Algorithmique
TP 3 : Arbres

L'exercice de ce TP doit être déposé sur l'espace "Rendu TP3".

Les parcours d'arbre

Étant donné un arbre, un *parcours* de cet arbre est un algorithme qui visite tous ses nœuds (ou sous arbres) et fait une opération (par exemple un affichage) avec les éléments qui y sont stockés.
Plusieurs ordres de parcours sont possibles : on peut parcourir un arbre "*en largeur*" (niveau par niveau), ou bien "*en profondeur*" (on finit d'explorer un sous-arbre avant de regarder son "frère"). Pour ce TP, nous nous intéressons à des parcours récursifs, qui sont des cas particuliers de parcours en profondeur.
On distingue notamment 3 grands types de parcours en profondeur correspondant à 3 ordres de traitement différents. Supposons que l'opération soit effectuée par la fonction `visite()` (qui travaille avec l'élément de l'arbre). Nous pouvons alors écrire ces algorithmes comme suit :

1. parcours préfixe :

```
parcours()
    visite()
    pour tout enfant f, f.parcours()
```

2. parcours postfixe :

```
parcours()
    pour tout enfant f,  f.parcours()
    visite()
```

3. parcours infixe (arbres binaires seulement) :

```
parcours()
    si on a un enfant gauche, g, g.parcours()
    visite()
    si on a un enfant droite, d, d.parcours()
```

Remarque : dans les algorithmes concrets qui se basent sur un parcours en profondeur, les fonctions `parcours` et `visite` peuvent retourner une valeur, auquel cas les valeurs retournées par la visite du nœud courant et les parcours de ses enfants doivent être "fusionnées" pour produire le résultat du parcours du nœud courant :

```
parcours()
    l = liste des retours de f.parcours() pour tous les enfants f
    retourner fusion(visite(), l)
```

1 Programmer une structure d'arbre binaire

On utilise pour cet exercice (voir aussi le fichier `Arbre.java`) un arbre binaire dont chaque nœud contient un entier.

1. Complétez l'implémentation des deux constructeur de la classe `Arbre`, qui représente un nœud de l'arbre. Pour représenter l'absence d'un enfant, on a un booléen `empty` qui permet de savoir si l'arbre est vide ou non. On fournit le constructeur `public Noeud()` qui crée l'arbre vide. Pensez à initialiser les enfants de vos arbres s'ils ne sont pas fournis dans les paramètres de votre constructeur.

2. Implémentez la méthode `isLeaf` qui vérifie si l'arbre est une feuille (c'est à dire que ses enfants sont vides).

Pour les fonctions récursives sur les arbres, il faut en général gérer le cas de base qui correspond à l'arbre vide (ou parfois également une feuille), puis faire la partie récursive qui s'appellera sur les enfants gauche et droit de l'arbre.

3. Implémentez la méthode récursive `equals` qui vérifie si deux arbres sont égaux. Pour être égaux, ils doivent contenir la même valeur, leurs enfants gauches doivent être égaux, tout comme leurs enfants droits.

4. Codez 3 méthodes récursives réalisant respectivement les parcours préfixe, postfixe et infixe de l'arbre. La fonction `visite` sera simplement un affichage de la valeur contenue dans le nœud. Testez soigneusement ces méthodes car elles vous seront utiles pour les questions qui suivent.

5. En vous inspirant des parcours précédents, implémentez la méthode `nombreNoeuds` qui compte le nombre de nœuds présents dans l'arbre. Les noeuds sont tous les sous arbres non vides présents.

6. À l'aide d'un parcours récursif, codez la méthode `hauteur`, qui calcule la hauteur de l'arbre. Petit rappel : la hauteur d'un arbre est la longueur du plus long chemin allant de la racine vers une feuille (une feuille a pour hauteur 0).

7. Complétez l'implémentation de la méthode `toString` retournant une chaîne de caractères représentant l'arbre.

8. Toujours en vous inspirant des parcours de l'arbre (et donc à l'aide d'une fonction récursive), codez la méthode `somme`, qui calcule la somme des valeurs de tous les nœuds de l'arbre.

9. Comme pour l'exercice précédent, implémentez la méthode `sommeFeuilles`, qui calcule la somme des valeurs de toutes les feuilles de l'arbre.

2 Arbre binaire de recherche

Un arbre binaire de recherche est un arbre tel que, pour chaque noeud, les valeurs à gauche sont plus petites que la valeur du noeud qui est elle-même plus petite que les valeurs à

droite. Quand on ajoute un une valeur dans un arbre binaire, on parcourt l'arbre en passant à gauche si on est plus petit que la valeur, à droite si on est supérieur ou égal. Quand on atteint `null`, on y place un nouveau noeud avec la valeur qu'on souhaite ajouter.

1. Implémentez la méthode `ajoutVal` qui ajoute la valeur entière dans l'arbre comme expliqué.

2. Implémentez un constructeur qui prend en paramètre un tableau d'entiers et renvoie l'arbre binaire de recherche correspondant à l'ajout des valeurs du tableau une après l'autre.

3. Testez votre constructeur sur différents tableaux, en affichant le `toString` de vos arbres. Qu'observez vous ?

4. (Facultatif) Implémentez la méthode `isDeRecherche` qui vérifie si l'arbre binaire est bien de recherche. Vous pouvez créer des méthodes supplémentaires pour vous aider.