Documentation
Indexeur de texte

Yoann Thomann
Mohammed Bouabdellah

ythomann@univ-mlv.fr

mbouabde@univ-mlv.fr

12/04/2010

Table des matières

1 Description générale **2**

2 Manuel Utilisateur **2**
2.1 Conseils d'utilisation 3
2.2 Recommandation 3

3 Manuel Developpeur **4**
3.1 Parseur de mot 4
3.2 Parseur de mot 4
3.3 Structures utilisées 5
3.4 Hachage ouvert 6
3.5 Gestion des listes 6
3.6 Création de la liste triée 7
3.7 Contrôle qualité 8

1 Description générale

Ce programme permet, à partir d'un texte, de représenter un dictionnaire en utilisant une table de hachage. Il indexe notamment des mots, en les associant à toutes les positions des phrases dans lequels ils peuvent apparaître dans le texte.

Il est donc possible en executant ce programme, pour un mot, de tester :

- l'appartenance d'un mot au texte
- l'affichage des positions d'un mot dans le texte
- l'affichage des phrases contenant un mot
- l'affichage de la liste triée des mots et de leur position
- l'affichage des mots commençant par un préfixe donné
- la sauvegarde de l'index

2 Manuel Utilisateur

Il existe deux manières d'éxecuter ce programme :

```
./Index [fichier]
```

et

```
./Index [option] [mot] fichier
```

Ces informations sont accessible a partir de l'executable via la commande :

```
yoann@hp_laptop# ./Index -h

SYNOPSIS:
Index [option] file
        or
Index [file]

Examples:
Index -a word file      | Check if word is in file.
Index -p word file      | Print word positions in file.
Index -P word file      | Print sentences containing word in file.
Index -l text           | Print sorted list of text's words.
Index -d word file      | Print words begining with word in the text.
```

```
Index -D out  file      | Save sorted list of file's words in out
    .DICO
Index -h out  file      | Print this help
```

La première commande donne accès à un menu permettant de manipuler les diverses fonctions de l'index.
La deuxième permet d'effectuer des opérations ponctuelles selon les arguments de la commande, cette deuxieme est particulièrement adaptée lors de l'utilisation du programme dans un script.
L'utilisateur pour ce faire peut rediriger la sortie standard, puis la traiter...

2.1 Conseils d'utilisation

Le fichier passé en argument doit être un fichier texte. Il est possible lors du traitement du texte de voir l'evolution du traitement en utilisant l'option verbose **-v**.
Example :

```
./Index -pv oreste Andromaque.txt
```

2.2 Recommandation

La principale recomandation d'utilisation dans le cas du test d'appartenance d'un mot dans le texte en mode interactif, un hash est systématiquement éffectué dans ce mode, ainsi si l'utilisateur souhaite uniquement rechercher la présence d'un mot dans le texte, il serait plus efficace d'utiliser la commande :

```
./Index -a word Colomba.txt
```

En effet, cette commande n'engendre elle pas la création d'un hash, elle recherche directement dans le fichier en le parcourant alors qu'en mode interractif, un hash du fichier est crée et le mot est recherché dans celui çi.

3 Manuel Developpeur

3.1 Parseur de mot

Pour récuperer les mots du textes, on aurait pu utiliser une fonction du type *fgets/fscanf*, mais ces fonctions 'voient' un mot est comme une suite de caractère séparés par des espaces, retour chariot, tabulation ; ce qui ne correspond pas à notre définition d'un mot.
En effet, un mot est pour nous une suite de caractères espacés par des espaces, retour chariot, tabulation, et divers caractères de ponctuations. Il nous a donc fallu récuperer les mots caractère par caractère.
Pour la récupération de mots, on peut distinguer 3 cas :

- fin de mot
- fin de phrase
- fin de texte

Face à ces 3 cas, on agit de la sorte : si une fin de mot se présente, on stocke le mot dans la table de hashage ou met a jour sa position (si déjà présent), si une fin de phrase se présente, on met à jour la valeur de l'offset (position dans le texte en octet) ; et enfin dans le cas d'une fin de texte on s'arrête là.

3.2 Parseur de mot

Le parseur de mot est assez important car il ne récupère que des mots qui possèdent des caractères alphanumériques. Des suites de caractères comme par exemple '****','*...*' ne sont pas considerés comme mots.
De plus, le parseur à la faculté de placer le curseur du fichier au bon endroit pour la lecture du prochain mot, en detectant la présence d'un caractère de fin de phrase après le mot. Ainsi dans la chaîne *'Bonjour !'*, le mot *Bonjour*, suivi d'un espace, est detecté comme étant un mot de fin de phrase, et non pas comme un mot courant d'une phrase.

Ce parseur correspond dans ce programme à la fonction, dont voici le prototype :

```
1 int get_word(FILE* stream, char* dest, size_t n);
```

Valeurs de retour :

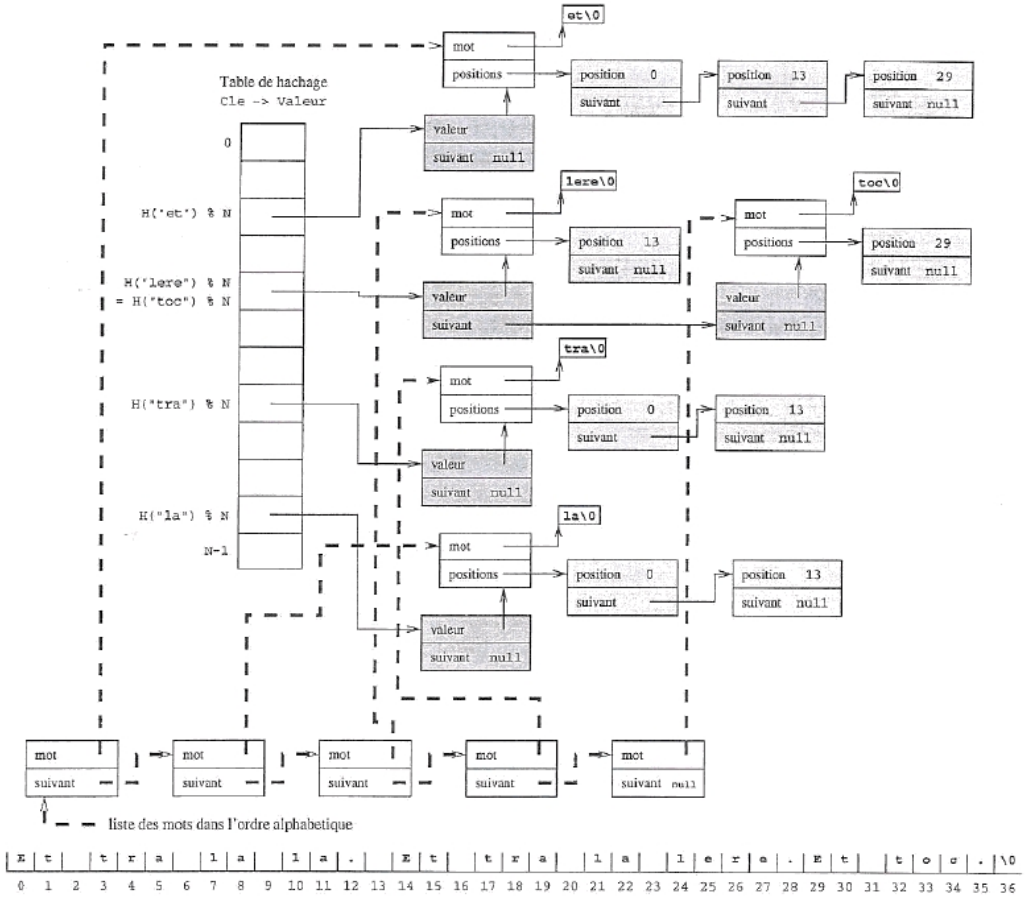
- EOF : Fin de fichier
- 1 : Fin de mot
- 2 : Fin de phrase

Paramètres :

- stream : flux dans lequel on récupère nos mots (bien ouvert au préalable)
- dest : chaîne de caractère dans lequel le mot sera stocké
- n : taille de dest, pour eviter des debordements

3.3 Structures utilisées

Le scheima général de l'architecture structurelle est le suivant.



Dans ce programme, on a besoin de représenter des mots. Pour ce faire, On définit un mot comme étant une chaîne de caractères accompagnée d'une liste de positions des phrases dans lequel il apparaît.
On utilise donc la structure suivante :

```
1 typedef struct cellword{
      char *word;
3     Lispos l;
  }Cellword;
```

Une liste de position est définie comme étant une position entière accompagnée d'un pointeur vers la position suivante.

```
typedef struct cellpos{
    long position;
    struct cellpos* next;
}Cellpos,*Listpos;
```

On a donc la posibilité de représenter un mot accompagné de la liste de ses positions. Il nous faut maintenant, pour manipuler des listes de mots, une structure permettant de lier ces mots entre eux.
Cette structure doit contenir un mot du type que l'on vient de définir, et un pointeur vers le prochain mot.

```
typedef struct cell{
    Cellword *value;
    struct cell* next;
}Cell,*List;
```

Pour detecter la fin d'une phrase ou la fin d'un mot, deux macros ont été definies :

```
#define END_OF_WORD(c) ((c==' ' || c==',' || c=='\n' || c=='\t'
    || c==';' || c=='*' || c=='-' || c=='\''))
#define END_OF_PHRASE(c) ((c=='.' || c=='?' || c=='!'))
```

Il est donc **très simple** pour un programmeur d'ajouter ses caractères de fin de mots/phrases.

3.4 Hachage ouvert

Pour stocker nos mots, on utilise un hachage ouvert. On a donc besoin d'un tableau assez grand qui contiendra des listes de mots. Il est usuel aussi pour mieu répartir le hachage d'utiliser **un nombre premier** comme taille de ce tableau.
Pour insérer un mot dans le tableau on utilise une fonction de hachage qui, à partir d'une chaîne de caractère, calcule une empreinte permettant de situer le mot dans notre table. La liste dans laquelle doit être comtenue mot est donc :
hashTable[hash(mot)].
L'avantage d'utiliser une table de hachage est que l'accès à un élement se fait en temps constant, avec un petit temps d'amortissement (parcours de la liste de mots).

Les cases de notre table contiennent donc des listes de mots triés. L'insertion dans notre table correspond à une insertion dans une liste dont la taille, si le tableau est assez grand, est majorée par une constante. L'insertion dans notre table de hachage possède donc une complexité constante.

3.5 Gestion des listes

Les listes de mots doivent être triées. L'insertion se fait donc en respectant l'ordre lexicographique. Etant donné que l'on dispose d'une liste simplement chainée, l'insertion est donc complexe. En effet, il faut deux curseurs parcourant la liste en stockant la cellule courante ainsi que la cellule précedente. Voici le prototype de la fonction effectuant cette insertion :

```
void insert_lexico_word(List *w, char* word, long offset);
```

Et pour éviter une complexité importante, cette fonction a le bon goût de rechercher le mot à ajouter pour eviter l'insertion de doublon et de trouver la bonne place où mettre le mot, et ce en un seul passage dans la liste.

En ce qui concerne la liste des position, on effectue des insertions en tête de liste. En effet, vu qu'il n'y a pas de contraintes sur l'ordre des positions on n'a pas besoin de reparcourir toute la liste.

3.6 Création de la liste triée

Après hachage complet du texte, on possède une multitude de listes triées dans le tableau de hash. Pour obtenir une seule liste triée, on doit alors fusionner toutes ces listes.

On cree alors une liste parallèle, on alloue **que** de quoi stocker un mot (Cellword) et un pointeur vers les mot suivant.

ATTENTION : On ne doit pas creer une nouvelle liste complète, mais seulement une liste parallèle pointant sur les mots contenus dans la table. En effet cela reviendrait relativement cher en place et en temps d'allouer

une nouvelle liste contenant des données déjà existantes. De plus, **il ne faut pas non plus fusionner directement les liste existantes entres elles**. En effet, cela effectuera un remaniements des liens entre les cellules et nous donnera bien une liste triée, mais aura aussi pour effet de corrompre notre table de hachage, puisque des mots possédant une clé X pourront être présent dans une liste de mots possédant une clé Y, ce qui est totalement à l'encontre du principe de hachage :

Soit f une fonction de hachage et E ensemble des mots du test :
$$\forall (x, y) \in E^2, (x \neq y \Rightarrow f(x) \neq f(y))$$
Une telle fonction doit être injective et donc ne pas retourner une même clé pour deux mots différents.

On a donc une liste pointant sur le champs *list->value* des mots existant. Enfin, pour creer notre liste, on parcours notre table et fusionne toutes les listes avec celle créee en parallèle.
Cette fusion correspond dans ce programme aux fonctions, dont voici les prototypes :

```
1 void merge_list(List *w1, const List w2);
  List create_sorted_list(List hash[]);
```

La fonction *create_sorted_list* parcours la table de hachage et fusionne en appellant *merge_list* toutes les listes de la table, puis renvoie cette liste triée.

3.7 Contrôle qualité

Une fois le programme fonctionnel, des tests ont été effectués, afin de vérifier et corriger des éventuels *bugs*.
Nous avons dans un premier temps utilisé le programme valgrind qui est un outil de debugging et de profiling.

```
valgrind ./Index -vp camion Colomba.txt
...
==4255== HEAP SUMMARY:
==4255==     in use at exit: 0 bytes in 0 blocks
==4255==   total heap usage: 10,043,999 allocs, 10,043,999 frees, 168,883,240 byte
...
==4255== ERROR SUMMARY: 0 errors from 0 contexts
```

On constate que a la fin de l'exécussion, la totalité de la mémoire utilisée est bien libérée, et que l'on a aucune erreur.

Dans un deuxième temps nous avons testé les *entrées* en utilisant le fichier **/dev/random**.

```
yoann@hp_laptop# ./Index -p tt /dev/random
^C
ctrl-c caught, exiting...
```

Apres un certain temps, on entre au clavier *ctrl-c* afin d'arréter le processus. On constate ici que l'on a pas rencontré d'erreurs ni d'erreur mémoire *segmentation fault*.