Vieubled Tanguy
Vernier Paul

IMA3

Rapport Projet Programmation Avancée :
Réalisation d'un correcteur orthographique

The spellchecker

POLYTECH LILLE

Université de Lille

Sommaire

Introduction

L'objectif de ce projet est de concevoir un correcteur orthographique en langage C. Ce correcteur devra détecter les erreurs d'orthographes présentent dans un texte, dans une phrase ou encore dans un mot. Pour cela on utilisera un dictionnaire comme référence. Ce dictionnaire pourra être une simple liste de mots, ou un texte.

Pour que le stockage des mots servant de référence est un impact minimal sur la mémoire tout en fournissant un temps de recherche relativement faible, un début de structure de donnée nous a été imposée, nous devions utiliser une structure sous forme d'arbre préfixe dans laquelle les mots ayant les même préfixes soient factorisés. Par exemple, les mots "Avion" et "Avis" ont un préfixe commun : "Avi", il faut donc que les lettre 'a', 'v' et 'i' ne soient stockées qu'une seule fois.

Cahier des charges

Nous devons définir et implémenter une structure de données permettant de stocker et de manipuler le dictionnaire/texte de référence sous la forme d'un arbre préfixe.
Une fois la structure de données implémentée, nous avons pour mission de charger un fichier texte (dictionnaire) et de le stocker en mémoire pour pouvoir s'en servir comme référence.
Enfin, lorsque le dictionnaire est correctement chargé dans le programme, nous devons analyser l'orthographe d'une phrase entrée par l'utilisateur ou d'un fichier texte, et retourner le nombre de fautes, ainsi que les mots concernés.

Le programme

Présentation de la structure de données choisie

Pour réaliser ce projet nous avons, en premier lieu, réfléchi sur la structure de données qui nous semblait la plus adaptée au cahier des charges.

Nous avons donc réfléchi à plusieurs solutions telles qu'une structure en arbres préfixes composés de tableau de 26 caractères (sans prendre en compte les caractères spéciaux). Cependant, nous nous sommes rendu compte que cette solution n'était pas optimale, bien qu'elle offre un temps de recherche très faible, l'espace mémoire occupé par le dictionnaire devient vite important surtout lorsque l'on souhaite gérer les différents caractères spéciaux propres à certaines langues.

Ensuite, nous avons retenue une solution qui semble plus optimale au niveau mémoire et toujours efficace niveau temps de recherche. Cette structure est une structure composée d'arbres et de listes chaînées :

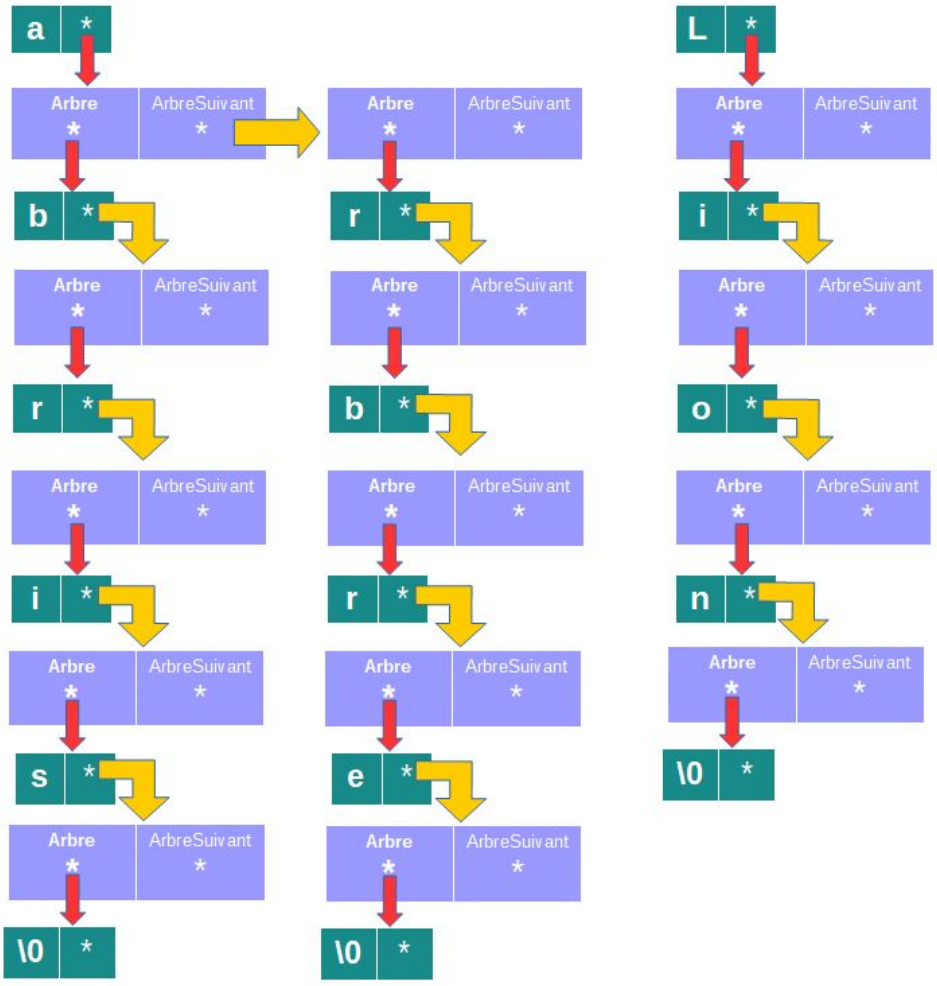
```
struct node {
  wchar_t lettre;
  struct cell* listeFils;
};

struct cell {
  struct node* arbre;
  struct cell* arbreSuivant;
};
```

Un noeud d'arbre est composé d'un caractère, que l'on appelle **'lettre'** et d'un pointeur de cellule qui représente tous les fils de ce noeud, la cellule correspondante à ce pointeur est elle-même composée d'un pointeur d'arbre (**'arbre')** et d'un pointeur de cellule (**'arbreSuivant'**).

Nous avons ensuite choisi de créer un tableau d'arbre en premier lieu avec tous les caractères pouvant commencer un mot ou être un mot. Ce tableau nous permet un recherche plus rapide puisqu'avec la première lettre d'un mot donné, nous savons dans quelle case du tableau nous devons commencer la recherche.

Pour expliquer le fonctionnement de notre structure, nous avons réaliser le schéma suivant :

Mots dans le dictionnaire :
-abris
-arbre
-lion



Présentation des principaux algorithmes

Dans un premier temps le programme commence par initialiser le tableau afin de mettre toutes les premières lettres potentielles de chaque mot.

```
void initialisation_tab_arbre(struct node tab[]) {
  int i = 0;
  for(wchar_t u = 'a'; u < A; u++) {
    tab[i].lettre = u; //ajout lettres minuscules
    tab[i].listeFils = NULL;
    i++;
  }
  tab[i].lettre = '?';
  tab[i+1].lettre = '!';
  tab[i+2].lettre = '\'';
  tab[i+3].lettre = '.';  //Ajout des caractères de ponctuation par défau
  tab[i+4].lettre = ':';
  tab[i+5].lettre = ';';
  for(int j = 0; j <= 5; j++) {
    tab[i+j].listeFils = NULL;
    insertion('\0', &(tab[i+j].listeFils));
  }
}
```

Dans cette fonction, nous commençons par entrer tous les caractères après “a”, nous ne rentrons donc pas les majuscules (que nous gérons plus tard) et certaines ponctuations.
Nous rentrons ensuite les ponctuations et nous insérons aussi “\0” (qui marque la fin d’un mot) afin de ne pas comptabiliser les ponctuations comme erreur car elles ne sont pas présentes dans le dictionnaire.

Nous chargeons ensuite le dictionnaire à l'aide cette fonction.

```
void remplir_dico(FILE* fd, struct node tab_arbre_prcp[]) {

  struct cell** localisationArbre = NULL;
  int cptmot = 0, indice = 0;
  wchar_t motLu[50];
  while(fwscanf(fd, L"%ls", motLu)==1) {
    int estUneLettre = 1;
  int i = 0;
    cptmot += 1;
     if((motLu[0] >= 'A') && (motLu[0] <= 'Z')) {
       localisationArbre = &tab_arbre_prcp[motLu[0]-65].listeFils;
     }
     else if((motLu[0] >= 'a') && (motLu[0] <= 'z')) {
       localisationArbre = &tab_arbre_prcp[motLu[0]-97].listeFils;
     }
     else if((motLu[0] < 'A') || (motLu[0] > 'z')) {
       indice = indice_lettre(tab_arbre_prcp, motLu[0]);
       localisationArbre = &tab_arbre_prcp[indice].listeFils;
       if(indice == -1) {
   wprintf(L"Erreur remplissage dico : L'un des caracteres n'est pas une lettre\n");
   wprintf(L"Mot : %ls incorrect\n", motLu);
   estUneLettre = 0;
       }
     }
     else {
       wprintf(L"Erreur remplissage dico : L'un des caracteres n'est pas une lettre\n");
       wprintf(L"Mot : %ls incorrect\n", motLu);
       estUneLettre = 0;
     }
     while((motLu[i] != '\0') && (estUneLettre == 1)) {
    i += 1;
  localisationArbre = insertion(motLu[i], localisationArbre);
     }
   }
  wprintf(L"\n");
  fclose(fd);
  wprintf(L"%d mots inseres dans le dictionnaire.\n", cptmot);
}
```

Dans cette fonction nous lisons le dictionnaire. La fonction commence par lire la première lettre du mot en charge afin de localiser l'adresse du pointeur associé à cette lettre pour entrer dans le bon arbre. Si le premier caractère n'est pas pris en charge par le programme la fonction informe l'utilisateur. Une fois l'adresse du pointeur récupérer nous pouvons commencer l'insertion :

```
struct cell ** insertion(wchar_t elem, struct cell** pL) {
  if(((*pL) == NULL) || ((*pL)->arbre->lettre > elem)) {
    ajout_tete(elem, pL);
    return &(*pL)->arbre->listeFils;
  }
  else if((*pL)->arbre->lettre == elem) {
    return &(*pL)->arbre->listeFils;
  }
  else {
    return insertion(elem, &(*pL)->arbreSuivant);
  }
}
```

Dans **‘insertion’** à l’image d’une liste chaînée nous comparons l’élément que nous voulons insérer afin de le placer de façon ordonnée, cependant après chaque insertion nous retournons l'adresse de la cellule suivant l’élément que nous venons d’insérer afin de savoir où insérer la suite du mot. Chaque mot se termine par le caractère **‘\0’** ce qui provoque la sortie de la boucle d’insertion.

La fonction **‘remplir_dico ’** se termine en informant le nombre de mots insérer.

Une fois que tous les mots du dictionnaire sont chargés, nous appelons la fonction **‘correction_txt’** qui a quasiment le même fonctionnement que **‘remplir_dico’** sauf que cette fonction compare les mots du texte avec ceux dans le dictionnaire au lieu de les insérer. Une autre différence avec la fonction **‘remplir_dico’** est que lorsqu’un mot finit par une ponctuation (**! ? . ; ,**) nous remplaçons cette dernière par le caractère **‘\0’** pour indiquer la fin du mot.
Lors de la comparaison, si la lettre qu’on regarde n’est pas présente dans notre structure, on sort de la fonction en indiquant une erreur. Lorsque toutes les lettres du mots sont présentes dans notre structure, on regarde si le caractère **‘\0’** est lui aussi présent dans la structure, si c’est le cas, cela indique que le mot est bien présent dans la structure et on sort de la fonction en indiquant que le mot est correct.

Nous avons une deuxième fonction de correction : **‘correction_mot’** qui a exactement le même fonctionnement que **‘correction_txt’**, la seule différence est que cette fonction vérifie les mots qui sont directement entrés par l’utilisateur au clavier, si l’utilisateur souhaite quitter le programme, il suffit qu’il entre le caractère **‘0’** et le programme quitte la fonction.

Enfin, lorsque le programme se termine, nous avons une dernière fonction de désallocation : **‘desallocationTableauArbre’** qui permet de libérer entièrement toutes les cellules mémoires utilisées par notre programme.

Limites du programme

Une limite principale de notre programme est le fait que la gestion des caractères accentués nous force à utiliser un fichier texte qui soit encodé en **'utf-8'**, si ce n'est pas le cas, le chargement du dictionnaire s'arrête dès qu'il arrive sur un caractère spécial (caractère accentué ou autre).
Une autre limite est le fait que certains caractères spéciaux très peu utilisés, tels que les symboles monétaire, peuvent entraîner un dysfonctionnement de notre programme.

Conclusion

Ce projet nous a permis d'améliorer et de confirmer nos compétences en programmation, en particulier sur les structures de données de types arbre et liste chaînée. Il nous a aussi fait découvrir l'outil **'git'** avec lequel nous avons pu travailler à distance et ainsi se répartir les tâches de travail.

Le cahier des charges étant plutôt simple en apparence, nous nous sommes vite rendu que le projet était plus complexe qu'il n'y paraissait, sachant que la structure que nous avons choisi d'utiliser comportée beaucoup de pointeurs, il fallait donc être assez à l'aise avec ces derniers.

Finalement, le projet a été vraiment enrichissant, et il nous a aussi aidé à développer notre capacité à travailler en équipe, nous permettant ainsi de pouvoir partager nos idées.