IN 103 - TD 01

Solution

Pour ce TD, il vous est donné le fichier `french.txt` qui contient une liste de 239185 mots en Français triés par ordre alphabétique.

Le but de ce TD est d'appliquer les techniques de recherche en table vue en cours magistral. Vous devrez constater la différence d'efficacité (donc de complexité) entre deux méthodes de recherche.

Dans le fichier `load_dic.(c|h)` vous est donnée la fonction :
char ∗∗load_file (**char** ∗fname, **unsigned int** ∗size)
qui permet de charger le contenu d'un fichier de mots en mémoire dans un tableau.

Q1 À partir du prototype donné de la fonction `load_file`, déduisez ce que fait cette fonction en terme de structures de données. Pourquoi ces types de paramètres et de retour ?

Solution

Le premier argument est un `char*` donc vraisemblablement une chaîne de caractères, le nom du fichier où lire les mots.

Le second argument est un pointeur sur entier positif. Vu que l'on doit récupérer un tableau de mots, on aura besoin de sa taille pour le manipuler. Ce second argument serait bien cette taille passée par adresse.

Pour terminer, la fonction retourne un `char**`. C'est donc un tableau de tableaux de caractères. Vu que l'on s'attend à récupérer un tableau de mots, c'est donc un tableau de chaînes de caractères.

Q2 La comparaison de chaînes de caractères en C ne se fait pas avec l'opérateur `==`. Pourquoi à votre avis ?

Solution

Une chaîne étant un tableau de `char`, l'opérateur `==` se contente de comparer les adresses des chaînes et non leur contenu.

Pour comparer 2 chaînes de caractères, on utilise la fonction :
int strcmp (**char** ∗s1, **char** ∗s2)
qui retourne 0 si les chaînes sont égales, une valeur < 0 si `s1` $<$ `s2`, une valeur > 0 si `s1` $>$ `s2`.

1 Recherche naïve

Q3 Écrivez une fonction qui prend en argument un mot et le tableau de mots et retourne un booléen disant si le mot a été trouvé dans le tableau. Cette fonction fera une recherche

séquentielle dans le tableau.

Solution

```
   ------------- naivedico.c -------------
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include "load_dic.h"


bool find_word_naive (char **words, char *word, unsigned int num_words)
{
  unsigned int i = 0 ;
  bool found = false ;

  while ((i < num_words) && (! found)) {
    if (strcmp (word, words[i]) == 0) found = true ;
    else i++ ;
  }

  return (found) ;
}

int main (int argc, char *argv[])
{
  char buffer[256] ;
  char **words ;
  unsigned int num_words = 0 ;

  if (argc != 2) {
    printf ("Error: wrong number of arguments. Expected <filename>.\n") ;
    return (1) ;
  }

  words = load_file (argv[1], &num_words) ;
  if (words == NULL) {
    printf ("Error: unable to open dico file '%s'.\n", argv[1]) ;
    return (1) ;
  }

  printf ("Number of words: %d.\n", num_words) ;
  while (gets (buffer)) {
    if (find_word_naive (words, buffer, num_words)) printf ("Found :)\n") ;
    else printf ("Not found :(\n") ;
  }

  return (0) ;
}
```

Q3 Quelle est la complexité de cette fonction en nombre de comparaisons ?

Solution

Chaque recherche va provoquer un parcours linéaire du tableau. Donc la complexité est linéaire en la taille du tableau (on ne considère pas la complexité de la comparaison de chaînes).

Dans la suite, on voudra lire des mots au clavier jusqu'à ce que ^ -d ait été pressé. La fonction :

char* gets (**char** *str)

permet de lire une chaîne au clavier en la rangeant dans le tableau (pré-alloué par vous) `str` et retourne un pointeur vers ce tableau. Le pointeur retourné permet juste de déterminer si une

chaîne a bien pu être lue : si oui, la valeur retournée est $\neq$ `NULL`, sinon, la valeur retournée est = `NULL` et cela signifie que la ≪fin de fichier≫ et atteinte (pression de ^ -d au terminal).

Q4 Complétez votre programme avec un `main ()` qui permette de saisir des mots et de vérifier s'ils existent dans le dictionnaire. Pour arrêter le programme, il suffira de presser ^ -d au terminal.

Q5 Testez votre programme. Il se peut qu'en fonction du codage des lettres accentuées, les mots contenant des accents ne soient pas trouvés.

Q6 Maintenant, petit test de performance, testez que tous les mots du dictionnaire sont bien dans le dictionnaire. Comment allez-vous vous y prendre ?

Solution

Entrer manuellement tous les mots au clavier risque d'être plus que fastidieux. Vous avez vu en MO101 la redirection de fichier sur l'entrée standard : utilisons-la.

```
./naive.x french.txt < french.txt
```

N'empêche que c'est long à terminer...

2 Recherche dichotomique

Dans cette section, nous allons implémenter une recherche dichotomique dans le tableau.

Q7 Esquissez l'algorithme. Quel sont les cas d'arrêt, quelle est la structure des traitements ?

Solution

Intéressons-nous d'abord à la structure de l'algorithme. À chaque appel récursif, il faut calculer l'indice au milieu du tableau. Ensuite on compare le mot se trouvant à et indice avec le mot recherché. S'ils sont égaux, on a trouvé et c'est fini. Si le mot recherché est plus petit que le mot courant, il faut aller chercher dans la partie ≪gauche≫ du tableau, sinon dans la partie ≪droite≫.

Donc, notre fonction de recherche doit prendre en argument le mot cherché, le tableau et les bornes entre lesquelles effectuer la recherche. Elle doit retourner un booléen pour dire si elle a trouvé ou non.

Nous avons identifié le cas d'arrêt avec succès, mais il reste le cas où le mot n'existe pas dans le dictionnaire. Dans ce cas, que se passe-t-il au cours de la récursion ? À chaque fois, on recherche avec la borne ≪droite≫ qui diminue ou la borne ≪gauche≫ qui augmente. À un moment, elles vont se croiser et l'indice de ≪droite≫ deviendra supérieur à celui de ≪gauche≫ et à ce moment on aura bien réduit l'espace de recherche à vide sans rien trouver. Donc, si `gauche > droite` il faut arrêter la recherche et signaler un échec.

Q8 Implémentez la fonction de recherche par dichotomie.

Solution

```
-------------- dichodico.c --------------
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

#include "load_dic.h"


bool find_word_dicho (char **words, char *word, int left, right)
{
  int mid ;
  int cmp ;

#ifdef DEBUG
  printf ("Search between %d %d\n", left, right) ;
#endif
  if (left > right) return (false) ;
  mid = (left + right) / 2 ;

  /* Is word > words[mid] ? */
  cmp = strcmp (word, words[mid]) ;
#ifdef DEBUG
  printf ("Index: %d %s ? %s Cmp: %d\n", mid, word, words[mid], cmp) ;
#endif
  if (cmp == 0) return (true) ;      /* word == words[mid] */

  /* word < words[mid] ? */
  if (cmp < 0) return (find_word_dicho (words, word, left, mid - 1)) ;

  /* Obviously word > words[mid]. */
  return (find_word_dicho (words, word, mid + 1, right)) ;
}



int main (int argc, char *argv[])
{
  char buffer[256] ;
  char **words ;
  unsigned int num_words = 0 ;

  if (argc != 2) {
    printf ("Error: wrong number of arguments. Expected <filename>.\n") ;
    return (1) ;
  }

  words = load_file (argv[1], &num_words) ;
  if (words == NULL) {
    printf ("Error: unable to open dico file '%s'.\n", argv[1]) ;
    return (1) ;
  }

  printf ("Number of words: %d.\n", num_words) ;

  while (gets (buffer)) {
    if (find_word_dicho (words, buffer, 0, num_words)) printf ("Found :)\n") ;
    else printf ("Not found :(\n") ;
  }

  return (0) ;
}
```

Q9 Refaites le test de la question **Q6**.

Q10 Quelle est la complexité de cette fonction en nombre de comparaisons ?

Solution

À chaque itération on divise l'espace de recherche par 2. Donc dans le pire des cas, il faudra diviser jusqu'à ce que l'espace de recherche soit de taille 1.

Donc pour un tableau de taille n, dans le pire des cas il faudra diviser k fois par 2 pour arriver à un espace de taille 1 :

$$\underbrace{(((n/2)/2)\dots)}_{k \text{ fois}} = 1$$

Donc $\frac{n}{2^k} = 1$
d'où $k = log_2(n)$

La complexité en nombre de comparaisons k est donc logarithmique en la taille n du tableau.