# Programmation C
# TP n° 7 : Arbres binaires de recherche

On considère les arbres binaires dont les nœuds sont étiquetés par des entiers. Rappelons lorsqu'un arbre est non vide et lorsque son sous-arbre gauche (resp. droit) est non vide, ce dernier est appelé *fils gauche* (resp. *droit*) de la racine. Un arbre sans fils droit ni fils gauche est appelé une *feuille*.
Un *arbre binaire de recherche* (ABR) est un arbre dont *chaque* nœud vérifie la propriété suivante (propriété des ABR) :

- Soit $n$ l'étiquette de ce nœud.
  Toutes les étiquettes du sous-arbre gauche du nœud sont strictement inférieures à $n$ ;
  Toutes les étiquettes du sous-arbre droit du nœud sont strictement supérieures à $n$.

Par exemple, trivialement, l'arbre vide est un ABR. L'arbre représenté ci-dessous est un ABR, de même que chacun de ses sous-arbres. Noter que la propriété des ABR implique que chaque valeur de l'arbre ne peut y apparaître qu'une seule fois.

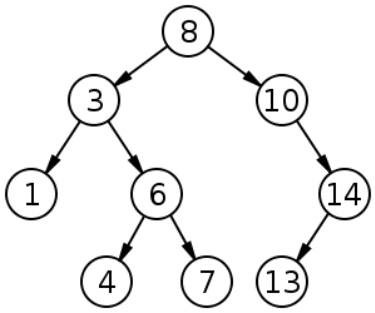


FIGURE 1 – Exemple d'arbre binaire de recherche.

Pour représenter les arbres en C, nous utiliserons les définitions suivantes :

```c
typedef struct node node;
typedef node *tree;
struct node{
  int val;
  tree left;
  tree right;
};
```

Par *arbre*, on entend "pointeur vers `struct node`". Par convention, l'arbre vide est représenté par le pointeur `NULL`. Un arbre non vide est représenté par un pointeur vers la structure représentant son nœud racine, le champ `val` de cette structure représentant son étiquette, les champs `left` et `right` représentant ses sous-arbres gauche et droit.

### Exercice 1 : Affichage

Écrire une fonction `void print_abr(tree t)` qui affiche dans l'ordre croissant les valeurs stockées dans l'ABR `t` (appel récursif sur le sous-arbre gauche, affichage de la racine, appel récursif sur le sous-arbre droit).

### Exercice 2 : Insertion d'une valeur et libération de la mémoire

Dans cet exercice comme dans les suivants, tout `malloc` devra être suivi d'un `assert` vérifiant le succès de l'allocation.

1. Écrire une fonction `tree leaf(int val)` allouant un arbre réduit à une feuille (c'est-à-dire un nœud dont les champs `left` et `right` sont `NULL`) étiquetée par `val`.
2. Écrire une fonction `tree insert_abr(tree t, int val)` ajoutant la valeur `val` à un ABR `t` si elle ne s'y trouve pas déjà, et renvoyant l'arbre résultant.

   Dans le cas d'un ajout, la valeur sera encapsulée dans une nouvelle feuille qui remplaçera l'un des sous-arbres vides de `t` de manière à ce que l'arbre résultant soit encore un ABR. Si la valeur se trouve déjà dans l'arbre, ce dernier sera simplement renvoyé tel quel. Servez-vous de la récurrence et de la fonction précédente.
3. Écrire une fonction `void free_tree(tree t)` qui libère **toute** la mémoire allouée pour un arbre.

### Exercice 3 : Recherche

1. Écrire une fonction `tree max_abr(tree t)` renvoyant un pointeur vers le nœud de l'ABR `t` étiqueté par la plus grande de ses valeurs, ou `NULL` si `t` est un arbre vide.
2. Symétriquement écrire `tree min_abr(tree t)` renvoyant un pointeur vers le nœud de l'ABR `t` étiqueté par la plus petite de ses valeurs, ou `NULL` si `t` est un arbre vide.
3. Écrire une fonction `tree search_abr(tree t, int val)` renvoyant un pointeur vers le nœud de l'ABR `t` étiqueté par `val` si ce nœud existe, et `NULL` sinon (la recherche sera bien sûr optimisée en tenant compte du fait que `t` est un ABR).

### Exercice 4 : Vérification de la propriété des ABR

1. Écrire une fonction `int check_abr(tree t)` renvoyant `1` si `t` est un ABR, et `0` sinon. Cette fonction sera récursive et utilisera à chaque étape les fonctions de recherche de maximum et minimum de l'exercice précédent.
2. Vérifier que la fonction `insert_abr` implémentée précédemment renvoie bien un ABR.

### Exercice 5 : Suppression d'une valeur

1. Écrire une fonction `tree delete_abr(tree t,int val)` supprimant la valeur `val` de l'ABR `t` si celle-ci s'y trouve, et renvoyant l'arbre résultant – celui-ci doit être encore un ABR. Si la valeur ne se trouve pas dans l'arbre, celui-ci sera simplement renvoyé tel quel. On distinguera les cas suivants :
   - le cas ou l'arbre est vide (retour de `NULL`) ;
   - le cas où la valeur à supprimer est strictement plus petite que l'étiquette de la racine (suppression par récurrence dans le sous-arbre gauche) ;
   - le cas où la valeur à supprimer est strictement plus grande que l'étiquette de la racine (suppression par récurrence dans le sous-arbre droit) ;
   - le cas où la valeur à supprimer est à la racine, mais cette racine est une feuille (libération de la racine, retour de `NULL`) ;
   - le cas où la valeur à supprimer est à la racine, mais cette racine n'a qu'un seul fils (libération de la racine, retour du fils) ;

– enfin, le cas où la valeur à supprimer est à la racine, et où cette racine a deux fils : on choisit l'un de deux fils, par exemple le gauche ; on échange l'etiquette de la racine et l'étiquette du nœud étiqueté par la plus grande des étiquettes du fils gauche ; on supprime récursivement la valeur dans le fils gauche.