TP – Classe Arbre Binaire

Principe

Dans ce TP, nous allons coder une structure de donnée dynamique, c'est-à-dire qui va grandir au fur et à mesure des besoins.

Plus précisément, nous allons développer une classe pour représenter un arbre binaire de recherche. Une telle structure de donnée est efficace pour effectuer des recherches très rapides, ou éliminer des doublons dans une liste de données très de grande taille, par exemple. Elle facilite aussi bien sûr la représentation des informations structurées selon une arborescence, comme les répertoires d'un système de fichiers, ou les instructions d'un programme qui peuvent contenir des blocs imbriqués les uns dans les autres.

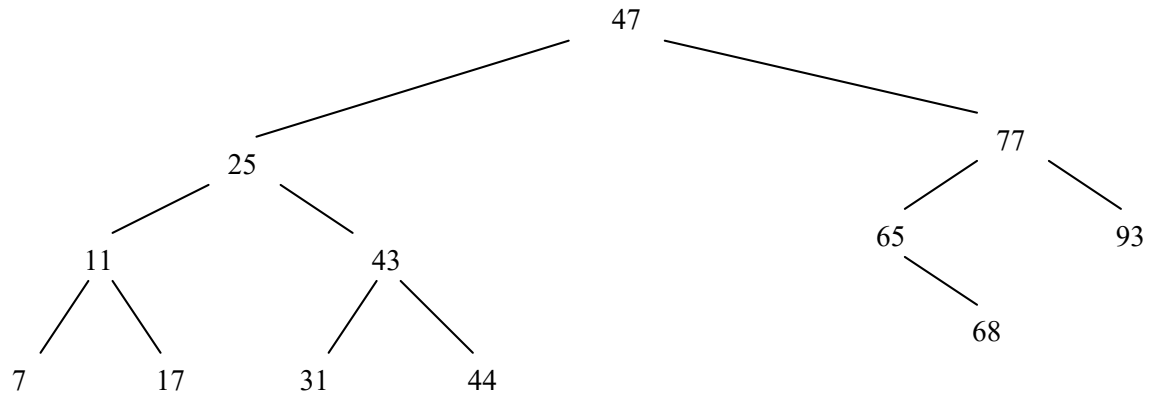
Un arbre binaire est constitué de nœuds. Chaque nœud contiendra une valeur présente dans l'arbre, ainsi qu'un pointeur vers un sous-arbre gauche et un pointeur vers un sous-arbre droit. Ces pointeurs peuvent éventuellement être égaux à nul.

Les nœuds qui n'ont pas de descendant sont appelés des **feuilles**. Le nœud situé en haut de la hiérarchie (c'est-à-dire le seul qui n'est pas situé dans un sous-arbre) est appelé le **nœud racine**.

Un **arbre binaire de recherche** possède en plus la caractéristique suivante : les valeurs du sous-arbre gauche sont inférieures à celle du nœud courant, et les valeurs du sous-arbre droit sont supérieures à celle du nœud courant.

Les arbres binaires manipulés dans ce TP ne pourront pas contenir de valeur en double.

Si par exemple les valeurs contenues dans l'arbre sont des entiers, un arbre binaire de recherche pourrait être :



Enoncé

Pour aboutir à cette représentation, nous devons développer deux classes : l'une pour représenter un nœud de l'arbre, l'autre pour représenter l'arbre binaire. En effet, il existe une différence importante entre ces deux notions, néanmoins voisines :

- un **noeud** devra toujours posséder une valeur et par définition ne pourra pas être vide dès lors qu'il aura été créé.
- **l'arbre binaire** pourra éventuellement être **vide**, et donc ne pas avoir de noeud racine (une référence nulle à la place).

La classe ***Nœud*** aura une structure récursive. Les classes écrites ***Nœud*** et ***ArbreBinaire*** seront génériques et paramétrées par le type des éléments de l'arbre.

Les méthodes à prévoir dans la classe ***Nœud*** sont les suivantes :

- Un **constructeur** avec en argument la valeur du nœud à créer. A la création, le nœud ne possèdera pas de sous-arbre, gauche ou droit.
- Une méthode ayant en paramètre une valeur à **insérer** dans l'arbre débutant au nœud courant. La méthode renverra un booléen indiquant si l'ajout a pu être effectué ou pas.
- Une méthode permettant de déterminer si la valeur argument **est présente** ou pas dans l'arbre débutant au nœud courant.
- Trois accesseurs : l'un pour la valeur du nœud courant, l'un pour le sous-arbre gauche, l'autre pour le sous-arbre droit.

Bien sûr, plusieurs des méthodes ci-dessus seront récursives.

La classe ***ArbreBinaire*** proposera les services suivants :

- Un **constructeur** par défaut. L'arbre construit sera vide.
- Une méthode ayant en paramètre une valeur à **insérer**. La méthode renverra un booléen indiquant si l'ajout a pu être effectué ou pas.
- Une méthode permettant de déterminer si la valeur argument **est présente** ou pas dans l'arbre.
- Une méthode qui affiche l'arbre en faisant apparaître les niveaux hiérarchiques.

Avec l'arbre donné en exemple, on obtiendra :

```
                7
          11
                17
    25
                31
          43
                44
47
          65
                68
    77
          93
```

Question 1

Développer la classe ***Nœud*** et la tester. Cette classe sera générique. Suivre les étapes suivantes :

1) Définir les attributs et le constructeur. Tester grâce à la méthode *testConstructeurSeul* (voir classe ***TestNoeud*** sur Moodle).

2) Coder la méthode ***estPresente.*** Elle est plus simple à coder que la méthode *inserer*. Tester partiellement ***estPresente*** grâce à la méthode *testEstPresenteSansDescendance*.

3) Coder la méthode ***inserer***. Se rappeler que l'arbre binaire ne contiendra pas de valeurs en double. Tester la méthode avec *testInserer*.

4) Compléter les tests des méthodes ***estPresente*** et ***inserer*** grâce à *testInsererEstPresente*.

5) Coder les accesseurs et les tester (aucune méthode de test n'est fournie).

Question 2

Développer la classe ***ArbreBinaire*** et la tester. Cette classe sera générique. Se rappeler qu'elle va envelopper la classe ***Nœud*** et servira à représenter un arbre vide, ce que la classe ***Nœud*** à elle seule ne pourra pas faire.

On retrouvera dans cette classe des méthodes équivalentes à celles de la classe *Nœud*. Elles sont codées selon le schéma suivant :

```
Si la racine est égale à nul alors
    Traiter ce cas particulier
Sinon
    Invoquer, sur le nœud racine, la méthode équivalente définie dans la
    classe Nœud
FinSi
```

Suivre la démarche suivante :

1) Définir l'attribut et le constructeur et tester.

2) Coder la méthode ***estPresente*** et tester**.**

3) Coder la méthode ***inserer*** et tester**.**

4) Coder la méthode qui réalise l'affichage et tester (à coder dans la classe ***Nœud*** et aussi dans ***ArbreBinaire***). Indication pour cette méthode d'affichage :

- dans la classe ***Nœud***, on peut définir une constante et une méthode ***afficheArbreNiveau*** :

```
/**
 * Constante égale au nombre d'espaces d'indentation à laisser lors de
 * l'affichage des différents niveaux de l'arbre
 */
private static final int DECALAGE = 5;
```

```java
/**
 * Affiche les valeurs contenues dans l'arbre débutant au noeud argument
 * Chaque fois que l'on descend d'un niveau dans l'arbre, les valeurs des
 * noeuds sont affichées (en colonne) à droite des précédentes.
 * @param niveau niveau de profondeur du noeud courant. Cette valeur sert à calculer
 *               sur quelle colonne il faut effectuer l'affichage
 */
public void afficheArbreNiveau(int niveau) {  . . .
```

- La valeur *niveau* sera augmentée de 1 lors des appels récursif sur les sous-arbres gauche et droit.

- Dans la classe *ArbreBinaire*, on définira aussi une méthode ***afficheArbreNiveau*** qui invoquera la méthode équivalente de la classe ***Noeud*** sur le noeud racine avec la valeur 0 pour l'argument *niveau*.

Question 3 – Parcours de l'arbre

a) Indiquer les valeurs obtenues si on parcourt l'arbre de la page 1 dans l'ordre :

- ✓ préfixe .....
- ✓ infixe .....
- ✓ postfixe .....

b) Ajouter à la classe ***Noeud*** une méthode qui renvoie une chaîne de caractères contenant les valeurs des nœuds à partir du nœud courant et parcourus dans l'ordre préfixe. Chaque valeur d'un nœud sera séparée de la suivante par 2 espaces.

Ajouter ensuite la même méthode dans la classe ***ArbreBinaire***. Dans le cas où l'arbre est vide, la méthode renverra la chaîne "Arbre vide".

Tester les méthodes obtenues.

c) Ajouter 2 méthodes à la classe ***Noeud*** qui effectuent le même traitement mais en parcourant les nœuds de manière infixe et postfixe.

Ajouter les 2 méthodes équivalentes à la classe ***ArbreBinaire***. Dans le cas où l'arbre est vide, elles renverront la chaîne "Arbre vide".

Tester les méthodes obtenues.

Question 4

Ajouter une méthode permettant de déterminer la ***hauteur*** de l'arbre courant. Un arbre vide à une hauteur égale à 0, un arbre réduit à sa racine une hauteur de 1. Dans les autres cas, la hauteur est égale au niveau de la feuille la plus éloignée de la racine.
Par exemple, la hauteur de l'arbre donné en exemple est égale à 4.
Tester la méthode obtenue.

Question 5

Ajouter une méthode permettant de déterminer si une valeur est ***présente et placée sur une feuille***.
Dans l'exemple donné précédemment :

- ✓ la valeur 68 remplit la propriété
- ✓ les valeurs 43, 65 et 88 ne remplissent pas cette propriété

Tester la méthode obtenue.

Question 6 (méthode très facile à écrire !)

Ajouter une méthode permettant de renvoyer **la plus grande valeur présente dans l'arbre**. Si l'arbre est vide la méthode renverra *null*.
Tester la méthode obtenue.

Question 7

Ajouter une méthode permettant de ***supprimer une valeur donnée à condition qu'elle soit située sur une feuille.*** La méthode renverra un booléen indiquant si la suppression a pu être effectuée.
Dans l'exemple donné précédemment :

- ✓ la valeur 68 peut être supprimée
- ✓ les valeurs 43, 65 et 88 ne le peuvent pas

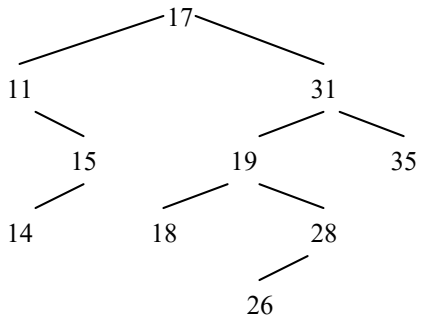
Tester la méthode obtenue.

Facultatif - Question 8 (méthode plus difficile)

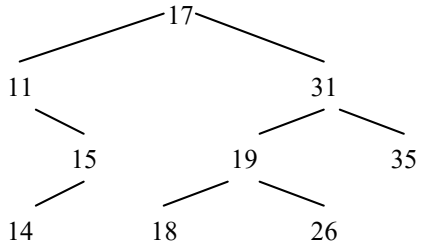
Ajouter une méthode permettant de **supprimer une valeur quelconque** dans l'arbre binaire. La méthode renverra un booléen indiquant si la suppression a été effectuée.
Tester la méthode obtenue.

Rappel

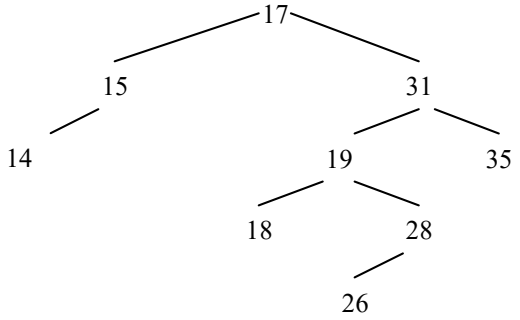
Supposons que l'arbre soit le suivant :



Si le sous-arbre droit du nœud à supprimer est vide, comme par exemple lors de la suppression de 28, il suffit de placer la racine du sous-arbre gauche à la place du nœud à supprimer. On obtiendra :

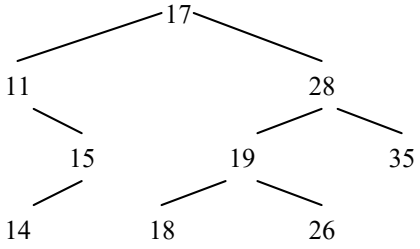


Si le sous-arbre gauche du nœud à supprimer est vide, comme par exemple lors de la suppression de 11, il suffit de placer la racine du sous-arbre droit à la place du nœud à supprimer. On obtiendra :



Sinon, dans le cas général aucun des sous-arbres du nœud à supprimer n'est vide. La valeur à supprimer peut être remplacée par la valeur la plus grande du sous-arbre gauche (de manière symétrique, on pourrait décider aussi de la remplacer par la valeur la plus petite du sous-arbre droit).

Supposons que l'on souhaite supprimer 31. La valeur la plus grande du sous-arbre gauche est 28. Il faut donc supprimer 28 de sa place initiale, et le mettre à la place de 31.



Indications

Il n'est pas judicieux d'utiliser la méthode de la question 7.

Supprimer un nœud de l'arbre implique de modifier le nœud père de celui-ci, soit pour actualiser son nœud racine du sous-arbre gauche, soit son nœud racine du sous-arbre droit.

La méthode récursive à écrire aura donc en paramètre le nœud père du nœud courant. Cette méthode sera ajoutée à la classe ***Nœud***. Elle réalisera le traitement pour tout nœud ayant un père, donc elle ne pourra pas gérer la suppression de la racine de l'arbre. C'est la méthode de la classe ***ArbreBinaire*** qui gèrera ce cas.

La spécification de la méthode ***supprimer*** de la classe ***Nœud*** sera la suivante :

```
    /**
     * Supprime la valeur argument si elle est présente à partir du noeud courant,
     * donc soit en tant que valeur du noeud courant, soit sur l'un des noeuds
     * descendants du noeud courant.
     * @param pere          noeud père du noeud courant
     * @param aSupprimer    valeur à supprimer
     * @return  un booléen égal à vrai ssi la suppression a pu se faire
     *          (si le noeud pere a pour valeur null, la méthode renvoie faux.
     *           En effet, la suppression exige de modifier éventuellement le noeud
     *           père. Son abscence empêche dont d'effectuer la suppression)
     */
    public boolean supprimer(Noeud<T> pere, T aSupprimer);
```

Cette méthode fera appel à la méthode suivante à prévoir aussi dans la classe **Noeud** :

```
    /**
     * Renvoie la référence sur le noeud contenant la plus grande valeur
     * à partir du noeud courant
     * @return  le noeud qui contient la plus grande valeur à partir
     *          du noeud courant, éventuellement le noeud courant lui-même
     */
    public Noeud<T> plusGrandNoeud();
```

Il faudra également ajouter un *setter* à la classe ***Noeud*** pour pouvoir modifier la valeur d'un nœud. Cette méthode sera invoqué par la méthode ***supprimer*** de la classe ***ArbreBinaire*** si celle-ci doit modifier la valeur de la racine de l'arbre.