TP n°3 – Arbres binaires

https://www.lri.fr/∼schevalier/teaching.html

26 mars 2019

En C, un arbre binaire est habituellement représenté par un pointeur sur sa racine. La racine est représentée par une structure de type nœud, contenant trois attributs : sa valeur, un pointeur vers son fils gauche (de type nœud) et un pointeur vers son fils droit (de type nœud).

1 Les structures

Créez un fichier `arbre.h` et placez-y les structures suivantes :

- `struct Noeud` : structure composée d'un champ `gauche` représentant un pointeur vers le fils gauche, d'un champ `droit` représentant un pointeur vers le fils droit, et d'un champ `cle` de type entier ;
- `struct Arbre` : structure composée d'un champ `racine` représentant la racine de l'arbre.

2 Les fonctions

Créez ensuite les fonctions suivantes dans un fichier `arbre.c` avec les prototypes associés dans le fichier `arbre.h`. Certaines de ces fonctions prennent en argument l'arbre binaire, alors que vous aurez besoin de les appeler récursivement sur les nœuds de l'arbre à partir de sa racine. Créez donc les fonctions auxiliaires prenant en argument un nœud de l'arbre que vous jugez nécessaires.

Attention ! Votre fichier `arbre.c` ne doit pas comporter de fonction `main` (voir la section 3).

2.1 Gestion de la mémoire

1. `Arbre *nouvel_arbre()` : crée (avec allocation mémoire) un nouvel arbre et renvoie un pointeur vers cet arbre.
2. `Noeud *nouveau_noeud(int val)` : crée (avec allocation mémoire) un nouveau nœud qui a pour clé l'entier `val` et renvoie un pointeur vers ce nœud.
3. `void free_arbre(Arbre *mon_arbre)` : libère la mémoire allouée à l'arbre (ainsi que pour tous ses nœuds).

2.2 Parcours d'un arbre binaire

4. `void parcours_prefixe(Arbre *mon_arbre)` : parcourt l'arbre reçu en argument dans l'ordre préfixe, en affichant les clés des nœuds.
5. `void parcours_postfixe(Arbre *mon_arbre)` : parcourt l'arbre reçu en argument dans l'ordre postfixe, en affichant les clés des nœuds.
6. `void parcours_infixe(Arbre *mon_arbre)` : parcourt l'arbre reçu en argument dans l'ordre infixe, en affichant les clés des nœuds.

2.3 Comptage

7. `int nombre_noeuds(Arbre *mon_arbre)` : compte et renvoie le nombre de nœuds de l'arbre reçu en argument.
8. `int nombre_feuilles(Arbre *mon_arbre)` : compte et renvoie le nombre de feuilles de l'arbre reçu en argument.
9. `int profondeur(Arbre *mon_arbre)` : calcule et renvoie la profondeur de l'arbre reçu en argument. ***Rappels*** *: Si elle existe, la racine d'un arbre a une profondeur de 1. La profondeur d'un arbre vide vaut zéro.*

2.4 Arbre binaire de recherche

Un arbre binaire de recherche A est un arbre binaire tel que pour tout nœud n de A, la clé de n est supérieure ou égale aux valeurs des clés de tous les nœuds du sous-arbre gauche de n, et inférieure aux valeurs des clés de tous les nœuds du sous-arbre droit de n.

10. `bool est_abr(Arbre *mon_arbre)` : renvoie `true` si l'arbre reçu en argument est un arbre binaire de recherche, et `false` sinon. ***Remarque*** *: Pour utiliser le type booléen, vous devez inclure le* header `stdbool.h`.
11. `Noeud *recherche_abr(Arbre *mon_arbre, int val)` : recherche un nœud dont la clé vaut `val` dans un arbre binaire de recherche `A` et renvoie un pointeur vers ce nœud s'il existe, ou `NULL` sinon. ***Remarque*** *: Il serait judicieux de s'assurer que l'arbre* `A` *est bien un arbre binaire de recherche avant de commencer la recherche.*

3 Le programme de test

Téléchargez le fichier `main_arbre.c` (`https://www.lri.fr/~schevalier/doc/teaching/M1BIBS/A-et-C/2018/TP3/main_arbre.c`). Ce programme de test vous permettra de tester votre implémentation au fur et à mesure, en commençant par des fonctions minimales. Compilez le tout avec :

```
gcc -Wall -g -o main_arbre arbre.c main_arbre.c
```

Les résultats attendus sont fournis dans le fichier `out_arbre.txt` (`https://www.lri.fr/~schevalier/doc/teaching/M1BIBS/A-et-C/2018/TP3/out_arbre.txt`).

4 Ça marche... mais est-ce correct ?

Même quand tout a l'air de fonctionner correctement, il peut y avoir des fuites de mémoire et d'autres erreurs. Un outil permettant d'identifier ces problèmes est `valgrind`. Si vous avez compilé en passant l'option `-g` à `gcc` comme indiqué dans la section précédente, lancez votre exécutable `main_arbre` via `valgrind` :

```
valgrind --track-origins=yes --leak-check=full ./main_arbre
```

Si tout se passe bien, vous devriez voir quelque chose comme ceci (en plus du résultat de l'exécution de votre programme) :

```
==6407== HEAP SUMMARY:
==6407==     in use at exit: 0 bytes in 0 blocks
==6407==   total heap usage: 70 allocs, 70 frees, 2,536 bytes allocated
==6407==
==6407== All heap blocks were freed -- no leaks are possible
==6407==
==6407== For counts of detected and suppressed errors, rerun with: -v
==6407== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```