Structures de données
TD7 Arbres Binaires équilibrés

1 Exercice 1 AVL

Arbre binaire de recherche dont la différence entre la hauteur du fils gauche et la hauteur du fils droit est au plus 1 (ou -1). Pour respecter cette contrainte on utilise des rotations.

- **doubleRotGauche :** rotDroite(r→fd) puis rotGauche(r) pour corriger déséquilibre du fils gauche du fils droit
- **doubleRotDroite :** rotGauche(r→fg) puis rotDroite(r) pour corriger déséquilibre du fils droit du fils gauche

Après chaque rotation, il faut mettre à jour la hauteur du fils (gauche si rotGauche sinon droit) puis la hauteur de la racine.

Pour chaque noeud on calcule et stocke sa hauteur : $h = 1 + \max(fg, fd)$. La hauteur d'un nouveau noeud est intialisée à 0, un noeud qui n'existe pas a une hauteur de **-1**. La hauteur d'un noeud est mise à jour après modification d'un fils.

2 Exercice 2 ARN

Une suppression dans un AVL peut entraîner un grand nombre de rotations, l'arbre rouge noir (ou bicolor) permet de palier cela.

Quand on insère un noeud on le colore en rouge puis on vérifie :

1. La racine est toujours noire
2. Il y a le même nombre de noeuds noirs sur tous les chemins
3. Les files d'un noeud rouge sont noirs (s'ils existent)

Si ce n'est pas le cas alors il faut recolorier des noeuds.

Les points 2 et 3 assurent que le plus grand chemin est au plus deux fois plus grand que le plus petit chemin. Ainsi comme tout chemin contient le même nombre de noeuds noirs :

— **Le plus petit chemin** : longueur n contient n noeuds noirs
— **Le plus grand chemin** : longueur $2n$ contient n noeuds noirs et au maximum n noeuds rouges

Dans cet arbre on conserve un pointeur vers son parent, ce pointeur est NULL pour la racine. La page française de wikipédia est très détaillée, à voir pour l'insertion : `https://fr.wikipedia.org/wiki/Arbre_bicolore#Insertion`

```
int verifARNcur(ARN* tree, int cur, int* n)
{
    // Si c'est une feuille
    if(tree==NULL) {
        if(*n == -1) {
            *n = cur;
            return 1;
        } else {
            return current == *n;
        }
    }
    // Sinon on verifie que c est un ABR
    if( (tree->fg!=NULL && tree->fg->valeur > tree->valeur) ||
        (tree->fd!=NULL && tree->fd->valeur < tree->valeur) ) {
            return 0;
    }

    if(tree->co == noir) { // si noir
       return verifARNcur(tree->fg, cur+1, hn) && verifARNcur(tree->fd, cur+1, hn);
    } else { // sinon rouge
        if( (tree->fg!=NULL && tree->fg->co == rouge)
           || (tree->fd!=NULL && tree->fd->co == rouge)) {
```

```
            return 0;
        } else {
            return verifARNcur(tree->fg, cur, hn) && verifARNcur(tree->fd, cur, hn);
        }
    }
}
int verifARN(ARN* tree)
{
    if(tree==NULL) return 1;
    else if(tree->co == rouge) return 0;
    else {
        int n = -1;
        return verifARNrec(tree, 0, &n);
    }
}
```

Que ce soit pour l'AVL ou l'ARN, les fonctions d'insertion, de recherche et de suppression sont en $O(h) = O(\log_2 n)$. L'AVL est plus équilibré mais l'ARN est plus économe.

Vous pouvez tester visuellement ces arbres sur ces pages : `https://www.cs.usfca.edu/~galles/visualization/RedBlack.html` `https://www.cs.usfca.edu/~galles/visualization/AVLtree.html` (les autres structures vues en cours doivent être sur le site)

3 Autre

3.1 Opérateur ternaire

```
int toto;
if(condition) {
    toto = 1;
} else {
    toto = 0;
}

// peut etre simplifie en :
//         (condition) ? valeur si vraie : valeur si faux
int toto = (condition) ? 1 : 0;
```