Lille iut Université de Lille

Examen du 9 novembre 2017

M3103 – Algorithmique avancée
DUT info 2ème année
2017–2018

Documents et calculatrices interdits.
Les trois exercices sont indépendants et pas nécessairement en ordre de difficulté croissante.
Vous rédigerez **chaque** exercice sur une **copie séparée**.

Exercice 1

Dans cet exercice, nous allons mettre des chaînes de caractères dans une table de hachage.

Question 1 : Pour le calcul du hashcode d'une chaîne, nous choisissons de faire la somme des rangs des lettres de la chaîne : le mot `a` aura pour hashcode 1, le mot `dut` aura pour hashcode 45 [1]. Quels sont les hashcodes des mots suivants :

Chaîne	Valeur de hachage
dut	
structure	
de	
donnees	
iut	
informatique	
programmation	
hpc	
java	

Question 2 : Programmez une fonction qui calcule ainsi le hashcode d'une chaîne de caractères.

Question 3 : Nous considérons maintenant une table de hachage de taille 17, initialement vide. Les collisions sont gérées par sondage linéaire. Ajoutez successivement les valeurs suivantes à la table de hachage. **Pour chaque élément, vous préciserez à quel index, il doit s'insérer, ainsi que les étapes de ce calcul.**

Élément	Index primaire	Sondages nécessaires
dut		
structure		
de		
donnees		
iut		
informatique		
programmation		
hpc		
java		

Question 4 : Puis supprimez la valeur `structure` en expliquant votre démarche.

Exercice 2

Question 5 : Expliquez où se situe le maximum d'un sous-arbre dans un arbre binaire de recherche. Détaillez votre réponse en fonction des différentes configurations possibles.

Question 6 : À partir de l'arbre binaire de recherche vide, ajoutez successivement les valeurs suivantes : 32, 17, 21, 29, 34, 12, 41, 7, 27, 35, 44, 25 et 18. **Détaillez toutes les étapes !**

Question 7 : Puis supprimez les valeurs 17 puis 32 en détaillant aussi votre démarche.

Votre classe d'arbre binaire de recherche est déclarée ainsi :

```
public class BST<E> {
    E element;
    BST<E> left;
    BST<E> right;
}
```

Question 8 : Programmez une méthode `isSubTreeSameRoot(BST<E> bst` adaptée à votre classe d'arbre de telle façon qu'elle retourne `true` si et seulement si l'instance courante (`this`) est un sous-arbre [2] de l'arbre passé en paramètre (`bst`) et que les deux arbres aient même racine (au sens `equals`).

Question 9 : Programmez une méthode `isSubTree` adaptée à votre classe d'arbre de telle façon qu'elle retourne `true` si et seulement si l'instance courante est un sous-arbre de l'arbre passé en paramètre. Pensez à utiliser la méthode de la question précédente ainsi que tout autre méthode auxiliaire que vous jugerez judicieuse.

Exercice 3

Le but de cet exercice est d'implémenter une classe de tableau creux (`SparseArray<E>`) à l'aide d'une `Map<Integer, E>`. Un tableau creux est un tableau qui ne stocke que les indices où les valeurs sont non-nulles. Par exemple, le tableau

3.14	0	0	0	0	0	2.81

serait stocké avec seulement deux associations : (0, 3.14) et (6, 2.81). Le premier élément correspond à l'indice, le second à la valeur stockée à cet indice.

```
public interface SparseArray<E> {
    /** Removes all key-value mappings from this SparseArray. */
    void  clear();

    /** Creates and returns a copy of this object. */
    SparseArray<E> clone();

    /** Removes the mapping from the specified key, if there was
    any. */
    void  delete(int key);

    /** Gets the Object mapped from the specified key, or null if
    no such mapping has been made. */
    public E get(int i);

    /** Adds a mapping from the specified key to the specified value,
    replacing the previous mapping from the specified key if there
    was one. */
    public void put(int i, E e);
}
```

Question 10 : Écrivez une classe Java qui implémente cette interface. Toutes les associations index vers valeur seront stockées dans une `Map<Integer, E>`.

N.B. N'oubliez pas le constructeur !

D'autre part voici un extrait de la JavaDoc de Map :

void	clear() Removes all of the mappings from this map (optional operation).
boolean	containsKey(Object key) Returns true if this map contains a mapping for the specified key.
boolean	containsValue(Object value) Returns true if this map maps one or more keys to the specified value.
Set<Map.Entry<K,V> >	entrySet() Returns a Set view of the mappings contained in this map.
boolean	equals(Object o) Compares the specified object with this map for equality.
V	get(Object key) Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
int	hashCode() Returns the hash code value for this map.
boolean	isEmpty() Returns true if this map contains no key-value mappings.
Set<K>	keySet() Returns a Set view of the keys contained in this map.
V	put(K key, V value) Associates the specified value with the specified key in this map (optional operation).
void	putAll(Map< ? extends K, ? extends V> m) Copies all of the mappings from the specified map to this map (optional operation).
V	remove(Object key) Removes the mapping for a key from this map if it is present (optional operation).
int	size() Returns the number of key-value mappings in this map.
Collection<V>	values() Returns a Collection view of the values contained in this map.