Lille iut Université de Lille

TP 6 : Arbres Binaires de Recherche (ABR)

Structures de données
DUT info 2ème année
2020–2021

Dans ce TP nous allons réviser les `Map`, tout d'abord en utilisant une `Map` pour compter les occurrences de mots dans un texte, puis implémentant cette interface à l'aide d'un arbre binaire de recherche. Nous ne nous occupons absolument pas de l'équilibrage de l'arbre. La seule propriété que nous garantissons sur l'arbre, c'est qu'il est un arbre de recherche.

Utilisation de `Map`

Question 1 : Faites un programme qui lit un texte (en UTF-8) et compte au fur et à mesure les occurrences des mots dans ce texte. Cette information sera conservée dans une `Map<String, Integer>`.

Question 2 : Après lecture du texte, votre programme affichera les cinq mots les plus courants dans le texte avec leur nombre d'apparitions.

Question 3 : Essayez votre programme sur le texte de *Nôtre-Dame de Paris* que vous aurez au préalablement téléchargé depuis le site.

Implémentation de `Map`

Pour cette partie, vous créerez une classe `BinarySearchTreeTest` pour vos tests, et vous l'enrichirez tout au long du TP par des tests pertinents. N'hésitez pas à vous inspirer des classes de tests utilisées tout au long des TP précédents.

Question 4 : Créez une classe `BinarySearchTree<K,V>` qui implémente `Map<K,V>`. Cette classe contiendra deux attributs `key` et `value` pour stocker une association. Vous y ajouterez deux attributs `left` et `right`, de type `BinarySearchTree<K,V>`. Ces deux attributs serviront à désigner les fils respectivement gauche et droit du noeud. Vous y inclurez enfin tout ce qui vous semble nécessaire pour la base de votre classe (constructeurs, `Comparator`[1]...).

Question 5 : Redéfinissez la méthode `toString`, de façon à ce qu'un appel à `System.out.println(myBST)` affiche[2] les associations contenues dans l'arbre dans l'ordre infixe. N'hésitez pas à faire d'autres affichages alternatifs : si, par exemple, les sous-arbres sont facilement identifiables (typiquement en les mettant entre parenthèses), cela vous facilitera le debugging.

Question 6 : Programmez une méthode `BinarySearchTree<K,V> search(K key)` qui localise l'emplacement potentiel d'une clé dans l'arbre. Programmez aussi les autres méthodes utilitaires sur les arbres binaires de recherche (telles `bstMin` ou `deleteMin` par exemple).

Question 7 : Programmez toutes les méthodes d'ajout et de suppression héritées de `Map`

Question 8 : Créez une classe `BSTIterator` implémentant l'interface `Iterator`. Afin de garder trace de l'élément couramment pointé ainsi que le chemin jusqu'à lui, cette classe pourra contenir un attribut de type `Stack<BinarySearchTree>`.

Question 9 : Programmez les méthodes de `BinarySearchTree` héritées de `Map`.

Question 10 : La version courante de `BSTIterator` oblige à faire une descente dans l'arbre à chaque fois que le noeud courant n'a pas de fils droit. Pour éviter cela, conservez tout la branche courante dans une `Stack` (remplacez l'attribut) ; puis reprogrammez les méthodes de l'`Iterator`.

Question 11 : Testez votre implémentation de `Map` avec le programme de la question 3.

Question 12 : Enrichissez votre classe `BinarySearchTree` pour qu'elle utilise le `Comparator` s'il est précisé à l'instanciation de la `Map` (via le constructeur), sinon elle utilise la comparaison par défaut, à savoir celle induite par le fait que les clés implémentent `Comparable`.

1. Si cela vous simplifie la vie, vous pouvez considérer que les clés implémentent l'interface `Comparable`.
2. Pensez à bien respecter la convention des `Collections` en Java.