TP 8

Juliusz Chroboczek, Yan Jurski et Gabriel Scherer

11 décembre 2014

On a vu au cours du TP précédent les arbres binaires d'entiers ; on travaillera dans ce TP sur des arbres binaires de chaînes de caractères (mots). La plupart des fonctions de base sont les mêmes, et c'est à considérer comme une reprise ou révision : si vous aviez déjà bien compris, les refaire ira vite, et sinon cela vous fera du bien.

On utilisera la structure de données suivante pour implémenter les arbres binaires de mots :

```
struct node {
   char *word;
   int count;
   struct node *left;
   struct node *right;
};
```

En parcourant un texte, vous pourrez rencontrer un mot donné plusieurs fois. Le champ `count` sert à compter le nombre de fois qu'un mot a été rencontré.

Rappel : Un arbre binaire de recherche (ABR) est un arbre binaire dont les nœuds respectent la contrainte suivante : chaque nœud porte une valeur qui est supérieure ou égale à toutes celles de son fils gauche, et cette valeur est aussi strictement inférieure à toutes celles de son fils droit. Ici, la "valeur" d'un nœud est son champ `word`, et on les comparera par ordre alphabétique tel qu'implémenté par la fonction de comparaison `strcasecmp`.

Exercice 1.

1. Écrivez une fonction qui construit un nouveau nœud à partir d'un mot, de son compteur, et deux fils.

   ```
   struct node *mknode(char *word, int count,
                       struct node *left, struct node* right);
   ```

2. `void print_tree(struct node *n)`

   `print_tree` affiche la valeur des nœuds de l'arbre de racine *n* dans l'ordre infixe. Chaque nœud sera affiché sous le format `"%s: %d\n"` : le mot, son compteur, et un retour à la ligne.

3. `void free_tree(struct node *n)`

   `free_tree` libère l'espace mémoire occupé par l'arbre de racine *n*. Modifez votre programme pour vous en servir, et vérifiez à l'aide de `valgrind` que tout l'espace est libéré

Exercice 2.

1. Écrivez une fonction

   ```
   struct node *insert_by_word(char *word, struct node *tree)
   ```

   qui ajoute le mot `word` dans un ABR avec un compte de `1` s'il était absent, et incrémente son compte de `1` s'il est présent. La fonction renvoie l'ABR modifié.
2. Écrivez une fonction

   ```
   struct node *histogram(FILE *f)
   ```

   qui renvoie le nombre d'occurences (le nombre de fois qu'il apparaît) de chaque mot du fichier représenté par `f`. Pour lire le prochain mot (en ignorant les ponctuations etc.) on pourra utiliser la fonction `getword` disponible en

   ```
   http://gallium.inria.fr/~scherer/teaching/eidd/getword.c
   ```

Exercice 3.

1. Écrivez une fonction

   ```
   void print_above(struct node *histo, int count)
   ```

   qui affiche tous les mots de l'arbre `histo` apparaissant plus de `count` fois, dans l'ordre alphabétique et le format `"%s: %d\n"`.
   Écrire un programme qui lit un fichier sur l'entrée standard (`stdin`), et affiche tous les mots qui reviennent plus de `20` fois.

Exercice 4. Les structures d'ABR sur `struct node *` que l'on a utilisé sont ordonnées selon le champ `word`. On va maintenant construire des ABR ordonnés selon `count`.

1. Écrivez une fonction

   ```
   struct node *insert_by_count(char *word, int count, struct node *tree)
   ```

   qui insère un mot et son compte dans un ABR ordonné selon le champ `count`.
2. Écrivez une fonction

   ```
   struct node *copy_tree_by_count(struct node *source, struct node *dest)
   ```

   qui copie tout l'arbre `source` dans un arbre `dest` ordonné selon `count`, et renvoie l'ABR `dest` ainsi modifié.
3. Écrivez une fonction

   ```
   int print_best(struct node *tree, int k)
   ```

   qui affiche les `k` éléments de `tree` (ordonné selon `count`) au compte maximal, dans le format `"%s: %d\n"`, et renvoie le nombre d'éléments réellement affiché (il peut y avoir moins de `k` mots).
   Écrivez un programme qui lit l'entrée standard et affiche les 5 mots qui y apparaissent le plus souvent, et leur compte.