TP n°21 – Arbres binaires de recherche

On choisit le type suivant pour représenter les noeuds d'un arbre binaire de recherche en C.

```c
struct noeud {
 int cle; //etiquette
 struct noeud* gauche;
 struct noeud* droit;
 struct noeud* parent;
}
typedef struct noeud noeud;
```

Comme on veut un type mutable, un arbre binaire de recherche sera représenté par le type `noeud*`.

Remarque : Par rapport à la représentation habituelle d'un arbre, on a ajouté le champ `parent` pour facilement se déplacer dans l'arbre de manière itérative. Le champ `parent` pointe vers le noeud père de chaque noeud, excepté pour la racine, où c'est `NULL`. Cet ajout **n'est pas** nécessaire pour implémenter un ABR en C.

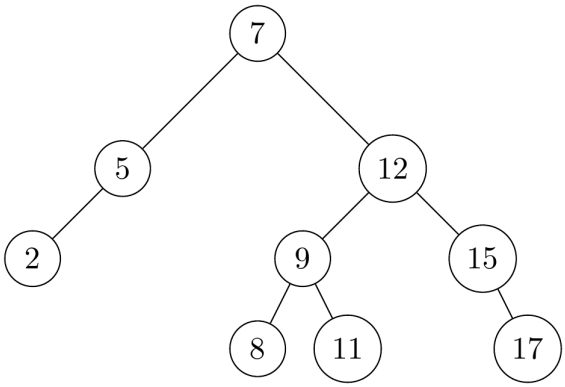
Autre remarque : On manipulera des `noeud*`, donc si `monnoeud` est un `noeud*` on accède à ses champs par `monnoeud->gauche`, `monnoeud->droite`, `monnoeud->parent`. De plus on créera évidemment tous les arbres de ce TP avec `malloc`.

1 Fonctions élémentaires

Un fichier vous est fourni avec une fonction `noeud* new_node(int etiquette)` qui crée un nouvel arbre binaire réduit à une feuille et une fonction `noeud* enracine(noeud* g, noeud* d, int etiquette)` qui crée un nouvel arbre binaire dont la racine est étiquetée `etiquette` et dont les sous-arbres sont `g` et `d` (Attention : `g` et `d` doivent être distincts et ne pas déjà être utilisés dans un autre arbre).

On vous fournit aussi une fonction `noeud* copie_arbre(noeud* a, noeud* parent)` qui permet de faire une copie d'un arbre a.

Le fichier contient enfin dans le `main` la définition d'un ABR `arbre_exemple` qui représente l'arbre suivant :



- **Q1.** Écrire une fonction `noeud* frere(noeud* n)` qui renvoie le frere d'un noeud ou `NULL` s'il n'existe pas.
- **Q2.** Écrire une fonction `noeud* oncle(noeud* n)` qui renvoie le frere du père d'un noeud ou `NULL` s'il n'existe pas.
- **Q3.** Écrire une fonction récursive `void free_arbre(noeud* n)` qui libère toute la mémoire attribuée à l'arbre de racine `n`.
- **Q4.** Que se passe-t'il si on applique la fonction précédente à un noeud qui n'est pas la véritable racine de l'arbre ?

2 Arbres binaires de recherche en C

- **Q5.** Écrire une fonction récursive `noeud* recherche(int etiquette, noeud* abr)` qui renvoie un pointeur vers le noeud de `abr` dont l'étiquette est `etiquette`, ou `NULL` si un tel noeud n'existe pas.
- **Q6.** Écrire une fonction qui cherche (et renvoie) de manière itérative le noeud d'étiquette minimale d'un arbre binaire de recherche. Il faut d'abord se rappeler où se trouve le noeud d'étiquette minimale de l'arbre.
- **Q7.** Écrire une fonction `noeud* ajout_noeud(noeud* abr, int etiquette)` qui ajoute un noeud d'étiquette `etiquette` et renvoir l'abr obtenu.
- **Q8.** Écrire une fonction qui supprime le noeud qui a une certaine étiquette. La technique est dans le cours, vous pouvez implémenter chaque cas séparément et les tester au fur et à mesure.

3 Implémentation d'un dictionnaire

- **Q9.** Modifier `struct noeud` pour pouvoir implémenter un dictionnaire. **Indices** : les tuples ça n'existe pas en C, on a déjà un champ clé et les valeurs seront des entiers.

- **Q10.** Définir un type pour le dictionnaire.
- **Q11.** Écrire une fonction qui crée un dictionnaire vide et une fonction qui supprime la mémoire allouée à un dictionnaire.
- **Q12.** Écrire une fonction qui recherche une clé dans un dictionnaire et renvoie sa valeur.
- **Q13.** Écrire une fonction qui modifie la valeur associée à une clé dans un dictionnaire.
- **Q14.** Écrire une fonction qui supprime une clé du dictionnaire.
- **Q15.** Écrire une fonction qui ajoute une clé (et sa valeur dans le dictionnaire). Cela peut nécessiter de mettre à jour les fonctions déjà implémentées.

4 Parcours infixe et arbres binaires de recherche

- **Q16.** Écrire une fonction qui réalise le parcours infixe d'un arbre binaire de recherche. Tester sur des exemples. Que remarque-t-on ?
- **Q17.** Prouver cette conjecture.
- **Q18.** En déduire un algorithme de tri des valeurs d'une liste (ou d'un tableau) qui utilise un ABR. Quelle est sa complexité ?