

## TP ALGO n°3

### Arbres

#### Exercice 1 :

##### Partie I : Représentation d'un arbre binaire de recherche

On veut représenter un arbre binaire de recherche en mémoire. Pour cela on crée une classe *ArbreRech* qui contient:

- Trois variables d'instance
  - Info : un entier
  - filsGauche : un nœud (c'est à dire un Arbre de recherche)
  - filsDroit : un nœud (c'est à dire un Arbre de recherche)
- Un premier constructeur
- Un second constructeur
- Une méthode qui insère une valeur passée en paramètre dans l'arbre

1) Ecrire le premier constructeur dont la signature est la suivante:

```
public ArbreRech(int info)
```

2) Ecrire le second constructeur dont la signature est la suivante:

```
public ArbreRech(int info, ArbreRech filsGauche, ArbreRech filsDroit)
```

3) Ecrire la méthode insérer qui insère l'info passée en paramètre dans l'arbre

```
public static ArbreRech inserer(int x, ArbreRech a)
```

4) Générer *toString()*

5) Dans la classe *Main* (mais pas dans la fonction main !!), ajouter :

```
public static ArbreRech saisieArbre() {
    Scanner entree=new Scanner(System.in);
    ArbreRech arbre= null;
    int x;
    do {
        System.out.print("\ninfo du noeud : ");
        x = entree.nextInt();
        if (x != -1) {
            arbre= ArbreRech.inserer(x, arbre);
        }
    }while (x!=-1);
    return arbre;
}
```

Vous pouvez maintenant tester vos fonctions en saisissant des arbres à l'aide de

cette fonction (exemples d'arbres dans le td).

## Partie II : Opérations simples sur les arbres binaires de recherche

- 1) Écrivez une méthode *nbNoeuds* pour calculer le nombre de nœuds de l'arbre.

```
Public static int nbNoeuds (ArbreRech rac)
```

- 2) Écrivez une méthode *hauteur* pour calculer la hauteur de l'arbre c'est-à-dire le nombre d'arcs sur le plus long chemin dans l'arbre. Pourquoi cette méthode est-elle une méthode de classe?

```
public static int hauteur ( ArbreRech arbre )
```

- 3) Écrivez une méthode de parcours en préfixe de l'arbre.

```
public static void parcoursPréfixe (ArbreRech a)
```

- 4) Écrivez une méthode de parcours en infixe de l'arbre et une méthode de parcours en postfixe de l'arbre.

## Partie III : Utilisation des arbres binaires de recherche

- 1) Écrivez une méthode *chercher* qui cherche si l'info passée en paramètre est dans l'arbre.

```
public static ArbreRech chercher (int x, ArbreRech a)
```

- 2) Écrivez une méthode qui supprime l'info passée en paramètre de l'arbre. Vous devrez probablement écrire des méthodes additionnelles, du type *supprimer\_noeud*.

```
public static ArbreRech supprimer (int x, ArbreRech a)
```

- 3) Écrivez une méthode qui effectue une rotation gauche de l'arbre passé en paramètre.  
Écrivez une méthode qui effectue une rotation droite.

```
Public static ArbreRech rotationG (ArbreRech a)
```

- 4) Écrivez une méthode qui vérifie qu'un arbre binaire est un arbre AVL.

```
public static boolean équilibre (ArbreRech arbre)
```

- 5) On souhaite écrire une méthode permettant d'équilibrer un arbre binaire de recherche.  
On s'appuie sur l'algorithme suivant:

**Algorithme.** Soit  $A$  un arbre,  $G$  et  $D$  ses sous-arbres gauche et droit.  
On suppose que  $|h(G)-h(D)|=2$ . Si  $h(G)-h(D)=2$ , on fait une rotation droite, mais précédée d'une rotation gauche de  $G$  si  $h(g) < h(d)$  (on note  $g$  et  $d$  les sous-arbres gauche et droit de  $G$ ). Si  $h(G)-h(D)=-2$  on opère de façon symétrique.

Ecrire la méthode *équilibrer*.

```
public static ArbreRech équilibrer (ArbreRech a)
```

- 6) Quand on fait un ajout ou une suppression dans un arbre binaire on peut ne pas obtenir un arbre équilibré. Ré-écrire les méthodes d'insertion et de suppression précédentes, en rééquilibrant l'arbre à chaque étape.

## Exercice 2 \*

La résolution de problèmes par exploration de l'espace des solutions est une problématique classique en intelligence artificielle. On s'intéresse à des problèmes dont la solution s'exprime comme une séquence d'actions. Le principe consiste à explorer (en le construisant au fur et à mesure) l'arbre des solutions possibles où, partant d'un état courant, on construit les états suivants en énumérant les actions possibles et leur effet sur l'état.

On considère le problème de la sortie d'un labyrinthe, c'est à dire aller d'une case de Départ à une case Sortie. La classe MazeGenerator vous est donnée, elle permet de créer un labyrinthe avec la commande:

```
MazeGenerator labyrinthe = new MazeGenerator( HEIGHT: 11, WIDTH: 55 );
```

Si vous faites plusieurs *run* ce sera à chaque fois un labyrinthe différent.

Vous pouvez afficher le labyrinthe créé avec la commande:

```
labyrinthe.printMaze();
```

Vous pouvez connaître les positions suivantes atteignables en un déplacement élémentaire à partir d'une position donnée avec la méthode *successeurs* du labyrinthe (classe MazeGenerator)

```
Queue<int[]> queue = labyrinthe.successeurs(x, y);
```

où (x,y) est la position courante et où *queue* est une liste de listes (de 2 éléments qui sont les coordonnées newx et newy d'une nouvelle position atteignable à partir de (x,y) en un déplacement).

Vous pouvez définir une classe MainMaze avec le code suivant, dans laquelle il est fait appel à la classe DFS pour trouver la solution de coût minimal.

```

public class MainMaze {
    public static void main(String[] args) {
        // Créer un labyrinthe de 10 lignes et 10 colonnes
        MazeGenerator labyrinthe = new MazeGenerator( HEIGHT: 11, WIDTH: 55); // Utiliser des dimensions impaires

        // Afficher le labyrinthe généré
        labyrinthe.printMaze();

        // Créer une instance de RechercheBFS avec le labyrinthe généré
        DFS recherche = new DFS(labyrinthe);

        // Lancer la recherche à partir de la position de départ (1, 1)
        if (recherche.rechercherSortie( startX: 1, startY: 1)) {
            System.out.println("Sortie trouvée !");
        } else {
            System.out.println("Pas de sortie trouvée.");
        }

        // Afficher le labyrinthe après la recherche
        recherche.afficherLabyrinthe();
    }
}

```

La classe *DFS* doit avoir une méthode *rechercheSortie* qui renvoie vrai ou faux suivant qu'une solution est trouvée. La classe *DFS* possède une méthode *afficheLabyrinthe* qui affiche le labyrinthe avec le chemin trouvé.

- 1) Ecrivez la class *DFS* qui implémente la résolution du problème en réalisant une recherche en profondeur d'abord
- 2) L'algorithme *DFS* trouve-t-il toujours la meilleure solution ? Expliquez pourquoi