Algorithmique
TP 4 : Arbres binaires de recherche

Les fichiers `NodeInt.java` ***et*** `BinTree.java` ***doivent être déposé sur l'espace "Rendu TP4".***

Un *Arbre Binaire de Recherche ou ABR (Binary Search Tree ou BST)* est un arbre binaire dont les étiquettes sont prises dans un ensemble X totalement ordonné (ici : les entiers) et dont le parcours **infixe** lit les étiquettes en ordre croissant.
Les ABR sont utilisés typiquement pour fournir des méthodes d'accès efficaces à ses éléments (recherche et insertion en temps logarithmique si l'arbre est équilibré).
Une application est le tri par arbre ("*tree sort*"), consistant à insérer les éléments à trier un à un dans un ABR (vide au départ) puis de lire ses étiquettes dans l'ordre infixe.

Les méthodes demandées à la fin du TP 3 peuvent être récupérées et adaptées pour ce TP, si vous les avez déjà faites.

1 Noeud d'un ABR

Dans le fichier `NodeInt.java` complétez l'implementation de la classe `NodeInt`, qui encode un nœud d'un arbre binaire. Pour ce TP (contrairement au TP3) on choisit la représentation suivante : un arbre est une classe possédant les attributs privés :

- `Integer val`, qui indique la clé entière du nœud de l'arbre.
- `NodeInt left` et `NodeInt right`, qui sont les références vers les fils (gauche et droit) du nœud; la valeur `null` de ces attributs indique l'absence du fils.

Les constructeurs `NodeInt(Integer val)`, qui construit un nœud avec la clé `val` et initialise `left` et `right` à `null`, et `NodeInt(Integer val, NodeInt lson, NodeInt rson)`, qui construit un nœud avec les fils indiqués, sont fournis. De même pour les accesseurs (de la forme `getAaa` et `setAaa`) qui permettront de lire et modifier les attributs *privés* de `NodeInt` lorsque cette classe sera utilisée par une autre (ici : `BinTree`).

1. Coder la méthode `public boolean isLeaf()` qui renvoie `true` si et seulement si le nœud est une feuille (c'est-à-dire si et seulement ses deux fils sont `null`).

2. Coder la méthode `public String toString()` qui affiche le nœud et ses fils.

3. Coder la méthode `boolean isIntBST(Integer min, Integer max)` qui renvoie `true` si et seulement si le nœud est la racine d'un ABR avec les clés k telles que `min`$< k \leq$`max`.

4. Ajouter des tests dans les méthodes `testIsLeaf`, `testToString` et `testIsIntBST`.

2 ABR et opérations en lecture✍

Le rôle de la classe `BinTree` est d'encapsuler une référence vers un nœud racine (instance de la classe `NodeInt`), afin de représenter un ABR (vide quand la référence vers la racine est `null`).
Par ailleurs, cette classe est censée ne représenter que des ABR. Cette propriété pourra être vérifiée en appelant la méthode `isBST` (on l'appele dans le constructeur avec paramètre et on lève une exception si elle retourne `false`.).
Les autres méthodes de cet exercice ne modifient pas l'arbre, mais partent du principe que l'arbre est un ABR.
Compléter le fichier `BinTree.java` :

1. Définir l'attribut privé `NodeInt root`, qui dénote la racine de l'arbre.
2. Définir le constructeur `BinTree()` qui construit l'arbre vide.
3. Définir le constructeur `BinTree(NodeInt r)` qui construit l'arbre de racine `r`.
4. Coder la méthode `String toString()` qui affiche l'arbre sous la forme d'une formule.
5. Coder la méthode `boolean isEmpty()` qui renvoie `true` si et seulement si l'arbre est vide.
6. Coder la méthode `int height()` qui calcule la hauteur de l'arbre, c'est à dire le maximum des profondeurs des nœuds de l'arbre. Par convention, la hauteur d'un arbre vide est -1.
7. Coder la méthode `boolean isBST()` qui renvoie `true` si et seulement si l'arbre est un arbre binaire de recherche.
8. Coder la méthode `Integer getMax()` qui calcule la valeur maximale dans l'arbre, en supposant qu'il s'agit d'un ABR. La méthode doit être non récursive et renvoyer `Integer.MIN_VALUE` si l'arbre est vide.
9. Coder la méthode `int less(Integer x)` qui compte le nombre de valeurs plus petites que x dans l'arbre, en supposant qu'il s'agit d'un ABR. La méthode doit être non récursive, donc utiliser une structure de pile fournie par les collections Java (par exemple l'interface `Deque` vue en TP 2).
10. Coder la méthode `boolean contains(Integer val)` qui dit si l'arbre contient la valeur passée en paramètre (il ne faut pas parcourir tout l'arbre, mais utiliser le fait que celui-ci est un ABR pour minimiser le nombre d'opérations).

3 Opérations en écriture et tri par Arbre (bonus)

1. Coder une méthode `void insert(Integer val)`, qui insère la valeur `val` dans l'arbre courant (cela implique de créer un nœud contenant `val`), en faisant en sorte que celui-ci reste un ABR s'il en était un.

Testez cette méthode en insérant des entiers aléatoires dans un arbre vide et en vérifiant que le résultat est bien un ABR.

2. Codez une méthode `static void treeSort(Integer[] elements)` qui trie le tableau passé en paramètre en utilisant l'algorithme de tri par arbre décrit dans l'introduction du sujet.

   Testez cette méthode.

3. Codez la méthode `boolean remove(Integer val)` qui implémente l'algorithme vu en cours pour supprimer la valeur `val` de l'arbre en préservant sa structure d'ABR. Cette méthode retourne `true` si un élément a effectivement été supprimé.