TRAVAUX PRATIQUES XVI

Arbres Binaires de Recherche

Le but de ce TP est de programmer les opérations vues en cours sur les arbres binaires de recherche.

Nous avons fait le cours en OCaml avec un style fonctionnel. Dans ce TP, on programmera en C avec un style impératif : nos fonctions vont modifier les ABR pris en argument.

Vous pourrez coder les fonctions récursivement ou itérativement, au choix.

Il est fortement recommandé pour ce TP de *faire des dessins !*.

.1 Compilation

On rappelle que les options suivantes doivent être utilisées à la compilation :

```
-Wall -Wextra -g -fsanitize=undefined,address
```

.2 Header guards

0. Allez lire la partie du cours sur les bonnes pratiques de programmation qui traite des `#define` et des headers guards. Appelez-moi si quelque chose n'est pas clair.

.3 `malloc`

1. Si vous n'êtes pas au point sur `malloc` et `free` (que prennent-elle en entrée ? que renvoient-elle ? effets secondaires ?) ; c'est le moment de me demander.

A Premières manipulation des arbres en C

Dans ce TP, on utilise la struct suivante pour représenter les arbres binaires de recherche. C'est celui du cours, agrémenté d'un emballage supplémentaires :

```c
struct noeud_s {
  struct noeud_s *gauche;
  int etiquette;
  struct noeud_s *droit;
};
typedef struct noeud_s noeud;

struct abr_s {
  int taille;
  noeud* racine;
};
typedef struct abr_s abr;
```

- Le type `noeud` correspond à un noeud d'un arbre (on peut le confondre avec un sous-arbre). Il s'agit d'un type « interne » : les utilisateurs de la structure de donnée n'y auraient pas accès.
- Le type `abr` correspond à un arbre binaire de recherche entier, sous forme d'un pointeur vers un noeud (sa racine). Un noeud vide est représenté par un pointeur nul (de type `noeud*`) ; un arbre vide est un `abr` dans lequel le champ `racine` contient un pointeur nul. Toutes les fonctions destinées à être fournies aux utilisateurs manipulent uniquement des `abr` .

2. Préparer un fichier `abr.c` et son header `abr.h` . Vous écrirez vos tests et votre fonction `main` dans un fichier séparé `TP19.c` . Assurez-vous de faire les bons "include" (`#include "abr.h"`) et d'utiliser des headers guards.

3. Écrire une fonction `noeud* nouveau_noeud(int x)` créant un nouveau noeud, qui est une feuille d'étiquette x.
4. Écrire une fonction `abr cree_abr_vide(void)` qui crée et renvoie un nouvel arbre, vide.
5. Écrire une fonction `void libere_noeud(noeud* n)` qui libère la mémoire utilisée par un noeud **et par tous ses descendants**.
6. Écrire une fonction `void libere_arbre(abr* a)` qui libère la mémoire utilisée par un arbre.

Pour vous aider à tester pour la suite, voici une fonction à mettre dans votre fichier `TP19.c` qui construit pour vous un arbre exemple.

```c
abr cree_test_abr(void) {
  noeud* n = nouveau_noeud(24);
  n->gauche = nouveau_noeud(5);
  n->droit = nouveau_noeud(42);
  n->gauche->gauche = nouveau_noeud(3);
  n->gauche->droit = nouveau_noeud(7);
  abr a = {.taille = 5, .racine = n};
  return a;
}
```

L'arbre renvoyé par cette fonction correspond à l'arbre suivant :

Terminal

```
abr* a = cree_test_abr ();
      24
      /  \
     5   42
    / \
   3   7
```

Bien entendu, rajoutez des tests ! À vous de trouver comment tester vos fonctions de manière pertinente (afficher certaines étiquettes bien choisies, voire toutes, ou encore la taille ou la hauteur de votre arbre, etc). Il vous est **fortement** conseillé de faire votre propre fonction d'affichage d'un arbre (par exemple, énumérer ses noeuds dans un certain ordre).

B Opérations sur les ABR

7. Écrire une fonction `int hauteur(abr const* a)` qui renvoie la hauteur d'un arbre.

B.1 Recherche dans un ABR

8. Écrire une fonction `bool mem_arbre(abr const* a, int x)` testant si une certaine clé est présente dans un arbre.

B.2 Insertion dans un ABR

9. Écrire une fonction `void insere_noeud(noeud* n, int x)` qui insère l'élément x dans l'arbre/sous-arbre *non vide* dont la racine est pointée par n. Si x est déjà présent dans l'arbre, elle ne fait rien.
10. Écrire une fonction `insere_arbre` qui ajoute un élément dans un arbre binaire de recherche : `void insere_arbre(abr* a, int x)` .
    Attention, le cas de l'arbre vide devra peut-être être triaté à part.

B.3 Parcours d'un ABR

Dans cette partie, on veut implémenter le parcours préfixe d'un ABR et stocker les noeuds rencontrés dans un **tableau**. Il y a deux façons de faire : par une méthode récursive, ou par une méthode impérative en utilisant une pile.

La longueur du tableau sera le nombre d'éléments de l'arbre : c'est pour cela qu'on le stocke dans le champ `taille` !

Nous sommes maintenant prêts à écrire notre parcours :

11. Écrire une fonction `int* arbre_vers_tableau(abr const* a)` qui renvoie un tableau contenant les étiquettes d'un arbre binaire de recherche dans l'ordre croissant de leurs valeurs.
Si vous travaillez en récursif, faire une fonction auxiliaire qui prend en argument le noeud en cours, le tableau où l'on écrit, l'indice où l'on écrit, et qui renvoie l'indice de la prochaine écriture peut être une bonne idée.

B.4 Tri par ABR

Application au tri d'un tableau, deuxième partie :

12. Écrire une fonction `abr abr_of_array(int const* arr, int len)` qui construit un arbre binaire de recherche en insérant un par un les `len` éléments de `arr` dans un ABR.

13. En déduire une fonction `void tri_abr(int* arr, int len)` qui trie en place un tableau d'entiers en utilisant un ABR, c'est à dire qu'elle modifie le `arr` passé en entrée de sorte à ce qu'il soit trié.

14. Avez-vous pensé à libérer **toute** la mémoire allouée ?

B.5 Suppression dans un ABR

15. Écrire une fonction `noeud* extrait_noeud_max(noeud* n, int* max_ptr)` qui supprime le minimum du sous-arbre passé en argument (de racine pointée par `n`), et renvoie la nouvelle racine du nouvel arbre ainsi obtenu. Notez que dans presque tous les cas, la nouvelle racine est la même que l'ancienne.
De plus cette fonction doit faire en sorte qu'après l'appel, `*min_ptr` soit être égal à l'étiquette du maximum supprimé (peu importe sa valeur initiale).

16. Avez-vous bien pensé à libérer la mémoire du noeud supprimé, dans la fonction précédente (et uniquement de ce noeud) ?

17. Écrire la fonction `void supprime_element(abr* a, int x)` qui supprime le noeud d'étiquette x dans l'arbre donné. Si x n'est pas dans l'arbre, la fonction ne fait rien.

C Pour aller plus loin : détermination expérimentale de la hauteur moyenne

18. Écrire un programme C ayant le comportement suivant :
- Il demande à l'utilisateur d'entrer dans le terminal deux entiers `max_power` et `rep_count` .
- Pour chaque entier k compris entre 4 et `max_power`, il génère `rep_count` fois un arbre binaire de recherche de taille 2^k en insérant les éléments de $[0, \dots, 2^k - 1]$ dans un ordre aléatoire. Pour chacun de ces arbres, il calcule la hauteur obtenue.
- Pour chacun des k, il affiche une ligne contenant la valeur de 2^k et la valeur moyenne des hauteurs obtenues sur les `rep_count` arbres générés, séparées par une espace.

19. Que constatez-vous ? Quelle relation la hauteur de l'arbre semble suivre en fonction de son nombre de noeud ? Est-ce cohérent avec le cours ?

20. Pour les plus motivés, écrivez un script pour générer un graphique semi-logarithmique et ainsi visualiser les hauteurs moyennes expérimentales obtenues en fonction de la taille de l'arbre, dans le langage de votre choix (par exemple python ; j'avoue préférer `gnuplot` pour les tracés rapides).