d’attente (*ArrayDeque<String>*). Elles sont placées dans une table.

La file à l’indice 0 contiendra les clients qui n’ont pas encore passé de commande.

La file à l’indice 1 contiendra les clients qui ont une commande d’une bouteille.

La file à l’indice 2 contiendra les clients qui ont une commande de deux bouteilles.

La file à l’indice 3 contiendra les clients qui ont une commande de trois bouteilles ou plus.

L’ajout se fera en fin de file et les retraits en tête.

Un client peut prendre place dans une des files à condition qu’il ne se trouve pas déjà dans une de celles-ci.

On décide d’ajouter l’ensemble (*HashSet<String>*) des clients en attente. La vérification d’existence est moins coûteuse dans un ensemble que dans une table de files!

On va retenir toutes les commandes dans une liste (*ArrayList<Commande>*). Cette liste permet de parcourir toutes les commandes selon l’ordre dans lequel elles ont été créées.

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 :

Actuellement, il y a 5 clients en attente.

Voici dans quel ordre seront traités les clients : cli6 cli7 cli3 cli1 cli5

Dans la file à l’indice 0 on retrouve cli6 et cli7. Ils n’ont pas encore de commande. Cli7 est arrivé après cli6. Cli6 sera le premier servi.

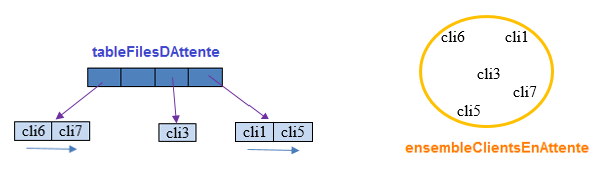
La file à l’indice 1 est vide.

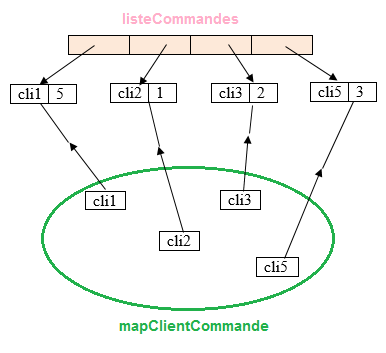
Dans la file à l’indice 2, on retrouve cli3 qui a pour l’instant commandé 2 bouteilles.

Dans la file à l’indice 3, on retrouve cli1 qui a commandé 5 bouteilles et cli5 qui en a commandé 3. L’ordre des clients dans cette file ne dépend pas du nombre de bouteilles mais du moment où ils ont passé demande.

Cli2 n’est pas en attente et a commandé 1 bouteille.

Cli4 a passé commande mais l’a supprimée, on ne retrouve plus aucune trace de ce client.





En plus de ces structures de données, l’attribut nombreBouteillesRestantes permet une implémentation efficace.

Prenez bien connaissance de tous les attributs de la classe avant de vous lancer dans la programmation !

Complétez la classe *VenteVin* 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*)

Utilisez uniquement les méthodes des classes *ArrayList*, *ArrayDeque, HashMap et HashSet* reprises dans le document API\_JAVA.

La classe *GestionVenteVin* va vous servir pour tester la classe *VenteVin*.

Vous pouvez la modifier.

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