Programmation & Algorithmique 2

TP - Séance 11

17 mai 2023

Matière visée : structures de données, généricité.

1 Arbres Binaires de Recherche

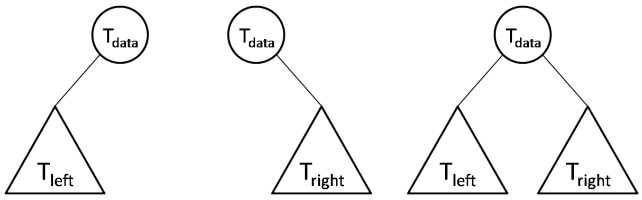
1.1 Définitions

Un *Arbre Binaire* (*Binary Tree* en anglais) est une structure de données composée d'éléments que nous appelons des *nœuds*. Le nœud initial, c'est-à-dire celui qui se trouve tout en haut de la représentation graphique de l'arbre, est appelé la *racine*. Chaque nœud peut avoir jusqu'à deux nœuds, appelés *fils*, en dessous de lui-même que nous appelons *fils gauche* et *fils droit*. Du point de vue de ces fils, le nœud dont ils sont issus est appelé *père*. Enfin, si un nœud n'a aucun fils, nous disons que ce nœud est une *feuille*.

Un *arbre binaire* T peut être défini récursivement comme suit :

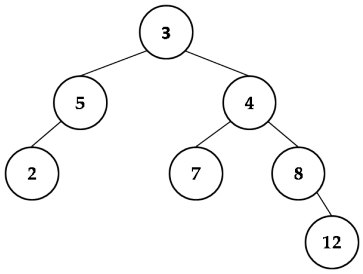
— Cas de base : une feuille

— Cas généraux :



où T_{data} est la donnée contenue dans la racine de notre arbre. T_{left} et T_{right} sont deux sous-arbres qui sont eux-mêmes deux arbres binaires qui suivent également la définition que nous venons de donner (c'est-à-dire qu'ils auront également leur propre racine ainsi que leurs sous-arbres).

Exemple d'arbre binaire (d'entiers) :



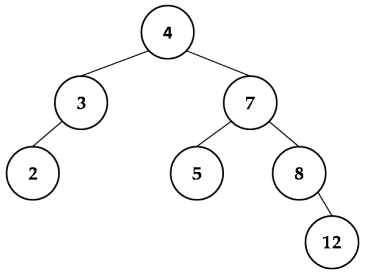
Le nœud contenant la valeur 2 est donc par exemple une feuille puisqu'il n'a ni fils gauche ni fils droit, ce qui représente notre cas de base. La structure formée par les nœuds contenant les valeurs 2 et 5 représente par contre déjà un cas général. C'est un arbre binaire où le nœud contenant la valeur 2 forme le sous-arbre gauche du nœud contenant la valeur 5, qui est la racine de l'arbre. Cet arbre sera lui-même le sous-arbre gauche du nœud contenant la valeur 3 et ainsi de suite.

Un *Arbre Binaire de Recherche* (*Binary Search Tree* en anglais) est un cas plus spécifique de l'arbre binaire. C'est un arbre binaire tel que pour tout nœud T de l'arbre :

— **tous les nœuds** du sous-arbre gauche (T_{left}) ont une valeur $< T_{data}$; et

— **tous les nœuds** du sous-arbre droit (T_{right}) ont une valeur $> T_{data}$.

Exemple d'arbre binaire de recherche (d'entiers) :



Dans cet exemple, vous pouvez voir que tous les éléments du sous-arbre gauche (c'est-à-dire 2 et 3) sont inférieurs à 4. De la même manière, tous les éléments du sous-arbre droit (c'est-à-dire 5, 7, 8 et 12) sont supérieurs à 4. Notez que cette propriété reste valable pour tous les sous-arbres de cet arbre. Par exemple, pour le sous-arbre droit, 5 est bien inférieur à 7. Nous avons également que 8 et 12 sont supérieurs à 7.

N'hésitez pas, à travers le TP, à faire des schémas pour les définitions où les algorithmes que vous avez du mal à comprendre en prenant exemple sur ceux qui vous sont montrés dans ce document. Les arbres de manière générale sont des structures qui sont facilement représentables visuellement. Cette visualisation peut énormément aider à la compréhension.

1.2 Classe BSTNode

Écrivez une classe `BSTNode` (*Binary Search Tree Node*). Un objet de cette classe représente un nœud d'un arbre binaire de recherche d'entiers. Implémentez-y les différents accesseurs.

Dans cette classe, nous aurons donc besoin de pouvoir stocker la donnée à l'intérieur du nœud ainsi que des références vers les fils de ce nœud. C'est en fait, en certains points, assez similaire à une implémentation d'une liste chainée si ce n'est que l'on peut avoir plus ou moins de références vers les éléments suivants. Le cas où chaque nœud a exactement un fils correspond en fait à une liste chainée. Un arbre binaire de recherche sera en fait un pointeur vers sa racine (qui est le `BSTNode` au sommet de l'arbre).

N'oubliez pas, pour cette étape et les suivantes, de tester à chaque fois que vos méthodes fonctionnent correctement.

1.3 Clonage d'arbres

Spécialisez la méthode `Object clone()` de la classe `Object` afin de pouvoir cloner les arbres. N'oubliez pas le clonage en profondeur !

1.4 Généricité

Jusqu'à présent, nous avons supposé que les données se trouvant dans les différents nœuds de notre arbre sont des entiers, mais nous pourrions très bien avoir d'autres types de données à l'intérieur.

Modifiez votre classe `BSTNode` pour qu'elle utilise un type générique pour ses données. En plus de la modification de la déclaration de la classe, il faut aussi s'assurer que les sous-arbres et les différents accesseurs soient déclarés correctement.

Vous y avez peut-être déjà pensé, mais vous aurez également besoin, pour les étapes suivantes, d'effectuer des comparaisons. Comme vous utilisez un type générique, vous ne pourrez pas utiliser les opérateurs habituels. Si votre type générique implémente une certaine interface tel que vous l'avez vu au cours théorique, vous pouvez être certain que vous pouvez utiliser une certaine méthode qui permet la comparaison. Vous pouvez vous assurer que votre type générique implémente une interface en écrivant à l'intérieur des balises (c'est-à-dire par exemple `<T extends Y<T>>` où `Y` est l'interface en question). Assurez-vous que la comparaison fonctionne avant de passer aux exercices suivant.

1.5 Égalité

Récursivement, deux arbres binaires de recherche sont égaux si et seulement si les données des deux racines sont égales **et** les deux sous-arbres respectifs sont égaux.

Ajoutez dans votre classe `BSTNode` une méthode `boolean equals(Object o)` qui permet de tester l'égalité de deux arbres binaires de recherche.

Notez que cette méthode est une méthode de la classe `Object` que vous allez spécialiser (n'oubliez pas l'annotation `@Override`). Un IDE peut vous aider à autogénérer ce test d'égalité (cherchez *Generate* dans votre IDE). Regardez également comment fonctionne la méthode `Objects.equals` (notez le `s` à la fin de `Objects`).

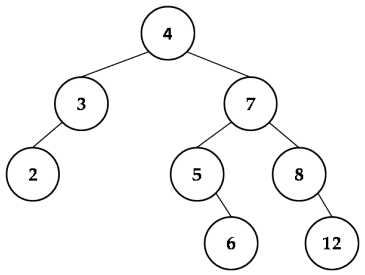
1.6 Recherche d'une donnée

Ajoutez dans votre classe `BSTNode` une méthode récursive `boolean search(T data)` où `T` est votre type générique et qui retourne `true` si la donnée `data` est contenu dans l'arbre, `false` sinon.

1.7 Insertion d'une donnée

Ajoutez dans votre classe `BSTNode` une méthode récursive `void insert(T data)` qui permet d'insérer un nœud contenant la valeur `data` dans l'arbre.

Si une variable `myTree` référence l'arbre binaire de recherche donné en exemple, `myTree.insert(6)` doit modifier l'arbre afin d'obtenir :



Avant d'implémenter, réfléchissez à la manière dont l'algorithme va procéder. Notez que le nouvel arbre doit toujours respecter la propriété d'arbre binaire de recherche.

1.8 Affichage des données

Ajoutez dans votre classe `BSTNode` une méthode récursive `void display()` qui affiche en console les valeurs contenues dans l'arbre par ordre croissant.

Testez votre méthode sur l'arbre donné en exemple.

1.9 Affichage des données d'un intervalle donné

Ajoutez dans votre classe `BSTNode` une méthode récursive `void display(T bi, T bs)` qui affiche en console les valeurs contenues dans l'arbre et dans l'intervalle $[bi, bs]$ par ordre croissant. Il vous est demandé de ne pas parcourir des nœuds « inutiles ».

Testez votre méthode sur l'arbre donné en exemple.

1.10 Suppression d'une feuille et d'un nœud avec un fils

Lorsque nous souhaitons supprimer un nœud, il y a trois possibilités. Soit ce nœud est une feuille, soit il a un fils, soit il en a deux.

Lorsque le nœud est une feuille, il n'y a aucun traitement supplémentaire à effectuer, nous pouvons simplement le supprimer. La conservation des propriétés de l'arbre binaire de recherche est évidente dans ce cas. Commencez par implémenter une méthode récursive `void remove(BSTNode<T> parent,`

`T data)` qui recherche la donnée `data` et qui supprime le nœud qui la contient uniquement si le nœud en question est une feuille. Si ce n'est pas le cas, cette méthode ne fait rien pour le moment. Le paramètre `parent` est le père du nœud que nous sommes en train de traiter dans l'itération actuelle. Pour simplifier votre travail, imaginez-vous que vous ne devez jamais traiter le cas où votre racine n'a pas de père (c.-à-d. le cas où parent est `null`). Vérifiez que votre méthode fonctionne avant de passer à l'étape suivante.

Lorsque le nœud a un unique fils, nous pouvons simplement remplacer ce nœud par son fils. Essayez cette opération sur l'exemple disponible dans ce document. Ensuite, commencez par justifier le fait que ce remplacement conserve les propriétés de l'arbre binaire de recherche pour vous en convaincre (une simple preuve informelle suffit). Enfin, modifiez votre méthode `remove` pour qu'elle prenne en compte ce nouveau cas.

Le cas où le nœud a deux fils est légèrement plus complexe. Si nous remplaçons le nœud par un de ses fils, les propriétés de l'arbre de binaire de recherche ne sont plus respectées. Essayez cette opération sur l'exemple dans le document pour vous en convaincre. Il ne vous est pas ici demandé d'implémenter cette opération, mais n'hésitez pas, si vous souhaitez entrainer vos capacités algorithmiques, à réfléchir à une possible solution (réfléchissez à comment obtenir l'élément minimum d'un arbre binaire de recherche).