Dans ce TP, on se propose d'implémenter les arbres en C, et plus particulièrement les **arbres binaires de recherche**.

1 Arbres en C

On propose le modèle de structure suivant pour représenter un nœud.

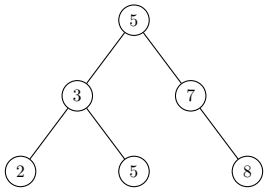
```
struct noeud {
  int cle;
  struct noeud* gauche;
  struct noeud* droit;
  struct noeud* parent;
}
typedef struct noeud noeud;
```

Un arbre est représenté par un pointeur sur la racine de cet arbre, et donc par le type `noeud*`. Un arbre vide est donc le pointeur `NULL`. Un nœud contient une clé, un pointeur vers son fils gauche et son fils droit, ainsi qu'un pointeur vers son nœud parent. La **racine** de l'arbre est le seul nœud dont le parent est `NULL`.

Un nœud d'un arbre n'est donc pas à proprement parler la racine de ce sous-arbre, car sont parent n'est pas nécessairement `NULL` : on dira que c'est sa **racine fictive**.

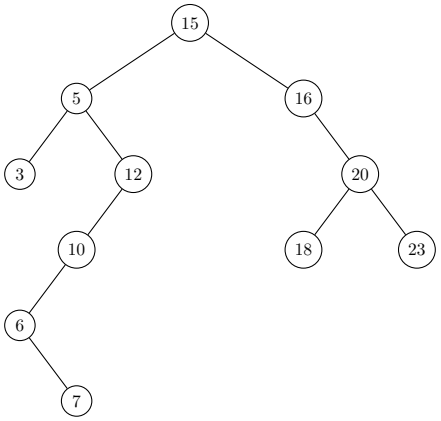
Exercice 1 (Création d'arbres en C) :

1. Écrire une fonction `noeud* feuille(int cle);` qui crée un arbre réduit à une feuille dont la clé est donnée en paramètre.
2. Écrire une fonction `bool est_feuille(noeud* n);` qui vérifie si un nœud, vide ou non, est une feuille.
3. Écrire une fonction `noeud *enracine(int cle, noeud* gauche, noeud* droit);` qui renvoie un arbre dont la racine est un nœud dont l'étiquette, le fils gauche et le fils droit sont donnés en arguments. On suppose que les nœuds donnés en arguments sont des racines différentes.
4. Pourquoi est-il important que `gauche` et `droit` soient des arbres (donc des racines) et pas des nœuds quelconques ? Que se passe-t-il si ces deux racines sont les mêmes ?



5. Implémenter en C l'arbre ci-dessus.

L'implémentation de l'arbre ci-dessous vous est également proposée dans le fichier `arbre_ex.c` pour vous permettre de tester vos fonctions.



Exercice 2 (Un peu de généalogie) :

1. Écrire une fonction `noeud* parent(noeud* n);` qui renvoie le parent d'un nœud, vide ou non, si ce parent existe et `NULL` sinon.
2. Écrire une fonction `noeud* grandparent(noeud* n);` qui renvoie le grand-parent d'un nœud, vide ou non, si ce grand-parent existe et `NULL` sinon.
3. Écrire une fonction `noeud* frere(noeud* n);` qui renvoie le frère d'un nœud, vide ou non, si ce frère existe et `NULL` sinon.
4. Écrire une fonction `noeud* oncle(noeud* n);` qui renvoie l'oncle (le frère du père) d'un nœud, vide ou non, si cet oncle existe et `NULL` sinon.

Exercice 3 (Fonctions élémentaires de parcours) :

1. Écrire une fonction récursive `int taille_rec(noeud* n);` qui renvoie le nombre de nœuds d'un sous-arbre de racine fictive n. Le nœud n peut être vide. Quelle est sa complexité ?
2. Écrire une fonction purement impérative `int profondeur_imp(noeud* n);` qui renvoie la profondeur d'un nœud, c'est-à-dire la longueur du chemin reliant ce nœud à la racine de son arbre. On suppose que le nœud n'est pas vide, ce que l'on vérifiera avec une assertion. La profondeur de la racine est 0. Quelle est sa complexité ?
3. Écrire une fonction récursive `void libere_arbre(noeud* n);` qui libère toute la mémoire allouée pour le sous-arbre de **racine fictive** n.
   Attention : en dehors des appels récursif, cette fonction ne doit absolument être appelée que sur une racine réelle d'un arbre. Pourquoi ?
4. Écrire une fonction récursive `void affiche_infixe_rec(noeud* n);` qui affiche les étiquettes d'un sous-arbre de **racine fictive** n dans l'ordre d'un parcours infixe des nœuds.
   Que faut-il modifier pour avoir un parcours préfixe ou suffixe ?

2 Arbres binaires de recherche en C

On rappelle que les arbres binaires de recherche sont des arbres binaires qui vérifient en plus la propriété d'ABR suivante : l'étiquette de chaque nœud est plus grande (ou égale) que celles des nœuds de son sous-arbre gauche, et plus petite (ou égale) que celles des nœuds de son sous-arbre droit.

Dans toute la suite, sauf mention explicite du contraire, on considère que les arbres binaires manipulés sont des arbres binaires de recherche.

Exercice 4 (Arbres binaires de recherche) :

1. Que faut-il apporter comme changements à la structure `noeud` pour représenter des arbres binaires de recherche ?
2. Écrire une fonction récursive `noeud* recherche(int cle, noeud* abr);` qui renvoie un pointeur sur un nœud d'un arbre binaire de recherche contenant une clé donnée en argument s'il en existe un et un pointeur `NULL` sinon.
   Quelle est sa complexité ?
3. Donner un version purement impérative `noeud* recherche_imp(int cle, noeud* abr);` de la fonction précédente.
4. Donner une fonction purement impérative `noeud* minimum_it(noeud* abr);` qui renvoie un pointeur vers un nœud de clé minimale d'un ABR, s'il en existe un, et un pointeur `NULL` sinon.
5. Même question avec le maximum d'un ABR en proposant cette fois-ci une fonction récursive `noeud* maximum_rec(noeud* abr);`.
6. Écrire une fonction purement impérative `noeud* insertion_imp(int cle, noeud* abr);` qui insère une nouvelle clé dans un arbre binaire de recherche. On créera une feuille à la bonne place, en n'oubliant pas de mettre à jour le nouveau parent ni de gérer correctement le cas où l'arbre est initialement vide.

7. Même question avec une fonction récursive `noeud* insertion_rec(int cle, noeud* abr);` qui insère une nouvelle clé dans un arbre binaire de recherche.

La suppression est une opération un peu plus délicate. Supposons que l'on cherche à supprimer un nœud z. Il y a trois cas :

(a) si z est une feuille, c'est-à-dire si ses deux fils sont `NULL`, alors on peut directement supprimer z ;

(b) si z n'a qu'un seul fils, alors on peut "détacher" z en reliant ce fils avec le parent de z ;

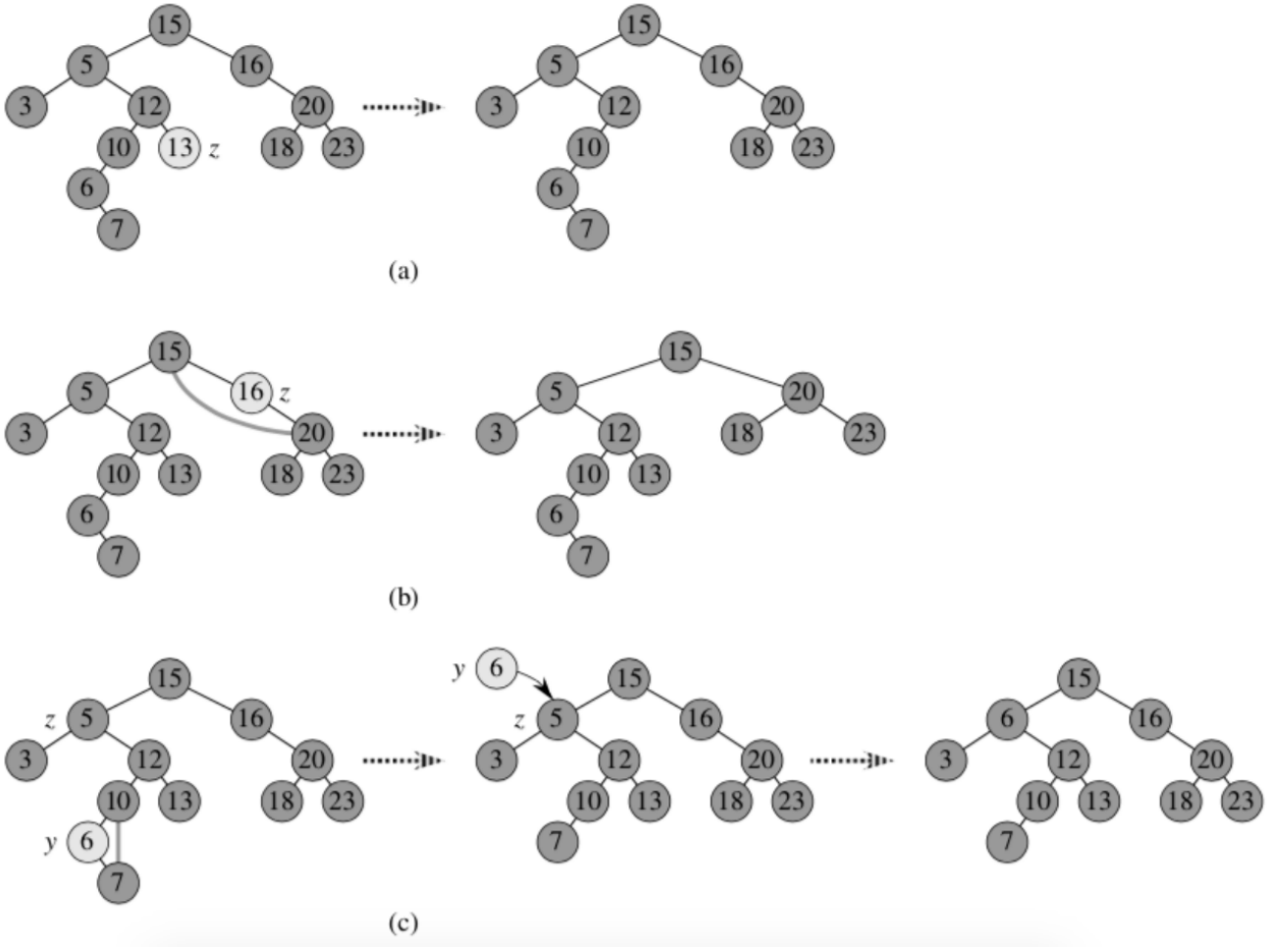
(c) en revanche, si z a deux fils la situation est plus complexe. Nous avons vu que l'on peut alors remplacer z par le minimum de son fils droit (ou par le maximum de son fils gauche) que l'on note y. On peut recopier la clé de y dans z puis supprimer y. Comme y est le minimum d'un sous-arbre, son fils gauche est nécessairement vide : on se retrouve dans le deuxième cas que l'on sait gérer.

La figure ci-dessous illustre ces trois cas de la suppression d'un nœud z dans un arbre binaire de recherche. Le nœud à supprimer effectivement est colorié en gris clair.

(a) z n'a pas d'enfants : on peut le supprimer directement.

(b) z n'a qu'un seul enfant : on peut le détacher.

(c) z a deux enfants : on le remplace par le minimum de son sous-arbre droit, appelé ici y. Ce nœud y n'a pas de fils gauche, et on peut donc le détacher.



8. Compléter la fonction `noeud* suppression_imp(int cle, noeud* abr);` fournie dans le fichier `arbre_ex.c` pour réaliser la suppression du minimum.