TP2: Arbres binaires de recherche

Le but de ce TP est d'implanter les algorithmes de recherche et de modification dans les ABR vus en cours.

Consignes
Télécharger le contenu du dossier TP2 sur moodle dans un nouveau dossier de votre espace personnel. Vous trouverez dans ce dossier les fichiers dont vous avez besoin. Les deux fichiers `ArbresBinaires.cc` et `ArbresBinaires.h` contiennent les définitions et opérations de base sur les nœuds et arbres binaires :

- la structure `ArbreBinaire` contient un champ, qui est un pointeur vers un `noeud` ;
- la structure `noeud` contient quatre champs : trois pointeurs vers les fils et le père, et une valeur entière ;
- `ArbreVide` et `creerNoeud` font ce que leur nom indique ;
- deux fonctions d'affichage des arbres binaires est fournie : la première est `dessinArbre(arbre, "nom")` où `arbre` est de type `ArbreBinaire*` et produit le fichier `nom.pdf` [1] qui représente graphiquement l'arbre `arbre`. Sous Linux, il faut le package GraphViz installé pour pouvoir l'utiliser. L'autre possibilité est d'utiliser la fonction `affichageGraphique(arbre)` qui produit un document postscript.

Le fichier `TP2.cc` **est celui que vous devez modifier** : il contient les trames des fonctions que vous avez à écrire, ainsi qu'un `main` qui contient des tests pour les différents exercices (à décommenter au fur et à mesure). Enfin un `Makefile` complète le tout : `make` compilera ce qu'il faut !

Remarque

- Vous ne devez modifier que `TP2.cc`, mais ne pas modifier les autres fichiers.
- Il est **impératif** de tester chaque fonction que vous écrivez !

À vous de jouer !

1. Compléter la fonction `void inserer(ArbreBinaire* arbre, noeud* x)` qui permet d'insérer le nœud `x` dans l'`arbre`. *Le test fournit dans le fichier* `TP2.cc` *doit créer un fichier* `exemple.pdf` *qui contient l'arbre du haut de la figure 1.*
2. Compléter les fonctions `void parcoursInfixe(noeud* x)` et `noeud* minArbre(noeud * x)` qui respectivement affiche un parcours infixe de l'arbre binaire et renvoie le nœud de valeur minimale dans cet arbre. *Pour tester ces fonctions, on les appelera sur* `arbre->racine`.
3. Compléter la fonction `noeud* recherche(ArbreBinaire* arbre, int k)` qui recherche un nœud de valeur `k` dans l'`arbre`. Si un tel nœud existe la fonction renvoit un pointeur sur ce nœud, sinon le pointeur `NULL`.
4. Compléter la fonction `noeud* successeur(noeud *x)` qui renvoie le nœud successeur du nœud *x* passé en paramètre. Si jamais `x` est le nœud de valeur maximale dans l'arbre, le pointeur `NULL` sera retourné.
5. Compléter les deux fonctions `void remplace(ArbreBinaire* arbre, noeud* x, noeud* z)` et `void supprimer(ArbreBinaire* arbre, noeud* z)` qui permettent respectivement de remplacer le nœud `x` par le nœud `z`, et de supprimer le nœud `z`. Penser à libérer la mémoire correspondante lors de la suppression d'un nœud. *Pour tester votre fonction, vous supprimerez successivement les nœuds de valeur 23, 16, 6 et 13 de l'arbre initial. La figure ci-dessous représente l'arbre initial et l'arbre obtenu après ces quatre suppressions.*
6. Écrire des fonctions de rotation gauche et droite, ainsi que des tests. Se renseigner sur les arbres rouge-noir et implanter les opérations correspondantes (insertion, suppression, etc.). Tester les algorithmes !

1. Ainsi que `nom.dot` utilisé pour produire le fichier PDF.

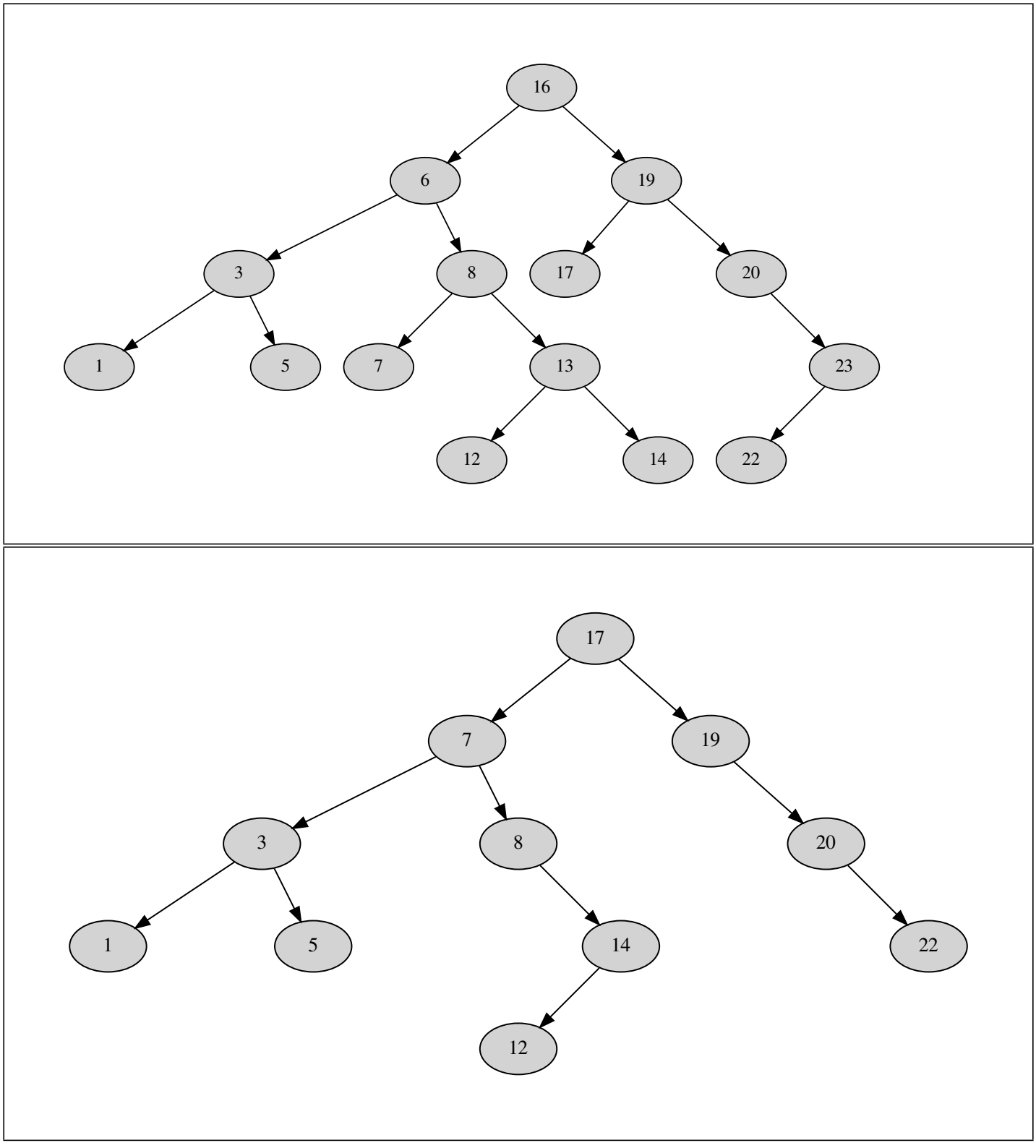

FIGURE 1 – En haut, l'arbre initial proposé dans le code, en bas le même arbre après la suppression des nœuds de valeur 23, 16, 6 et 13.