Programmation C
TP n° 7 : Arbres binaires

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*.

1 Arbres binaires en C

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

```
typedef struct node node;
struct node {
  int val;
  node *left;
  node *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.

La Figure 1 montre la manière dont un arbre peut être représenté en mémoire. Les quatre structures à droite avec leurs adresses (fictives) représentent les quatre noeuds de l'arbre. L'ordre des structures en mémoire peut être quelconque : il ne dépend que de l'ordre des allocations et du comportement de `malloc`. L'arbre lui-même est représenté par la valeur d'adresse de sa structure racine (`0x300`), de type pointeur vers `struct node`.

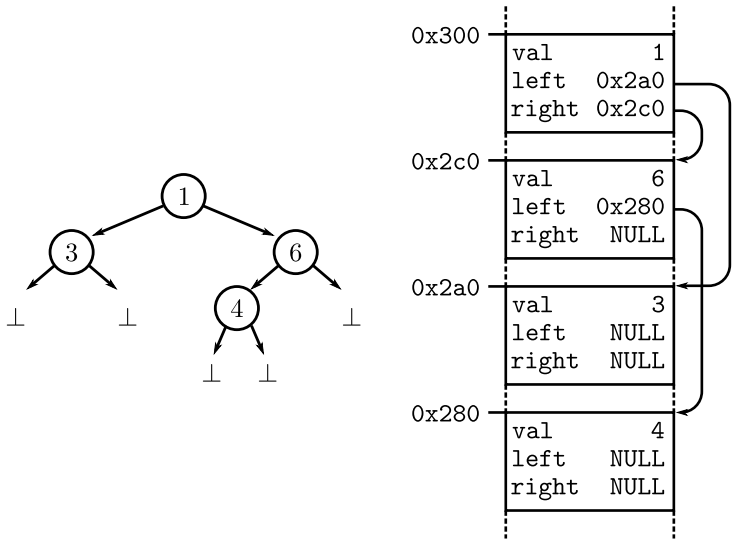


FIGURE 1 – Un arbre binaire et sa représentation en mémoire.

Vous trouverez sur Moodle dans le fichier `afficheur.c` un afficheur d'arbre : servez-vous de la fonction `pretty_print` pour tester les fonctions de cet énoncé.

Exercice 1 : Constructeur d'arbre

Écrire une fonction `node *cons_tree(int val, node *left, node *right)`. Cette fonction suppose que `tl`, `tr` sont deux arbres déjà construits, éventuellement de valeur `NULL`. La fonction doit allouer par `malloc` une nouvelle structure `node` (vérifier par `assert` le succès de cette allocation), initialiser ses champs `val`, `left`, `right` aux valeurs respectives des paramètres, puis renvoyer l'adresse d'allocation. L'arbre binaire de la Figure 1 peut par exemple être construit avec cette fonction de la manière suivante :

```
node *t;
t = cons_tree (1, cons_tree (3, NULL, NULL),
                  cons_tree (6, cons_tree (4, NULL, NULL), NULL));
```

Exercice 2 : Destructeur d'arbre

La récurrence permet de parcourir de manière naturelle une structure arborescente. Par exemple, pour libérer la totalité de la mémoire allouée pour un arbre, on peut effectuer le traitement suivant :

1. Si l'arbre est vide, il n'y a rien à faire.
2. Sinon :
   (a) On libère (récursivement) la mémoire allouée pour son sous-arbre gauche.
   (b) On libère (récursivement) la mémoire allouée pour son sous-arbre droit.
   (c) On libère la mémoire allouée pour son noeud racine.

Écrire une fonction `void free_tree(node *t)` libérant la totalité de la mémoire allouée pour l'arbre `t`.

Exercice 3 : Récurrence avec retour de valeur

Une fonction récursive effectuant un parcours d'arbre peut extraire de l'information de cet arbre et la renvoyer à l'appelant. Par exemple, la taille d'un arbre exprimée en nombre de noeuds peut être calculée de la manière suivante :

1. Si l'arbre est vide, on renvoie zéro.
2. Sinon :
   (a) On calcule (récursivement) la taille de son sous-arbre gauche.
   (b) On calcule (récursivement) la taille de son sous-arbre droit.
   (c) On renvoie un plus la somme des deux tailles précédentes.

Écrire les fonctions suivantes. Par convention, la somme des étiquettes d'un arbre vide est zéro. La profondeur d'un arbre est la longueur de sa plus grande branche – *e.g.* 0 pour l'arbre vide, 3 pour l'arbre représenté à la Figure 1 :

- `int size_tree(node *t)`, renvoyant la taille de l'arbre `t` ;
- `int sum_tree(node *t)`, renvoyant la somme des étiquettes de l'arbre `t` ;
- `int depth_tree(node *t)`, renvoyant la profondeur de `t`.

2 Arbres binaires de recherche

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.

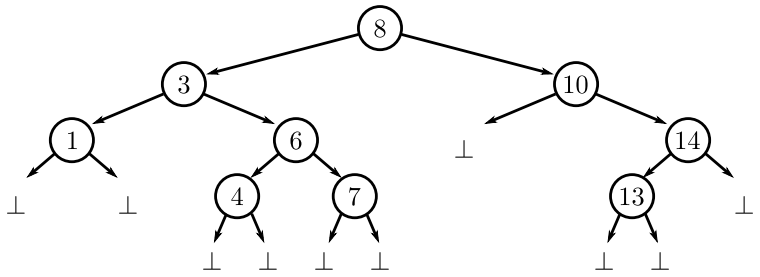


FIGURE 2 – Exemple d'arbre binaire de recherche.

Exercice 4 : Affichage

Écrire une fonction `void print_abr(node *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 5 : Insertion d'une valeur dans un ABR

Écrire une fonction `node *insert_abr(node *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 `cons_tree` pour construire la feuille.

Exemple 1. Si `t` représente l'ABR de la Figure 2, un appel de `insert_abr(t, 2)` renverra la valeur d'adresse de `t` (la racine de l'arbre n'est pas modifiée), `t` représentant l'arbre représenté à la Figure 3.

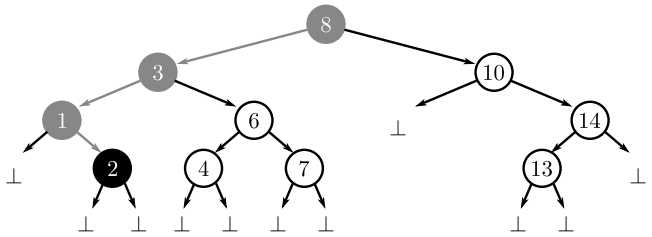


FIGURE 3 – Ajout de la valeur 2 dans l'ABR précédent.

Exemple 2. L'arbre de la Figure 3 devrait être en principe entièrement reconstruit par les instructions suivantes :

```
node *t = NULL;
int vals[10] = {8, 3, 1, 2, 6, 4, 7, 10, 14, 13};
for (int i = 0; i < 10; i++) {
  t = insert_abr (t, vals[i]);
}
```

Exercice 6 : Recherche

Écrire une fonction `node *search_abr(node *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 7 : Vérification de la propriété des ABR

1. Écrire une fonction `node *max_abr(node *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 `node *min_abr(node *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 `int check_abr(node *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 précédentes.

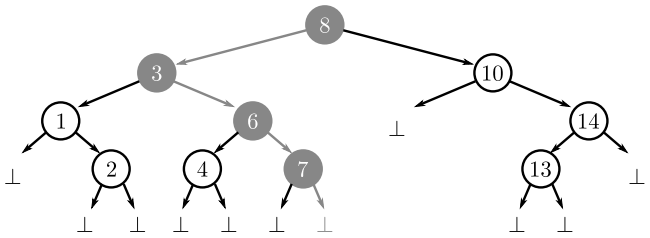
Exercice 8 : (Optionnel) Suppression de valeur

Écrire une fonction `node *delete_abr(node *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 :

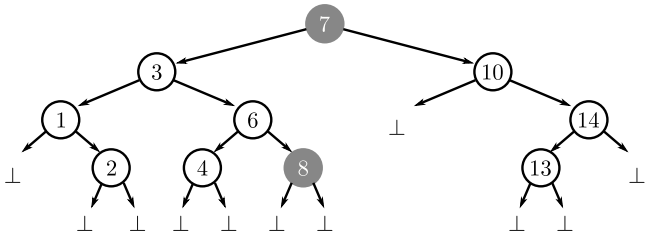
- le cas où l'arbre est vide (retour de `NULL`) ;
- le cas où la valeur à supprimer est strictement plus petite que l'étiquette de la racine (suppression par récurrence dans le sous-arbre gauche) ;
- le cas où la valeur à supprimer est strictement plus grande que l'étiquette de la racine (suppression 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.

Voici par exemple la manière dont la valeur 8 peut être supprimée dans l'ABR représenté à la Figure 3. Noter qu'après l'étape 2 et jusqu'à la suppression de 8 à gauche, l'arbre global n'est temporairement plus un ABR, mais le redevient après suppression de cette valeur. Son sous-arbre gauche est bien en revanche un ABR : l'appel récursif sur ce sous-arbre est donc légitime.

1. Recherche du max à gauche :



2. Échange de 8 et du max (sans recablage des nœuds) :



3. Suppression de 8 à gauche :