U.B.S. - I.U.T. de Vannes
Département Informatique

R3.02 – TP4 – Arbre binaire

J-F. Kamp

Septembre 2023

Quatrième TP d'exercices sur la manipulation de structures assez classiques (listes, arbres) enseignés en seconde année de BUT informatique. Les exercices sont à réaliser en Java, ils abordent également les notions de contrat et de généricité.

Mise en pratique des notions vues en cours

TP4 : Arbre binaire ordonné et type générique

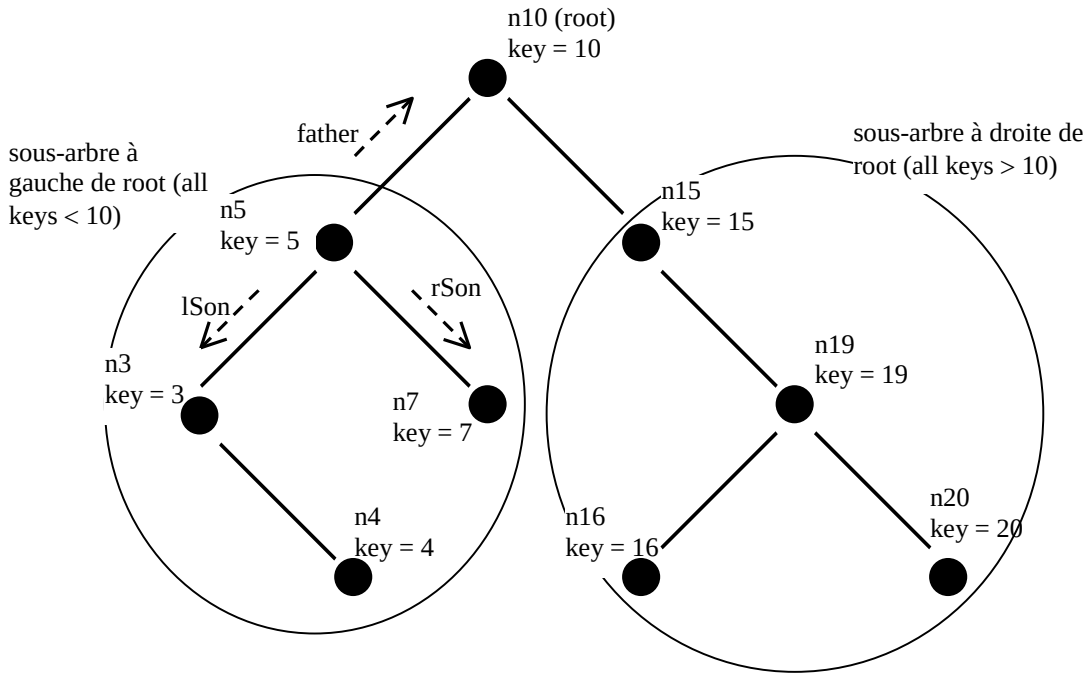
1. Objectifs

Construction de la classe *BinaryTreeTable.java* (paquetage *datastruct*) sous Eclipse qui implémente une structure de données en arbre binaire ordonné. On créera d'abord une interface *Table.java* et ensuite la classe *BinaryTreeTable.java* qui implémentera cette interface.

Pour ce TP, on demande de créer une interface et une classe avec type générique. Ce type paramétré (générique) *E* définira le type de la clé d'identification de chacun des nœuds de l'arbre et on définira aussi un type paramétré *T* pour la donnée stockée par le nœud. Pour le test, faire usage obligatoirement de *JUnit4*.

2. Rappels sur les arbres binaires ordonnés

La figure ci-dessous présente un exemple d'arbre binaire ordonné :



L'arbre binaire ordonné (verticalement) que nous étudions possède les caractéristiques suivantes :

- Chaque nœud possède une clé de recherche unique (cf. Table en BDD).
- Toutes les clés sont de même type (type paramétré recevant, par exemple, la valeur *Integer* sur le schéma ci-dessus) et doivent être ordonnées entre elles. Le type paramétré *E* doit dès lors forcément être de type *Comparable*.
- Chaque nœud stocke une donnée (semblable à un tuple d'une Table en BDD).
- Tous les nœuds sont reliés entre eux pour former un arbre.
- Chaque nœud possède, au plus, deux fils et un père : le fils de droite (right son, *rSon*), le fils de gauche (left son, *lSon*), et un père (*father*).
- Un nœud qui ne possède aucun fils (ni droit, ni gauche) est appelé feuille de l'arbre.
- Le seul et unique nœud de l'arbre qui ne possède pas de père s'appelle nœud racine (*root*) et se situe au sommet de l'arbre (voir schéma).

- Dans le cas d'un arbre binaire (2 fils au plus) ordonné (verticalement) comme montré ci-dessus, l'organisation des nœuds de l'arbre respecte obligatoirement le principe suivant : pour un nœud donné de clé égale à *k*, toutes les clés du sous-arbre de gauche ont une valeur nécessairement inférieure à *k* et, toutes les clés du sous-arbre de droite ont une valeur nécessairement supérieure à *k*.

Le but de ce TP est de coder et tester correctement sous Eclipse la classe *BinaryTreeTable.java* qui devra :

- implémenter l'interface *Table* qui contient exactement 3 méthodes publiques d'opérations classiques sur une table : *select(…)*, *insert(…)* et *delete(…)*,
- contenir 1 classe interne : *Node*,
- déclarer 1 attribut *Node root* : nœud racine, sommet de l'arbre,
- déclarer 1 attribut *boolean balance* : booléen qui décide (ou pas) de déclencher une phase de rééquilibrage de l'arbre après une insertion. Ce booléen peut être modifié par l'utilisateur de la classe par l'intermédiaire du modificateur *public void setBalance (boolean bal)*,
- posséder un accesseur *public Node getTheRoot()* qui renvoie simplement la référence sur l'attribut *root* de votre classe.

3. L'interface Table

L'interface *Table.java* du paquetage *datastruct* doit déclarer dans sa signature les types génériques *E* et *T*, le type *E* étant forcément sous-classe de *Comparable* (clé d'un nœud de l'arbre ordonné). Cette interface *Table* de signature `public interface Table<E extends Comparable<E>, T> {…}` contient la signature des 3 méthodes suivantes :

- *public T select (E key) ;*

  Cette méthode renvoie la donnée contenue dans le nœud correspondant à la clé de recherche *key* passée en paramètre. Renvoie *null* si la clé n'existe pas.

- *public boolean insert (E key, T data) ;*

  Si la clé n'existe pas déjà dans la table, cette méthode insert au bon endroit dans l'arbre un nouveau nœud dont la clé (*key*) et la donnée (*data*) sont passées en paramètres. Renvoie faux si l'insertion n'est pas possible.

- *public boolean delete (E key) ;*

  Détruit de l'arbre binaire le nœud correspondant à la clé passée en paramètre. Renvoie faux si la destruction n'est pas possible (i.e. la clé n'existe pas).

4. La classe BinaryTreeTable

Cette classe du paquetage *datastruct* implémente l'interface *Table* décrite ci-dessus. A nouveau, la classe doit déclarer dans sa signature les types génériques *E* et *T*, le type *E* étant forcément sous-classe de *Comparable*. La signature de la classe sera alors la suivante :

```
public class BinaryTreeTable<E extends Comparable<E>, T> implements Table<E, T> {…}
```

Le constructeur de la classe construit un arbre vide (*root* = *null*) et initialise le booléen *balance* à faux.

On trouvera ensuite une classe interne **publique** *Node* qui définira un nœud comme suit :

```
public class Node {

        // Attributs
        private Node lSon ;          // fils gauche (null si pas de fils gauche)
        private Node rSon ;          // fils droit (null si pas de fils droit)
```

```
        private Node father ;          // père (null si le nœud est root)
        private T theValue ;           // donnée stockée
        private E key ;                // clé unique

        // Constructeur
        public Node (…) {…}

        // Accesseurs
        public String getLabel() ;     // accesseur de la clé
        public Node getLeft() ;        // accesseur du fils gauche
        public Node getRight() ;       // accesseur du fils droit

        // Duplication
        public Node clone() ;          // duplication mémoire du nœud courant
}
```

5. L'affichage textuelle et graphique d'un arbre

Une première façon d'afficher un arbre et son contenu est l'affichage textuel. Pour ce faire, coder la méthode *public String toString()* qui renvoie une chaîne de caractères contenant toutes les informations contenues dans l'arbre binaire c'est-à-dire la clé et la donnée de chacun des nœuds (voir paragraphe 7).

Une deuxième manière d'afficher un arbre et son contenu et d'utiliser deux classes *TreeDraw.java* et *TreeDrawing.java* (paquetage *ihm*) permettant de faire l'affichage dans une fenêtre graphique. Ces 2 classes vous sont fournies. Le passage en paramètre d'un arbre dans l'interface graphique se faisant par référence, chaque fenêtre affichera un arbre identique même si cet arbre a été modifié. C'est pourquoi il faut passer en paramètre la copie de l'arbre pour avoir l'affichage de l'arbre actuel. Dès lors on rajoutera dans *BinaryTreeTable* deux méthodes publiques :

- *public BinaryTreeTable<E, T> clone()* qui renvoie une copie (duplication mémoire) exacte de l'arbre binaire courant. Cette méthode appellera une méthode privée récursive *void computeClone(Node nodeToCopy, Node newFather)* qui construit le nouvel arbre en copiant récursivement tous les nœuds de l'arbre original. Le paramètre *nodeToCopy* est le noeud à recopier et à raccorder à son père et le paramètre *newFather* est le père auquel doit être raccordé le nouveau nœud recopié.
- *public void showTree()* qui affiche dans une fenêtre graphique tous les nœuds de l'arbre binaire de même que les branches entre les nœuds. Cette méthode doit simplement instancier un objet de type *TreeDraw* et ensuite appeler la méthode *drawTree()* sur cet objet (voir les classes interface graphique *TreeDraw* et *TreeDrawing* qui vous sont fournies).

D'autres méthodes privées dans la classe *BinaryTreeTable* seront à développer.

6. L'algorithme d'insertion

La méthode *insert (E key, T data)* doit insérer un nouveau nœud au bon endroit dans l'arbre binaire.

Le nouveau nœud à insérer (*newN*) doit se raccrocher à un nœud père *nodeFather* de l'arbre qui ne possède pas déjà 2 fils (c'est soit une feuille, soit un nœud avec un seul fils). L'opération se déroule en 2 étapes :

1. Opération de recherche du père *nodeFather* à l'aide de la méthode privée *private Node seekFather (E key)* : à partir de *root* et par comparaison de *key* (la clé du nouveau nœud *newN* à insérer) avec la clé de chaque nœud rencontré, descendre dans l'arbre jusqu'à l'impasse c'est-à-dire jusqu'à obtenir *theNode.rSon == null* ou *theNode.lSon == null*. Le nœud *theNode* devient le père recherché. Exemple sur le schéma ci-dessus : le nouveau nœud n17 de clé = 17 aura nécessairement comme père le nœud n16 de clé = 16.
2. Raccrocher le nouveau nœud *newN* au père *nodeFather*. Dans l'exemple du nœud n17 de clé = 17 à raccrocher au nœud n16, comme 17 est + grand que 16, n17 devient le fils droit de n16.

7. L'algorithme de parcours complet d'un arbre binaire par ordre croissant des clés

La méthode *toString()* nécessite de devoir parcourir l'ensemble des nœuds de l'arbre (sans en oublier un seul).

Suivre le principe suivant : partant de la racine (*tmp = root*), descendre le + profondément possible à gauche (*tmp = tmp.lSon*) dans l'arbre binaire. Dès qu'il n'y a plus de fils à gauche (*tmp.lSon == null*) on récupère les informations (clé + donnée) concernant le nœud *tmp* (pour les ajouter à la chaîne de caractères en cours de construction) et on recommence l'opération sur le sous-arbre à droite de *tmp* (*tmp = tmp.rSon*). On montre que de cette manière, on parcourt forcément tout l'arbre binaire suivant l'ordre croissant des clés.

L'algorithme récursif s'écrit (bien le comprendre avant de le coder) :

```
Initialisation de l'appel récursif dans la méthode toString() :
String ret = this.getInfo(this.root) ;

private String getInfo ( Node theN ) {
 String infosLNode, infosRNode, infosNode ;
 String ret ;

 if ( theN != null ) {
       infosLNode = getInfo ( theN.lSon ) ;
       infosRNode = getInfo ( theN.rSon ) ;
       infosNode = new String ( "\nclé=" + theN.key.toString() + "\tdata=" +
theN.theValue.toString() ) ;
       ret = new String ( infosLNode + infosNode + infosRNode ) ;
 }

 else ret = new String ("") ;

 return ret ;
}
```

8. L'algorithme de recherche

La méthode *select (E key)* doit effectuer une recherche de nœud dans l'arbre binaire sur base d'une clé de recherche *key*. Cette méthode appelle obligatoirement la méthode privée récursive *Node findNode (Node theNode, E key)* qui renvoie le nœud correspondant à la clé de recherche et non pas la donnée stockée dans le nœud.

Le principe de la méthode récursive est le suivant :

Initialisation : on part du sommet de l'arbre (*tmp = root*)

Si il y a égalité des clés alors « trouvé » et on arrête la recherche : le nœud *theNode* est renvoyé.

Sinon Si (*key < tmp.key*) Alors se positionner sur le fils gauche (*tmp = tmp.lSon*)

Sinon se positionner sur le fils droit (*tmp = tmp.rSon*)

9. L'algorithme de suppression

La méthode *public boolean delete (E key)* doit supprimer de l'arbre le nœud identifié par la clé *key*.

1. La première étape consiste à rechercher d'abord le nœud (*nodeToDel*) à supprimer avec la méthode récursive privée *Node findNode (Node theNode, E key)*.
2. Une fois *nodeToDel* identifié, il faut le supprimer de l'arbre (méthode privée *private void delete (Node theNode)*). Trois cas sont possibles :

2.1. *nodeToDel* est une feuille de l'arbre (aucun fils, le nœud n4 dans l'exemple ci-dessus) : cas trivial, il suffit de couper correctement toutes les références sur *nodeToDel*.

2.2. *nodeToDel* n'a qu'un seul fils (le nœud n15 dans l'exemple ci-dessus) qu'on appellera *nodeToDelSon* (nœud n19 dans l'exemple) : cas relativement simple car il suffit de raccorder correctement *nodeToDelSon* au père de *nodeToDel* (nœud n10 dans l'exemple) et de couper toutes les références sur *nodeToDel*.

2.3. *nodeToDel* a deux fils (le nœud n5 par exemple). Le principe est le suivant : on va rechercher, dans le sous-arbre gauche de *nodeToDel*, le nœud *theGNode* ayant la clé la + grande (il s'agit du nœud n4 dans l'exemple). On remplace ensuite *nodeToDel* par ce nouveau nœud *theGNode* (en remplaçant simplement la donnée et la clé de *nodeToDel* par la donnée et la clé de *theGNode* sans modifier les références de *nodeToDel* vers ses fils et son père). Cette opération de remplacement de *nodeToDel* par *theGNode* respecte bien le principe d'arbre ordonné car *theGNode* aura forcément une clé de valeur supérieure à toutes les clés de son sous-arbre gauche. Le nœud *nodeToDel* ayant maintenant les bonnes valeurs (donnée et clé), il faut maintenant s'occuper de la suppression du nœud *theGNode* => appel récursif de *private void delete (Node theNode)* avec comme nouveau paramètre *theGNode*.

10. Rééquilibrage d'un arbre déséquilibré

10.1. Arbre équilibré et hauteur d'un noeud

Un arbre binaire est dit « équilibré » si en chacun de ses nœuds, la différence entre la hauteur du sous-arbre gauche et la hauteur du sous-arbre droit est au plus de 1 (ou -1). Cet arbre est alors appelé « arbre AVL » et les opérations d'insertion, de recherche et de suppression deviennent particulièrement efficaces.

Par définition, la hauteur *h* d'un nœud *n* est la longueur du plus long chemin de ce nœud aux feuilles qui dépendent de ce nœud *n*. Cette longueur *h* s'exprime en nombre de nœuds pour aller de *n* jusqu'au nœud feuille y compris *n* et le nœud feuille.

Cas particuliers : un arbre vide a une hauteur zéro, un arbre réduit à sa racine a une hauteur 1.

Exemples (voir schéma page 1) : la hauteur du nœud n5 est égale à 3 (n5, n3, n4), la hauteur du nœud n7 est égale à 1 (n7), la hauteur du nœud n15 est égale à 3. En n10 (*root*), il y a équilibre car *h(n5)-h(n15) = 0*, en n15, il y a déséquilibre car *h(n19)-h(null) = 2*. Donc l'arbre de la page 1 n'est pas un arbre AVL.

On demande d'écrire la méthode *private int computeH (Node theN)* de la classe *BinaryTreeTable* qui renvoie la hauteur du nœud passé en paramètre (cas particulier : si *theN* est *null* alors sa hauteur vaut zéro).

Il est fortement recommandé de résoudre le problème par récursivité : en effet, la hauteur d'un nœud est égale à la plus grande des hauteurs de ses fils gauche et droit plus 1.

H(node) = max (H(node.lSon), H(node.rSon)) + 1

De plus, rajouter dans la classe la méthode **publique** *public boolean isAVL()* qui permet à l'utilisateur de savoir (retour *true/false*) si son arbre est AVL. Pour ce faire, l'usage de la méthode *computeH (...)* ci-dessus est indispensable.

10.2. Rééquilibrage d'un arbre binaire

La méthode *public void setBalance (boolean bal)*, modificateur à coder, permet simplement de modifier l'attribut *boolean balance* de la classe. Cet attribut est important car c'est en se basant sur sa valeur que l'on décidera, après une insertion, de rééquilibrer (ou pas) l'arbre binaire en appelant la méthode *void balanceTheTree (Node theN)* (voir plus loin).

D'abord lire attentivement le cours sur le rééquilibrage. Dans ce cours on vous explique en résumé que :

- le rééquilibrage, si nécessaire, se fait après chaque insertion d'un nœud dans (de) l'arbre binaire,
- le rééquilibrage consiste à d'abord identifier le nœud déséquilibré (*theN*). Un nœud déséquilibré est tel que la différence entre la hauteur de son sous-arbre gauche et la hauteur de son sous-arbre droit est supérieur ou égale à 2 (ou -2),
- si le sous-arbre « lourd » se trouve à gauche du noeud *theN* alors l'équilibre en *theN* est rétabli soit par rotation droite, soit par double rotation : gauche puis droite,
- si le sous-arbre « lourd » se trouve à droite du noeud *theN* alors l'équilibre en *theN* est rétabli soit par rotation gauche, soit par double rotation : droite puis gauche.

10.3. Les méthodes

Ecrire les méthodes privées qui permettent d'effectuer des rotations autour d'un nœud :

```
• private void rightRotation ( Node theN );     // rotation droite autour du noeud theN

• private void leftRotation ( Node theN );      // rotation gauche autour du noeud theN

• private void leftRightRotation (Node theN); // rotation gauche puis droite autour du
                                               // noeud theN
• private void rightLeftRotation (Node theN); // rotation droite puis gauche autour du
                                               // noeud theN
```

Pour finir, écrire la méthode privée *private void balanceTheTree (Node theN)* qui effectue un rééquilibrage de l'arbre après insertion d'un nœud dans l'arbre. Cette méthode doit donc d'abord identifier le nœud déséquilibré et ensuite procéder au rééquilibrage de l'arbre par rapport à ce noeud.

Attention : pour cette dernière méthode, le paramètre *theN* n'est PAS le nœud déséquilibré (celui-là il faut le trouver !) mais le nouveau nœud inséré.

Faire appel à cette méthode *void balanceTheTree(...)* dans la méthode d'insertion (*insert*) à condition que le booléen *balance* soit à vrai (sinon, pas de rééquilibrage).