TME2 : Langage C: tables de hachage

1. Objectif

Objectif

Ce TME se décompose en deux parties: La première partie porte sur l'analyse d'un programme C existant. Dans la seconde partie vous devrez modifier ce programme pour introduire de nouvelles fonctionnalités. L'objectif est triple :

1. Il doit vous permettre de compléter l'auto-évaluation de vos connaissances des outils de développement C que vous avez commencée dans le précédent TME. Si vous ne savez pas répondre directement aux questions posées, vous devez trouver les réponses dans les documentations (man, web), ou auprès de vos camarades.
2. Il introduit de nouveaux outils permettant l'indentation automatique d'un programme source (outil *indent*), ou l'écriture d'une documentation (outil *man*).
3. Il présente une structure de données, la table de hachage, très utilisée dans tous les programmes où on a besoin de rechercher un objet par son nom.

Il vous offre également un modèle de programme, avec Makefile et man pour vos futurs développements.

Vous devez commencer par créer un répertoire *tme2* et y copier tous les fichiers se trouvant dans :

```
/users/enseig/encadr/cao/tme2
}}}}æ
```

Ce répertoire contient tous les fichiers sources permettant de générer un programme qui utilise une table de hachage pour compter le nombre total de mots d'un fichier texte quelconque, ainsi que le nombre d'occurrences de chaque mot.

```
* {{{Makefile .....}}} description du processus de construction de l'exécuta
* {{{main.c, main.h .....}}} programme principal source et déclarations.
* {{{count.c, count.h .....}}} analyseur lexical et fonctions de comptage et d'affic
* {{{dico.c, dico.h.....}}} fonctions de gestion de la table de hachage.
* {{{hash.c, hash.h.....}}} fonction générale de calcul de l'index à partir de la
* {{{man1/tool.1 .....}}} fichier au format man
```

= Principe des tables de hachage =

Une table de hachage est une structure de données permettant de représenter des ensembles d'éléments, où chaque élément est un couple de la forme (clé, donnée). La clé est le plus souvent une chaîne de caractères. La donnée peut être une valeur ou un pointeur sur une structure plus ou moins complexe. Le principal objectif d'une table de hachage est d'accélérer la recherche d'un élément par sa clé.

Pour représenter un ensemble de couples (clé, data) la méthode la plus simple consiste à les stocker dans une unique liste chaînée. La recherche d'un élément se fait alors par un parcours de la liste, ce qui n'est pas très efficace si le nombre d'éléments est grand.

Pour accélérer la recherche, on crée un tableau de listes chaînées. On définit ensuite une fonction, que l'on nomme '''fonction de hachage''', qui calcule un index à partir de la clé. Tout élément doit être rangé dans la liste chaînée associée à la case du tableau définie par l'index calculé par la fonction de hachage.

```
[[Image(Diapositive1.jpg, nolink)]]
```

Dans la pratique, il n'est pas possible d'éviter les collisions: deux éléments ayant des clés différentes peuvent avoir le même index de hachage, et seront donc stockés dans la même liste chaînée.

Pour que la méthode soit efficace, il faut cependant que les éléments se répartissent aussi uniformément que possible dans les différentes cases du tableau, et qu'il n'y ait qu'un petit nombre d'éléments dans chaque liste chaînée. Le nombre de cases du tableau doit donc être du même ordre de grandeur que le nombre total d'éléments à stocker.

Il existe plusieurs méthodes de calcul de l'index. La fonction `hashindex()` contenue dans le fichier `hash.c` implémente celle proposée par Donald Knuth.

La propriété principale d'une table de hachage est que, si la taille de la table est du même ordre de grandeur que le nombre d'éléments, alors le temps de recherche est en $O(1)$, c'est à dire indépendant du nombre d'éléments, même pour un million d'éléments, comme s'il s'agissait d'un simple tableau.

= Etape 1 : Questions sur le code fourni =

== A) Le Makefile ==

Les premières questions portent sur le fichier [attachment:Makefile Makefile]

Completez la liste des dépendances pour les cibles : `{{{statt, main.o, ...}}}`, puis re-écrivez

- `{{{$@}}` : désigne le fichier cible d'une règle. [[BR]]
- `{{{$<}}` : désigne le premier fichier de la liste des fichiers source d'une règle. [[BR]]
- `{{{$^}}` : désigne la liste des fichiers source d'une règle. [[BR]]

- * '''QA1''' Pourquoi est-il préférable de regrouper la définition des commandes et paramètres a
- * '''QA2''' A quoi servent les options `-p`, `-g`, `-Wall`, `-Werror`, `-ansi` ? (man gcc)
- * '''QA3''' Comment demander l'optimisation maximale du compilateur ? (man gcc)
- * '''QA4''' L'option `-p` est présente dans `LDLFLAGS` et `CFLAGS`, pourquoi n'est-ce pas le cas de `-g`
- * '''QA5''' Que fait la règle `indent` ? quelle est la signification des flags utilisés par le pr

== B) Le programme main ==

Les questions suivantes portent sur le programme principal [attachment:main.c main.c]

Ce fichier contient la fonction `main()` et la fonction `getarg()` qui effectue l'analyse de la ligne

- * '''QB1''' Expliquez à quoi sert chacun des fichiers inclus au début du fichier `'main.c'`
- * '''QB2''' A quoi sert le fichier `'main.h'` ? A quoi servent les 2 premières lignes et la de
- * '''QB3''' Expliquez le fonctionnement de la fonction `getopt()` (`{{{man 3 getopt}}}`) [[BR]]
Vous ajouterez plus tard dans la fonction `getarg()` l'option `-s` qui demande de fournir des st
- * '''QB4''' A quoi sert l'appel `'return'` a la fin de la fonction `main()` ?
- * '''QB5''' A quoi sert l'appel système `'exit'` à la fin de la fonction `usage()` ?
- * '''QB6''' Quels sont les appels système utilisés dans ce fichier `main.c` ? Quelle précaution d
- * '''QB7''' Où sont définies les fonctions standards de la bibliothèque C ?
- * '''QB8''' Qu'est-ce qu'un filtre unix ? Que faut-il faire pour transformer ce programme en fi

== C) Le dictionnaire ==

Le fichier [attachment:dico.c dico.c] rassemble les fonctions d'accès à une table de hachage uti

- * '''QC1''' Quel est l'encombrement (en nombre d'octets) des structures créées en mémoire par l
- * '''QC2''' Pourquoi la structure `{{{dico_item_t}}}` définie dans le fichier [attachment:dico.h
- * '''QC3''' Quelle est la seule fonction capable d'introduire un nouvel item dans le dictionnai
- * '''QC4''' Dans la fonction `dico_get()` comment fait-on pour tester le retour de l'appel systèm
- * '''QC5''' A quoi sert la structure `{{{dico_iterator_t}}}` définie dans le fichier [attachment:
- * '''QC6''' La fonction `hashindex()` peut-elle provoquer une erreur de segmentation ? Comment y

== D) La fonction de comptage des mots ==

Le fichier [attachment:count.c count.c] contient le code de la fonction `count()`, qui compte le m et la fonction `result_count()` qui affiche les résultats.

- * '''QD1''' Le mot clé `'static'` est utilisé de deux manières différentes dans le fichier `'co`
- * '''QD2''' Dans la fonction `token()`, pourquoi ne peut-on pas utiliser l'appel système `malloc()`
- * '''QD3''' La fonction `token()` doit renvoyer un nouveau `"token"` (mot) du fichier texte analysé

- * '''QD4''' Pourquoi as-t-on mis une étoile devant l'argument 'numero' de la fonction token()
- * '''QD5''' La fonction result_count() utilise-t-elle des fonctions d'accès spécifiques pour eff

= Etape 2 : Modifications du programme =

== A) affichage des numéros de ligne ==

Le programme qui vous est fourni affiche, pour chaque mot présent dans le fichier texte analysé, le nombre d'occurrences de ce mot. Vous devez modifier le programme de façon à ce qu'il indique en plus, pour chaque mot, les numéros de toutes les lignes où le mot est présent.

Le fichier Makefile est un fichier texte. Si on lance le programme {{{statt}}} sur le fichier M

```

{{{
$ ./statt Makefile
      dico.o : 2 occurrences
      des : 3 occurrences
      : : 2 occurrences
      clean : 3 occurrences
/dev/null : 2 occurrences
      2> : 2 occurrences
      count.o : 2 occurrences
      # : 3 occurrences
}}}

```

Après modification, il devra afficher:

```

{{{
$ ./statt Makefile

      dico.o : 2 occurrences lignes: 23  22
      des : 3 occurrences lignes: 14   8   2
      : : 2 occurrences lignes: 22  19
      clean : 3 occurrences lignes: 37  34
/dev/null : 2 occurrences lignes: 37  34
      2> : 2 occurrences lignes: 37  34
      count.o : 2 occurrences lignes: 20  19
      # : 3 occurrences lignes: 14   8   2
}}}

```

Pour introduire cette nouvelle fonctionnalité, il faut:

- * Définir un type de liste chaînée pour le stockage des numéros de ligne. Cette structure contiendra
- * Changer la structure dico_item_s afin d'ajouter un champ contenant un pointeur sur la liste chaînée
- * Modifier la fonction count() pour ajouter un nouveau numéro de ligne dans la liste chaînée
- * Modifier la fonction result_count() pour parcourir les listes chaînées contenant les numéros de ligne

== B) statistiques sur la table de hachage ==

Vous donnerez également des statistiques sur l'usage de la table de hachage.

Vous fabriquerez pour cela une fonction {{{void dico_info(dico_root_t *)}}} dans le fichier {{{dico.c}}}

```

{{{
Informations sur la table de routage
Taux de remplissage      : 79.21 %
Longueur moyenne des listes : 1.40
Longueur maximale des listes : 4
}}}

```

== C) création d'un manuel en ligne ==

Vous devez enfin écrire la page de manuel pour le programme 'statt'.

Le fichier [attachment:tool.1 tool.1] se trouve dans le répertoire man1, et contient une page de manuel générique contenant des commentaires pour expliquer la syntaxe.

La structure de cette page est standard, c'est celle utilisée pour les commandes unix.

Renommez le fichier 'tool.1' en 'statt.1', et modifiez son contenu pour documenter le programme 'statt'.

Pour rendre ce manuel utilisable en ligne, ajoutez le répertoire {{{.}}} à la variable d'environnement

```

{{{
export MANPATH=./$MANPATH
}}}

```

}}}

Il suffit de taper la commande `{{{ man statt }}}}` dans le répertoire `tme2` pour afficher la docu

= Compte-Rendu =

Pour la première partie de ce TME, vous rédigerez un compte-rendu contenant les réponse aux questions posées, en respectant la numérotation.

Pour la seconde partie, une démonstration des modifications introduites dans le programme vous sera demandée au début du prochain TME.