Rapport

Jules HABLOT
Rémi GATTAZ

Table des matières

Présentation du sujet

Nous souhaitons faire un lexique de tous les mots présents dans un texte, et pour les retrouver facilement nous ajoutons à chaque mot le numéro de la ligne où il est présent et après combien de caractères le mot commence sur cette ligne. Etant donné que nous ne connaissons pas la taille du texte avant de le lire en entier alors nous devons créer une structure dynamique qui aura exactement la taille nécessaire permettant de contenir tout le lexique.

Choix de conception

Structure des données

Nous pouvons modéliser la structure de donnée nécessaire à notre problème de **gestion de mémoire** par un embriquement de ***listes chaînées***. Nous avons donc choisi de gérer un dictionnaire comme un pointeur vers la permière case chainée. Il est important d'avoir un **pointeur vers ce dictionnaire** car lors d'un ajout en tête il serai trop couteux de devoir déplacer tout le contenu d'un dictionnaire pour créer une place libre en début liste chainée. Ainsi avec un pointeur nous pouvons nous abstraire de ce problème facilement.
Ensuite nous devons spécifier ce que chaque case contient. La case est coupée en deux, la deuxième partie servant à trouver la case suivante par un pointeur vers cette case. La première doit cond contenir un mot avec toutes les informations.

Un **mot** est composé d'une **suite de lettres** mais aussi des **emplacements** où il est présent dans le texte.
Commençons par modéliser les positions. Nous ne savons pas si le mot va être présent plusieurs fois dans le texte nous avons donc besoin ici aussi d'une liste chaînée qui grandit en fonction de la répétion du mot dans le texte. Nous avons contruit cette structure de listes d'emplacement sur le même principe que le dictionnaire, avec la deuxième partie de la case qui pointe vers la suivante et la première qui contient un emplacement (**couple d'entiers** ligne, colonne).
Il nous reste à voir ce qu'est une suite de lettres. Le sujet nous oblige à jouer avec des entiers de différentes tailles pour mettre nos lettres dedans. Ceci est basé sur le fait que dans notre alphabet nous n'avons que 26 lettres et donc que nous pouvons les représenter sur **seulement 5 bits** alors que normalement un caractère simple (*char*) est **sur 8 bits**. Nous devons donc gérer une liste d'entiers de stockage à taille variable (*uint8/16/32/64_t*). Dans un **entier de stockage**, les lettres sont stockées sur les bits de poids faibles mais la premier lettre est dans le bits de poids forts de bits utilisés pour stocker. Le schéma du sujet reprend très bien cette description. Ne sachant pas la taille des mots, nous devons là encore faire une **liste chaînée d'entiers de stockage**.

Pour résumer, nous avons une liste chaînée (dictionnaire) de pointeurs vers des listes chaînées (vers des mot). Dans un mot nous avons choisi de mettre un pointeur vers la tête de la liste de ses lettres et vers la tête de la liste de ses positions mais aussi pour les deux listes vers la queue des listes (le dernier élément) pour faciliter l'ajout d'un élément.

Notre structure de donnée est décrite dans le fichier *types.h*.

Fonctions

Nos fonctions doivent **être économes en mémoire** car le texte peut être très gros. Notre structure de donnée ne doit pas être passé en argument, et donc copier à chaque appel de fonctions. Nous avons donc choisi de donner le plus souvent possible des **pointeurs vers la structure de donnée**.
Dès que nous avons commencé à coder, nous avons pu voir que certaines portions de code se trouvaient à plusieurs endroits, nous avons donc choisi d'en faire des fonctions auxiliaires, de même que certaines variables que nous avons rendu globale dans les fichiers où elle était utilisée.
De plus, nous avons essayé de nous **abstraire** le plus possible de **la taille des entiers de stockage**, ce qui rend notre application viable pour toutes les tailles de stockage possibles.
Enfin nous avons choisi de **ne pas faire d'appel récursif** dans nos fonctions par soucis de **taille de pile**, celle-ci pouvant se remplir rapidement si nous sommes en présence d'un grand texte avec beaucoup de fois les mêmes mots. En effet, trop de listes imbriquées avec une fonction récursive pour chaque liste pourrait saturer la pile.
Finalement, nous avons essayer de **nommer nos fonctions le plus judicieusement possible**, ainsi que les variables, qui sont d'ailleurs réduites au strict minimum.

Choix de programmation

Taille des maillons

Afin de pouvoir changer facilement la taille des maillons plus tard, nous avons décidé de définir dans le fichier *maillon.c* une **constante**. Appelée *NB_LETTRES_MAILLON*, cette constante contient le nombre de lettres qu'il est possible de mettre dans un maillon. Cette constante est calculée en divisant le nombre de bit par la taille d'une lettre (5 bits).
Grâce à cette constante, il a suffit de changer le typedef *maillon_t* pour changer la taille du maillon et que cela soit appliqué dans tout le programme.

Compare

Contrairement à ce qui était proposé, nous n'avons pas définie une fonction compare_mots qui prend en paramètres deux mots. A la place, nous avons défini une fonction qui prend en paramètre un mot et une chaine de caractère.

Nous aurions pu faire ce choix mais cela nous aurais forcé à exposer la fonction ou à effectuer des copies des listes d'emplacement. Nous avons donc décidé de ne pas opter pour cette solution.

Autres

Nous avons choisi de coder notre application en C car le C est un langage proche de la mémoire et de machine au sens matérielle. Nous avons hésité à faire des fonctions en assembleur pour gérer les entiers de stockage, mais nous ne voulions pas rentrer dans un niveau d'abstraction inférieur. Le C nous permet de manipuler les bits facilement.

Nous gérons tous les **choix d'affichages** et de **gestion de fichiers** dans notre *main*, car l'application doit s'arrêter proprement si elle rencontre un problème au cours de son exécution.

Nous avons essayé de ne faire que des fonctions essentielles, nous les utilisons toutes. Les **fonctions** compiquées, que nous avons mis du temps à dévélopper, sont **commentées** en détails pour une meilleure (re)lecture.

Nous avons, dans notre gestion de structure de donnée, pensé à nettoyer la mémoire à la fin de l'exécution de notre programme. Pour cela nous avons **libéré** (free) la mémoire utilisée.

Organisation logicielle

Notre application est décomposée en 4 fichiers :
- maillon.c : fonctions pour abstraction du maillon
- mot.c : fonctions pour abstraction de la structure **mot**
- dictionnaire.c : fonctions abstraction du dictionnaire
- main.c : programme principal

Dans cette liste, **chaque fichier dépend du fichier du dessus**. C'est *maillon.c* qui est donc le niveau le plus bas de notre abstraction. Nous voulons respecter cette hiérarchie pour que notre application soit la plus sûre possible. Les fonctionnalités essentiels entre les différents niveaux sont dans les *.h* mais les autres sont privées et restent au sein du fichier.

C'est uniquement notre programme principale qui dépent de la librarie *tokenize*.

Organisation logistique

Etant donné que nous étions tous les deux familliers avec l'utilisation de git, nous avons décidé pour ce projet de mettre en place un dépot git. Ceci nous as permis de partager et mettre en commun nos sources très facilement.

Comment compiler et utiliser notre application

L'application dico dépend de la librairie tokenize. Cette librairie est présente dans le dossier *lib/* et le **makefile** a été écrit afin que la compilation fonctionne sur les plateformes Linux et Mac OS X. Pour pouvoir éxecuter le programme, il est cependant nécessaire d'ajouter le chemin vers le dossier lib correspondant dans la variable d'environnement LD_LIBRARY_PATH pour linux et DYLD_LIBRARY_PATH pour OSX. Sourcer le fichier **setenv.sh** à la racine permet d'effectuer cette opération pour les *Linux* **64 bits** et *Mac OS X*. Pour les **32 bits**, il faudra modifier le ficher sourcé pour mettre la bonne librairie.

La compilation et l'exécution de notre projet se fait donc de la façon suivante :

```
$ source ../setenv.sh
$ cd src
$ make
$ ./dico [fichier]
```

Le paramètre *[fichier]* est optionnel.

Le programme permet de récupérer le texte entré au clavier par un utilisateur et de produire et afficher le dictionnaire contenant tous les mots de ce texte. Si le nom d'un fichier est fourni et que celui-ci existe, alors c'est le dictionnaire du texte dans ce fichier qui sera produit.

Dans le cas où la constante *DEBUG* est défini pendant la compilation, en plus de l'affichage classique, le dictionnaire sera affiché sous forme de liste de mots. Cela permet de **visualiser** facilement les **structures de données** utilisées pour stocker les mots dans le dictionnaires.

Exemples d'éxecution

Avec des maillons de *32 bits* sans avoir défini la constante *DEBUG* :

```
$ ./dico ../Examples/hugo.txt

Dictionnaire :
    aller (2,6)
    branches (2,42)
    ce (2,20)
    dit (2,29)
    ecouter (2,12)
    faner (1,24)
    je (1,1)
    laisserai (1,7)
    les (1,30) (2,38)
    ne (1,4)
    on (2,26)
    pas (1,17)
    pervenches (1,34)
    qu (2,23)
    sans (2,1)
    se (1,21)
    sous (2,33)
```

Avec des maillons de *16 bits* en ayant défini la constante *DEBUG* :

```
$ ./dico ../Examples/hugo.txt

Dictionnaire :
    aller (2,6)
    branches (2,42)
    ce (2,20)
    dit (2,29)
    ecouter (2,12)
    faner (1,24)
    je (1,1)
    laisserai (1,7)
    les (1,30) (2,38)
    ne (1,4)
    on (2,26)
    pas (1,17)
    pervenches (1,34)
    qu (2,23)
    sans (2,1)
    se (1,21)
    sous (2,33)
```

```
Maillons :
|   [ {all} -> {er }
|     |
|     |--> {2, 6}   ]
|
|   [ {bra} -> {nch} -> {es }
|     |
|     |--> {2, 42}  ]
|
|   [ {ce }
|     |
|     |--> {2, 20}  ]
|
|   [ {dit}
|     |
|     |--> {2, 29}  ]
|
|   [ {eco} -> {ute} -> {r  }
|     |
|     |--> {2, 12}  ]
|
|   [ {fan} -> {er }
|     |
|     |--> {1, 24}  ]
|
|   [ {je }
|     |
|     |--> {1, 1}   ]
|
|   [ {lai} -> {sse} -> {rai}
|     |
|     |--> {1, 7}   ]
|
|   [ {les}
|     |
|     |--> {1, 30} ->  {2, 38}  ]
|
|   [ {ne }
|     |
|     |--> {1, 4}   ]
|
|   [ {on }
|     |
|     |--> {2, 26}  ]
|
|   [ {pas}
|     |
|     |--> {1, 17}  ]
|
|   [ {per} -> {ven} -> {che} -> {s  }
|     |
```

```
|     |--> {1, 34}  ]
|
|   [ {qu }
|     |
|     |--> {2, 23}  ]
|
|   [ {san} -> {s  }
|     |
|     |--> {2, 1}   ]
|
|   [ {se }
|     |
|     |--> {1, 21}  ]
|
|   [ {sou} -> {s  }
|     |
|     |--> {2, 33}  ]
|
```

Limites / Extentions

Améliorations faites

Notre manière d'implémenter les fonctions et procédures de gestions de listes de stockage pour les lettres, nous permet de **faire varier rapidement la taille des entiers de stockage**, lors de la compilation. Pour cela, il faut définir pendant la compilation la constante *INT8*, *INT16* ou *INT64*, modification à apporter dans les flags de notre **Makefile**. Si aucune de ces constantes n'est définie, ce sont des entiers de 32 bits qui seront utilisés.

Nous pouvons gérer des **lettres majuscules** simplement, mais elles seront stockées comme des lettres miniscules, car il s'agit du même mot, qu'il soit en début de phrase ou pas.

Nous avons essayé de **représenter** le mieux possible, dans le ternimal, les **listes chaînées** constituant notre structure de donnée. Ainsi nous avons ajouté un mode pour voir l'intérieur des listes, **détaillant chaque case**. Ceci passe par l'affichage des maillons, avec les cases vides, et des listes de positions. Cette amélioration est très utile pour voir les différences entre les tailles d'entiers de stockage, car on distingue bien la taille de chaque maillon. Pour voir cette affichage, il faut définir pendant la compilation la constante *DEBUG*.

Limites

Nous n'avons pas géré les **accents** sur les lettres. Ceci aurait pourtant été relativement facile en rajoutant quelques règles dans la fonction *char_to_num*. Puisque ce n'est pas géré, dans le cas ou un mot contient un accent, le mot sera dans le dictionnaire considéré comme terminé à l'emplacement de l'accent dans le mot. Ce comportement se vérifie pour n'importe quel caractère qui n'est pas un lettre en majuscule ou minuscule.