TP7 – Arbres binaires de recherche

Projet de programmation M1

17 Novembre 2015

Les arbres binaires de recherche sont une structure très pratique pour représenter des ensembles ordonnés. Dans ce TP, on se propose de les implémenter pour les entiers `int`.

Admettons qu'on veuille représenter un ensemble S de n entiers. On veut pouvoir facilement réaliser les opérations suivantes :

— Décider si $x \in S$.
— Éliminer/ajouter un élément y à S.
— Trouver le maximum/minimum de S.

Exercice 1. Imaginons qu'on utilise les tableaux dynamiques de la semaine dernière. Combien de temps serait nécessaire pour résoudre chacune des opérations précédentes ?

Afin de gagner en efficacité, on va utiliser une structure de données arborescente, c'est-à-dire qu'on va ranger nos entiers dans un arbre ayant de bonne propriété. Un *arbre binaire de recherche* (ABR) est un arbre binaire enraciné dont les sommets sont étiquetés par des entiers. Soit v un sommet de l'arbre étiqueté par la valeur x. Soit v_g et v_d ses fils gauche et droite respectivement. Les ABR ont cette propriété : les étiquettes du sous-arbres enraciné en v_g sont toutes plus petites que x. Celle du sous-arbre enraciné en v_d sont toutes plus grandes que x. Schématiquement : Par exemple, l'ensemble $\{0, 1, 2, 5, 7, 10\}$: peut-être

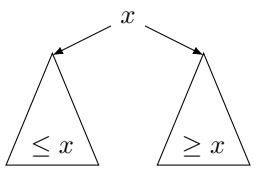

FIGURE 1 – Un schéma des ABR

représenter par :

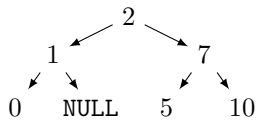

FIGURE 2 – Un exemple d'ABR

On représentera ces arbres par la structure C suivante :

```
struct abr {
  int label;
  struct abr *g;
  struct abr *d;
};
typedef struct abr abr;
```

où `g` sera un pointeur vers le fils gauche et `d` un pointeur vers le fils droit. Lorsque le nœud n'a pas de fils, ce pointeur vaudra `NULL`.

Exercice 2. On va commencer par les fonctions d'ajout et de recherche de sommets :

1. Écrire une fonction `abr *singleton_tree(int x)` qui renvoie un pointeur vers un arbre contenant seulement une feuille étiquetée `x`.
2. Écrire une fonction `abr *add(abr *r, int x)` qui ajoute la valeur `x` à l'arbre dont la racine se trouve à l'adresse donnée par le pointeur `r`. On ajoutera la valeur au bon endroit, à une feuille de l'arbre, pour conserver la propriété qui nous intéresse sur les étiquettes. On utilisera l'allocation dynamique pour créer le nouveau nœud. Votre fonction retournera un pointeur vers la nouvelle feuille.
3. Écrire une fonction `abr *array_to_tree(int *t, int n)` qui créé un ABR à partir d'un tableau d'entier de taille n et renvoie un pointeur vers la racine de cet ABR.
4. Écrire une fonction `abr *search(abr *r, int x)` qui cherche si l'élément `x` est dans l'arbre dont la racine est `r`. Si `x` est trouvé, on renverra un pointeur vers le nœud qui le contient. Sinon, on renverra `NULL`.
5. Écrire des fonctions `min` et `max` pour rechercher le minimum et le maximum d'un arbre. À vous de décider quels sont leurs arguments et type de retour.
6. Écrire une fonction `print_tree(abr *r)` qui écrit les valeurs de l'arbre pointé par `r` dans l'ordre croissant.
7. Combien d'opérations sont-elles nécessaires pour effectuer les fonctions précédentes sur un arbre de profondeur d ?

Exercice 3. Cet exercice est dédié à la suppression d'un sommet. Admettons qu'on veuille supprimer 2 de l'arbre représenter à la figure 2. Cela va casser l'arbre en deux. Il faut trouver un moyen de réorganiser simplement ces deux arbres pour obtenir un nouvel ABR.

1. Expliquer pourquoi cela ne pose pas de problème lorsque la valeur que vous devez enlever est soit une feuille ou un nœud qui n'a qu'un seul fils.
2. On souhaite désormais enlever un nœud qui a deux fils. Montrer qu'on peut le remplacer par le minimum du sous-arbre droit ou par le maximum du sous-arbre gauche. Montrer aussi que ces nœuds ont au plus un fils.
3. En déduire une implémentation de la fonction `abr *remove_node(abr *r, int x)` qui enlève la valeur `x` de l'arbre. Si l'arbre n'est pas vide après cet opération, on renverra `r`. Sinon, on renverra `NULL`. On n'oubliera pas d'utiliser `free` pour libérer la mémoire.

Exercice 4. Écrire une fonction `abr *crible(int n)` qui crée un ABR contenant les nombres premiers compris entre 1 et n. On utilisera pour cela l'algorithme du crible d'Erathostène et on essaiera de faire en sorte que la profondeur de cet arbre soit $O(\log(n))$.