Programmation Avancée

TP n°6 : Les arbres

Simon Forest

4 Mars 2021

Exercice 1 : Arbres binaires de recherche d'entiers

Dans cet exercice, on écrit les fonctions permettant de manipuler des arbres binaires de recherche sur des entiers. On utilise pour cela le type vu en cours qui est :

```
typedef struct arb {
  int val;
  struct arb *gauche, *droite;
} arb;
```

Essayez de faire cet exercice sans copier-coller ce qui a été vu en cours, l'idée étant que vous devriez être capable de faire tout ce qui suit vous-même.

1. Écrire une fonction `arb* arb_singleton(int v)` qui crée un `arb` avec un seul nœud dont le contenu est `v`.

2. Écrire une fonction `void arb_insert(arb* a, int v)` qui insert un nœud de contenu `v`. La fonction devra maintenir la propriété des arbres binaires de recherche, c'est-à-dire que le nœud inséré devra :
   - être « à gauche » de tout nœud dont le champ `.val` est plus grand que lui ;
   - être « à droite » de tout nœud dont le champ `.val` est plus petit que lui.

   Aussi, si un nœud avec la même valeur est déjà présent dans l'arbre, on n'en insérera pas une autre copie.

3. Écrire une fonction `int arb_trouver(arb* a, int v)` renvoie `1` s'il y a un nœud de contenu `v` présent dans l'arbre `a`, et `0` sinon. On utilisera la propriété des arbres binaires de recherche pour lire le moins de nœuds possible.

4. Écrire une fonction `void arb_detruire(arb* a)` qui libère l'arbre. Il faudra bien appeler `free` récursivement sur tous les nœuds.

5. Écrire une fonction `int arb_taille(arb* a)` qui renvoie le nombre de nœuds de l'arbre.

6. Tester l'implémentation avec le code fourni sur Moodle.

7. Vérifier avec `valgrind` que toute la mémoire allouée sur le tas est bien libérée.

Exercice 2 : Arbres pour les chaines de caractères

Les arbres binaires de recherche ne sont pas réservés aux entiers, mais peuvent être utilisés avec n'importe quel type que l'on peut ordonner. En particulier, après avoir donné une façon d'ordonner les chaines de caractères, on peut faire des arbres qui traitent ce type de données.

1. Écrire une fonction `my_strcmp(char *s1, char *s2)` qui permet de comparer deux chaines de caractères `s1` et `s2`. La comparaison doit être lexicographique, c'est-à-dire que `s1` doit être considérée strictement plus petite que `s2` si :

- soit `s1` est un préfix strict de `s2` ;
- soit `s1` n'est pas un préfix de `s2` et la première paire de caractères `c1` et `c2` de `s1` et `s2` qui diffère vérifie que `c1 < c2`.

La fonction renverra `-1` si `s1` est plus petit que `s2`, `0` en cas d'égalité, et `1` si `s1` est plus grand que `s2`. (Une fonction `strcmp` existe déjà dans la librairie standard, mais l'exercice consiste à la réécrire).

2. Adapter le type `arb` de l'exercice 1 en un type `arb_str` qui stocke des chaines de caractères au lieu d'entiers. Écrire des fonctions `arb_str_singleton`, `arb_str_insert`, `arb_str_trouver`, `arb_str_detruire`, `arb_str_taille` analogues à celles de l'exercice 1 pour ce type.
3. Adapter le code de test de l'exercice précédent pour tester votre structure. Afin d'avoir une chance raisonnable de faire des requêtes sur des mots déjà insérés dans la structure, on ne considérera que des chaines de caractères aléatoires de taille 4 n'utilisant que les lettres `'a'`, `'b'` et `'c'`. Utiliser `valgrind` pour vérifier que la mémoire a bien été désallouée.

Exercice 3 :

Une autre façon de stocker un ensemble de chaînes de caractères avec des arbres est donnée par les *tries* : ce sont des arbres qui ne sont plus nécessairement binaires et où chaque nœud correspond à une lettre de l'alphabet contenue par l'une des chaînes stockées dans l'arbre. Si un certain mot `m` est stocké dans la *trie*, ses lettres apparaîtront alors dans l'ordre comme nœuds sur une branche de l'arbre partant de la racine. Par exemple, la *trie* présentée Figure 1 est l'arbre contenant les mots `"cat"`, `"cathy"`, `"dig"`, `"dog"`. Les nœuds correspondants à des fins de mots sont marqués par la couleur orange. Dans cet exercice, on cherche à représenter de tels arbres.

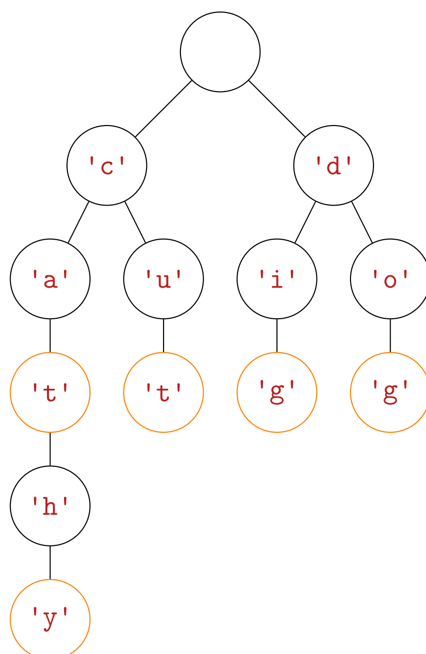

Figure 1 – Un exemple de *trie*

Pour simplifier, on suppose que les chaînes à stocker ne contiendront que des lettres minuscules entre `'a'` et `'z'`. On utilisera le type suivant pour représenter les nœuds des *tries* :

```
typedef struct noeud {
  char lettre;
```

```c
  int fin_mot;
  struct noeud* enfants[26];
} noeud;
```

L'attribut `lettre` correspondra à la lettre associée au nœud (il vaudra `0` pour la racine). L'attribut `fin_mot` vaudra `1` dans le cas où le nœud correspond à la fin d'un mot, et `0` sinon. Le tableau `enfants` contiendra les pointeurs vers les nœuds enfants correspondants à chacune des lettres possibles.

1. Écrire une fonction `noeud* trie_new()` qui crée une *trie* n'ayant qu'un nœud racine.
2. Écrire une fonction `int trie_trouver(noeud* racine, char* mot)` qui détermine si un mot est présent dans l'arbre.
3. Écrire une fonction `noeud* trie_inserer_lettre(noeud* node, char lettre)` qui insert un nœud pour la lettre `lettre` comme enfant de `node` et renvoie un pointeur vers le nœud créé.
4. Écrire une fonction `void trie_inserer(noeud* racine, char* mot)` qui insert un mot dans la *trie*.
5. Écrire une fonction `void trie_detruire(noeud* racine)` qui désalloue complètement la *trie*.
6. Écrire une fonction `int trie_taille(noeud* racine)` qui renvoie le nombre de chaînes différentes stockée dans la *trie*.
7. Tester les fonctions avec le code fourni sur Moodle. Utiliser `valgrind` pour vérifier que la mémoire a bien été désallouée.
8. Changer la définition de `noeud` par

   ```c
   typedef struct noeud {
     char lettre;
     int fin_mot;
     struct noeud *frere, *enfant;
   } noeud;
   ```

   où `frere` pointera sur le prochain nœud frère et `enfant` vers le premier enfant du nœud. Faire à nouveau les questions précédentes avec cette nouvelle définition.