Énoncé mini projet Algorithmique : Arbre binaire de recherche

Enseignant responsable : Mr. Sahbi Bahroun

Enseignant du cours: Mr. Adel Khalfallah

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 "\0" indique la fin d'un mot. Cela est nécessaire pour savoir quels préfixes d'un mot forment des mots du dictionnaire.

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 :

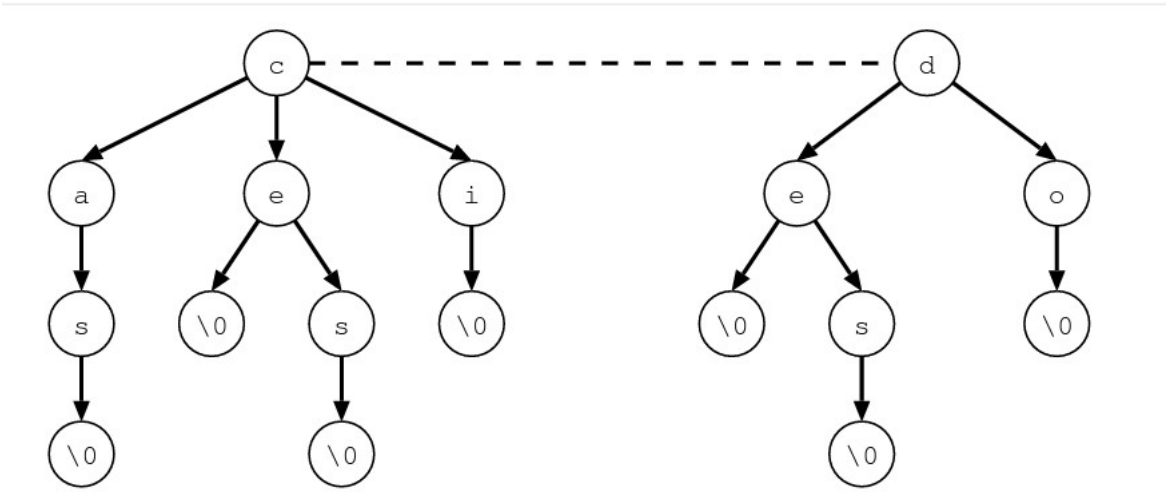

Figure 1 : dictionnaire sous forme d'arbre n-aire

Dans cet exemple, le caractère \0permet 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).

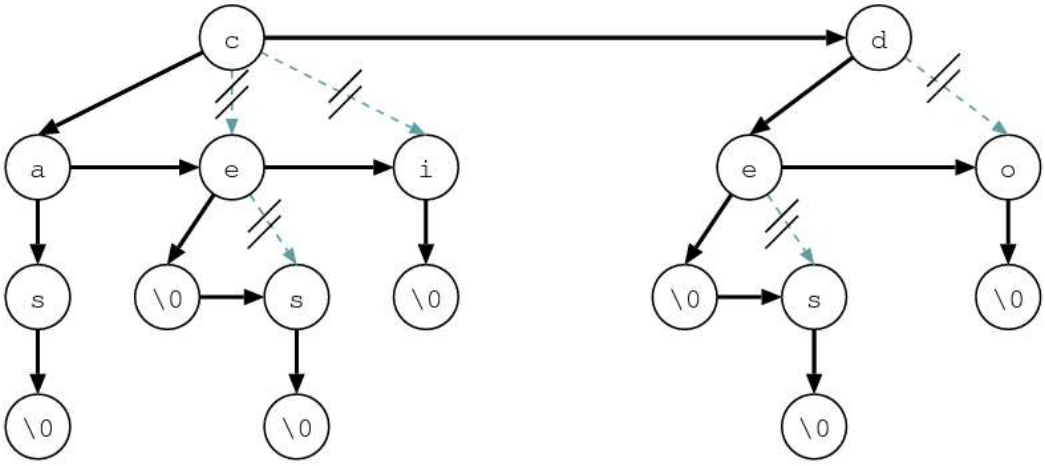

Figure 2 : Transformation en arbre binaire

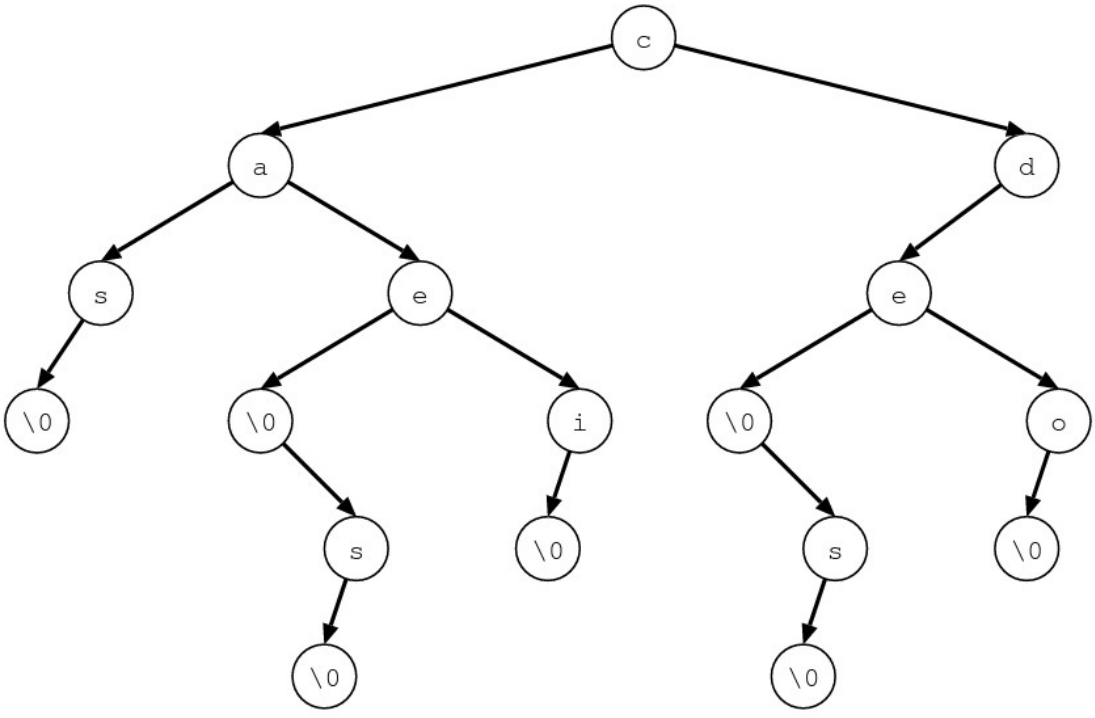

Figure 3 : Arbre binaire redressé

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 "\0", cet attribut ne sera pas utilisé (valeur 0). Par contre, sur les feuilles de l'arbre, i.e. les nœuds contenant le caractère "\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 structures 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).

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.

Programme principal

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 1: *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 présenté 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 1 ING 2021 (ISI) */
/* 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"); }
 /* ----------------------------------------------------*/
```

Annexe 2: *Création du dictionnaire*

Je vais donc créer un fichier dico.txt **dans le même répertoire que mon projet**. Mettez un nombre de mots pas moins de 100. Les mots sont écrits un mot par ligne. Exemple :

Maison

Bleu

Avion

Abeille

Immeuble

Garage

...

Une fois que j'aurai terminé de coder le programme, on ajouterai évidemment des tooonnes de mots tordus comme XYLOPHONE, ou à rallonge comme ANTICONSTITUTIONNELLEMENT. Mais pour le moment, retournons à nos instructions.

Préparation des nouveaux fichiers

La lecture du « dico » va demander pas mal de lignes de code (du moins, j'en ai le pressentiment). Je prends donc les devants en ajoutant un nouveau fichier à mon projet : dico.c (qui sera chargé de la lecture du dico).

Dans la foulée, je crée le dico.h qui contiendra les prototypes des fonctions contenues dans dico.c.

Dans dico.c, je commence par inclure les bibliothèques dont j'aurai besoin ainsi que mon dico.h.

A priori, comme souvent, j'aurai besoin de stdio et stdlib ici. En plus de cela, je vais être amené à piocher un nombre au hasard dans le dico, je vais donc inclure time.h comme on l'avait fait pour notre premier projet « Plus ou Moins ». Je vais aussi avoir besoin de string.hpour faire un strlen vers la fin de la fonction :

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>


#include "dico.h"
```

La fonction piocherMot

Cette fonction va prendre un paramètre : un pointeur sur la zone en mémoire où elle pourra écrire le mot. Ce pointeur sera fourni par le main().
La fonction renverra un int qui sera un booléen : 1 = tout s'est bien passé, 0 = il y a eu une erreur.

Voici le début de la fonction :

```
int piocherMot(char *motPioche)
{
  FILE* dico = NULL; // Le pointeur de fichier qui va contenir notre fichier
  int nombreMots = 0, numMotChoisi = 0, i = 0;
  int caractereLu = 0;
```

Je définis quelques variables qui me seront indispensables. Comme pour le main(), je n'ai pas pensé à mettre toutes ces variables dès le début, il y en a certaines que j'ai ajoutées par la suite lorsque je me suis rendu compte que j'en avais besoin.

Les noms des variables parlent d'eux-mêmes. On a notre pointeur sur fichier dico dont on va se servir pour lire le fichier dico.txt, des variables temporaires qui vont stocker les caractères, etc.

Notez que j'utilise ici un int pour stocker un caractère (caractereLu) car la fonction fgetc que je vais utiliser renvoie un int. Il est donc préférable de stocker le résultat dans un int.

Passons à la suite :

```
dico = fopen("dico.txt", "r"); // On ouvre le dictionnaire en lecture seule


// On vérifie si on a réussi à ouvrir le dictionnaire
if (dico == NULL) // Si on n'a PAS réussi à ouvrir le fichier
{
  printf("\nImpossible de charger le dictionnaire de mots");
  return 0; // On retourne 0 pour indiquer que la fonction a échoué
  // À la lecture du return, la fonction s'arrête immédiatement.
}
```

Je n'ai pas grand-chose à ajouter ici. J'ouvre le fichier dico.txt en lecture seule ("r") et je vérifie si j'ai réussi en testant si dico vaut NULL ou non. Si dico vaut NULL, le fichier n'a pas pu être ouvert (fichier introuvable ou utilisé par un autre programme). Dans ce cas, j'affiche une erreur et je fais un **return** 0.

Pourquoi un **return** là ? En fait, l'instruction **return** commande l'arrêt de la fonction. Si le dico n'a pas pu être ouvert, la fonction s'arrête là et l'ordinateur n'ira pas lire plus loin. On retourne 0 pour indiquer au main que la fonction a échoué.

Dans la suite de la fonction, on suppose donc que le fichier a bien été ouvert.

```
// On compte le nombre de mots dans le fichier (il suffit de compter les entrées \n
do
{
  caractereLu = fgetc(dico);
  if (caractereLu == '\n')
    nombreMots++;
} while(caractereLu != EOF);
```

Là, on parcourt tout le fichier à coups de fgetc (caractère par caractère). On compte le nombre de \n (entrées) qu'on détecte. À chaque fois qu'on tombe sur un \n, on incrémente la variable nombreMots.

Grâce à ce bout de code, on obtient dans nombreMots le nombre de mots dans le fichier. Rappelez-vous que le fichier contient un mot par ligne.

```
numMotChoisi = nombreAleatoire(nombreMots); // On pioche un mot au hasard
```

Ici, je fais appel à une fonction de mon cru qui va génèrer un nombre aléatoire entre 1 et nombreMots (le paramètre qu'on envoie à la fonction).

C'est une fonction toute simple que j'ai placée aussi dans dico.c (je vous la détaillerai tout à l'heure). Bref, elle renvoie un nombre (correspondant à un numéro de ligne du fichier) au hasard qu'on stocke dans numMotChoisi.

```
// On recommence à lire le fichier depuis le début. On s'arrête lorsqu'on est arrivé au bon mot
```

```
rewind(dico);
while (numMotChoisi > 0)
{
   caractereLu = fgetc(dico);
   if (caractereLu == '\n')
      numMotChoisi--;
}
```

Maintenant qu'on a le numéro du mot qu'on veut piocher, on repart au début grâce à un appel à rewind(). On parcourt là encore le fichier caractère par caractère en comptant les \n. Cette fois, on décrémente la variable numMotChoisi. Si par exemple on a choisi le mot numéro 5, à chaque entrée la variable va être décrémentée de 1.

Elle va donc valoir 5, puis 4, 3, 2, 1… et 0.

Lorsque la variable vaut 0, on sort du **while**, la condition numMotChoisi > 0 n'étant plus remplie.

Ce bout de code, que vous devez impérativement comprendre, vous montre donc comment on parcourt un fichier pour se placer à la position voulue. Ce n'est pas bien compliqué, mais ce n'est pas non plus « évident ».

Assurez-vous donc de bien comprendre ce que je fais là.

Maintenant, on devrait avoir un curseur positionné juste devant le mot secret qu'on a choisi de piocher.

On va le stocker dans motPioche (le paramètre que la fonction reçoit) grâce à un simple fgets qui va lire le mot :

```
/* Le curseur du fichier est positionné au bon endroit.
On n'a plus qu'à faire un fgets qui lira la ligne */
fgets(motPioche, 100, dico);


// On vire le \n à la fin
motPioche[strlen(motPioche) - 1] = '\0';
```

On demande au fgets de ne pas lire plus de 100 caractères (c'est la taille du tableau motPioche, qu'on a défini dans le main). N'oubliez pas que fgets lit toute une ligne, y compris le \n.

Comme on ne veut pas garder ce \n dans le mot final, on le supprime en le remplaçant par un \0. Cela aura pour effet de couper la chaîne juste avant le \n.

Et… voilà qui est fait ! On a écrit le mot secret dans la mémoire à l'adresse de motPioche.

On n'a plus qu'à fermer le fichier, à retourner 1 pour que la fonction s'arrête et pour dire que tout s'est bien passé :

```
fclose(dico);
```

```
  return 1; // Tout s'est bien passé, on retourne 1
}
```

Pas besoin de plus pour la fonction piocherMot !

La fonction nombreAleatoire

C'est la fonction dont j'avais promis de vous parler tout à l'heure. On tire un nombre au hasard et on le renvoie :

```
int nombreAleatoire(int nombreMax)
{
  srand(time(NULL));
  return (rand() % nombreMax);
}
```

La première ligne initialise le générateur de nombres aléatoires, comme on a appris à le faire dans le premier TP « Plus ou Moins ».
La seconde ligne prend un nombre au hasard entre 0 et nombreMax et le renvoie. Notez que j'ai fait tout ça en une ligne, c'est tout à fait possible, bien que peut-être parfois moins lisible.

Le fichier dico.h

Il s'agit juste des prototypes des fonctions. Vous remarquerez qu'il y a la « protection » #ifndef que je vous avais demandé d'inclure dans tous vos fichiers .h (revoyez le chapitre sur le préprocesseur au besoin).

```
#ifndef DEF_DICO
#define DEF_DICO


int piocherMot(char *motPioche);
int nombreAleatoire(int nombreMax);


#endif
```

Le fichier dico.c

Voici le fichier dico.c en entier :

```
/*
Jeu du Pendu


dico.c
------


Ces fonctions piochent au hasard un mot dans un fichier dictionnaire
```

```c
pour le jeu du Pendu
*/


#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>


#include "dico.h"


int piocherMot(char *motPioche)
{
  FILE* dico = NULL; // Le pointeur de fichier qui va contenir notre fichier
  int nombreMots = 0, numMotChoisi = 0, i = 0;
  int caractereLu = 0;
  dico = fopen("dico.txt", "r"); // On ouvre le dictionnaire en lecture seule


  // On vérifie si on a réussi à ouvrir le dictionnaire
  if (dico == NULL) // Si on n'a PAS réussi à ouvrir le fichier
  {
     printf("\nImpossible de charger le dictionnaire de mots");
     return 0; // On retourne 0 pour indiquer que la fonction a échoué
     // À la lecture du return, la fonction s'arrête immédiatement.
  }


  // On compte le nombre de mots dans le fichier (il suffit de compter les
  // entrées \n
  do
  {
     caractereLu = fgetc(dico);
     if (caractereLu == '\n')
        nombreMots++;
  } while(caractereLu != EOF);


  numMotChoisi = nombreAleatoire(nombreMots); // On pioche un mot au hasard


  // On recommence à lire le fichier depuis le début. On s'arrête lorsqu'on est arrivé au bon
mot
  rewind(dico);
  while (numMotChoisi > 0)
  {
     caractereLu = fgetc(dico);
     if (caractereLu == '\n')
```

```
        numMotChoisi--;
    }


    /* Le curseur du fichier est positionné au bon endroit.
    On n'a plus qu'à faire un fgets qui lira la ligne */
    fgets(motPioche, 100, dico);


    // On vire le \n à la fin
    motPioche[strlen(motPioche) - 1] = '\0';
    fclose(dico);


    return 1; // Tout s'est bien passé, on retourne 1
}


int nombreAleatoire(int nombreMax)
{
    srand(time(NULL));
    return (rand() % nombreMax);
}
```