INF2010 – Structures de données et algorithmes
Automne 2017
Travail Pratique 4
Arbres binaires de recherche

Objectifs :

- Implémenter un arbre binaire de recherche.
- Implémenter un arbre AVL (type d'arbre binaire de recherche balancé).
- Développer une meilleure compréhension de l'utilité des arbres binaires de recherche.
- Maîtriser les différentes façons de parcourir un arbre binaire.

Qu'est-ce qu'un arbre binaire de recherche?

Un arbre binaire de recherche ou *binary search tree (BST)* est un arbre binaire qui respecte les deux propriétés suivantes:

- La valeur du fil gauche d'un nœud donné est inférieure à la valeur du nœud courant.
- La valeur du fils droit d'un nœud donné est supérieure à la valeur du nœud courant.

En quoi est-ce utile?

Un arbre binaire de recherche a diverses utilités. Les éléments suivants en sont des exemples:

- Ajout et retrait d'éléments de façon efficace (O(log(n)) en cas moyen).
- Recherche d'un élément de façon efficace (O(log(n)) en cas moyen).
- Obtention du minimum et du maximum de façon efficace (O(log(n)) en cas moyen).
- Obtention des éléments triés de façon efficace (O(n) en cas moyen).

Exercice 1: Fonctionnalités de base (1.5 pts)

Pour le premier exercice, on vous demande d'implémenter diverses fonctionnalités de base d'un arbre binaire de recherche afin de mieux en comprendre le fonctionnement. Vous devez ainsi apporter des changements à la classe `BST` du fichier `BST.java`. Les diverses méthodes de la classe `BstMain` (dans le fichier `BstMain.java`) vous permettront de tester les fonctionnalités que vous aurez complétées.

1) Ajout d'un élément (0.5 pts)

Il vous est ici demandé d'implémenter la méthode `insert()`. Cette dernière doit vous permettre d'ajouter un élément à l'arbre binaire de recherche tout en le conservant valide, c'est-à-dire, qu'après chaque insertion, l'arbre doit toujours respecter les deux critères donnés plus haut. Veuillez noter qu'on ne vous demande pas ici de balancer l'arbre, mais simplement

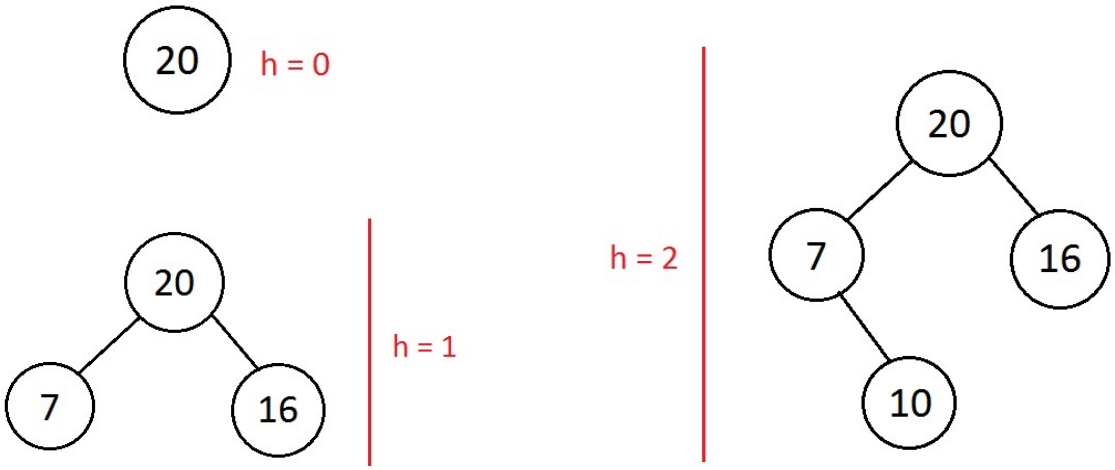
d'ajouter la valeur passée en paramètre à l'endroit approprié dans l'arbre. Aussi, **veuillez ignorer les doublons, une valeur ne devrait donc pas se trouver dans l'arbre plus d'une fois**.

2) Recherche d'un élément (0.5 pts)

Vous devez ensuite compléter la méthode `contains()` de la classe `BST` afin d'indiquer si l'arbre contient ou non la valeur passée en paramètre.

3) Calcul de la hauteur de l'arbre (0.5 pts)

De plus, vous devez compléter la méthode `getHeight()` qui retourne la hauteur de l'arbre. Cette méthode vous sera d'ailleurs utile plus tard afin de déterminer si un arbre est balancé ou non. Les figures suivantes servent d'exemples de la valeur de la hauteur en fonction de l'arbre.



Exercice 2: Parcours d'arbre (1 pt)

Il existe différentes façons de parcourir un arbre binaire, les plus fréquemment utilisées sont les suivantes :

- Parcours pré-ordre (parent, fils gauche, fils droite).
- Parcours post-ordre (fils gauche, fils droite, parent).
- Parcours en ordre ou ascendant (fils gauche, parent, fils droite).
- Parcours en ordre inverse ou descendant (fils droite, parent, fils gauche).
- Parcours par niveau.

Il vous est donc demandé, dans cet exercice, d'implémenter ces 5 types de parcours dans les méthodes correspondantes de la classe `BST`. Encore une fois, les méthodes de la classe `MainBst` vous permettront de tester votre implémentation.

Plus de détails sur ces parcours d'arbre sont aussi disponibles dans les diapositives du cours 5.

Exercice 3: Arbres AVL (2 pts)

Tel que mentionné précédemment, les arbres binaires de recherche sont des structures de données permettant d'effectuer plusieurs opérations de façon très efficace. Cependant, il peut arriver qu'un arbre se retrouve débalancé, c'est-à-dire, que la hauteur de certains sous-arbres soit plus grande que celle de leur frère. Les opérations effectuées sur un arbre débalancé ne sont alors plus aussi efficaces. En effet, la complexité asymptotique en pire cas des opérations mentionnées plus haut est O(n) plutôt que O(log(n)) en cas moyen.

Afin d'y remédier, des arbres binaires de recherche balancés ont été inventés. Les arbres AVL en sont d'ailleurs un type. Ces derniers utilisent ainsi des rotations autour d'un nœud débalancé afin de revenir à un état dit balancé, c'est-à-dire, un état où la différence en hauteur entre un fils de gauche et un fils de droite n'est jamais plus de 1.

1) Ajout d'un élément (0.4 pts)

Dans cet exercice, on vous demande alors de compléter l'implémentation de la méthode `insert()` de la classe `AvlTree`.

2) Balancement (1.6 pts)

De manière à compléter l'insertion dans un arbre AVL, vous allez aussi devoir compléter les quatre méthodes de balancements couvrant chacun un cas possible, soit :

- Cas débalancement droite-droite (`balanceRightRight()`).
- Cas débalancement droite-gauche (`balanceRightLeft()`).
- Cas débalancement gauche-gauche (`balanceLeftLeft()`).
- Cas débalancement gauche-droite (`balanceLeftRight()`).

La classe `AvlMain` vous permettra ensuite de tester votre implémentation de ces opérations.
Pour plus ample information sur les arbres AVL, les différents cas de débalancement et les types de rotations effectuées afin de corriger les débalancements, vous pouvez vous référer aux diapositives du cours 6.

Exercice 4: Applications réelles (0.5 pts)

Afin de démontrer l'utilité des arbres binaires de recherche dans la vie réelle, nous avons décidé de représenter une bibliothèque très simple à l'aide de l'interface `IBibliotheque`.

La bibliothèque possède donc une collection de livres (représentés par leur titre) avec laquelle on veut pouvoir effectuer diverses opérations :

- Ajouter un livre à la bibliothèque (`ajouterLivre()`).
- Trouver si un livre est présent dans la bibliothèque (`contientLivre()`).
- Afficher les livres en ordre alphabétique (`afficherLivresAlpha()`).
- Afficher les livres en ordre alphabétique inverse (`afficherLivresAlphaInverse()`).

Une implémentation de cette interface a déjà été complétée à l'aide d'une liste dans la classe `BibliothequeListe`. Cependant, comme vous pouvez le constater dans les commentaires présents dans le code source, il est possible d'effectuer certaines opérations de façon plus efficace à l'aide d'un arbre binaire de recherche.

Il vous est ainsi demandé de compléter l'implémentation de la classe `BibliothèqueBst` à l'aide de la classe `AvlTree` complétée précédemment. La classe `BibliothèqueMain` vous permettra ensuite de tester cette partie du travail.

Instructions pour la remise

Le travail doit être fait par équipe de 2 personnes et doit être remis via Moodle au plus tard le :

- 6 novembre avant 23h59 pour le groupe 4, soit Mardi (B2)
- 12 novembre avant 23h59 pour le groupe 2, soit Lundi (B2)
- 13 novembre avant 23h59 pour le groupe 3, soit Mardi (B1)
- 19 novembre avant 23h59 pour le groupe 1, soit Lundi (B1)

Veuillez envoyer vos fichiers .java **seulement**, dans un **seul répertoire**, le tout dans une archive de type ***.zip** (et seulement **zip**, pas de **rar**, **7z**, etc.) qui portera le nom : **inf2010_lab4_MatriculeX_MatriculeY.zip**, où MatriculeX < MatriculeY.

Les travaux en retard seront pénalisés de 20 % par jour de retard. Aucun travail ne sera accepté après 4 jours de retard. Si votre dépôt ne respecte pas la nomenclature définie ci-dessus, 0.5 point de pénalité sera appliqué.