***NOM :***

***PRENOM :***

***LOGIN D’EXAMEN :***

***NUMERO PC :***

***LOCAL :***

***Haute Ecole Léonard de Vinci***

Institut Paul Lambin Session de septembre 2019

Examen de SD

A. Legrand – P. VanEerdenbrugghe – L. Lecharlier – A. Dupont

Année d’études : 1ère Informatique

Durée de l’examen : 2 h 30 ; pas de sortie durant les 60 premières minutes

**MISE EN GARDE :**

**L’examen se déroule sur PC.**

**De nombreuses classes tests vous sont proposées.**

**Si une méthode demandée ne fonctionne pas, vous allez être tenté de la solutionner surtout si vous en avez besoin pour la suivante.**

**MAIS le temps va passer vite.**

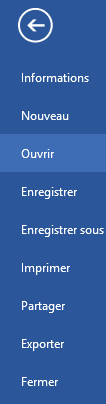
**Contrairement à ce qu’on vous demande pendant l’année, passez à une autre méthode même si la précédente n’est pas bonne.**

**En cas d’erreurs, vous n’aurez pas le maximum, mais peut-être pas 0 !**

Déroulement de l’examen :

* **Notez sur ce questionnaire vos nom, prénom, local, numéro de PC et votre login Windows d’examen.**
* **A la racine de votre Z:/ vous trouverez un zip contenant les fichiers dont vous avez besoin. « Dézippez »-le. (Extract here)**
* **A la demande du workspace  mettez Z :**
* **Créez un projet Java qui porte votre Nom\_Prenom et placez-y les classes données**
* **Mettez votre nom et votre prénom dans toutes les classes à compléter et complétez ces classes avec Eclipse.**
* **N’effacez pas votre workspace et son contenu !!!**
* **L’examen se termine quand vous aurez montré au surveillant le répertoire créé portant votre nom contenant vos classes et remis ce questionnaire. Pensez à éteindre votre poste de travail (Arrêter le système).**

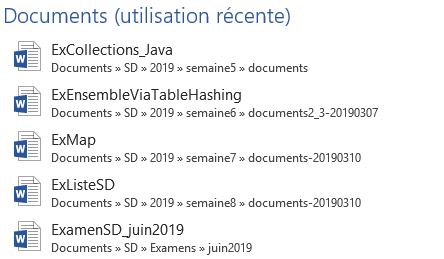
1. **Liste chaînée LRU (8 points)**



Le menu ouvrir de Word présente les documents récemment utilisés.

Ce nombre est limité. Il sera de 5 dans les exemples.

Exemple :

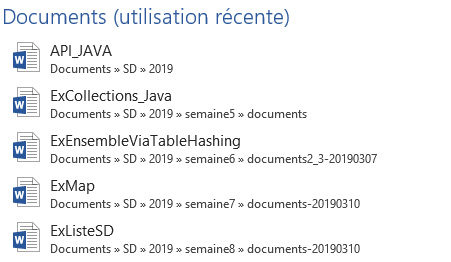


L’ordre d’apparition est important. C’est le document *ExCollections\_Java* qui apparait en premier, car il est le document qui a été ouvert le plus récemment.

Ensuite c’est *ExEnsembleViaTableHashing*, l’avant dernier à avoir été ouvert et ainsi de suite.

Le fait d’ouvrir un document qui n’était pas encore présent dans la liste a comme conséquence que celui-ci apparaît maintenant en premier lieu (MRU – Most Recently Used) et que le dernier document de la liste disparait si le nombre max de documents à faire apparaître est atteint.

Le document qui (éventuellement) disparait est celui qui a été le moins récemment utilisé (LRU – Least Recently Used).



Dans l’exemple, le document *API\_JAVA* vient d’être ouvert. Il apparaît en premier lieu.

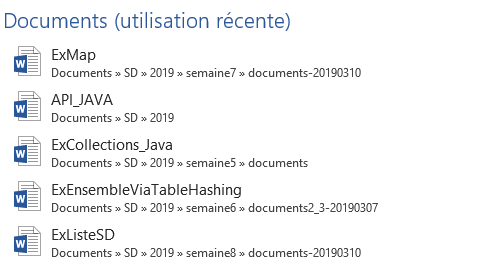
Le document *ExCollections\_Java* apparait maintenant en deuxième, …

Le document *ExamenSD\_juin2019* n’apparait plus pour respecter la capacité max qui est de 5.

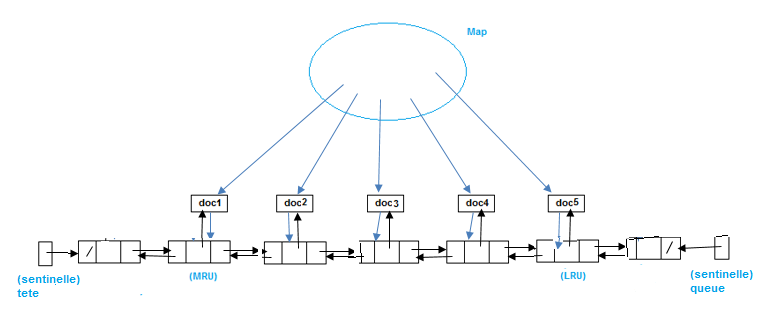
Si on ouvre un document qui se trouve déjà dans la liste, il est déplacé en premier.

Il devient le document le plus récemment utilisé.

Dans l’exemple qui suit, c’est le document *ExMap* qui a été ouvert :



Voici l’implémentation qui a été choisie afin d’optimiser le mécanisme de LRU :



Les documents sont placés dans une liste doublement chaînée avec sentinelles.

Il y a deux sentinelles : une sentinelle de tête et une sentinelle de queue.

Ces sentinelles sont des nœuds « bidon » qui permettent d’éviter de tester des cas particuliers dans les méthodes qui modifient la liste.

Le nœud qui suit la sentinelle de tête contient le document le plus récemment utilisé (MRU).

Les documents apparaissent dans la liste du plus récemment utilisé au moins récemment utilisé.

Pour trouver le nœud contenant un document qu’on recherche, on utilise un *map*.

La *clé* est le document recherché, la *valeur* associée est le nœud contenant ce document.

Justifiez (en français) pourquoi l’utilisation de ce *map* est intéressante.

(On s’attend, dans cette explication à voir apparaître des ordres de complexité.)

On vous demande de compléter la classe *DocumentsLRU*.

Les attributs et le constructeur sont donnés.

Le constructeur construit une liste sans aucun document. Elle contient donc uniquement les 2 sentinelles. Il reçoit en paramètre le nombre maximum de documents à faire apparaître.

Vous allez compléter les 2 méthodes *public* :

**void ouvrirDocument(String document)** qui place ce document dans la liste selon le mécanisme LRU expliqué ci-dessus.

**String toString()** qui renvoie une chaîne de caractères avec tous les documents contenus dans la liste.

Par souci de clarté, nous vous proposons de compléter quelques méthodes *private*.

Vous pouvez en introduire d’autres.

**Noeud donnerNoeud(String document)**

**void supprimer(Noeud noeud)**

**void ajouterMRU(String document)**

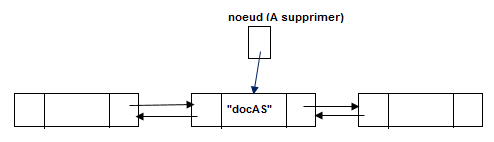
**void supprimerLRU()**

Avant de programmer, complétez les schémas qui suivent, comme au cours.

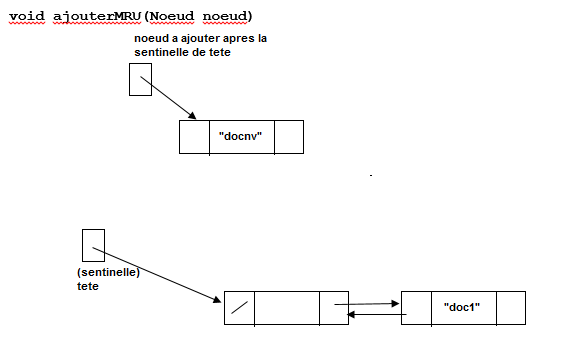
(Dans le répertoire de l’examen, vous trouverez le document *pdf* de la semaine 3 où apparaît un tel schéma complété.)

**void supprimer(Noeud noeud)**

(Le nœud à supprimer n’est ni la sentinelle de tête, ni celle de queue)



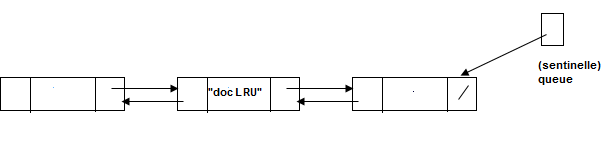
Instructions :



Instructions :

**void supprimerLRU()**

(Le nœud à supprimer n’est pas la sentinelle de tête.)



Instructions :

La classe *TestDocumentsLRU* vous est donnée.

Voici ce qu’elle devrait afficher à l’écran :

Au depart :

ouvrir doc1

doc1

ouvrir doc2

doc2 doc1

ouvrir doc3

doc3 doc2 doc1

ouvrir doc2

doc2 doc3 doc1

ouvrir doc4

doc4 doc2 doc3 doc1

ouvrir doc5

doc5 doc4 doc2 doc3 doc1

ouvrir doc6

doc6 doc5 doc4 doc2 doc3

ouvrir doc3

doc3 doc6 doc5 doc4 doc2

ouvrir doc3

doc3 doc6 doc5 doc4 doc2

ouvrir doc1

doc1 doc3 doc6 doc5 doc4

ouvrir doc5

doc5 doc1 doc3 doc6 doc4

1. **API JAVA : Application Brocante (6 points)**

L’organisateur d’une brocante désire informatiser l’attribution des différents emplacements.

Il y a 2 types d’emplacement :

Les emplacements de type A sont des stands d’animations : bar, pêche au canard, stand de tir, …

Les emplacements de type B sont prévus pour les brocanteurs.

L’attribution des emplacements se déroule différemment selon son type.

Pour le type A :

Le demandeur a la possibilité de choisir son emplacement.

Mais comme ceux-ci sont en nombre limité, il ne pourra en réserver qu’au maximum 1 de ce type.

Pour le type B

Ces emplacements sont attribués de façon automatisée.

Le demandeur ne peut pas les choisir.

Il n’y a aucune limite dans le nombre de réservation de ce type.

L’organisateur aimerait bien garder une trace des attributions.

Voici l’implémentation qui a été choisie :

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

L’exposant sera représenté par un *String*.

La classe *Emplacement* est fournie.

Un emplacement est représenté par son numéro (*int*). La numérotation des emplacements commence à 0.

Il possède 1 type (*char*) : A ou B

Il possède un exposant (*String*) (*null* si pas d’exposant)

Pour attribuer de façon automatisée les emplacements de type B, la classe *Brocante* va utiliser une **pile**.

Cette pile contiendra les emplacements libres de type B.

Un exposant ne peut réserver qu’un emplacement de type A. Ces exposants seront placés dans un **ensemble** afin de les retenir.

Chaque emplacement est gardé dans une **table**.

L’indice de la table correspond au numéro de l’emplacement.

Cela permet de garder une trace des attributions.

Dans le schéma qui suit, on trouve une brocante avec 8 emplacements.

table des emplacements

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |

|  |  |  |
| --- | --- | --- |
| 7 | ‘A’ | «b3» |

|  |  |  |
| --- | --- | --- |
| 3 | ‘A’ | / |

|  |  |  |
| --- | --- | --- |
| 5 | ‘B’ | «b1» |

|  |  |  |
| --- | --- | --- |
| 2 | ‘B’ | / |

|  |  |  |
| --- | --- | --- |
| 4 | ‘B’ | «b2» |

|  |  |  |
| --- | --- | --- |
| 6 | ‘B’ | «b2» |

|  |  |  |
| --- | --- | --- |
| 1 | ‘A’ | «b1» |

|  |  |  |
| --- | --- | --- |
| 0 | ‘B’ | / |

|  |
| --- |
|  |
|  |

«b3»

«b1»

pile des emplacements

de type ‘B’ libres

Ensemble des exposants

avec des emplacements de type ‘A’

Le constructeur reçoit en paramètre une table reprenant les types des emplacements.

Il va créer les différents emplacements. Il va placer chaque emplacement dans la table des emplacements et éventuellement dans la pile.

Ne perdez pas de vue que les piles et la table référencient des mêmes objets !!!

Pour l’exemple ci-dessus, voici la table reçue :

Voici un exemple de table passée en paramètre :

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| ‘B’ | ‘A’ | ‘B’ | ‘A’ | ‘B’ | ‘B’ | ‘B’ | ‘A’ |

L’emplacement 0 est de type ‘B’

L’emplacement 1 est de type ‘A

…

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

Pour la pile, vous utiliserez un objet de la classe *ArrayDeque* et pour l’ensemble, un objet de la classe *HashSet.*

Contrainte : utilisez uniquement les méthodes reprises dans le document API\_JAVA !

Testez votre implémentation.

La classe *TestBrocante* propose déjà quelques tests basiques.

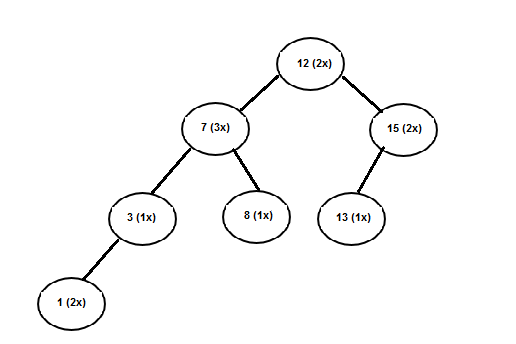
A vous de la compléter !(Cette classe ne sera pas évaluée)

1. **ABR (6 points)**

Dans le cadre de l’examen, l’arbre de recherche utilisé est un arbre d’entiers où tout entier situé dans un sous-arbre de gauche devra être inférieur à l’entier contenu dans la racine de cet arbre. Tout entier situé dans un sous-arbre de droite devra lui être supérieur.

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

Voici un exemple d’un tel arbre :



Cet arbre contient 12 entiers : 1 1 3 7 7 7 8 12 12 13 15 15

L’implémentation choisie pour l’examen est la même que celle utilisée pendant l’année.

Cependant la classe *NœudEntier* possède un attribut supplémentaire : nombreOccurrences

Ce nombre doit toujours être positif (>0).

// classe interne

**public** **class** NoeudEntier {

**private** **int** entier;

**private** **int** nombreOccurrences;

**private** NoeudEntier gauche;

**private** NoeudEntier droit;

}

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

Vous pouvez introduire d’autres méthodes (private).

La classe *TestABRDEntiers* permet de tester les méthodes complétées avec l’arbre ci-dessus.

(Pour l’arbre ci-dessus, la méthode toString() renverra :

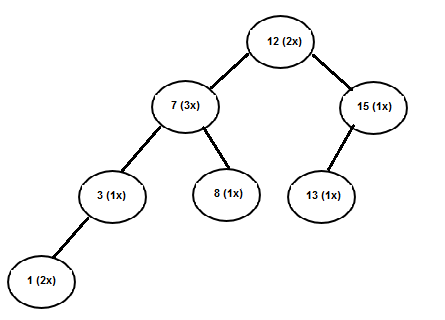
[ [ [ [ 1(2x) ] 3(1x) [ ] ] 7(3x) [ 8(1x) ] ] 12(2x) [ [ 13(1x) ] 15(2x) [ ] ] ])

La méthode **contientPlusieurs(int entier)** vérifie que l’entier passé en paramètre est present dans l’arbre et possède plus d’une occurrence. Cette méthode est une méthode récursive.

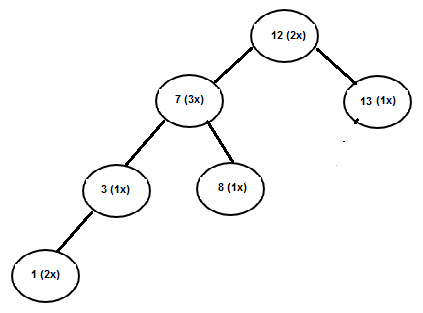
La méthode **supprimeMax()** va supprimer une fois le plus grand entier de l’arbre.

Après suppression, l’arbre doit continuer à être un ABR avec les propriétés citées ci-dessus. Cette méthode est une méthode itérative ou récursive. Dans le cadre de l’examen, on vous impose de l’écrire de façon récursive !

L’arbre ci-dessus après appel de cette méthode :



Si on appelle encore une fois cette méthode, l’arbre devient :



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

Dans cette classe, il vous reste à compléter les méthodes **remplirFile(), hasNext()** et **next()**.

La classe *Iterateur****Dec*** possède un attribut : une file d’entiers (*ArrayDeque<Integer>*).

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

La méthode hasNext() vérifie si la file est non vide.

La méthode next() « défile ».

Le but de cet itérateur est de parcourir les entiers de l’arbre selon **l’ordre décroissant**.

Il faut donc « enfiler » les entiers dans la file de façon à respecter ce parcours. C’est la méthode remplirFile() qui se charge de remplir la file. Il s’agit d’une méthode récursive.

Notez que l’itérateur renverra autant de fois un entier que son nombre d’occurrences le spécifie.

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

15 15 13 12 12 8 7 7 7 3 1 1