IUT des Pays de l'Adour - RT2
Informatique - Module IC2 - Algorithmique Avancée

Contrôle de TP

Dictionnaire & Arbres Binaires

mercredi 20 mars 2013
durée : 3h
6 pages

Les programmes de correction orthographique ont besoin de tester rapidement si un mot fait partie du dictionnaire ou non, ainsi que d'ajouter facilement des mots au dictionnaire. Une version simple de dictionnaire s'obtient par une représentation arborescente. L'arbre représentant le dictionnaire comporte des nœuds dont le nombre de fils est variable (on parle d'arbre n-aire). Chaque nœud représente une lettre (de type `char`) d'un mot du dictionnaire et a pour fils les lettres pouvant arriver immédiatement derrière. Pour une recherche plus efficace, ces fils sont classés par ordre alphabétique. Le dictionnaire lui-même est (un pointeur sur) la liste des initiales de mots. Comme pour les chaînes de caractères en C, le caractère $'\backslash 0'$ indique la fin d'un mot. Cela est nécessaire pour savoir quels préfixes d'un mot forment des mots du dictionnaire.

Représenter des arbres dont le nombre de fils peut varier est relativement complexe en soi. Afin de simplifier le problème, nous allons plutôt utiliser des arbres binaires interprétés de façon un peu particulière. L'idée est la suivante : chaque nœud de l'arbre binaire `N` est le fils d'un autre nœud `P` (sauf pour la racine). Le fils droit de `N` dans l'arbre binaire correspond au fils suivant de `P` dans l'arbre de départ, et le fils gauche de `N` dans l'arbre binaire correspond au premier de ses fils dans l'arbre de départ. Dans le cas du dictionnaire, donc, le fils droit est un pointeur vers une autre lettre possible en même position et le fils gauche est un pointeur vers la suite possible du mot. Les choses seront plus claires sur un exemple...

Exemple

Si le dictionnaire se compose des mots `"cas"`, `"ce"`, `"ces"`, `"ci"`, `"de"`, `"des"` et `"do"`, sa représentation sous forme d'arbre n-aire sera la suivante (figure 1) :

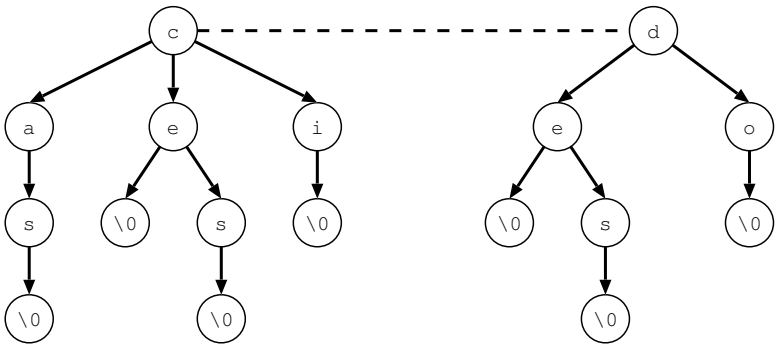

Figure 1 – dictionnaire sous forme d'arbre n-aire

Dans cet exemple, le caractère $'\backslash 0'$ permet de dire que `"ce"` est un mot en même temps qu'une partie de `"ces"`, alors que `"ca"` n'est pas un mot, mais seulement une partie de `"cas"`.

Pour traduire cette représentation en arbre binaire, il suffit de bouger certaines flèches. Nous obtenons ainsi l'arborescence décrite à la figure 2. Nous pouvons ensuite tourner un peu l'arbre pour avoir une vision d'arbre binaire classique (figure 3).

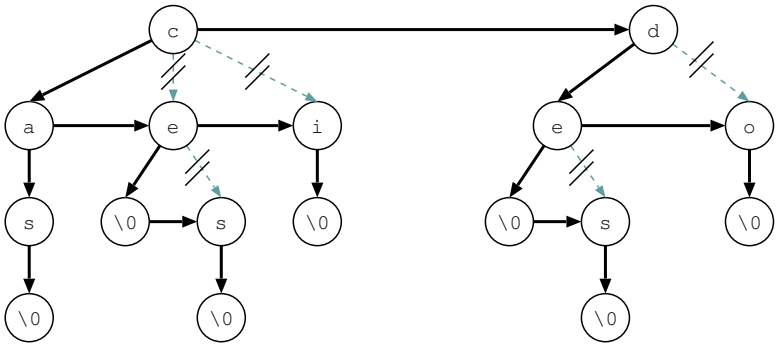

FIGURE 2 – transformation en arbre binaire

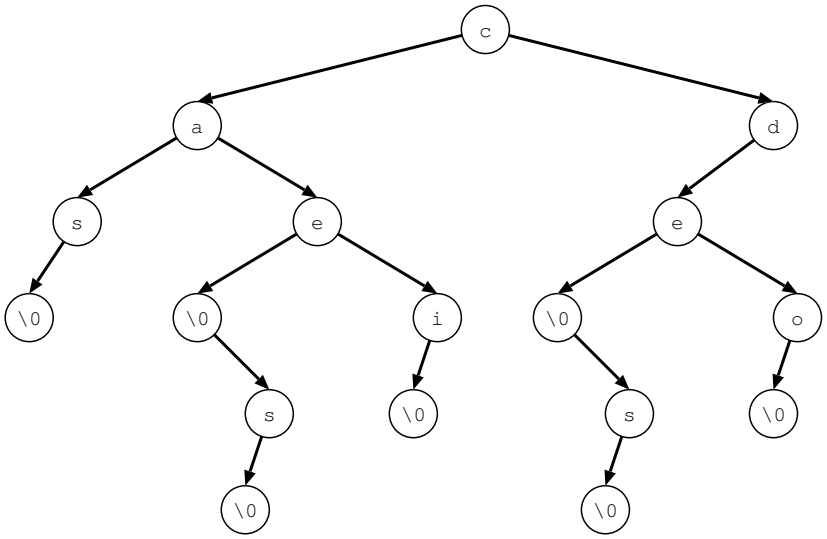

FIGURE 3 – arbre binaire "redressé"

En utilisant la représentation binaire, l'interprétation est la suivante : descendre vers la gauche correspond à passer à la lettre suivante dans le corps d'un mot ; descendre vers la droite correspond à passer à une autre lettre en même position. Notez bien que pour une plus grande efficacité, les lettres qui se suivent de fils droit en fils droit sont ordonnées.

⋆ ⋆ ⋆

TRAVAIL À RÉALISER

L'objectif de ce sujet consiste à développer une (mini) application en langage C qui construit un tel dictionnaire à partir des mots fournis "en dur" par le programme principal. Par rapport à la structure présentée précédemment, nous allons placer un attribut supplémentaire à chaque nœud de l'arbre : un entier. Sur les nœuds intermédiaires, i.e. ceux contenant un caractère différent de $'\backslash 0'$, cet attribut ne sera pas utilisé (valeur `0`). Par contre, sur les feuilles de l'arbre, i.e. les nœuds contenant le caractère $'\backslash 0'$, cet attribut

contiendra le nombre d'occurrences du mot en question. De cette manière, une fois que nous aurons construit ce dictionnaire nous pourrons éditer un certain nombre de statistiques quant au nombre d'apparitions de chaque mot passé en paramètre : nombre total de mots, nombre de mots différents, etc...

☺ **Définition des structures de données et des primitives (fonctions élémentaires) d'accès à ces structure de données ⇒ cette partie vous est donnée : fichiers `arbre.h` et `arbre.c`**

Il s'agit de définir les types de données permettant de stocker un arbre binaire. Afin d'éviter de devoir manipuler cette structure de données directement (i.e. champ par champ), il vous est conseillé de développer une série de fonctions permettant de manipuler "proprement" les arbres binaires. Le reste de votre travail sera basé sur cet ensemble de fonctions. Un exemple de telle bibliothèque de fonctions vous est fourni ci-dessous.

```
/* ------------------------------------------------------ */
/* Primitives de gestion des arbres                       */
/* ------------------------------------------------------ */
TArbre arbreConsVide(void);
int    arbreEstVide(TArbre a);
TArbre arbreCons(char c, int n, TArbre fg, TArbre fd);
char   arbreRacineLettre(TArbre a);
int    arbreRacineNbOcc(TArbre a);
TArbre arbreFilsGauche(TArbre a);
TArbre arbreFilsDroit(TArbre a);
void   arbreSuppr(TArbre a);
/* ------------------------------------------------------ */
```

La fonction `arbreConsVide` retourne un arbre vide (i.e. sans aucun nœud). La fonction `arbreEstVide` nous permet de savoir si l'arbre passé en paramètre est vide ou non. La fonction `arbreCons` prend en paramètre une lettre, un entier, ainsi que deux sous-arbres. Elle construit alors un nouveau nœud avec la lettre et l'entier, puis y greffe les deux sous-arbres. Nous obtenons alors un nouvel arbre dont la racine est le nœud que nous venons de créer. Les fonctions `arbreRacineLettre` et `arbreRacineNbOcc` retournent respectivement la lettre et l'entier stockés dans le nœud racine de l'arbre passé en paramètre. Les fonctions `arbreFilsGauche` et `arbreFilsDroit` retournent respectivement le sous-arbre fils gauche et le sous-arbre fils droit de l'arbre passé en paramètre. La fonction `arbreSuppr` permet de détruire proprement l'arbre passé en paramètre (i.e. libère toute la mémoire qui aurait été allouée par les appels successifs à la fonction `arbreCons`).

⚠ **RAPPEL : cette partie vous est donnée ⇝ fichiers `arbre.h` et `arbre.c`**

1. **Fonctions "évoluées" sur un dictionnaire**

   Dans cette partie, il s'agit de développer des fonctions nous permettant par la suite de manipuler un dictionnaire sans se soucier de savoir s'il s'agit d'un arbre binaire ou autre. Une liste non-exhaustive de telles fonctions vous est fournie ci-dessous.

```
/* ------------------------------------------------------ */
/* Primitives de gestion d'un dictionnaire                */
/* ------------------------------------------------------ */
void   dicoAfficher(TArbre a);
void   dicoInsererMot(char mot[], TArbre *pa);
int    dicoNbOcc(char mot[], TArbre a);
int    dicoNbMotsDifferents(TArbre a);
int    dicoNbMotsTotal(TArbre a);
/* ------------------------------------------------------ */
```

On y retrouve la fonction `dicoAfficher` permettant d'afficher le contenu d'un dictionnaire, la fonction `dicoInsererMot` pour y insérer un nouveau mot (ou incrémenter son nombre d'occurrences si ce mot existe déjà dans le dictionnaire), la fonction `dicoNbOcc` qui retourne le nombre d'apparitions d'un mot (et nous permet également de savoir si ce mot est présent ou non dans le dictionnaire), ainsi que différents fonctions calculant des statistiques.

2. **Programme principal**

   Et oui, il faut bien en arriver là un jour si on veut pouvoir tester tout ce que l'on a fait avant... et corriger les erreurs...

   Comme lors des derniers TP, il vous est demandé de réaliser un petit programme permettant de tester les différentes fonctionnalités (et donc fonctions) de votre application. Parmi les obligations figurent :
   - création d'un dictionnaire vide
   - ajouts de mots au dictionnaire existant
   - affichage du contenu du dictionnaire (avec les occurrences de chaque mot)
   - test d'existence d'un mot dans le dictionnaire (avec son nombre d'occurences le cas échéant)
   - statistiques : nombre total de mots, nombre de mots différents, etc...
   - suppression de toutes les entrées du dictionnaire

⋆ ⋆ ⋆

Annexe : Exemple d'exécution

Voici un exemple de programme principal permettant de tester les quelques fonctions de manipulation d'un dictionnaire dont les entêtes vous ont été données ci-dessus. Le résultat de l'exécution de ce programme vous est présentée à la fin de ce document. L'idée est simplement de vérifier que les insertions de mots dans le dictionnaire se déroulent comme prévu (notamment avec les "préfixe") et que les recherches de mots parcourent cet arbre correctement.

```
/* ------------------------------------------------------ */
/* Eval TP IC2 2013 (MM)                                  */
/* fichier "projet.c"                                     */
/* ------------------------------------------------------ */
#include <stdio.h>
#include "dico.h"
/* ------------------------------------------------------ */
int main(int argc, char **argv)
{
  TArbre dico;
  char buffer[100];

  dico = arbreConsVide();

  strcpy(buffer, "gallon");
  dicoInsererMot(buffer, &dico);
  dicoAfficher(dico);
  printf("\n");
```

```
  strcpy(buffer, "munier");
  dicoInsererMot(buffer, &dico);
  dicoAfficher(dico);
  printf("\n");

  strcpy(buffer, "abenia");
  dicoInsererMot(buffer, &dico);
  dicoAfficher(dico);
  printf("\n");

  strcpy(buffer, "munier");
  dicoInsererMot(buffer, &dico);
  dicoAfficher(dico);
  printf("\n");

  strcpy(buffer, "mumu");
  dicoInsererMot(buffer, &dico);
  dicoAfficher(dico);
  printf("\n");

  strcpy(buffer, "alpha");
  dicoInsererMot(buffer, &dico);
  strcpy(buffer, "alphabeta");
  dicoInsererMot(buffer, &dico);
  strcpy(buffer, "al");
  dicoInsererMot(buffer, &dico);
  dicoAfficher(dico);
  printf("\n");

  printf("\"%s\" \t -> %d\n", "gallon", dicoNbOcc("gallon",dico));
  printf("\"%s\" \t\t -> %d\n", "mumu", dicoNbOcc("mumu",dico));
  printf("\"%s\" \t -> %d\n", "munier", dicoNbOcc("munier",dico));
  printf("\"%s\" \t -> %d\n", "gastro", dicoNbOcc("gastro",dico));
  printf("\"%s\" \t -> %d\n", "lespine", dicoNbOcc("lespine",dico));
  printf("\"%s\" \t\t -> %d\n", "aaa", dicoNbOcc("aaa",dico));
  printf("\n");
}
/* ------------------------------------------------------ */
```

La trace d'exécution de ce programme pourrait ressembler à ceci :

```
"gallon" [1]

"gallon" [1]
"munier" [1]

"abenia" [1]
"gallon" [1]
"munier" [1]

"abenia" [1]
"gallon" [1]
"munier" [2]

"abenia" [1]
"gallon" [1]
"mumu" [1]
"munier" [2]

"abenia" [1]
"al" [1]
"alpha" [1]
"alphabeta" [1]
"gallon" [1]
"mumu" [1]
"munier" [2]

"gallon"         -> 1
"mumu"           -> 1
"munier"         -> 2
"gastro"         -> 0
"lespine"        -> 0
"aaa"            -> 0
```