# TP n°3 Arbres Binaire de Recherche

## Introduction

Ce TP est directement associé au contenu de la séance de TD : il concerne la réalisation d’opérations de base s’appliquant aux arbres binaires de recherche contenant des valeurs entières qui doivent toutes être différentes.

La structure générale de l'application est fournie et elle est présentée succinctement au début de ce document, les travaux à réaliser étant quant à eux, présentés dans la suite.

## Présentation générale de l'application fournie

### Structure de l’application

L’application à compléter est réalisée de façon modulaire ; chaque module étant constitué d’un fichier de spécification et du fichier d’implémentation associé. Elle est constituée des fichichiers suivants :

### Le fichier principal : tp3.c

Le fichier tp3.c contient la fonction principale qui se chargera d’effectuer le test des différentes fonctions de traitement des arbres binaire de recherche.

### Le fichier abr.c

Ce fichier contient l’implémentation des fonctions de gestion d’un arbre binaire de recherche. C’est essentiellement dans ce fichier que vous interviendrez.

### Le fichier testABR.c

Il contient l’implémentation des différentes fonctions de test qui seront appelées dans la fonction principale présente dans tp3.c.

**Pour chaque fonction réalisée, vous indiquerez dans le compte-rendu quel est le principe de base utilisé pour la définir et vous insérerez sa définition ainsi que les résultats des tests effectués.**

## Première partie : opérations de base

Quelques fonctions de base ont déjà été définies, comme la fonction :

noeudABR\_t \* creer1Noeud (elt\_t e, const noeudABR\_t \*g, const noeudABR\_t \*d) ;

C’est la fonction présentée lors du TD, qui crée un nœud et initialise ses trois composantes avec les valeurs des paramètres transmis.

Complétez la fonction void afficherABR(const noeudABR\_t \* a) qui permet d’afficher le contenu de l’arbre désigné par le pointeur a, de façon à ce que les valeurs contenues soient affichées par valeur croissante. Un saut de ligne sera effectué à la fin de l’affichage. Le test de cette fonction est réalisé par une fonction déjà écrite dans le fichier testAbr.c : void test\_afficherABR(void);

Donnez dans le compte-rendu la représentation graphique du dernier arbre construit dans la fonction de test.

Complétez la fonction void afficherInverseABR(const noeudABR\_t \* a) qui permet d’afficher le contenu de l’arbre désigné par le pointeur a, de façon à ce que les valeurs contenues soient affichées par valeur décroissante. Un saut de ligne sera effectué à la fin de l’affichage. Le test de cette fonction est réalisé par une fonction déjà écrite dans le fichier testAbr.c : void test\_afficherInverseABR(void);

Écrire de façon récursive la fonction permettant d’insérer un élément dans un arbre binaire de recherche. Cette fonction ne réalisera effectivement l’insertion que si l’élément à insérer était **absent de l’arbre** : après insertion il ne doit pas y avoir plusieurs valeurs identiques rangées dans l’arbre. Le test de cette fonction est réalisé par une fonction déjà écrite dans le fichier testAbr.c : void test\_insererABR(void) ;

Développez la fonction permettant de vérifier qu’un élément donné est présent dans un arbre binaire de recherche : spécifier son prototype dans le fichier abr.h, l’implémenter dans le fichier abr.c et réaliser la fonction de test dans testAbr.c

Écrire la fonction calculant la hauteur, ainsi que la fonction calculant le nombre de nœuds d’un arbre binaire de recherche, puis compléter la fonction test\_insererABR() de manière à afficher la hauteur et le nombre d’éléments des arbres construits.

Rappeler dans le compte-rendu les expressions donnant les nombres minimum et maximum de nœuds d’un arbre de hauteur h, ainsi que celles donnant la hauteur minimale et maximale d’un arbre binaire composé de n nœuds.

## Deuxième partie : copie et suppression d’un ABR

Développez la fonction chargée de supprimer un arbre binaire de recherche en récupérant tous les espaces de mémoire qui lui avaient été alloués. Cette fonction retournera la valeur NULL : noeudABR\_t \* supprimerABR(noeudABR\_t \* a); Il n’y aura pas de tests spécifiques à réaliser pour cette fonction ; elle est appelée dans les fonctions de test, ou dans la fonction principale dès qu’un arbre binaire n’est plus utile.

Développez de manière récursive la fonction noeudABR\_t \* copierABR (noeudABR\_t \* arbre) qui crée une copie d’un arbre binaire de recherche, puis écrire et exécuter la fonction test\_copieABR() de façon à valider cette fonction. Cette fonction permettra de s’assurer que les copies d’un arbre vide, d’un arbre d’un seul nœud et d’un arbre de plusieurs nœuds s’effectuent correctement. Les arbres construits et leurs copies doivent être détruits en fin d’exécution de la fonction de test.

Développez la fonction permettant de vérifier que deux arbres binaires sont égaux (même forme et mêmes valeurs) : int egalABR (const noeudABR\_t \* arbre1, const noeudABR\_t \* arbre2);

Écrire et tester la fonction permettant de vérifier le bon comportement de cette fonction.

Développer la fonction permettant de supprimer le nœud contenant une valeur donnée dans un arbre binaire désigné par le pointeur sur sa racine :  noeudABR\_t \* supprimer1EltABR(elt\_t e, noeudABR\_t \* pRac);  Le test de cette fonction est réalisé par une fonction déjà écrite dans le fichier testABR.c : void test\_supprimer1EltABR(void);

## Troisième partie

L’objectif est maintenant de générer la représentation graphique d’un arbre binaire de recherche en s’appuyant sur l’un des outils de la suite Graphviz. Il faut pour cela produire le fichier au format .DOT puis exécuter la commande **dot** pour générer l’image PNG .

Quel type de parcours d’un arbre binaire a permis de produire le fichier .dot ci-dessous ?

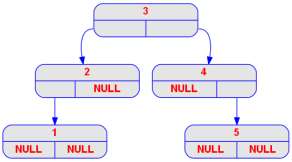
digraph Abr {

node [

fontname = "Arial bold"

fontsize = "14"

fontcolor = "red"

 style = "rounded, filled"

shape = "record"

fillcolor = "grey90"

color = "blue"

width = "2"

]

edge [ color = "blue" ]

3 [label = "{<c> 3 | { <g> | <d>}}"];

3:g -> 2;

2 [label = "{<c> 2 | { <g> | <d>}}"];

2 [label = "{<c> 2 | { <g> | <d> NULL}}"];

2:g -> 1;

1 [label = "{<c> 1 | { <g> | <d>}}"];

1 [label = "{<c> 1 | { <g> NULL | <d> NULL}}"];

3:d -> 4;

4 [label = "{<c> 4 | { <g> | <d>}}"];

4 [label = "{<c> 4 | { <g> NULL | <d>}}"];

4:d -> 5;

5 [label = "{<c> 5 | { <g> | <d>}}"];

5 [label = "{<c> 5 | { <g> NULL | <d> NULL}}"];

}

Développez la fonction genererDotPng() qui prend en charge la production dans un fichier préalablement ouvert, des lignes décrivant les nœuds et les arcs d’un arbre – lignes de couleur bleue dans l’exemple ci-dessus –.

Cette fonction est utilisée par la fonction creerDotABR() qui crée le fichier .dot, y copie le préambule - digraph Abr { ... -, puis génère le fichier .png.

Le test de cette fonction est réalisé par la fonction test\_creerDotPng().