TP8 : arbres binaires de recherche et programmation dynamique

Algorithmique et projet de programmation — M1 mathématiques

22 novembre 2016

1 Arbres binaires de recherche

On va coder une petite *bibliothèque* implémentant les arbres binaires de recherche vus en cours (*Binary Search Trees* en anglais). Rappelons qu'un arbre binaire de recherche est un arbre binaire où chaque noeud contient un élément strictement supérieur aux noeuds de son sous-arbre gauche et strictement inférieur aux noeuds de son sous-arbre droit (on a aussi les variantes non strictes d'un des deux côtés). On peut y faire une recherche d'élément, une suppression ou un ajout en $O(h)$ avec h la hauteur de l'arbre. En particulier, si l'arbre est équilibré, alors $h \sim \ln n$ avec n le nombre de noeuds et toutes ces opérations sont donc en $O(\ln n)$. Cette structure peut être vue comme intermédiaire entre la structure de tableau (accès en $O(1)$, mais ajout et suppression en $O(n)$) et celle de liste chaînées (ajout en $O(1)$, mais accès et suppression en $O(n)$). De ce fait, on peut s'en servir pour représenter les ensembles finis de manière assez efficace.

On va donc créer une bibliothèque qui fait cela : en particulier, l'utilisateur n'aura pas à connaître l'implémentation des arbres binaires de recherche. Tout ce à quoi il aura accès sera une collection de fonction permettant la manipulation d'ensembles finis d'entiers. Commençons donc par créer un fichier `bst.c` qui contiendra toutes les subtilités de l'implémentation que l'on veut cacher à l'utilisateur. Il devra tout d'abord contenir la structure représentant un noeud de l'arbre binaire :

```
struct bst_node_s {
  int key;
  struct bst_node_s *left, *right;
}
```

et tout un tas de fonction que l'on va coder par la suite.

L'utilisateur de notre bibliothèque n'a pas besoin de connaître cette structure : tout ce dont il a besoin est de savoir qu'il y a un type représentant les ensembles finis d'entiers, et que celui-ci prend la forme d'un pointeur vers une structure (obscure pour lui[1]). On crée donc un fichier `bst.h` (l'interface avec l'utilisateur) contenant ce qui suit :

```
#ifndef BST_H
#define BST_H
```

1. C'était exactement notre cas lors de l'utilisation du type `FILE*` dans la manipualtion de fichiers : on était alors utilisateur et on se moquait bien du détail de la structure `FILE` du moment qu'on avait un descriptif des fonctions manipulant cette structure.

```c
// bst_t types finite sets of integers with relatively fast searching,
// adding and removing.
typedef struct bst_node_s* bst_t;

// some prototypes incoming...

#endif
```

L'usage des macros sera expliquée par la suite, faisons en abstraction pour le moment. Ce fichier déclare donc qu'il y a un type `struct bst_node_s` et qu'on défini `bst_t` comme un raccourci pour désigner `struct bst_node_s*`. Et c'est tout ! L'utilisateur n'a donc accès qu'à ce type `bst_t`, censé représenter les ensembles finis d'entiers. N'oubliez pas d'inclure `bst.h` dans `bst.c`, ainsi que dans le fichier `.c` utilisant la bibliothèque : `#include "bst.h"`.

Exercice 1. Compléter `bst.c` avec les fonctions de bases vues en cours manipulant les arbres binaires de recherche : insertion d'un élément, suppression d'un élément, recherche d'un élément, recherche du maximum, recherche du minimum.

On ajoutera aussi les fonctions suivantes, simplifiant la vie à l'utilisateur :

```c
// creates a void BST
bst_t bst_init();

// destructs a BST (freeing any allocated memory)
void bst_destruct(bst_t root);

// creates a BST containing only a root; in particular, it should
// allocate sufficient memory on the heap
bst_t bst_sing(int key);

// prints out the BST
void bst_print (bst_t root);
```

Exercice 2. Rajouter la possibilité de supprimer un élément d'un arbre binaire de recherche. On rappelle que pour cela, il est nécessaire d'avoir une fonction supprimant le minimum d'un sous-arbre.

2 Plus long sous-mot commum

Rappelons qu'un alphabet Σ est juste un ensemble fini dont on appelle les éléments *lettres*. Un mot w sur un alphabet Σ est une séquence finie $(a_1, \dots, a_n)$ avec tous les $a_i \in \Sigma$. On ommettra les parenthèses et les virgules, non nécessaires, et on notre $w = a_1 \dots a_n$. La longueur d'un mot est simplement le nombre de lettres qui le composent. L'ensemble de tous les mots sur un alphabet Σ est noté Σ^*.

Un sous-mot d'un mot $w = a_1 \dots a_n \in \Sigma^*$ est un mot $u \in \Sigma^*$ tel qu'il existe des indices $1 \leq i_1 < \dots < i_k \leq n$ tels que $u = a_{i_1} \dots a_{i_k}$. En français, c'est dire qu'en supprimant certaines lettres de w, on tombe sur u. On s'intéresse ici à développer un algorithme renvoyant un sous-mot le plus grand possible commun à deux mots $w, w' \in \Sigma^*$ donnés.

En écrivant $w = a_1 \dots a_n$ et $w' = b_1 \dots b_m$ (avec $a_i, b_j \in \Sigma$), on peut caractériser les sous-mots communs de taille maximale comme suit :

- soit n ou m est nul et le seul sous-mot commun est le séquence vide ε de longueur 0,
- soit $a_n = b_m = x$, et tout sous-mot commun maximal u de $a_1 \dots a_{n-1}$ et $b_1 \dots b_{m-1}$ donne un sous-mot commun maximal ux de w et w',
- soit $a_n \neq b_m$ et un sous-mot commun maximal de w et w' est un sous-mot maximal commun de w et $b_1 \dots b_{m-1}$ ou de $a_1 \dots a_{n-1}$ et w'.

Exercice 3. En déduire un algorithme récursif donnant la longueur d'un sous-mot maximal de deux mots donnés w et w'.

Cet algorithme est-il efficace ? Pourquoi ?

Pour éviter de répéter des calculs déjà effectués, on va utiliser une technique de *programmation dynamique* appelée *memoisation*. Elle consiste simplement à stocker les valeurs déjà calculées d'une fonction, afin qu'à un appel ultérieure, on puisse retourner directement la valeur au lieu de la recalculer bêtement.

Exercice 4. Écrire une fonction :

```
int memoized_lcs (char *v, char *w, int i, int j, int **computed);
```

prenant en paramètres deux mots $v = v_1 \dots v_n$ et $w = w_1 \dots w_m$, deux indices i et j, et une matrice d'entiers `computed` et renvoyant la longueur d'un plus long sous-mot commun (*Longest Common Subsequence* en anglais) de $v_1 \dots v_i$ et $w_1 \dots w_j$. On suppose que la matrice `computed` est de taille $(n+1) \times (m+1)$ (avec n la taille de v et m celle de w) et contient en case $(i\,,j)$

- soit la longueur, déjà calculée, d'un plus long sous-mot commun à $v_1 \dots v_i$ et $w_1 \dots w_j$,
- soit -1 si la valeur n'a pas encore été calculée.

En déduire une fonction efficace :

```
int lcs (char *v, char *w);
```

renvoyant la longueur d'un plus long sous-mot commun à v et w.

Exercice 5. Améliorer les fonctions précédentes pour retourner effectivement un sous-mot maximal commun de v et w, et non plus seulement sa longueur.

3 Pour le 29 novembre

Exercice 6. Dans la fonction `memoized_lcs`, utilise-t-on toutes les cases de la matrice ?

Modifier la fonction pour stocker les valeurs déjà calculées non plus dans une matrice mais dans un arbre binaire de recherche (utilisez une version légèrement modifiée de votre bibliothèque). Ainsi, on paye un peu plus cher l'accès aux valeurs calculées ($O(\ln n)$ pour n valeurs calculées contre $O(1)$ dans le cas de la matrice) mais on économise de l'espace mémoire sinon gaspillé[2].

2. On pourrait croire que cela est inutile à l'heure actuelle, avec les RAM importantes dont l'on dispose. Mais il ne faut pas oublier que nos exemples sont ici des "jouets" comparés à l'utilisation qui est réellement faite des algorithmes ici étudiés. D'autre part, on peut aussi penser à l'implémentation de ces algorithmes sur des systèmes embarqués disposant de très peu de mémoire vive (système de bord d'une voiture, montre intelligente, robot, etc.).