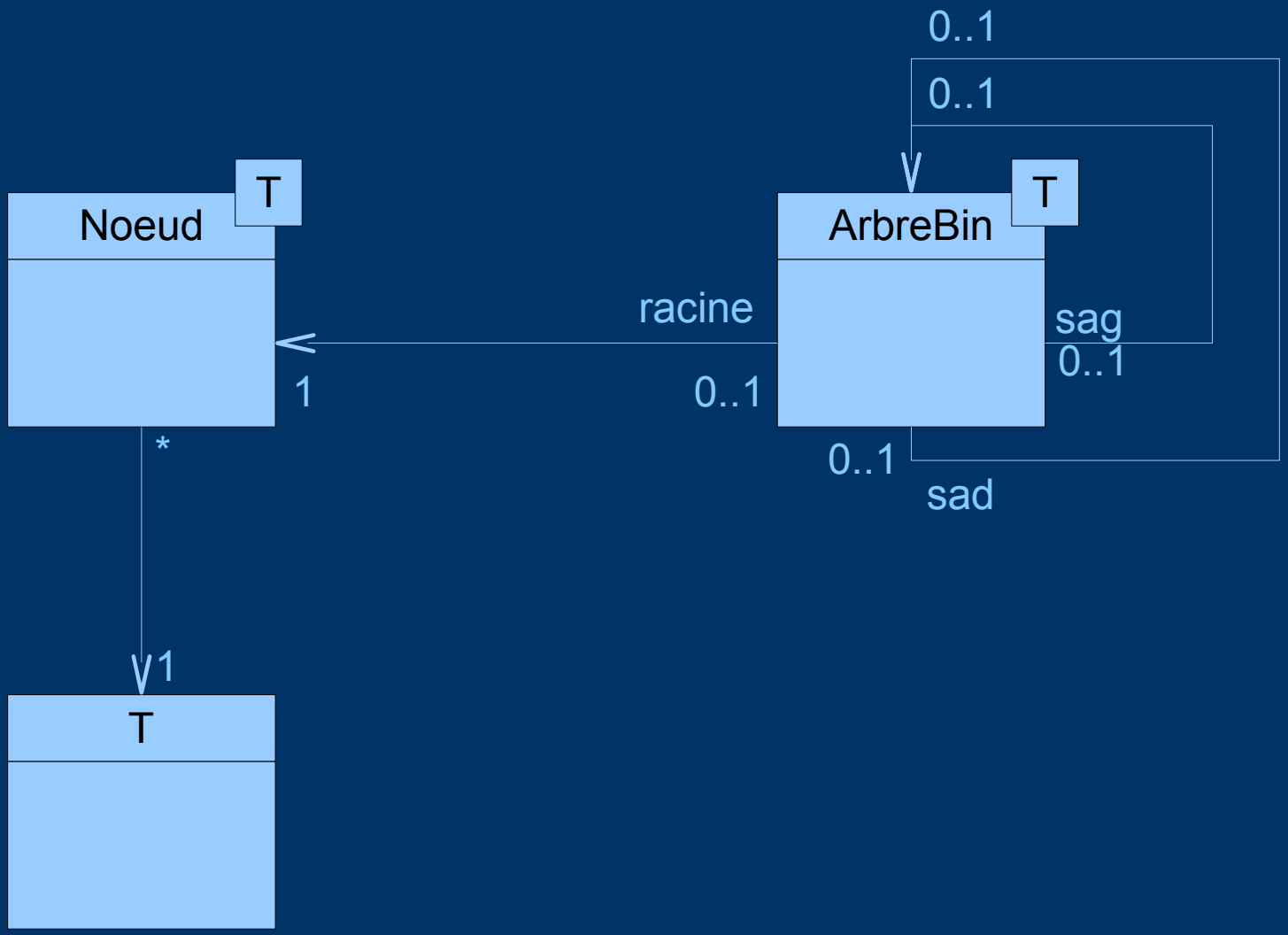
Refactoring guidé par les tests
Application à la classe ArbreBin

Bruno Mermet
2010

Contexte

- Une implantation d'arbre binaire a été définie par Gaële Simon pour les étudiants de 2ème année de l'IUT du Havre
- Dans le contexte particulier du cours de programmation objet en question, la structure choisie se justifie.
- Si l'on souhaite rendre cette classe utilisable dans un contexte externe
  - Des choix différents doivent être faits
  - Certaines parties du code sont à réécrire
- P.S. : j'ai quelques fois « sali » un peu le code initial pour mieux illustrer le propos

Structure actuelle



Code actuel (1)

```
package correction1;

/*Classe ArbreBinBase (G. Simon - octobre 2006) */

/* Classe générique permettant de manipuler un arbre binaire. Cette classe utilise la classe Noeud permettant de
représenter un noeud de l'arbre. Cette classe donne les éléments de base pour représenter et parcourir un arbre
binaire. Elle ne comporte pas de méthode permettant de calculer des propriétés de l'arbre; */

import enonce.Noeud;
import java.util.Vector;

public class ArbreBinBase<T> {

private Noeud<T> racine;          /* racine de l'arbre */
private ArbreBinBase<T> sag;    /* sous-arbre gauche de la racine */
private ArbreBinBase<T> sad;    /* sous-arbre droit de la racine */

public enum Position {gauche, droit}; // permet de distinguer les deux positions possibles d'un noeud fils

/* Constructeurs */
public ArbreBinBase(T val) {
    racine = new Noeud<T>(val);
}

public ArbreBinBase(T val, ArbreBinBase<T> fg, ArbreBinBase<T> fd) {
racine = new Noeud<T>(val);
sag = fg;
sad = fd;
}
```

Code actuel (2)

```
/* Accesseurs */

public Noeud<T> getRacine() {return racine;}
public T getDonneeRacine() {return racine.getDonnee();}

public ArbreBinBase<T> getSAG() {return sag;}
public ArbreBinBase<T> getSAD() {return sad;}

/*
 * Accesseurs en écriture permettant d'ajouter un sous-arbre au noeud racine
 * Attention : s'il en existe déjà un à la même position, il est "écrasé" !
 */
public void setSAG(ArbreBinBase<T> fg) {sag = fg;}
public void setSAD(ArbreBinBase<T> fd) {sad = fd;}

/*
 * Accesseurs en écriture permettant d'ajouter un fils au noeud racine.
 * Attention : s'il en existe déjà un à la même position, il est écrasé.
 */
public void setFilsg(T val) {sag = new ArbreBinBase<T>(val);}
public void setFilsd(T val) {sad = new ArbreBinBase<T>(val);}
```

Code actuel (3)

```
/* rend en résultat le sous-arbre dont le noeud d'identifiant idNoeud est racine si le noeud existe dans l'arbre, rend null sinon.
Méthode utilisée dans la première méthode d'ajout. */
public ArbreBinBase<T> rechercheId(int idNoeud) {
    ArbreBinBase<T> sa;

    if (idNoeud == racine.getId()) {
        return this;
    }
    else {
        sa = null;
        if (sag != null) {sa = sag.rechercheId(idNoeud);}
        if ((sa == null) && (sad != null)) { sa = sad.rechercheId(idNoeud);}
        return sa;
}}

/* Recherche la première occurence d'une donnée dans l'arbre. Le résultat est le sous-arbre dont le noeud racine contient cette
donnée. Le parcours est de type en profondeur d'abord gauche/droite. Si la donnée n'existe pas dans l'arbre, le résultat est null.
Méthode utilisée notamment dans getIdDonnee. */
public ArbreBinBase<T> rechercheDonnee(T donnee) {
    ArbreBinBase<T> sa;

    if (donnee.equals(racine.getDonnee())) {
        return this;
    }
    else {
        sa = null;
        if (sag != null) {sa = sag.rechercheDonnee(donnee);}
        if ((sa == null) && (sad != null)) { sa = sad.rechercheDonnee(donnee);}
        return sa;
 }}
```

Code actuel (4)

```
/* Facultatif */
/* Peut-être utilisé dans le main pour ne pas avoir à calculer les identifiants des différents noeuds. Au lieu d'appeler ajout(5,2,Position.gauche) pour ajouter la donnée 2 comme fils gauche du noeud d'identifiant 5 dans l'arbre ab, on peut appeler : ajout(ab.getIdDonnée(3),2,Position.gauche) si 5 est l'identifiant d'un noeud dont la valeur est 3. */
public int getIdDonnee(T donnee) {
    ArbreBinBase<T> sa;

    sa = rechercheDonnee(donnee);
    if (sa == null) {return -1;}
    else {return sa.getRacine().getId();}
}

/* Dans cette méthode d'ajout, on ajoute un sous-arbre a un noeud spécifié par son identifiant. La position du sous-arbre est passée en troisième paramètre. On ne vérifie pas s'il existe déja un sous-arbre à la position indiquee. Si oui, il est écrasé. Attention : la méthode peut cependant retourner false et donc l'ajout ne pas se faire si l'identifiant du noeud père n'est pas trouvé dans l'arbre. */
public boolean ajout(int idPere, ArbreBinBase<T> sa, Position p) {
    ArbreBinBase<T> sap;

    sap = rechercheId(idPere);
    if (sap != null) {
        /* on fait l'ajout */
        if (p == Position.gauche) {sap.setSAG(sa);}
        else {sap.setSAD(sa);}
        return true;
    }
    else { return false; }
}
```

Code actuel (5)

```
public int ajout2(int idPere, T donnee, Position p) throws Exception {
        ArbreBinBase<T> sap;
        int ret;

        if (racine.getId() == idPere) {
            ArbreBinBase ab = new ArbreBinBase(donnee);
            if (p == Position.gauche) {sag = ab;}
            else {sad = ab;}
            return ab.racine.getId();
        }
        try {
            if (sag != null) {
                ret = sag.ajout2(idPere, donnee, p);
                return ret;
            }
        } catch (Exception e) {}
        try {
            if (sad != null) {
                    ret = sad.ajout2(idPere, donnee, p);
                    return ret;
            }
        } catch (Exception e) {}
        throw new Exception();
    }

    /* Similaire à la méthode précédente sauf qu'ici on ajoute un fils et non plus un sous-arbre. */
public boolean ajout(int idPere, T fils, Position p) {
   return (ajout(idPere,new ArbreBinBase<T>(fils),p));
}
```

Code actuel (6)

```
/*Affichage "textuel" du contenu de l'arbre. Le paramètre esp est une chaîne de caractères utilisée comme
séparateur des différents éléments de l'arbre lors de l'affichage. */
private void affiche(String esp) {
    System.out.println(esp+racine);
    if (sag != null) {
        sag.affiche(esp+"  ");
    }
    if (sad != null) {
        sad.affiche(esp+"  ");
    }
}

/*Affichage "textuel" du contenu de l'arbre. Le séparateur des différents éléments de l'arbre utilisé lors de
l'affichage est une chaîne vide. */
public void affiche() {
    System.out.println("----");
    affiche("");
}
```

Code actuel (7)

```
/* Méthodes mettant en oeuvre les 3 types de parcours en profondeur d'abord d'un arbre. Elles renvoient un chaîne de caractères contenant la liste des données des noeuds dans l'ordre correspondant au parcours. */
public String parcoursInfixe() {
    String s1,s2;
    if (sag != null) {s1 = sag.parcoursInfixe();} else {s1 = "";}
    if (sad != null) {s2 = sad.parcoursInfixe();} else {s2 = "";}
    return s1+racine.toString()+s2;
}

public String parcoursPostfixe() {
    String s1,s2;
    if (sag != null) {s1 = sag.parcoursPostfixe();} else {s1 = "";}
    if (sad != null) {s2 = sad.parcoursPostfixe();} else {s2 = "";}
    return s1+s2+racine.toString();
}

public String parcoursPrefixe() {
    String s1,s2;
    if (sag != null) {s1 = sag.parcoursPrefixe();} else {s1 = "";}
    if (sad != null) {s2 = sad.parcoursPrefixe();} else {s2 = "";}
    return racine.toString()+s1+s2;
}
```

Code actuel (8)

```
/* Même principe que pour les méthodes précédentes mais pour le parcours en largeur. La file que nécessite la
mise en oeuvre de ce parcours est mise en oeuvre par un vecteur de sous-arbres binaires. */
public String largeur() {
    Vector<T> lues = new Vector<T>(); /* pour stocker le résultat du parcours */
    Vector<ArbreBinBase<T>> aTraiter = new Vector<ArbreBinBase<T>>(); /* la file */
    ArbreBinBase<T> courant;

    lues.add(racine.getDonnee());
    aTraiter.add(this);

    while (aTraiter.size() != 0) {
        courant = aTraiter.get(0);
        if (courant.sag!= null) {
            lues.add(courant.sag.getRacine().getDonnee());
            aTraiter.add(courant.sag);
        }
        if (courant.sad != null) {
            lues.add(courant.sad.getRacine().getDonnee());
            aTraiter.add(courant.sad);
        }
        aTraiter.removeElementAt(0);
    }
    return lues.toString();
}
```

Code actuel (9)

```
/** la taille d'un arbre est le nombre de noeuds de l'arbre */
    public int taille() {
        int tailleSAG = 0;
        int tailleSAD = 0;

        if (sag != null) {
            tailleSAG = sag.taille();
        }
        if (sad != null) {
            tailleSAD = sad.taille();
        }
        return 1+tailleSAG+tailleSAD;
    }

    public int hauteur() {
        int hauteurSAG = -1;
        int hauteurSAD = -1;

        if (sag != null) {
            hauteurSAG = sag.hauteur();
        }
        if (sad != null) {
            hauteurSAD = sad.hauteur();
        }
        return 1+Math.max(hauteurSAG, hauteurSAD);
    }
```

```
    public int nbFeuilles() {
        int feuillesSAG = 0;
        int feuillesSAD = 0;

        if (isFeuille()) {
            return 1;
        }
        if (sag != null) {
            feuillesSAG = sag.nbFeuilles();
        }
        if (sad != null) {
            feuillesSAD = sad.nbFeuilles();
        }
        return feuillesSAG+feuillesSAD;
    }

    private boolean isFeuille() {
        return sag == null && sad == null;
    }
```

Code actuel (10)

```
public static void main(String[] args) {

/* l'instruction suivante utilise l'auto-boxing */
ArbreBinBase<Integer> ai = new ArbreBinBase<Integer>(1); /* la racine de l'arbre est 1. son identifiant est 1. */

/* création d'une  partie de l'arbre du cours */

ai.ajout(1,4,Position.gauche);
ai.ajout(1,5,Position.droit);
ai.ajout(2,6,Position.gauche); /* l'identifiant de 4 est 2 */
ai.ajout(2,7,Position.droit);
ai.ajout(3,8,Position.gauche); /* l'identifiant de 5 est 3 */
ai.ajout(6,9,Position.gauche); /* l'identifiant de 8 est 6 */
ai.ajout(6,10,Position.droit); /* l'identifiant de 8 est 6 */

/* création du même arbre en utilisant la méthode getIdDonnee(). attention ! : cette façon de faire n'est pas efficace
du tout car pour calculer l'identifiant de chaque noeud, on reparcourt l'arbre ! */

/* ai.ajout(ai.getIdDonnee(1),4,Position.gauche); ai.ajout(ai.getIdDonnee(1),5,Position.droit);
 * ai.ajout(ai.getIdDonnee(4),6,Position.gauche); ai.ajout(ai.getIdDonnee(4),7,Position.droit);
 * ai.ajout(ai.getIdDonnee(5),8,Position.gauche); ai.ajout(ai.getIdDonnee(8),9,Position.gauche);
 * ai.ajout(ai.getIdDonnee(8),10,Position.droit); */

ai.affiche();

System.out.println("infixe : "+ai.parcoursInfixe());System.out.println("postfixe : "+ai.parcoursPostfixe());
System.out.println("prefixe : "+ai.parcoursPrefixe());System.out.println("largeur : "+ai.largeur());
// Affichage de taille, nbFeuilles et hauteur
System.out.println("taille : " + ai.taille()); System.out.println("nb de feuilles : " + ai.nbFeuilles());
System.out.println("hauteur : " + ai.hauteur());
}}
```

Etape 1 : épuration

Code actuel (1)

Inutile : pris en charge par svn

```
package correction1;

/*Classe ArbreBinBase (G. Simon - octobre 2006) */

/* Classe générique permettant de manipuler un arbre binaire. Cette classe utilise la classe Noeud permettant de
représenter un noeud de l'arbre. Cette classe donne les éléments de base pour représenter et parcourir un arbre
binaire. Elle ne comporte pas de méthode permettant de calculer des propriétés de l'arbre; */

import enonce.Noeud;
import java.util.Vector;
```

Inutile : paraphrase du code

```
public class ArbreBinBase<T> {

private Noeud<T> racine;          /* racine de l'arbre */
private ArbreBinBase<T> sag;      /* sous-arbre gauche de la racine */
private ArbreBinBase<T> sad;      /* sous-arbre droit de la racine */

public enum Position {gauche, droit}; // permet de distinguer les deux positions possibles d'un noeud fils

/* Constructeurs */
public ArbreBinBase(T val) {
    racine = new Noeud<T>(val);
}

public ArbreBinBase(T val, ArbreBinBase<T> fg, ArbreBinBase<T> fd) {
racine = new Noeud<T>(val);
sag = fg;
sad = fd;
}
```

Code actuel (3)

```
/* rend en résultat le sous-arbre dont le noeud d'identifiant idNoeud est racine si le noeud existe dans l'arbre, rend null sinon.
Méthode utilisée dans la première méthode d'ajout. */
public ArbreBinBase<T> rechercheId(int idNoeud) {
    ArbreBinBase<T> sa;

    if (idNoeud == racine.getId()) {
        return this;
    }
    else {
        sa = null;
        if (sag != null) {sa = sag.rechercheId(idNoeud);}
        if ((sa == null) && (sad != null)) { sa = sad.rechercheId(idNoeud);}
        return sa;
}}
```

Les « else » après un return alourdissent inutilement la structure du code

```
/* Recherche la première occurence d'une donnée dans l'arbre. Le résultat est le sous-arbre dont le noeud racine contient cette
donnée. Le parcours est de type en profondeur d'abord gauche/droite. Si la donnée n'existe pas dans l'arbre, le résultat est null.
Méthode utilisée notamment dans getIdDonnee. */
public ArbreBinBase<T> rechercheDonnee(T donnee) {
    ArbreBinBase<T> sa;

    if (donnee.equals(racine.getDonnee())) {
        return this;
    }
    else {
        sa = null;
        if (sag != null) {sa = sag.rechercheDonnee(donnee);}
        if ((sa == null) && (sad != null)) { sa = sad.rechercheDonnee(donnee);}
        return sa;
 }}
```

Code actuel (4)

```
/* Facultatif */
/* Peut-être utilisé dans le main pour ne pas avoir à calculer les identifiants des différents noeuds. Au lieu d'appeler ajout(5,2,Position.gauche) pour ajouter la donnée 2 comme fils gauche du noeud d'identifiant 5 dans l'arbre ab, on peut appeler : ajout(ab.getIdDonnée(3),2,Position.gauche) si 5 est l'identifiant d'un noeud dont la valeur est 3. */
public int getIdDonnee(T donnee) {
    ArbreBinBase<T> sa;

    sa = rechercheDonnee(donnee);
    if (sa == null) {return -1;}
    else {return sa.getRacine().getId();}
}

/* Dans cette méthode d'ajout, on ajoute un sous-arbre a un noeud spécifié par son identifiant. La position du sous-arbre est passée en troisième paramètre. On ne vérifie pas s'il existe déja un sous-arbre à la position indiquee. Si oui, il est écrasé. Attention : la méthode peut cependant retourner false et donc l'ajout ne pas se faire si l'identifiant du noeud père n'est pas trouvé dans l'arbre. */
public boolean ajout(int idPere, ArbreBinBase<T> sa, Position p) {
    ArbreBinBase<T> sap;

    sap = rechercheId(idPere);
    if (sap != null) {
        /* on fait l'ajout */
        if (p == Position.gauche) {sap.setSAG(sa);}
        else {sap.setSAD(sa);}
        return true;
    }
    else {return false; }
}
```

Code actuel (5)

```
public int ajout2(int idPere, T donnee, Position p) throws Exception {
        ArbreBinBase<T> sap;
        int ret;

        if (racine.getId() == idPere) {
            ArbreBinBase ab = new ArbreBinBase(donnee);
            if (p == Position.gauche) {sag = ab;}
            else {sad = ab;}
            return ab.racine.getId();
        }
        try {
            if (sag != null) {
                ret = sag.ajout2(idPere, donnee, p);
                return ret;
            }
        } catch (Exception e) {}
        try {
            if (sad != null) {
                    ret = sad.ajout2(idPere, donnee, p);
                    return ret;
            }
        } catch (Exception e) {}
        throw new Exception();
    }

    /* Similaire à la méthode précédente sauf qu'ici on ajoute un fils et non plus un sous-arbre. */
public boolean ajout(int idPere, T fils, Position p) {
   return (ajout(idPere,new ArbreBinBase<T>(fils),p));
}
```

Commentaire obsolète : ajout2 a été rajouté entre temps entre la précédente méthode et celle-ci.

Code actuel (10)

```
public static void main(String[] args) {

/* l'instruction suivante utilise l'auto-boxing */
ArbreBinBase<Integer> ai = new ArbreBinBase<Integer>(1); /* la racine de l'arbre est 1. son identifiant est 1. */

/* création d'une  partie de l'arbre du cours */

ai.ajout(1,4,Position.gauche);
ai.ajout(1,5,Position.droit);
ai.ajout(2,6,Position.gauche); /* l'identifiant de 4 est 2 */
ai.ajout(2,7,Position.droit);
ai.ajout(3,8,Position.gauche); /* l'identifiant de 5 est 3 */
ai.ajout(6,9,Position.gauche); /* l'identifiant de 8 est 6 */
ai.ajout(6,10,Position.droit); /* l'identifiant de 8 est 6 */

/* création du même arbre en utilisant la méthode getIdDonnee(). attention ! : cette façon de faire n'est pas efficace
du tout car pour calculer l'identifiant de chaque noeud, on reparcourt l'arbre ! */

/* ai.ajout(ai.getIdDonnee(1),4,Position.gauche); ai.ajout(ai.getIdDonnee(1),5,Position.droit);
 * ai.ajout(ai.getIdDonnee(4),6,Position.gauche); ai.ajout(ai.getIdDonnee(4),7,Position.droit);
 * ai.ajout(ai.getIdDonnee(5),8,Position.gauche); ai.ajout(ai.getIdDonnee(8),9,Position.gauche);
 * ai.ajout(ai.getIdDonnee(8),10,Position.droit); */

ai.affiche();

System.out.println("infixe : "+ai.parcoursInfixe());System.out.println("postfixe : "+ai.parcoursPostfixe());
System.out.println("prefixe : "+ai.parcoursPrefixe());System.out.println("largeur : "+ai.largeur());
// Affichage de taille, nbFeuilles et hauteur
System.out.println("taille : " + ai.taille()); System.out.println("nb de feuilles : " + ai.nbFeuilles());
System.out.println("hauteur : " + ai.hauteur());
}}
```

Etape 2 : application du pattern « Cas Particulier »

Motivation

- Le code est émaillé de nombreux tests sur la « nullité » des sous-arbres gauche ou droit
  - If (sag == null) {…}
  - If (sad != null) {…}
- Ces tests correspondent à des cas particuliers et nuisent à la lisibilité de l'algorithme
- Principe du pattern « cas particulier »
  - Créer une sous-classe Singleton de ArbreBinBase
  - L'instance de cette sous-classe sera mise à la place de « null »
  - Les appels faits sur sag et sad seront transmis à cette instance, qui renverra la valeur « particulière » associée

Classe ArbreBinNull

```
package correction2;

class ArbreBinNull extends ArbreBinBase {
    private static ArbreBinNull instance = null;

    public ArbreBinNull() {}

    public static ArbreBinNull getInstance() {
        if (instance == null) {
            instance = new ArbreBinNull();
        }
        return instance;
    }

    @Override
    public int taille() {return 0;}

    @Override
    public int nbFeuilles() {return 0;}

    @Override
    public int hauteur() {return -1;}

    @Override
    public ArbreBinBase2 rechercheId(int idNoeud) {
        return getInstance();
    }
    @Override
    public ArbreBinBase2 rechercheDonnee(Object donnee) {
        return getInstance();
    }
    @Override
    public String parcoursInfixe() {
        return "";
    }
    @Override
    public String parcoursPrefixe() {
        return "";
    }
    @Override
    public String parcoursPostfixe() {
        return "";
    }
    @Override
    public void affiche(String s) {
    }
    @Override
    public int getIdRacine() {return -1;}
    @Override
    public boolean isNull() {return true;}
}
```

Classe ArbreBin revue (1)

```
package correction2;

import enonce.Noeud;
import java.util.Vector;

public class ArbreBinBase<T> {
private Noeud<T> racine;
private ArbreBinBase2<T> sousArbreGauche;
private ArbreBinBase2<T> sousArbreDroit;

public enum Position {gauche, droit}; /* permet de
distinguer les deux positions possible d'un noeud fils */

public ArbreBinBase(T val) {
    racine = new Noeud<T>(val);
    sousArbreGauche = ArbreBinNull.getInstance();
    sousArbreDroit = ArbreBinNull.getInstance();
}

public ArbreBinBase(T val, ArbreBinBase2<T> fg,
ArbreBinBase2<T> fd) {
    racine = new Noeud<T>(val);
    sousArbreGauche = fg;
    sousArbreDroit = fd;
}
/* constructeur réservé aux sous-classes */
protected ArbreBinBase() {}
```

```
public Noeud<T> getRacine() {return racine;}
public T getDonneeRacine() {return racine.getDonnee();}

public int getIdRacine() {return racine.getId();}

public ArbreBinBase<T> getSAG() {
    return sousArbreGauche;
}
public ArbreBinBase<T> getSAD() {
    return sousArbreDroit;
}

public void setSAG(ArbreBinBase<T> fg){
    sousArbreGauche = fg;
}
public void setSAD(ArbreBinBase<T> fd) {
    sousArbreDroit = fd;
}

public void setFilsg(T val) {
    sousArbreGauche = new ArbreBinBase<T>(val);
}
public void setFilsd(T val) {
    sousArbreDroit = new ArbreBinBase<T>(val);
}
```

Classe ArbreBin revue (2)

```
/* rend en résultat le sous-arbre dont le noeud
d'identifiant idNoeud est racine si le noeud existe dans
l'arbre, rend null sinon. Méthode utilisée dans la première
méthode d'ajout. */
public ArbreBinBase<T> rechercheId(int idNoeud) {
    ArbreBinBase<T> sa;

    if (idNoeud == racine.getId()) {
        return this;
    }
    sa = sousArbreGauche.rechercheId(idNoeud);
    if (sa.isNull()) {
        sa = sousArbreDroit.rechercheId(idNoeud);}
    return sa;
}

/* Recherche la première occurence d'une donnée dans
l'arbre. Le résultat est le sous-arbre dont le noeud racine
contient cette donnée. Le parcours est de type en
profondeur d'abord gauche/droite. Si la donnée n'existe
pas dans l'arbre, le résultat est null. Méthode utilisée
notamment dans getIdDonnee. */
public ArbreBinBase<T> rechercheDonnee(T donnee) {
    ArbreBinBase<T> sa;

    if (donnee.equals(racine.getDonnee())) {
        return this;
    }
    sa = sousArbreGauche.rechercheDonnee(donnee);
    if (sa.isNull()) {
        sa = sousArbreDroit.rechercheDonnee(donnee);
    }
    return sa;
}

/* Facultatif */
/* Peut-être utilisé dans le main pour ne pas avoir à calculer
les identifiants des différents noeuds. Au lieu d'appeler
ajout(5,2,Position.gauche) pour ajouter la donnée 2 comme
fils gauche du noeud d'identifiant 5 dans l'arbre ab, on peut
appeler : ajout(ab.getIdDonnée(3),2,Position.gauche) si 5 est
l'identifiant d'un noeud dont la valeur est 3. */
public int getIdDonnee(T donnee) {
    ArbreBinBase<T> sa;

    sa = rechercheDonnee(donnee);
    return sa.getIdRacine();
}
```

Classe ArbreBin revue (3)

```
/* Dans cette méthode d'ajout, on ajoute un sous-arbre a
un noeud spécifié par son identifiant. La position du
sous-arbre est passée en troisième paramètre. On ne
vérifie pas s'il existe déja un sous-arbre à la position
indiquee. Si oui, il est écrasé. Attention : la méthode peut
cependant retourner false et donc l'ajout ne pas se faire si
l'identifiant du noeud père n'est pas trouvé dans l'arbre.*/
public boolean ajout(int idPere, ArbreBinBase<T> sa,
Position p) {
    ArbreBinBase<T> sap;

    sap = rechercheId(idPere);
    if (! sap.isNull()) {
        /* on fait l'ajout */
        if (p == Position.gauche) {sap.setSAG(sa);}
        else {sap.setSAD(sa);}
        return true;
    }
}

/*Similaire à la méthode précédente sauf qu'ici on ajoute
un fils et non plus un sous-arbre. */
public boolean ajout(int idPere, T fils, Position p) {
return (ajout(idPere,new ArbreBinBase<T>(fils),p));
}
```

```
/*
 * Affichage "textuel" du contenu de l'arbre
 * Le séparateur des différents éléments de l'arbre utilisé
 * lors de l'affichage est une chaîne vide.
 */

public void affiche() {
    System.out.println("----");
    affiche("");
}

        /*
 * Affichage "textuel" du contenu de l'arbre
 * Le paramètre esp est une chaîne de caractères utilisée
comme séparateur
 * des différents éléments de l'arbre lors de l'affichage.
 * Cette méthode n'est que "protégée" pour pouvoir être
redéfinie dans les sous-classes */
protected void affiche(String esp) {
    System.out.println(esp+racine);
    sousArbreGauche.affiche(esp+"  ");
    sousArbreDroit.affiche(esp+"  ");
}
```

Classe ArbreBin revue (4)

```
/* Méthodes mettant en oeuvre les 3 types de parcours
en profondeur d'abord d'un arbre. Elles renvoient un
chaîne de caractères contenant la liste des données des
noeuds dans l'ordre correspondant au parcours. */
public String parcoursInfixe() {
    String s1,s2;

    s1 = sousArbreGauche.parcoursInfixe();
    s2 = sousArbreDroit.parcoursInfixe();
    return s1+racine.toString()+s2;
}

public String parcoursPostfixe() {
    String s1,s2;

    s1 = sousArbreGauche.parcoursPostfixe();
    s2 = sousArbreDroit.parcoursPostfixe();
    return s1+s2+racine.toString();
}

public String parcoursPrefixe() {
    String s1,s2;

    s1 = sousArbreGauche.parcoursPrefixe();
    s2 = sousArbreDroit.parcoursPrefixe();
    return racine.toString()+s1+s2;
}
```

```
/* Même principe que pour les méthodes précédentes mais pour
le parcour en largeur. La file requise par ce parcours est mise en
oeuvre par un vecteur de sous-arbres binaires. */
public String largeur() {
    Vector<T> lues = new Vector<T>(); /* pour stocker le
résultat du parcours */
    Vector<ArbreBinBase<T>> aTraiter = new
Vector<ArbreBinBase<T>>(); /* la file */
    ArbreBinBase<T> courant;

    lues.add(racine.getDonnee());
    aTraiter.add(this);

    while (aTraiter.size() != 0) {
        courant = aTraiter.get(0);
        if (! courant.sousArbreGauche.isNull()) {
            lues.add(courant.sousArbreGauche.
getRacine().getDonnee());
            aTraiter.add(courant.sousArbreGauche);
        }
        if (! courant.sousArbreDroit.isNull()) {
            lues.add(courant.sousArbreDroit.
getRacine().getDonnee());
            aTraiter.add(courant.sousArbreDroit);
        }
        aTraiter.removeElementAt(0);
    }
    return lues.toString();
}
```

Classe ArbreBin revue (5)

```
    /* la taille d'un arbre est le nombre de noeuds de
l'arbre */
    public int taille() {
        int tailleSAG;
        int tailleSAD;

        tailleSAG = sousArbreGauche.taille();
        tailleSAD = sousArbreDroit.taille();
        return 1+tailleSAG+tailleSAD;
    }

    public int nbFeuilles() {
        int feuillesSAG;
        int feuillesSAD;

        if (isFeuille()) {
            return 1;
        }
        feuillesSAG = sousArbreGauche.nbFeuilles();
        feuillesSAD = sousArbreDroit.nbFeuilles();
        return feuillesSAG+feuillesSAD;
    }

    private boolean isFeuille() {
        return sousArbreGauche.isNull() &&
sousArbreDroit.isNull();
    }
```

```
    public int hauteur() {
        int hauteurSAG;
        int hauteurSAD;

        hauteurSAG = sousArbreGauche.hauteur();
        hauteurSAD = sousArbreDroit.hauteur();
        return 1+Math.max(hauteurSAG, hauteurSAD);
    }

    public boolean isNull() {
        return false;
    }
```

Etape 3 : Gestion des erreurs via des exceptions

Motivation

Simplifier le code en « sortant » les cas des noeuds non trouvés

Classe NoeudInexistantException

```
package correction3;


public class NoeudInexistantException extends Exception {

    public NoeudInexistantException(String s) {
        super("Le Noeud " + s + "n'existe pas dans l'arbre");
    }
}
```

Classe ArbreBinNull

```
package correction3;

class ArbreBinNull extends ArbreBinBase {
    private static ArbreBinNull instance = null;

    public ArbreBinNull() {}

    public static ArbreBinNull getInstance() {
        if (instance == null) {
            instance = new ArbreBinNull();
        }
        return instance;
    }

    @Override
    public int taille() {return 0;}

    @Override
    public int nbFeuilles() {return 0;}

    @Override
    public int hauteur() {return -1;}

    @Override
    public ArbreBinBase3 rechercheId(int idNoeud)
throws NoeudInexistantException {
        throw new
NoeudInexistantException("d'identifiant "+idNoeud);
    }

    @Override
    public ArbreBinBase3 rechercheDonnee(Object donnee)
throws NoeudInexistantException {
        throw new NoeudInexistantException("contenant la
donnée "+donnee);
    }

    @Override
    public String parcoursInfixe() {
        return "";
    }

    @Override
    public String parcoursPrefixe() {
        return "";
    }

    @Override
    public String parcoursPostfixe() {
        return "";
    }

    @Override
    public void affiche(String s) {
    }

    @Override
    public int getIdRacine() {return -1;}

    @Override
    public boolean isNull() {return true;}
}
```

Classe ArbreBin (modifications)

```
public ArbreBinBase<T> rechercheId(int idNoeud)
throws NoeudInexistantException {
    ArbreBinBase<T> sa;
    if (idNoeud == racine.getId()) {return this;}
    try {
        sa = sousArbreGauche.rechercheId(idNoeud);
        return sa;
    } catch (NoeudInexistantException nie) {
        sa = sousArbreDroit.rechercheId(idNoeud);
        return sa;
    }
}

public ArbreBinBase<T> rechercheDonnee(T donnee)
throws NoeudInexistantException {
    ArbreBinBase<T> sa;
    if (donnee.equals(racine.getDonnee())) {
        return this;
    }
    try {
        sa = sousArbreGauche.rechercheDonnee(donnee);
        return sa;
    } catch (NoeudInexistantException nie) {
        sa = sousArbreDroit.rechercheDonnee(donnee);
        return sa;
    }
}
```

```
public int getIdDonnee(T donnee) throws
NoeudInexistantException {
    ArbreBinBase<T> sa;

    sa = rechercheDonnee(donnee);
    return sa.getIdRacine();
}

public void ajout(int idPere, ArbreBinBase<T> sa,
Position p) throws NoeudInexistantException {
    ArbreBinBase<T> sap;

    sap = rechercheId(idPere);
    if (p == Position.gauche) {
        sap.setSAG(sa);
    } else {
        sap.setSAD(sa);
    }
}

public void ajout(int idPere, T fils, Position p) throws
NoeudInexistantException {
    ajout(idPere, new ArbreBinBase<T>(fils), p);
}
```

Code simplifié

Code simplifié

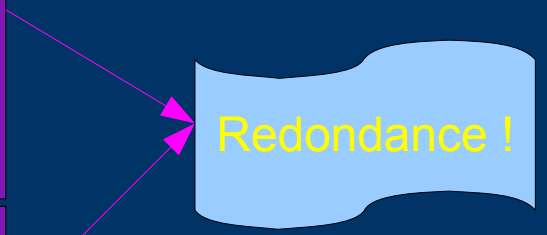
Etape 4 : Suppression de la redondance dans largeur

largeur

```
public String largeur() {
    Vector<T> lues = new Vector<T>(); /* pour stocker le résultat du parcours */
    Vector<ArbreBinBase<T>> aTraiter = new Vector<ArbreBinBase<T>>(); /* la file */
    ArbreBinBase<T> courant;

    lues.add(racine.getDonnee());
    aTraiter.add(this);

    while (aTraiter.size() != 0) {
        courant = aTraiter.get(0);
        if (! courant.sousArbreGauche.isNull()) {
            lues.add(courant.sousArbreGauche.getRacine().getDonnee());
            aTraiter.add(courant.sousArbreGauche);
        }
        if (! courant.sousArbreDroit.isNull()) {
            lues.add(courant.sousArbreDroit.getRacine().getDonnee());
            aTraiter.add(courant.sousArbreDroit);
        }
        aTraiter.removeElementAt(0);
    }
    return lues.toString();
}
```



Problème et principe de la solution

- Solution primaire
  - Introduire une nouvelle méthode
- Mais 2 variables doivent être modifiées par chacune des 2 parties :
  - lues
  - Atraiter
- Utiliser des paramètres de sortie ?
  - Peu clair
- Solution
  - Passer par une nouvelle classe

Mise en oeuvre
Classe parcoursEnLargeur

```
package correction3;

import java.util.ArrayList;
import java.util.LinkedList;

class ParcoureurEnLargeur<T> {
    private ArrayList<T> lues;
    private LinkedList<ArbreBinBase<T>> aTraiter;

    ParcoureurEnLargeur(ArbreBinBase<T> racine) {
        lues = new ArrayList<T>();
        aTraiter = new
LinkedList<ArbreBinBase<T>>();
        lues.add(racine.getDonneeRacine());
        aTraiter.push(racine);
        executer();
    }
```

N.B. : on optimise en remplaçant Vector par ArrayList et LinkedList

```
    private void executer() {
        ArbreBinBase<T> courant;
        while (aTraiter.size() != 0) {
            courant = aTraiter.remove();
            traiterSousArbre(courant.getSAG());
            traiterSousArbre(courant.getSAD());
        }
    }

    private void traiterSousArbre(ArbreBinBase<T>
arbre) {
        if (! arbre.isNull()) {
            lues.add(arbre.getDonneeRacine());
            aTraiter.add(arbre);
        }
    }

    String getResultat() {
        return lues.toString();
    }
}
```

Mise en oeuvre
Classe ArbreBinBase

```
public String largeur() {
    ParcoureurEnLargeur<T> parcoureur = new ParcoureurEnLargeur<T>(this);
    return parcoureur.getResultat();
}
```

Etape 5 : Transformation des méthodes « affiche » en appels à des méthodes toString()

Motivation

- Augmenter la testabilité du code

Affichage Avant/Après

AVANT

```
private void affiche(String esp) {
    System.out.println(esp+racine);
    if (sag != null) {
        sag.affiche(esp+"  ");
    }
    if (sad != null) {
    sad.affiche(esp+"  ");
    }
}


public void affiche() {
    System.out.println("----");
nnnnaffiche("");
}
```

APRES

```
public void affiche() {
        System.out.print(this);
    }

    @Override
    public String toString() {
        return "----\n"+toString("");
    }


    protected String toString(String esp) {
        StringBuilder sb = new StringBuilder(esp+racine+"\n");
        if (!isFeuille()) {
            // pour distinguer la position gauche ou droit dans le
cas d'un noeud ayant 1 seul fils
            sb.append(sousArbreGauche.toString(esp + "  "));
            sb.append(sousArbreDroit.toString(esp + "  "));
        }
        return sb.toString();
    }
```

Etape 6 : Définition de Jeux de Test Junit 4

Test de la classe Noeud<T> (1)

```
package correction3;

import org.junit.*;
import static org.junit.Assert.*;

public class NoeudTest {
    @Test
    public void testResetCompteurInstance() {
        System.out.println("resetCompteurInstance");
        Noeud.resetCompteurInstance();
        Noeud<String> s = new Noeud<String>("coucou");
        assertEquals(1,s.getId());
        Noeud<String> s2 = new Noeud<String>("coucou");
        Noeud.resetCompteurInstance();
        Noeud<String> s3 = new Noeud<String>("coucou");
        assertEquals(1,s3.getId());
    }

    @Test
    public void testGetDonnee() {
        System.out.println("getDonnee");
        String donnee = "coucou";
        Noeud<String> n = new Noeud<String>(donnee);
        assertSame(donnee, n.getDonnee());
    }
```

Test de la classe Noeud<T> (2)

```
@Test
    public void testGetId() {
        System.out.println("getId");
        Noeud.resetCompteurInstance();
        Noeud<Integer> n1 = new Noeud<Integer>(100);
        Noeud<Integer> n2 = new Noeud<Integer>(200);
        assertEquals(1,n1.getId());
        assertEquals(2,n2.getId());
    }

    @Test
    public void testSetDonnee() {
        System.out.println("setDonnee");
        Noeud<String> n1 = new Noeud<String>("Hello");
        Noeud<String> n2 = new Noeud<String>("World");
        String s1 = "foo";
        String s2 = "bar";
        n1.setDonnee(s1);
        n2.setDonnee(s2);
        assertSame(s1,n1.getDonnee());
        assertSame(s2,n2.getDonnee());
    }
```

Test de la classe Noeud<T> (3)

```
@Test
public void testEquals() {
    System.out.println("equals");
    Noeud<String> n1 = new Noeud<String>("Hello");
    Noeud<String> n2 = new Noeud<String>("World");
    Noeud<String> n3 = new Noeud<String>("Hello");
    assertTrue(n1.equals(n1));
    assertTrue(n2.equals(n2));
    assertFalse(n1.equals(n2));
    assertFalse(n1.equals(n3));
    assertFalse(n1.equals("Hello"));
}

@Test
public void testEqualsDonnee() {
    System.out.println("equalsDonnee");
    Noeud<Integer> n1 = new Noeud<Integer>(100);
    Noeud<Integer> n2 = new Noeud<Integer>(200);
    Noeud<Integer> n3 = new Noeud<Integer>(100);
    assertTrue(n1.equalsDonnee(n1));
    assertTrue(n1.equalsDonnee(n3));
    assertFalse(n1.equalsDonnee(n2));
}
```

Test de la classe Noeud<T> (4)

```
    @Test
    public void testToString() {
        System.out.println("toString");
        Noeud<String> n1 = new Noeud<String>("Hello");
        Noeud<Integer> n2 = new Noeud<Integer>(100);
        assertEquals("Hello",n1.toString());
        assertEquals(new Integer(100).toString(),n2.toString());
    }

    @Test
    public void testToStringId() {
        System.out.println("toStringId");
        Noeud.resetCompteurInstance();
        Noeud<String> n1 = new Noeud<String>("Hello");
        Noeud<Integer> n2 = new Noeud<Integer>(100);
        assertEquals("(1:Hello)",n1.toStringId());
        assertEquals("(2:"+new Integer(100).toString()+")",n2.toStringId());
    }

}
```

Test de la classe ArbreBinNull (1)

```
package correction3;

import org.junit.*;
import static org.junit.Assert.*;

public class ArbreBinNullTest {

    public ArbreBinNullTest() {
    }

    @Test
    public void testGetInstance() {
        System.out.println("getInstance");
        ArbreBinNull arbre1 = ArbreBinNull.getInstance();
        ArbreBinNull arbre2 = ArbreBinNull.getInstance();
        assertSame(arbre1, arbre2);
    }

    @Test
    public void testTaille() {
        System.out.println("taille");
        ArbreBinNull instance = ArbreBinNull.getInstance();
        assertEquals(0, instance.taille());
    }
```

Test de la classe ArbreBinNull (2)

```
@Test
public void testNbFeuilles() {
    System.out.println("nbFeuilles");
    ArbreBinNull instance = ArbreBinNull.getInstance();
    assertEquals(0, instance.nbFeuilles());
}

@Test
public void testHauteur() {
    System.out.println("hauteur");
    ArbreBinNull instance = ArbreBinNull.getInstance();
    int expResult = -1;
    int result = instance.hauteur();
    assertEquals(expResult, result);
}

@Test(expected=NoeudInexistantException.class)
public void testRechercheId() throws Exception {
    System.out.println("rechercheId");
    ArbreBinNull instance = ArbreBinNull.getInstance();
    ArbreBinBase result = instance.rechercheId(1);
}

@Test(expected=NoeudInexistantException.class)
public void testRechercheDonnee() throws Exception {
    System.out.println("rechercheDonnee");
    Object donnee = null;
    ArbreBinNull instance = ArbreBinNull.getInstance();
    ArbreBinBase result = instance.rechercheDonnee(donnee);
}
```

Test de la classe ArbreBinNull (3)

```
@Test
public void testParcoursInfixe() {
    System.out.println("parcoursInfixe");
    ArbreBinNull instance = ArbreBinNull.getInstance();
    String expResult = "";
    String result = instance.parcoursInfixe();
    assertEquals(expResult, result);
}

@Test
public void testParcoursPrefixe() {
    System.out.println("parcoursPrefixe");
    ArbreBinNull instance = ArbreBinNull.getInstance();
    String expResult = "";
    String result = instance.parcoursPrefixe();
    assertEquals(expResult, result);
}

@Test
public void testParcoursPostfixe() {
    System.out.println("parcoursPostfixe");
    ArbreBinNull instance = ArbreBinNull.getInstance();
    String expResult = "";
    String result = instance.parcoursPostfixe();
    assertEquals(expResult, result);
}
```

Test de la classe ArbreBinNull (4)

```
    @Test
    public void testToString_0arg() {
        System.out.println("toString()");
        ArbreBinNull instance = ArbreBinNull.getInstance();
        assertEquals("\n",instance.toString());
    }

    @Test
    public void testToString_1arg() {
        System.out.println("toString(String)");
        ArbreBinNull instance = ArbreBinNull.getInstance();
        assertEquals("toto\n",instance.toString("toto"));
    }

    @Test
    public void testGetIdRacine() {
        System.out.println("getIdRacine");
        ArbreBinNull instance = ArbreBinNull.getInstance();
        assertEquals(-1, instance.getIdRacine());
    }

    @Test
    public void testIsNull() {
        System.out.println("isNull");
        ArbreBinNull instance = ArbreBinNull.getInstance();
        assertTrue(instance.isNull());
    }
}
```

Test de la classe ArbreBin<T> (1)

```
package correction3;

import correction3.ArbreBinBase.Position;
import org.junit.*;
import static org.junit.Assert.*;

public class ArbreBinBaseTest {
    public ArbreBinBaseTest() {}

    @Before
    public void setUp() {
        Noeud.resetCompteurInstance();
    }

    @Test
    public void testGetRacine() {
        System.out.println("getRacine");
        ArbreBinBase<String> instance = new
ArbreBinBase("coucou");
        Noeud result = instance.getRacine();
        assertSame("coucou", result.getDonnee());
        assertEquals(1,result.getId());
        try {
            instance.ajout(1, "coucou2", Position.gauche);
            assertEquals(1,result.getId());
            assertSame("coucou",result.getDonnee());
        } catch (NoeudInexistantException nie) {
            fail("problème");
        }
    }

    @Test
    public void testGetDonneeRacine() {
        System.out.println("getDonneeRacine");
        String s = "coucou";
        ArbreBinBase<String> instance = new
ArbreBinBase(s);
        String result = instance.getDonneeRacine();
        assertSame(s, result);
    }

    @Test
    public void testGetIdRacine() {
        System.out.println("getIdRacine");
        String s = "coucou";
        ArbreBinBase<String> instance = new
ArbreBinBase(s);
        int expResult = 1;
        int result = instance.getIdRacine();
        assertEquals(expResult, result);
        String s2 = "coucou2";
        try {
            instance.ajout(1, s2, Position.gauche);
            ArbreBinBase<String> fils = instance.getSAG();
            assertEquals(2,fils.getIdRacine());
        } catch (NoeudInexistantException nie) {
            fail("problème");
        }
    }
```

Test de la classe ArbreBin<T> (2)

```
    @Test
    public void testRechercheId() throws Exception {
        System.out.println("rechercheId");
        ArbreBinBase<Integer> arbre = arbre3();
        ArbreBinBase<Integer> arbreRetourne =
arbre.rechercheId(1);
        assertEquals(1,arbreRetourne.getIdRacine());
        arbreRetourne = arbre.rechercheId(2);
        assertEquals(2,arbreRetourne.getIdRacine());
        arbreRetourne = arbre.rechercheId(5);
        assertEquals(5,arbreRetourne.getIdRacine());
    }

    @Test(expected=NoeudInexistantException.class)
    public void testRechercheId_EX() throws Exception {
        System.out.println("rechercheId inexistant");
        ArbreBinBase<Integer> arbre = arbre3();
        ArbreBinBase<Integer> arbreRetourne =
arbre.rechercheId(9);
    }
```

```
    @Test
    public void testRechercheDonnee() throws Exception {
        System.out.println("rechercheDonnee");
        ArbreBinBase<Integer> arbre = arbre3();
        ArbreBinBase<Integer> arbreRetourne =
arbre.rechercheDonnee(10);

assertEquals(10,arbreRetourne.getDonneeRacine().intVal
ue());
        arbreRetourne = arbre.rechercheDonnee(20);

assertEquals(20,arbreRetourne.getDonneeRacine().intVal
ue());
        arbreRetourne = arbre.rechercheDonnee(50);

assertEquals(50,arbreRetourne.getDonneeRacine().intVal
ue());
    }

    @Test(expected=NoeudInexistantException.class)
    public void testRechercheDonnee_EX() throws
Exception {
        System.out.println("rechercheDonnee inexistant");
        ArbreBinBase<Integer> arbre = arbre3();
        ArbreBinBase<Integer> arbreRetourne =
arbre.rechercheId(90);
    }
```

Test de la classe ArbreBin<T> (3)

```
@Test
public void testGetIdDonnee() throws Exception {
    System.out.println("getIdDonnee");
    ArbreBinBase<Integer> arbre = arbre3();
    assertEquals(1, arbre.getIdDonnee(10));
    assertEquals(2, arbre.getIdDonnee(20));
    assertEquals(5, arbre.getIdDonnee(50));
}

@Test(expected=NoeudInexistantException.class)
public void testGetIdDonnee_EX() throws Exception {
    System.out.println("getIdDonnee inexistant");
    ArbreBinBase<Integer> arbre = arbre3();
    arbre.getIdDonnee(90);
}

@Test
public void testAjout_3args_1() throws Exception {
    System.out.println("ajout avec sous-arbre possible");
    ArbreBinBase<String> arbre = new
ArbreBinBase("coucou");
    Position p = Position.gauche;
    ArbreBinBase<String> sag = new
ArbreBinBase("coucou2");
    arbre.ajout(1, sag, p);
    assertSame(sag,arbre.getSAG());
    p = Position.gauche;
    ArbreBinBase<String> sagsag = new
ArbreBinBase("coucou3");
    arbre.ajout(2, sagsag, p);
    assertSame(sagsag,arbre.getSAG().getSAG());
    p = Position.droit;
    ArbreBinBase<String> sad = new
ArbreBinBase("coucou4");
    arbre.ajout(1, sad, p);
    assertSame(sad,arbre.getSAD());
    p = Position.droit;
    ArbreBinBase<String> sad2 = new
ArbreBinBase("coucou5");
    arbre.ajout(1, sad2, p);
    assertSame(sad2,arbre.getSAD());
}

@Test(expected = NoeudInexistantException.class)
public void testAjout_3args_1_bis() throws Exception
{
    System.out.println("ajout avec sous-arbre
impossible");
    ArbreBinBase<String> arbre = new
ArbreBinBase("coucou");
    Position p = Position.gauche;
    ArbreBinBase<String> sag = new
ArbreBinBase("coucou2");
    arbre.ajout(10, sag, p);
}
```

Test de la classe ArbreBin<T> (4)

```
    @Test
    public void testAjout_3args_2() throws Exception {
        System.out.println("ajout avec valeur");
        ArbreBinBase<Integer> arbre = new
ArbreBinBase(10);
        arbre.ajout(1, 20, Position.droit);
        arbre.ajout(2, 30, Position.gauche);
        assertEquals(30,arbre.getSAD().getSAG().
              getDonneeRacine().intValue());
    }

    @Test
    public void testToString_0args() {
        System.out.println("toString()");
        ArbreBinBase<Integer> arbre = arbre3();
        String expected = "----\n10\n  20\n    40\n    50\n
70\n      \n  30\n    \n    60\n";
        assertEquals(expected,arbre.toString());
    }
    @Test
    public void testParcoursInfixe() {
        System.out.println("parcoursInfixe");
        ArbreBinBase<Integer> instance = arbre3();
        String expResult = "402070501030 60";
        String result = instance.parcoursInfixe();
        assertEquals(expResult, result);
    }
```

```
    @Test
    public void testParcoursPostfixe() {
        System.out.println("parcoursPostfixe");
        ArbreBinBase instance = arbre3();
        String expResult = "40705020603010";
        String result = instance.parcoursPostfixe();
        assertEquals(expResult, result);
    }

    @Test
    public void testParcoursPrefixe() {
        System.out.println("parcoursPrefixe");
        ArbreBinBase instance = arbre3();
        String expResult = "10204050703060";
        String result = instance.parcoursPrefixe();
        assertEquals(expResult, result);
    }

    @Test
    public void testLargeur() {
        System.out.println("largeur");
        ArbreBinBase instance = arbre3();
        String expResult = "[10, 20, 30, 40, 50, 60, 70]";
        String result = instance.largeur();
        assertEquals(expResult, result);
    }
```

Test de la classe ArbreBin<T> (5)

```
    @Test
    public void testTaille() {
        System.out.println("taille");
        ArbreBinBase<String> instance = new
ArbreBinBase("coucou");
        assertEquals(1, instance.taille());
        try {
            instance.ajout(1, "coucou2", Position.gauche);
            assertEquals(2, instance.taille());
        } catch (NoeudInexistantException nie) {
            fail("problème");
        }
        try {
            instance.ajout(2, "coucou3", Position.gauche);
            assertEquals(3, instance.taille());
        } catch (NoeudInexistantException nie) {
            fail("problème");
        }
        try {
            instance.ajout(1, "coucou4", Position.droit);
            assertEquals(4, instance.taille());
        } catch (NoeudInexistantException nie) {
            fail("problème");
        }
    }
```

```
    @Test
    public void testNbFeuilles() {
        System.out.println("nbFeuilles");
        ArbreBinBase<String> instance = new
ArbreBinBase("coucou");
        assertEquals(1, instance.nbFeuilles());
        try {
            instance.ajout(1, "coucou2", Position.gauche);
            assertEquals(1, instance.nbFeuilles());
        } catch (NoeudInexistantException nie) {
            fail("problème");
        }
        try {
            instance.ajout(1, "coucou3", Position.gauche);
            assertEquals(1, instance.nbFeuilles());
        } catch (NoeudInexistantException nie) {
            fail("problème");
        }
        try {
            instance.ajout(1, "coucou4", Position.droit);
            assertEquals(2, instance.nbFeuilles());
        } catch (NoeudInexistantException nie) {
            fail("problème");
        }
    }
```

Test de la classe ArbreBin<T> (6)

```
    @Test
    public void testHauteur() throws Exception {
        System.out.println("hauteur");
        ArbreBinBase<String> instance = new
ArbreBinBase("coucou");
        assertEquals(0, instance.hauteur());
        instance.ajout(1, "coucou2", Position.gauche);
        assertEquals(1, instance.hauteur());
        instance.ajout(1, "coucou3", Position.droit);
        assertEquals(1, instance.hauteur());
        instance.ajout(3, "coucou4", Position.droit);
        assertEquals(2, instance.hauteur());
    }

    @Test
    public void testIsNull() {
        System.out.println("isNull");
        ArbreBinBase<String> instance = new
ArbreBinBase("coucou");
        assertFalse(instance.isNull());
    }

    private static ArbreBinBase<Integer> arbre1() {
        ArbreBinBase<Integer> arbre = new
ArbreBinBase(10);
        return arbre;
    }

    private static ArbreBinBase<Integer> arbre2() {
        ArbreBinBase<Integer> arbre = new
ArbreBinBase(10);
        try {
            arbre.ajout(1, 20, Position.gauche);
            arbre.ajout(1, 30, Position.droit);
        } catch (NoeudInexistantException nie) {}
        return arbre;
    }

    private static ArbreBinBase<Integer> arbre3() {
        ArbreBinBase<Integer> arbre = new
ArbreBinBase(10);
        try {
            arbre.ajout(1, 20, Position.gauche);//(10;20;-)
            arbre.ajout(1, 30, Position.droit); //(10;20;30)
            arbre.ajout(2, 40, Position.gauche);//(10;
(20;40;-);30)
            arbre.ajout(2, 50, Position.droit); //(10;
(20;40;50);30)
            arbre.ajout(3, 60, Position.droit); //(10;(20;40;50);
(30;-;60))
            arbre.ajout(5, 70, Position.gauche);//(10;(20;40;
(50;70;-));(30;-;60))
        } catch (NoeudInexistantException nie) {}
        return arbre;
    }
}
```

Tests globaux

```
package correction3;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({NoeudTest.class,ArbreBinNullTest.class,ArbreBin
BaseTest.class})
public class TestsGlobaux {

}
```

Etape 7 : Simplification du constructeur de la classe Noeud

Motivations

- Rappel du constructeur de la classe noeud :

```
public Noeud(T val) {
    id = ++compteur;
    donnee = val;
}
```

- Analyse : 2 rôles
  - Obtention d'un nouvel identifiant
  - Affectation de cet identifiant au noeud

- Correction

  Faire gérer l'identifiant à une autre classe, qui deviendra du même coup réutilisable

Classe CompteurManager

```
package correction4;

public class CompteurManager {

    private int id;
    private static CompteurManager instance=null;
    private final static int INIT = 0;

    private CompteurManager() {
        reset();
    }

    public static CompteurManager getInstance() {
        if (instance == null) {
            instance = new CompteurManager();
        }
        return instance;
    }

    public int nextValeur() {
        return ++id;
    }

    public void reset() {
        id = INIT;
    }

    protected int getPremiereValeur() {
        return INIT+1;
    }
}
```

Classe Noeud (modifications)

```
public class Noeud<T> {

    private static CompteurManager compteur ; /* compteur d'instances de la classe Noeud*/
    private int id;                           /* identifiant du noeud courant */
    private T donnee;                         /* donnée utile du noeud courant */

    static {
        compteur = CompteurManager.getInstance();
    }

    public Noeud(T val) {
        id = compteur.nextValeur();
        donnee = val;
    }

    // methode requise pour les tests
    static void resetCompteurInstance() {
        compteur.reset();
    }
```

Classe CompteurManagerTest

```
package correction4;

import org.junit.*;

public class CompteurManagerTest {

    public CompteurManagerTest() {}

    @Test
    public void testGetInstance() {
        System.out.println("getInstance");
        CompteurManager c1 = CompteurManager.getInstance();
        CompteurManager c2 = CompteurManager.getInstance();
        assertNotNull(c1);
        assertSame(c1,c2);
    }

    @Test
    public void testNextValeur() {
        System.out.println("nextValeur");
        int MAX = 1000;
        CompteurManager instance =
CompteurManager.getInstance();
        int [] valeurs = new int[MAX];
        for (int i = 0 ; i < MAX ; i++) {
            valeurs[i] = instance.nextValeur();
        }
        boolean duplicata = false;
        ext: for (int i = 0 ; i < MAX-1 ; i++) {
            for (int j = i+1 ; j < MAX ; j++) {
                if (valeurs[i] == valeurs[j]) {
                    duplicata = true ;
                    break ext;
                }
            }
        }
        assertFalse(duplicata);
    }

    @Test
    public void testReset() {
        System.out.println("reset");
        CompteurManager instance =
CompteurManager.getInstance();
        for (int i = 0 ; i < 10 ; i++) {
            instance.nextValeur();
        }
        instance.reset();
        int expResult = instance.getPremiereValeur();
        int result = instance.nextValeur();
        assertEquals(expResult, result);
    }
}
```

Tests globaux

```
package correction3;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;

@RunWith(Suite.class)
@Suite.SuiteClasses({NoeudTest.class,ArbreBinNullTest.class,ArbreBin
BaseTest.class,CompteurManagerTeset.class})
public class TestsGlobaux {

}
```

Etape 8 : Modification des types de retour des parcours

Motivations

- Problème

  Les parcours actuels transforme l'arbre en chaîne, ce qui restreint leur usage à un seul cas

- Solution

  Réécrire les méthodes de parcours pour qu'elles ne renvoient plus une chaîne mais une liste

Modification de ArbreBin<T>

```
public List<T> parcoursInfixe() {
    List<T> s1, s2;
    ArrayList<T> retour;
    retour = new ArrayList<T>();
    s1 = sousArbreGauche.parcoursInfixe();
    retour.addAll(s1);
    retour.add(racine.getDonnee());
    s2 = sousArbreDroit.parcoursInfixe();
    retour.addAll(s2);
    return retour;
}

public List<T> parcoursPostfixe() {
    List<T> s1, s2;
    ArrayList<T> retour;
    retour = new ArrayList<T>();
    s1 = sousArbreGauche.parcoursPostfixe();
    retour.addAll(s1);
    s2 = sousArbreDroit.parcoursPostfixe();
    retour.addAll(s2);
    retour.add(racine.getDonnee());
    return retour;
}
```

```
    public List<T> parcoursPrefixe() {
        List<T> s1, s2;
        ArrayList<T> retour;
        retour = new ArrayList<T>();
        retour.add(racine.getDonnee());
        s1 = sousArbreGauche.parcoursPrefixe();
        retour.addAll(s1);
        s2 = sousArbreDroit.parcoursPrefixe();
        retour.addAll(s2);
        return retour;
    }

    public List<T> largeur() {
        ParcoureurEnLargeur<T> parcoureur = new
ParcoureurEnLargeur<T>(this);
        return parcoureur.getResultat();
    }
```

Modification de ArbreBinNull

```
@Override
public List parcoursInfixe() {
    return new ArrayList();
}

@Override
public List parcoursPrefixe() {
    return new ArrayList();
}

@Override
public List parcoursPostfixe() {
    return new ArrayList();
}
```

Modification de ParcoursEnLargeur<T>

- AVANT

```
String getResultat() {
    return lues.toString();
}
```

- APRES

```
List<T> getResultat() {
    return lues;
}
```

Modification de ArbreBinTest

```
@Test
public void testParcoursInfixe() {
    System.out.println("parcoursInfixe");
    ArbreBin<Integer> instance = arbre3();
    List<Integer> expResult = new ArrayList<Integer>();
    expResult.add(40); expResult.add(20);
    expResult.add(70); expResult.add(50);
    expResult.add(10); expResult.add(30);
    expResult.add(60);
    List<Integer> result = instance.parcoursInfixe();
    assertEquals(expResult, result);
}

@Test
public void testParcoursPostfixe() {
    System.out.println("parcoursPostfixe");
    ArbreBin instance = arbre3();
    List<Integer> expResult = new ArrayList<Integer>();
    expResult.add(40); expResult.add(70);
    expResult.add(50); expResult.add(20);
    expResult.add(60); expResult.add(30);
    expResult.add(10);
    List<Integer> result = instance.parcoursPostfixe();
    assertEquals(expResult, result);
}
```

```
@Test
public void testParcoursPrefixe() {
    System.out.println("parcoursPrefixe");
    ArbreBin instance = arbre3();
    List<Integer> expResult = new ArrayList<Integer>();
    expResult.add(10); expResult.add(20);
    expResult.add(40); expResult.add(50);
    expResult.add(70); expResult.add(30);
    expResult.add(60);
    List<Integer> result = instance.parcoursPrefixe();
    assertEquals(expResult, result);
}

@Test
public void testLargeur() {
    System.out.println("largeur");
    ArbreBin instance = arbre3();
    List<Integer> expResult = new ArrayList<Integer>();
    expResult.add(10); expResult.add(20);
    expResult.add(30); expResult.add(40);
    expResult.add(50); expResult.add(60);
    expResult.add(70);
    List<Integer> result = instance.largeur();
    assertEquals(expResult, result);
}
```

Modification de ArbreBinNullTest

```
@Test
public void testParcoursInfixe() {
    System.out.println("parcoursInfixe");
    ArbreBinNull instance = ArbreNull.getInstance();
    List expResult = new ArrayList();
    List result = instance.parcoursInfixe();
    assertEquals(expResult, result);
}

@Test
public void testParcoursPrefixe() {
    System.out.println("parcoursPrefixe");
    ArbreBinNull instance = ArbreNull.getInstance();
    List expResult = new ArrayList();
    List result = instance.parcoursPrefixe();
    assertEquals(expResult, result);
}

@Test
public void testParcoursPostfixe() {
    System.out.println("parcoursPostfixe");
    ArbreBinNull instance = ArbreNull.getInstance();
    List expResult = new ArrayList();
    List result = instance.parcoursPostfixe();
    assertEquals(expResult, result);
}
```