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

R3.02 – TP4 – Arbre binaire

J-F. Kamp

Octobre 2024

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

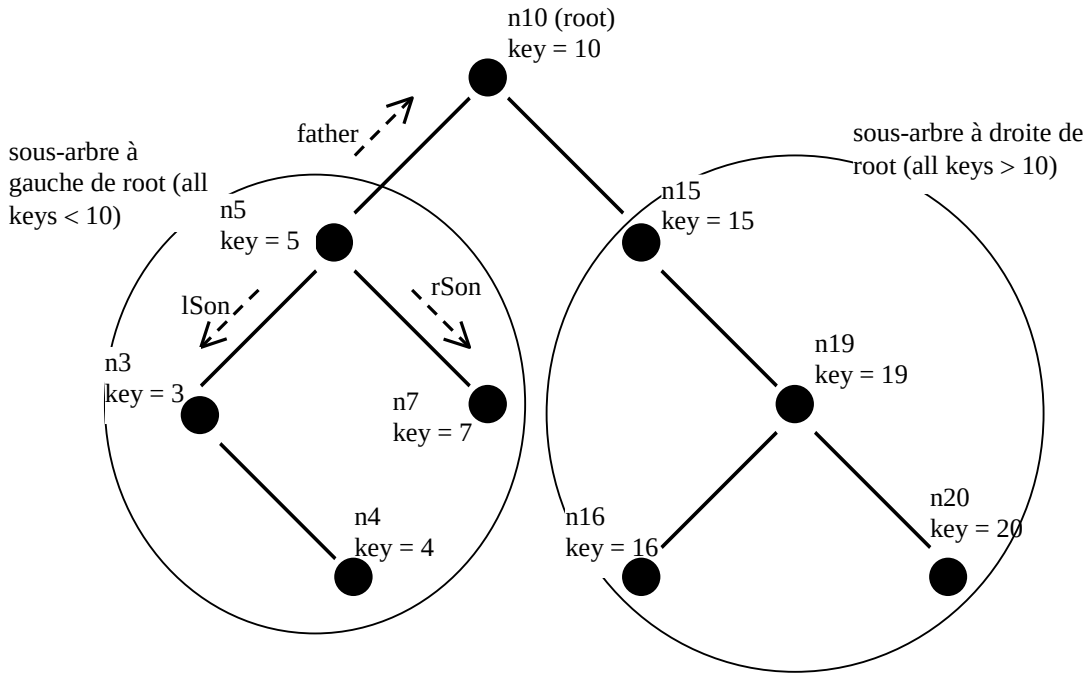
1. Objectifs

Construction de la classe *ArbreBinaire.java* (paquetage *structdonnees*) sous Eclipse qui implémente une structure de données en arbre binaire ordonné. On créera d'abord une interface *Table.java* et ensuite la classe *ArbreBinaire.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 (classe *TestArbreBinaire* qui appartiendra au paquetage *structdonnees*), 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 (*filsD*), le fils de gauche (*filsG*) et un père (*pere*).
- 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 (*racine*) 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 *ArbreBinaire.java* qui devra :

- implémenter l'interface *Table* qui contient exactement 3 méthodes publiques d'opérations classiques sur une table : *obtenir(…)*, *inserer(…)* et *supprimer(…)*,
- contenir 1 classe interne : *Noeud*,
- déclarer 1 attribut *Noeud racine* : sommet de l'arbre.

3. L'interface Table

L'interface *Table.java* du paquetage *structdonnees* 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 obtenir (E cle) ;*

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

- *public boolean inserer (E cle, T donnee) ;*

  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 (*cle*) et la donnée (*donnee*) sont passées en paramètres. Renvoie faux si l'insertion n'est pas possible.

- *public boolean supprimer (E cle) ;*

  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 ArbreBinaire

Cette classe du paquetage *structdonnees* 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 ArbreBinaire<E extends Comparable<E>, T> implements Table<E, T> {…}
```

Le constructeur de la classe construit un arbre vide (*racine* = *null*).

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

```
public class Noeud {

        // Attributs
        private Noeud filsG ;          // fils gauche (null si pas de fils gauche)
        private Noeud filsD ;          // fils droit (null si pas de fils droit)
        private Noeud pere ;           // père (null si le nœud est racine)
        private T donnee ;             // donnée stockée
        private E cle ;                // clé unique
```

```
        // Constructeur
        public Noeud (…) {…}

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

        // Duplication
        public Noeud cloner() ;       // 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 **en tout premier lieu** 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 pour le codage).

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 *ArbreBinaire* deux méthodes publiques :

- *public ArbreBinaire<E, T> cloner()* qui renvoie une copie (duplication mémoire) exacte de l'arbre binaire courant. Cette méthode appellera une méthode privée récursive *void fabriquerClone(Noeud noeudACopier, Noeud nouveauPere)* qui construit le nouvel arbre en copiant <u>récursivement</u> tous les nœuds de l'arbre original. Le paramètre *noeudACopier* est le noeud à recopier et à raccorder à son père et le paramètre *nouveauPere* est le père auquel doit être raccordé le nouveau nœud recopié.
- *public void afficherArbre()* 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 <u>privées</u> dans la classe *ArbreBinaire* seront à développer.

6. L'algorithme d'insertion

La méthode *inserer (E cle, T donnee)* doit insérer un nouveau nœud au bon endroit dans l'arbre binaire.

Le nouveau nœud à insérer (*nouvNoeud*) doit se raccrocher à un nœud père *noeudPere* 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 *noeudPere* à l'aide de la méthode privée *private Noeud chercherPere (E cle)* : à partir de *racine* et par comparaison de *cle* (la clé du nouveau nœud *nouvNoeud* à insérer) avec la clé de chaque nœud rencontré, descendre dans l'arbre jusqu'à l'impasse c'est-à-dire jusqu'à obtenir *leNoeud.filsD == null* ou *leNoeud.filsG == null*. Le nœud *leNoeud* 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 *nouvNoeud* au père *noeudPere*. 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 = racine*), descendre le + profondément possible à gauche (*tmp = tmp.filsG*) dans l'arbre binaire. Dès qu'il n'y a plus de fils à gauche (*tmp.filsG == 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.filsD*). 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.obtenirInfo(this.racine) ;

private String obtenirInfo ( Noeud leNoeud ) {
 String infosNoeudG, infosNoeudD, infosNoeud ;
 String ret ;

 if ( leNoeud != null ) {
       infosNoeudG = obtenirInfo ( leNoeud.filsG ) ;
       infosNoeudD = obtenirInfo ( leNoeud.filsD ) ;
       infosNoeud = new String ( "\nclé=" + leNoeud.cle.toString() + "\tdonnée=" +
leNoeud.donnee.toString() ) ;
       ret = new String ( infosNoeudG + infosNoeud + infosNoeudD ) ;
 }

 else ret = new String ("") ;

 return ret ;
}
```

8. L'algorithme de recherche

La méthode *obtenir (E cle)* doit effectuer une recherche de nœud dans l'arbre binaire sur base d'une clé de recherche *cle*. Cette méthode appelle obligatoirement la méthode privée récursive *Noeud trouverNoeud (Noeud leNoeud, E cle)* 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 = racine*)

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

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

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

9. L'algorithme de suppression

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

1. La première étape consiste à rechercher d'abord le nœud (*noeudASup*) à supprimer avec la méthode récursive privée *Noeud trouverNoeud (Noeud leNoeud, E cle)*, méthode déjà codée,
2. Une fois *noeudASup* identifié, il faut le supprimer de l'arbre (méthode privée *private void supprimer (Noeud leNoeud)*). Trois cas sont possibles :

2.1. *noeudASup* 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 *noeudASup*.

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

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

10. 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 (*racine*), 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 calculerH (Noeud leN)* de la classe *ArbreBinaire* qui renvoie la hauteur du nœud passé en paramètre (cas particulier : si *leN* 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(noeud) = max (H(noeud.filsG), H(noeud.filsD)) + 1

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