***NOM :***

***PRENOM :***

***LOGIN D’EXAMEN :***

***NUMERO PC :***

***LOCAL :***

***Haute Ecole Léonard de Vinci***

Institut Paul Lambin Session de juin 2017

Examen de SD-Fichiers

G. Seront - A. Legrand – J-L Collinet – L. Leleux – 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)**
* **Créez votre workspace sur votre Z : et appelez-le : workspace**

**Si nécessaire, switchez votre workspace afin qu’il y soit positionné.**

* **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 contenant vos classes et remis ce questionnaire. Pensez à éteindre votre poste de travail (Arrêter le système).**

1. **ABR (4 points)**

Voici la convention qui a été choisie pour les arbres de recherche (ABR) de cette question :

Tout entier situé dans un sous-arbre de gauche devra être inférieur à l’entier situé dans la racine de ce sous-arbre. Tout entier situé dans un sous-arbre de droite devra lui être supérieur ou égal.

a) Dessinez l’ABR obtenu en partant de l’arbre vide et en insérant successivement les entiers :

12 20 14 9 15 17 10

(Donnez l’arbre final. Chaque nœud y sera présenté par un simple rond contenant sa valeur.)

b) Considérez l’ABR obtenu en a).

Donnez les entiers dans l’ordre qu’ils seront parcourus lors d’un parcours en in-ordre.

c) Donnez un ABR le mieux équilibré possible qui possède les mêmes entiers que l’ABR obtenu en a)

d) Remplissez le tableau suivant en mettant les coûts des opérations

(O(1), O(logN), O(N), O(NlogN), O(N2),…) :

|  |  |  |
| --- | --- | --- |
|  | ABR | ABR équilibré |
| recherche |  |  |
| ajout |  |  |
| suppression |  |  |

1. **Liste chaînée (8 points)**

Vous allez complétez la classe *ListeCaracteres*.

Elle est implémentée via pointeurs.

Elle possède comme attribut un nœud de tête.

Chaque nœud contient un caractère et référencie le nœud suivant.

Une liste vide a le nœud de tête à null.

La liste est une structure de données récursive.

Toutes les méthodes de cette classe peuvent s’écrire de façon récursive.

Par exemple :

/\*\*

\* calcule le nombre de fois qu'apparait le caractere passe en parametre \* @param caractereRecherche

\* @return le nombre d'occurrences du caractere

\*/

public int nombreOccurrences(char caractereRecherche){

return nombreOccurrences(tete,caractereRecherche);

}

private int nombreOccurrences(NoeudCaractere noeud, char caractereRecherche) {

if(noeud == null)

return 0;

if(noeud.caractere==caractereRecherche)

return 1 + nombreOccurrences(**noeud.suivant**,caractereRecherche);

return nombreOccurrences(**noeud.suivant**, caractereRecherche);

}

La méthode nombreOccurrences(char caractereRecherche)appelle la méthode nombreOccurrences(NoeudCaractere noeud,char caractereRecherche)en donnant le nœud de tête en paramètre.

C’est la méthode appelée qui est récursive. La méthode y est rappelée avec le nœud suivant.

La méthode toString() qui vous est donnée a aussi été écrite de façon récursive !

Vous allez écrire 2 méthodes de façon itérative et 2 méthodes de façon récursive.

Le comportement des 4 méthodes à compléter est décrit dans les commentaires qui les précèdent dans le fichier java. Veillez à respecter ce qui y est décrit.

Vous pouvez ajouter d’autres méthodes que celles qui vous ont été données.

La classe *TestListeCaracteres* permet de tester ces 4 méthodes.

Attention, ce n’est pas parce que tous les tests d’une méthode ont réussi que celle-ci est forcément correcte.

Voici les 4 méthodes demandées et les indications à suivre pour leur implémentation:

1) La méthode inserer(int position, char caractere)insère un nouveau nœud avec le caractère donné à la position donnée.

Le premier nœud de la liste se trouve en position 0.

Il peut être possible d’ajouter un nouveau nœud entre 2 nœuds de la liste, mais également au début ou juste derrière le dernier nœud existant, mais pas plus loin.

Vous écrirez cette méthode de façon itérative.

2) La méthode contientExAequo() vérifie la présence d’ex-aequo dans la liste.

Cette méthode va utiliser un ensemble (*HashSet<Character>*) qui va retenir tous les caractères rencontrés au fur et à mesure de son parcours.

La présence de cet ensemble permet une implémentation en O(N).

Vous écrirez cette méthode de façon itérative.

3) La méthode contient(char caractereRecherche)vérifie si le caractère recherché est présent dans la liste.

Vous écrirez cette méthode de façon récursive.

.

4) La méthode nombreCaracteresTable(char[] tableCaracteresRecherches) calcule le nombre de caractères de la liste qui se trouvent parmi les caractères de la table passée en paramètre.

Vous écrirez cette méthode de façon récursive.

La recherche dans un ensemble est plus efficace que dans une table.

Vous placerez les caractères de la table dans un ensemble (*HashSet<Character>*).

Ce remplissage ne sera fait qu’une fois dans la méthode appelante.

Cet ensemble sera donné en paramètre à la méthode appelée.

1. **Deque – UnshuffleSort (8 points)**

a) Un *deque* (double ended queue) est une structure de données dans laquelle les ajouts et les retraits peuvent se faire aux 2 extrémités.

Voici l’implémentation qui a été choisie pour l’examen:

Un *deque* peut être implémenté via une table.

Pour obtenir un ordre de complexité en O(1) pour les opérations d’ajout et de suppression, il faut laisser de la place libre en début de table.

Le premier élément prendra place au milieu de la table.

Lors d’un ajout (au début ou en fin) s’il n’y a plus de place, on double la capacité de la table et on recentre le mieux possible les éléments en milieu de table.

Exemple :

Au départ :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  |  |  |  |

indicePremier = -1 indiceDernier = -1 taille = 0

ajouterEnPremier("a")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  |  | "a" |  |  |

indicePremier = 2 indiceDernier = 2 taille = 1

ajouterEnPremier("b")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  | "b" | "a" |  |  |

indicePremier = 1 indiceDernier = 2 taille = 2

ajouterEnDernier("c")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
|  | "b" | "a" | "c" |  |

indicePremier = 1 indiceDernier = 3 taille = 3

ajouterEnPremier("d")

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 |
| "d" | "b" | "a" | "c" |  |

indicePremier = 0 indiceDernier = 3 taille = 4

ajouterEnPremier("e")

SOIT :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  | "e" | "d" | "b" | "a" | "c" |  |  |  |

indicePremier = 2 indiceDernier = 6 taille = 5

SOIT :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|  |  |  | "e" | "d" | "b" | "a" | "c" |  |  |

indicePremier = 3 indiceDernier = 7 taille = 5

On vous demande de compléter la classe *DequeImpl*. Cette classe implémente l’interface *Deque*. Le comportement des méthodes à compléter est décrit dans les commentaires. Veillez à respecter ce qui y est décrit.

La classe *TestDequeImpl* permet de tester la classe *DequeImpl*.

Attention, ce n’est pas parce que tous les tests d’une méthode ont réussi que celle-ci est forcément correcte.

b) L’*UnshuffleSort* est un algorithme de tri.

Vous avez écrit, ce semestre, une version simplifiée qui utilise une liste de piles.

Dans la version de base, cet algorithme de tri utilise une liste de *deques*.

L’algorithme comporte deux étapes. La première consiste à répartir les entiers à trier dans un nombre variable de *deques*.

Lorsque tous les entiers auront été répartis, la deuxième étape se chargera de remplir la table à renvoyer.

Les 2 étapes sont basées sur le principe suivant :

La liste des *deques* devra toujours être triée en utilisant le premier entier de chaque *deque* comme clef de tri. Le *deque* aussi est trié.

Dans le cadre de l’examen, on ne vous demande pas d’écrire tout l’algorithme, mais uniquement la méthode placerEntier(int entier) qui est utilisée dans la 1ère étape :

Chaque entier sera placé dans le premier *deque* qui peut le contenir.

Si aucun *deque* ne peut contenir l’entier, un nouveau *deque* avec cet entier sera placé en fin de liste.

Exemple :

Au départ la liste est vide.

placerEntier(3)

|  |
| --- |
| 3 |

Un *deque* est placé dans la liste. L’entier 3 est ajouté en premier.

placerEntier(5)

|  |  |
| --- | --- |
| 3 | 5 |

Le *deque* peut contenir l’entier 5 tout en respectant le tri du *deque*. 5 est placé en dernier.

placerElement(2)

|  |  |  |
| --- | --- | --- |
| 2 | 3 | 5 |

Le *deque* peut contenir l’entier 2 tout en respectant le tri du *deque*, 2 est placé en premier.

placerElement(4)

|  |  |  |
| --- | --- | --- |
| 2 | 3 | 5 |

|  |
| --- |
| 4 |

Le *deque* ne peut contenir l’entier 4. Le tri n’est pas respecté si on ajoute 4 en premier ou en dernier. Un nouveau *deque* avec 4 est placé en fin de liste.

placerElement(8)

|  |
| --- |
| 4 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 5 | 8 |

Le 1er *deque* peut contenir l’entier 8 tout en respectant le tri du *deque*. 8 est placé en dernier.

placerElement(7)

|  |  |
| --- | --- |
| 4 | 7 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 5 | 8 |

Le 1er *deque* ne peut contenir l’entier 7 tout en respectant le tri du *deque*.

Le 2ème deque peut contenir l’entier 7 tout en respectant le tri du *deque*. 7 est placé en dernier.

placerElement(6)

|  |  |
| --- | --- |
| 4 | 7 |

|  |  |  |  |
| --- | --- | --- | --- |
| 2 | 3 | 5 | 8 |

|  |
| --- |
| 6 |

Le 1er *deque* ne peut contenir l’entier 6 tout en respectant le tri du *deque*.

Le 2ème *deque* ne peut contenir l’entier 6 tout en respectant le tri du *deque*.

Un nouveau *deque* avec 6 est placé en fin de liste.

On vous demande de compléter la classe *UnshuffleSor*t.

Pour la liste vous utiliserez un objet de la classe *Linked<>.*

Pour les *deques*, vous utiliserez des objets de la classe *DequeImpl* que vous avez implémentée.

La méthode toString() peut être très simple. Elle va vous servir pour vos tests.

Voici un exemple de ce que pourrait être l’affichage du String renvoyé :

2 3 5 8

4 7

6

La classe *TestUnshuffleSort* place les entiers de l’exemple ci-dessus dans cet ordre.