

IMA 3^{ème} année

Programmation Avancé

TP7 Arbres d'entiers

1 Objectifs

- Savoir créer et utiliser une bibliothèque d'arbres d'entiers
- Savoir coder quelques algorithmes de base sur les arbres

Préparation : Avant de commencer, copier le répertoire `~wrudamet/public/IMA3/TP7/` qui contient :

- Les fichiers `trees.c` et `trees.h`, contenant des fonctions de base de traitement d'arbres
- Un fichier `main.c` utilisant ces fonctions
- Un sous-répertoire `samples/` qui contient des fichiers de données `.txt` et des fichiers `.dot` et `.pdf` qui permettent de visualiser les arbres correspondant à ces données (section 2).
- Un fichier `tree2pdf.c` qui contient un deuxième main.

2 Fonctions de base dans les ABR

Dans cette première partie, il vous est demandé d'implémenter les fonctions définies dans `tree.h` en utilisant le `main` fourni pour tester (corriger tous les *warnings* à la compilation et utilisez *valgrind*) :

1. `cons_tree(...)` qui construit un ABR (*fait en cours*).
2. `mk_empty_tree(...)`, `is_empty(...)` et `is_leaf(...)`, tous des fonctions auxiliaires. (*Fait en cours*)
3. `add(...)` qui ajout un élément dans un ABR (arbre binaire de recherche, donc **ordonné!**). (*Fait en cours*)
4. `print_tree(...)` qui imprime un ABR par parcours récursif, en ordre ascendant (*infixe*).
Vérifier que la fonction de parcours précédente donne la même suite de valeurs pour les trois fichiers `unspecified.txt`, `balanced.txt`, `degenerated.txt` alors que ce sont des arbres différents (voir les dessins dans le répertoire `samples/`)
5. `load_tree(...)` qui construit un arbre d'entiers à partir d'un fichier.
6. `free_tree(...)` qui libère la mémoire utilisé par un arbre binaire.
7. Écrire une fonction d'impression `print_rec_edges(...)` qui imprime les couples (*père =L=> fils*) pour les fils gauches, et (*père =R=> fils*) pour les fils droits, d'un arbre donné. Ne pas imprimer les sous-arbres vides. À part la flèche *=L/R=>*, l'impression donne des arbres correspondant aux fichiers `.dot` fournis dans le répertoire `samples/`
8. Archiver le fichier `trees.o` dans une bibliothèque `libtrees.a` (commande `ar`).

3 Impression des arbres sous forme graphique

Les fichiers `.pdf` ont été générés à partir des fichiers `.dot` (fournis dans `samples/`). Le format `.dot` est un format de représentation textuelle simple d'arbres¹. Visualisez le format de ces fichiers dans votre éditeur de texte préféré ou par la commande `less`². À partir de ce format `.dot`, la commande `dot` permet de générer une version visualisable (`.pdf`, `.png`, ...) comme suit (en pdf par exemple) :

```
dot -Tpdf samples/balanced.dot -o samples/balanced.pdf
```

L'objectif est alors de générer de tels fichiers pour des arbres construits à partir de fichiers de données (tels que ceux fournis dans `samples/*.txt`).

Préparation : Un fichier `tree2pdf.c` plus conséquent est fourni qui :

1. Génère, par appel à la fonction `generate_dot`, les fichiers `.dot` correspondants aux fichiers de données dont les noms ont été passés en paramètre (e.g. à partir de `samples/balanced.txt` on génère `samples/balanced.dot`).
2. Transforme ces fichiers `.dot` en `.pdf` par appel système à la commande `dot`.

Travail à faire :

1. Programmer la fonction `recursive_dot` qui traite le cas général
2. Tester ensuite par la commande : `./tree2pdf samples/*.txt` Vous devez retrouver les fichiers `.dot` et `.pdf`.

1. Et plus généralement de graphes exploitables par des logiciels tels que <http://www.graphviz.org>
2. digraph signifie *directed graph*, les arbres en font partie du fait de leur orientation `pere->fils`

4 Annexes

```
1 /*
2  * By Walter Rudametkin
3  * Modified from Bernard Carre
4  */
5
6 #include <stdio.h>
7 #include <stdlib.h>
8 #include <stdbool.h>
9
10 typedef struct node {
11     int val;
12     struct node *left;
13     struct node *right;
14 }Node, *PtNode, *Tree;
15 /* These typedefs are optionnel, you may use them */
16
17 /* Constructs a new tree from a value for the root node, a
18    ↪ given left tree and a given right tree */
19 //struct node * cons_tree(int, struct node *, struct node *);
20 void cons_tree(struct node **, int, struct node *, struct node *);
21
22 /* Make an empty tree */
23 //struct node * mk_empty_tree();
24 void mk_empty_tree(struct node **);
25
26 /* Is the tree empty ? */
27 bool is_empty(struct node *);
28
29 /* Is the tree a leaf ? */
30 bool is_leaf(struct node *);
31
32 /* Add the value (int) to the binary search tree,
33    * it must be ordered.
34    * Do not verify the presence of the value,
35    * duplicate values are valid.
36    */
37 void add(struct node **, int);
38
39 /* Print the values of the tree in ascendant order */
40 void print_tree(struct node *);
41
42 /* Build a tree adding values of the file */
43 void load_tree(FILE *, struct node **);
44
45 void free_tree(struct node **);
46
47 void print_rec_edges(struct node *t);
48
49
50 /**
51  * PART 2
52  */
53 void generate_dot(struct node *, FILE *);
54
55 void recursive_dot(struct node *, FILE *fp);
```

```
8 4 2 1 3 6 5 7 12 10 9 11 14 13 15
```

```
samples/balanced.txt
```

```
1 digraph G {
2     8 -> 4;
3     4 -> 2;
4     2 -> 1;
5     2 -> 3;
6     4 -> 6;
7     6 -> 5;
8     6 -> 7;
9     8 -> 12;
10    12 -> 10;
11    10 -> 9;
12    10 -> 11;
13    12 -> 14;
14    14 -> 13;
15    14 -> 15;
16 }
```

```
samples/balanced.dot
```

```
9 4 2 11 3 6 5 15 12 10 8 1 14 13 7
```

```
samples/unspecified.txt
```

```
1 digraph G {
2     9 -> 4;
3     4 -> 2;
4     2 -> 1;
5     2 -> 3;
6     4 -> 6;
7     6 -> 5;
8     6 -> 8;
9     8 -> 7;
10    9 -> 11;
11    11 -> 10;
12    11 -> 15;
13    15 -> 12;
14    12 -> 14;
15    14 -> 13;
16 }
```

```
samples/unspecified.dot
```

```
trees.h
```