Université d'Orléans — L3 Info / PASD
Informatique — 2016-17

Travaux Dirigés sur Machine n°3 — Arbres binaires de recherche — TDM noté (4h)

L'objectif de ce TDM est de construire une structure de données pour des arbres binaires de recherche génériques.

Le TDM est divisé en deux parties. La *première partie* consiste à mettre en place la structure et les fonctions associées. Un programme principal est disponible pour tester sur des arbres dont les nœuds contiennent des entiers. La *deuxième partie* consiste à utiliser les arbres binaires de recherche pour manipuler un dictionnaire de sigles.

Exercice 1. La structure `arbre`

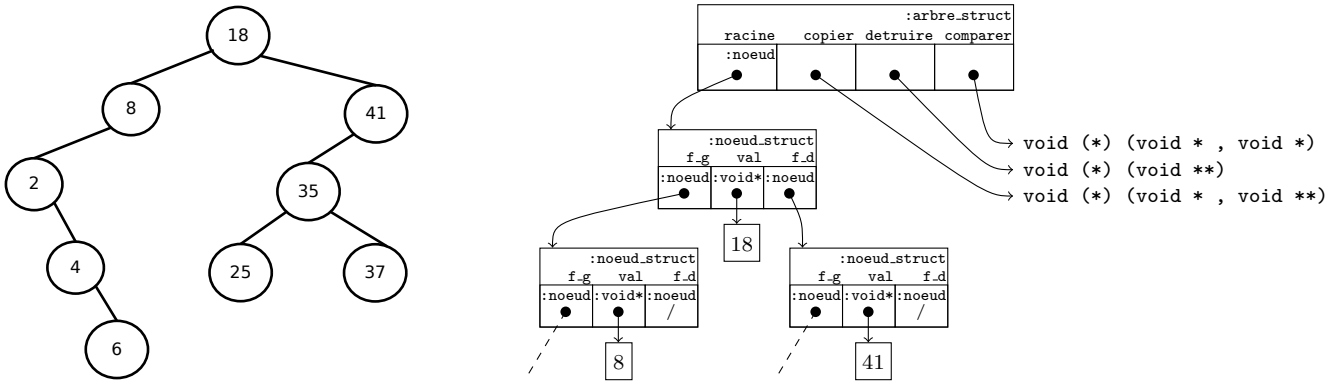
Dans l'archive `TDM3.tgz` disponible sur Celene se trouvent les fichiers :

- `arbres.h` contenant le `typedef` du type `arbre` et les en-têtes des fonctions (visibles) à développer ainsi que leur documentation,
- `arbres.c` contenant les structures `noeud_struct` et `arbre_struct` et les squelettes de toutes les fonctions à développer.
- `test_arbres_int.c` et `test_arbres_int_in.txt` pour tester vos fonctions, et
- `Makefile` pour compiler et exécuter les différents tests.

Les fichiers contiennent les commentaires nécessaires aux développements demandés quand la documentation, les noms des fonctions et cet énoncé ne suffisent pas.

La structure combinatoire/logique proposée correspond au dessin de la Figure 1(a). Le haut de sa représentation en mémoire est illustrée par la Figure 1(b) où `f_g` est un raccourci pour `fils_gauche` et `f_d` pour `fils_droit`. Il s'agit d'une structure à deux niveaux : les `struct arbres_struct` servent de socle aux arbres alors que les `struct noeud_struct` servent à représenter les nœuds de l'arbre.

(1) Dessiner *dans le compte-rendu* la représentation d'un arbre vide.



(a) Arbre binaire

(b) Haut de la représentation en mémoire

FIGURE 1 – Exemple d'arbre binaire de recherche.

Pour avoir une structure de données générique, la valeur des nœuds est un pointeur de type `void *`; on ne connaît pas le type des valeurs stockées dans l'arbre. Les fonctions `copier`, `comparer` et `detruire` permettent de manipuler convenablement ces valeurs.

Comme ce sont des arbres binaires de recherche, les valeurs doivent pouvoir être comparées les unes aux autres. La fonction `int (*comparer) (void* val1, void* val2)` sert à cela. Par la suite, on appelle *clé* la partie d'une valeur sur laquelle se fait la comparaison. Deux valeurs différentes peuvent avoir une même clé (c.-à-d. que `comparer` retourne zéro). Cela est très utile pour la réalisation de dictionnaires. La fonction `comparer` renvoie un entier plus grand ou égal à 1 lorsque la clé de `val1` est plus grande que celle de `val2`, 0 si elles sont égales et un entier inférieur ou égal à -1 sinon.

Dans un arbre binaire de recherche, pour chaque nœud, tous les nœuds du sous-arbre gauche (resp. droit) ont des valeurs dont la clé est inférieure (resp. supérieure) à celle du nœud considéré. De plus, tous les nœuds de l'arbre ont des clés distinctes : on n'insérera pas une valeur si sa clé est déjà présente. La fonction `comparer` sert donc à savoir si une valeur doit être avant, après ou à la place d'une autre.

L'arbre de la Figure 1(a) correspond à un arbre vide auquel on a successivement ajouté 18, 8, 2, 41, 35, 25, 4, 8, 6, et finalement 37 (pour la comparaison, on utilise l'ordre naturel des entiers).

Les fonctions d'affichage prennent en argument le flux où afficher et un pointeur vers une fonction permettant d'afficher une valeur, une valeur.

(2) Expliquer *dans le compte-rendu* pourquoi on doit fournir un tel pointeur.

Attention, pour l'insertion, la recherche et la suppression, il faut trouver la bonne position dans l'arbre. Afin de ne pas écrire trois fois la recherche, une fonction annexe est à faire (`arbre_chercher_position`).

(3) Expliquer *dans le compte-rendu* la signature de cette fonction.

(4) Expliquer *dans le compte-rendu* la complexité de cette fonction (sous l'hypothèse que l'arbre est « équilibré » : pour chaque nœud, il y a à peu près autant de nœud sous chacun des fils).

Attention, la fonction `arbre_supprimer` doit laisser l'arbre de recherche dans un état cohérent. Si cela est facile pour une feuille, c'est plus complexe pour un nœud interne. Si le nœud que l'on veut supprimer n'a pas de fils droit alors on le remplace par son fils gauche. Sinon le nœud doit échanger sa valeur avec celle du fils le plus à gauche de son fils droit et on se ramène alors au cas précédent.

(5) Expliquer *dans le compte-rendu* ce que signifie `\pre` pour Doxygene. Modifier `arbres.h` pour en donner des exemples (et écrire le code correspondant dans `arbres.c`).

Implanter les fonctions de manipulation des arbres et des nœuds dans `arbres.c`.

Le programme `test_arbres_int.c` sert à tester ces fonctions dans le cas où les valeurs des nœuds sont des entiers (la valeur est réduite à la clé). Le `Makefile` s'utilise de la manière suivante :

— `make test_arbres_int` compile et engendre l'exécutable du programme `test_arbre_int.c` fourni,
— `make test_arbres` exécute le test et affiche les résultats à l'écran, et
— `make memoire_int` exécute le test avec valgrind et vérifie si la mémoire est correctement gérée.

Exercice 2. Le dictionnaire de sigles

Dans cette application, on propose d'utiliser les arbres binaires de recherche pour manipuler un dictionnaire de sigles. Désormais les valeurs des nœuds de l'arbre sont définis par deux chaînes de caractères (avec la représentation du C) :

```
typedef struct sigle_struct * sigle ;

struct sigle_struct {
   char * court ;    /* le sigle */
   char * details ;  /* la description du sigle */
};
```

(6) Dessiner *dans le compte-rendu* la représentation d'un nœud enregistrant le sigle
{ `.court = "WWW" , .detail = "World Wide Web"` } .

Implanter les fonctions de création, de destruction et d'affichage d'un `sigle`. Ces fonctions sont préfixées par `sigle_` .

Écrire ensuite les fonctions qui permettent de créer un arbre dont les valeurs sont des sigles avec pour clé le champ `sigle`. Ces fonctions sont postfixées par `_sigle` pour les distinguer des précédentes. La fonction de comparaison est destinée à définir l'ordre lexicographique sur les sigles.

Le programme `test_arbres_sigle.c` fournit un squelette de ces fonctions ainsi qu'un programme principal qui lit un fichier de sigles, crée l'arbre binaire de recherche et teste les fonctions écrites.

Pour les tests, le `Makefile` contient les commandes similaires aux précédentes pour `test_arbres_sigle.c`.

Exercice 3. Remise

(Consignes habituelles.)