TP : vérificateur orthographique

El KHALDI Ahannach Yassin
ZENNOU Massine

BUT:

Programmer un vérificateur orthographique en C qui, après avoir construit un dictionnaire à partir d'un fichier donné, parcourra un texte et affichera les mots n'appartenant pas au dictionnaire. Cela en implémentant de trois structures de données pour stocker le dictionnaire, une table de hachage , un arbre préfixe , un arbre lexicographique

Table de Hachage:

Dans notre programme nous utilisons une fonction de hachage qui associe a chaque mot un nombre entier entre 0 et 2000, cet entier est la somme de des codes ascii des lettres du mot.

```c
#define MAX 50
#define TAILLE_HACHAGE 2000

int hash(char* mot_t){
    int a=0;
    char mot[MAX];
    strncpy(mot,mot_t,sizeof(mot));
    for(unsigned int i=0;i<strlen(mot_t);i++){
        a=a+mot[i];
    }
    return a%TAILLE_HACHAGE;
}
```

Notre table de hachage est une tableau de 2000 cases, chaque case étant une liste chainée et stocke les élément en collision sous une liste chainée dans un même maillon.

```c
typedef struct noued{
    char word[MAX];
    struct noued* next;
}noued;
noued* hash_tab[TAILLE_HACHAGE];

void ajouter(noued** pile,noued* sommet){
    sommet->next=*pile;
    *pile=sommet;
}
```

Arbre Préfixe:

Pour cette méthode d'implémentation, nous utilisons une liste chaînée de noeuds (chaque noeud représente une lettre), dans ce noeud l'élément suivant est une table de 26 noeuds vide en premier temps, et une variable est_mots qui est true si le mot est final.

La table de nœuds qui est remplie par un nœud en case i si la lettre i (lettre d'ordre i dans alphabet) existe sinon ce maillon reste null indiquant que cette combinaison n'existe pas.

Tout cela est stocké sur un noeud global noté tree dans notre cas

```
#define MAX 50

noeud* creer_noeud(){
    noeud*  n=malloc(sizeof(noeud));
    for(int i=0;i<NUM_ALPHA;i++){
        n->fils[i]=NULL;
    }
    n->est_mot=false;//tous les mot ne sont pas final a leur creation;
    return n;
}
```

Pour ce faire, nous utilisons une fonction ajouter mot, qui parcourt l'arbre et ajoute la chaque lettre en fils (si elle n'existe pas déjà)

```
void ajouter_mot(arbre* tree,char* mot){//ajouter un mot a une arbre
    if(*tree==NULL)
    {
        (*tree)=creer_noeud();
    }
    noeud* tmp=(*tree);

    unsigned int longueur_mot=strlen(mot);

    for(unsigned int i=0;i<longueur_mot;i++){

        if(tmp->fils[(unsigned int)mot[i]-(unsigned int)('a')]==NULL)
        {

            tmp->fils[(unsigned int)mot[i]-(unsigned int)('a')]=creer_noeud();
        }

        tmp=tmp->fils[(unsigned int)mot[i]-(unsigned int)('a')];

    }

    tmp->est_mot=true;
}
```

Pour implémenter notre dictionnaire, nous utilisons la fonction Fichier_en_arbre qui effectue l'implémentation de chaque mot de notre dictionnaire (FR.txt) dans notre arbre (tree)

```c
arbre Fichier_en_arbre(char* fichier){
    arbre tree=NULL;
    char chaine[50];
    FILE* f=fopen(fichier,"r");
    if(f!=NULL){
        while(fgets(chaine,50,f)){
            chaine[strlen(chaine)-1]='\0';
            ajouter_mot(&tree,chaine);
        }
    }
    fclose(f);
    return tree;
}
```

Arbre Lexicographique:

Dans cette méthode on utilise une liste chaînée 2D de maillons représentant une lettre chacun, ayant comme suivant un fils et un frère, et un bool indiquant si le mot est final ou non.

Le fils est rempli lors d'un début d'un mot par la première lettre, ensuite la deuxième lettre est ajoutée comme frère d'ordre 1, en augmentant d'ordre à chaque ajout, lorsque la lettre d'ordre n n'existe pas dans notre chaîne, celle ci est ajoutée comme frère en ordre n en tenant compte des lettres précédentes

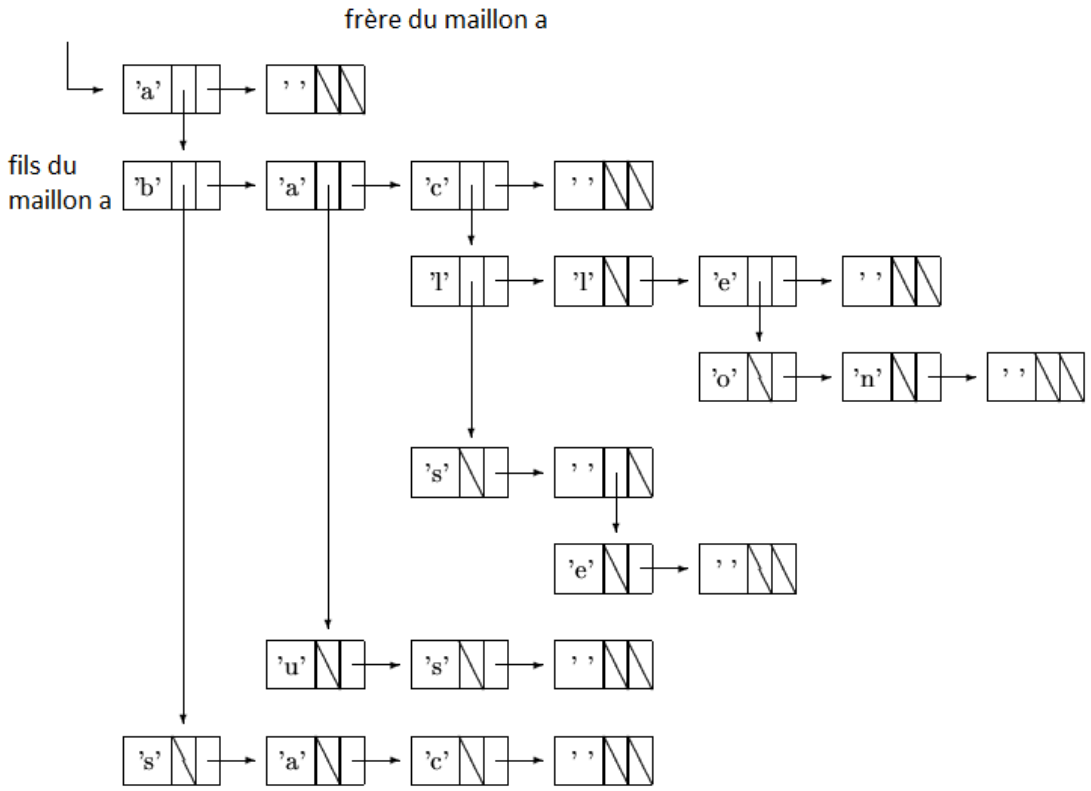
```c
typedef struct maillion{
    char lettre;
    struct maillion* fils;
    struct maillion* frere;
    bool est_mot;
}maillion,maillon_t;

typedef maillion* arbre;
```

Prenant un exemple simple, pour le dictionnaire suivant:

a
bac
balle
ballon
bas
base
bus
sac

On peut le représenter de la façon suivante:



Fonction Trouver:

Notre fonction trouver a pour but de mettre tous les mots du text qui ne sont pas dans le dictionnaire dans un fichier résultat.

```c
void trouver_(char* text,arbre* a){//mettre tous les mot du text qui ne sont pas dans dans le dictionnaire dans un fihier resultat.
    char mot[MAX];
    for(int i=0;i<MAX;i++){
        mot[i]='\0';
    }
    char c;
    int i=0;
    FILE* resultat=fopen("./tests/resultats_arbre_prefix","w");
    if(resultat==NULL){
        printf("erreur d'ouverture du fichier\n");
        exit(1);
    }
    fputs("************************************Methode de l'arbre_prefix*******************************\n\n",resultat);
    fputs("les mot qui sont dans le dictionnaire mais pas dans le texte sont: \n\n",resultat);
    FILE* fichier=fopen(text,"r");
    do{//caratere par caractere jusqu'a trouver un caratere speciale;
        c=fgetc(fichier);
        if((c>='a'&& c<='z') || (c>='A' && c<='Z')){
            mot[i]=c;
            i++;
        }else if(mot[0]!='\0'){
            if(!search(*a,mot)){
                fputs(mot,resultat);
                fputs("\n",resultat);
            }
            for(int i=0;i<MAX;i++){
                 mot[i]='\0';
            }
            i=0;
        }
    }while(c!=EOF);
    fclose(fichier);
    fclose(resultat);
}
```

Comparaison des trois méthodes:

La différence entre les trois méthodes en terme de performance et de vitesse de compilation est principalement dû à la mémoire qu' utilise chacune des méthode et de la façon avec laquelle on parcourt notre structure que ça soit arbre ou table de hachage évidemment on aura un temps d'exécution plus grand dans la table de hachage car c'est principe basique de stockage des arbre et il refait a chaque fois des iteration qui peuvent être réduit si on stocke le dictionnaire et voici les résultats de l'exécution de chacune des méthode :

```
[phelma@localhost dictionnaire]$ ./bin/hashtab FR.txt a_la_recherche_du_temps_perdu.txt



********************************************************************************
le temps de l'execution de la methode table de hashage est : 11.510000 s
le resultat est dans le dossier tests
********************************************************************************
```

```
[phelma@localhost dictionnaire]$ ./bin/arbre_prefix FR.txt a_la_recherche_du_temps_perdu.txt



********************************************************************************
le temps de l'execution de la methde arbre prefix est : 0.480000 s
le resultat est dans le dossier tests
********************************************************************************
```

```
[phelma@localhost dictionnaire]$ ./bin/arbre_lexi FR.txt a_la_recherche_du_temps_perdu.txt



********************************************************************************
le temps d'execution pour l'arbre lexicograpphique est 0.830000 s
le resultat est dans le dossier tests
********************************************************************************
```

Sources:

https://fr.wikipedia.org/wiki/Trie_(informatique)
https://www.enseignement.polytechnique.fr/informatique/profs/Philippe.Chassignet/00-01/TD/td_6.html
https://tdinfo.phelma.grenoble-inp.fr/2ASEOCALGO/TD/TD6/
https://pageperso.lis-lab.fr/~benjamin.monmege/diu_eil_semaine4/cours/cours9_slides.pdf