TME2 : Langage C: tables de hachage

Objectif

Ce second TME porte plutôt sur l'analyse d'un programme C existant, que vous devrez modifier. L'objectif est triple :

1. Il doit d'une part vous permettre de complêtez l'auto-évaluation de vos connaissances des outils de developpement C que vous avez commencée dans le précédent TME, en vous posant des questions auxquelles vous devriez savoir répondre. Si ce n'est pas le cas, 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*), la constructtion d'une bibliothèque C (outil *ar*), 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 récupérer l'archive attachment:tme2.tar.gz ou créer un répertoire *tme2* et y copier tous les fichiers se trouvant dans :

```
/users/enseig/encadr/cao/tme2
```

Ce répertoire contient un programme utilisant une table de hachage. Le travail demandé comporte deux phases. Dans un premier temps vous devez analyser le code fourni, et **rédiger** des réponses aux questions portant sur ce code. Dans un deuxième temps, vous devrez modifier ce programme, pour introduire de nouvelles fonctionnalités.

Le programme fourni compte le nombre de mots d'un fichier texte et indique le nombre total de mots dans le fichier, ainsi que le nombre de mots différents, et le nombre d'occurences de chaque mot.

Description des sources et principe des tables de hachage

- `Makefile ..... description du processus de construction de l'exécutable.`
- `main.c, main.h ..... programme principal source et déclarations.`
- `count.c, count.h ..... algorithme de parcours d'un fichier texte en vue de comptage.`
- `hte.c, hte.h ..... fonction de création des tables de hachage et déclaration`

de toutes fonctions de gestion.
- `dico.c, dejavu.c, namealloc.c ..` fonctions de gestion des tables pour trois types d'usage
- `man1/tool.1 ...................` fichier au format man

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, data). Le plus souvent la clé est une chaîne de caractères. La donnée peut être un nombre ou une structure de données 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 liste chainée. 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 créé un tableau de listes chainées. Le nombre de cases de ce tableau doit être de l'ordre de grandeur du nombre maximal d'éléments que l'on souhaite gérer. Si on veut gérer 1000 éléments, il faut un tableau d'environ 1000 cases. On définit ensuite une fonction, que l'on nomme **fonction de hachage**, qui calcule l'index d'une case à partir de la clé. Tout élément doit être rangé dans la liste chainée associée à la case du tableau définie par l'index calculé par la fonction de hachage.

Pour que cette méthode soit efficace, il faut que les éléments se répartissent aussi uniformément que possible dans les différentes cases de la table. Il existe plusieurs méthodes de calcul de l'index, nous vous donnons celle proposé par Donald Knuth. Dans la pratique, il n'est pas possible d'éviter les collisions, et deux éléments ayant des clés différentes peuvent avoir le même index de hachage.

En résumé, pour ranger 1000 éléments, on fabrique un tableau de 1000 listes chainées avec l'espoir que la plupart des listes ne contiendront qu'un seul élément et qu'il n'y aura qu'un tout petit nombre de listes contenant plus d'un élément.

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. Les deux principales actions sont l'ajout d'un élément (add) et la recherche d'un élément (get).

- La fonction get() prend en paramètre la clé de l'élément recherché. Si l'élément existe, elle rend la donnée associée.
- La fonction add() prend en paramètre le couple (clé, data). Si l'élément existe, elle change sa donnée, sinon elle créé l'élément.

Etape 1 : Questions sur le code fourni

Le Makefile

Les premières questions portent sur le fichier attachment:"Makefile"

1. Completez la liste des dépendances pour les cibles : `main.o ... namealloc.o`.
2. Réécrivez les commandes en utilisant les variables automatiques : `$@ $< $^`
    - `$@` : désigne le fichier cible d'une règle.
    - `$<` : désigne le premier fichier de la liste des fichiers source d'une règle.
    - `$^` : désigne la liste des fichiers source d'une règle.
3. Pourquoi est-il préférable de regrouper la definition des commandes et paramètres au début du Makefile?
4. A quoi servent les options -p, -g, -wall, -werror, -ansi ? (man gcc)
5. Comment demander l'optimisation maximale du compilateur ? (man gcc)

6. L'option -p est présente dans LDFLAGS et CFLAGS, pourquoi n'est-ce pas le cas de -g ? (man gcc)
7. Que fait la regle indent ? quelle est la signification des flags utilisés par le programme indent ? (man indent)

Le programme main

Les questions suivantes portent sur le programme principal attachment:main.c. Ce fichier contient la fonction main() et la fonction getarg() qui effectue l'analyse de la ligne de commande.

- Expliquez à quoi sert chacun des fichiers inclus au début du fichier *main.c*
- A quoi sert le fichier "main.h" ?
- Expliquez le fonctionnement de la fonction getopt (`man 3 getopt`)

  Ajoutez dans la fonction getarg() le traitement de l'option -h, qui affiche un texte rappelant le format de la ligne de commande.
  Vous ajouterez plus tard l'option -s qui demande de fournir des statistiques concernant l'utilisation de la tables de hachage.

- A quoi sert l'appel de return a la fin de la fonction main() ?
- Pourquoi y-a-t-il exit() à la fin de la fonction usage() ?
- Quels sont les appels systeme utilisés dans ce fichier main.c ?
- Quelle precaution doit on prendre lors de leur utilisation ?
- Ou sont definies les fonctions standard de la bibliothèque C ?
- Qu'est-ce qu'un filtre unix ?
- Que faut-il faire pour transformer ce programme en filtre ?

et sur le fichier de prototype associé attachment:main.h

- A quoi servent les lignes les 2 premières lignes et la dernière ?
- Pourquoi inclure `stdio.h` ici ?

Les fonctions de bases de la tables de hachage

Questions sur les déclaration du fichier attachment:hte.h

- Les types `hte_item_t` et `hte_data_t` sont des structures dont le contenu n'est pas défini ici.

  Leur contenu n'est pas défini non plus dans le fichier `hte.c`, en revanche il est défini dans les fichiers `dico.c`, `dejavu.c` et `namealloc.c` Quelle est alors la contrainte d'usage de ces types dans le fichier `hte.c` ? Quel est l'intérêt de cette écriture ?

- Dans la définition des prototypes de fonctions, le nom des paramètre est-il nécessaire ? si non pourquoi les mettre ?

et sur les fichiers de créations des tables attachment:hte.c

- Faîte un dessin représentatif de la valeur de la variable locale `root` de la fonction `hte_create` si `nb_list==10` juste avant l'appel return.
- La fonction `hte_hash` peut-elle provoquer une erreur de segment ? Comment y remedier proprement ?
- Dans la fonction `hte_create` comment fait-on pour tester le retour de `malloc` ? à quoi celà sert-il ?

Le service *dictionnaire*

Le fichier attachment:dico.c rassemble les fonctions d'accès à une table de hachage utilisé comme dictionnaire.

- Le type `hte_data_t` n'est pas défini ici. Est-ce grave ? Où devra-t-il être défini ?
- Pourquoi teste-on key dans la fonction `hte_get` ?
- Pourquoi définit on la structure `hte_item_s` ici ?
- Dans la structure `hte_item_s` le champ KEY est un tableau de taille indéfinie.

  Quel différence y aurait-il avec une définition du type `char *KEY` ?

- A quoi sert la fonction `perror` ?

L'utilisation du dictionnaire pour compter des occurences de mots

Le code se trouve dans le fichier attachment:count.c

- Le mot clé **static** est utilisé de trois manières différentes. Quel est son effet ?
- Aurait on pu écrire `static char * buffer = malloc(1024)` ou encore `char * buffer = malloc(1024);`. Pourquoi ?
- La fonction `token` est censé rendre un nouveau token (mot) du fichier `infile` à chaque appel. Que pensez-vous de son comportement, est-il satisfaisant d'une manière générale?.
- Pourquoi as-ton mis une étoile devant l'argument `numero` de la fonction `token`.
- La fonction result_count utilise des fonctions d'accès spécifiques pour effectuer le parcours des items présents dans la table.

  Pourquoi ne peut on pas faire un parcours directement? Dans quel ordre vont être affichés les items ?

Les autres services possibles

Vous trouverez deux autres services possibles utilisant des tables de hachage dans les fichiers, attachment:dejavu.c et attachment:namealloc.c. Ces services ne sont pas utilisés dans ce TME2, mais ils vous seront utiles lors des TME futurs. Vous pourrez faire alors une édition de liens avec la bibliothèque libhte.a.

La fonction dejavu() permet de répondre à la question "ai-je déjà vu cette chaine de caractères". La fonction namealloc() est très utilisée dans la chaîne de CAO Alliance, et permet de garantir que si deux chaines de caractères sont identiques alors elles seront rangées à la même adresse.

Etape 2 : Evolution du programme

Vous devrez modifier le programme de façon à ce qu'il indique, pour chaque mot, les numéros de toutes les lignes où le mot est présent. Ainsi le programme fourni appliqué sur le fichier Makefile (c'est un fichier texte) produit:

```
prompt$ ./statt Makefile
         namealloc.o : 2 occurences
               hte.o : 2 occurences
                true : 2 occurences
              dico.o : 2 occurences
                 des : 3 occurences
                   : : 2 occurences
```

```
               clean : 3 occurences
            /dev/null : 2 occurences
                  2> : 2 occurences
             count.o : 2 occurences
                   # : 3 occurences
          Definition : 3 occurences
            dejavu.o : 2 occurences
                  -p : 2 occurences
                  || : 2 occurences
              indent : 2 occurences
                   = : 9 occurences
               $(RM) : 2 occurences
               statt : 4 occurences
              main.o : 2 occurences
                  de : 2 occurences
            libhte.a : 3 occurences
116 mots dont 80 différents
```

Après modification, il devra afficher:

```
prompt$ ./statt Makefile
         namealloc.o : 2 occurences lignes: 23  22
               hte.o : 2 occurences lignes: 23  22
                true : 2 occurences lignes: 37  34
              dico.o : 2 occurences lignes: 23  22
                 des : 3 occurences lignes: 14   8   2
                   : : 2 occurences lignes: 22  19
               clean : 3 occurences lignes: 37  34
            /dev/null : 2 occurences lignes: 37  34
                  2> : 2 occurences lignes: 37  34
             count.o : 2 occurences lignes: 20  19
                   # : 3 occurences lignes: 14   8   2
          Definition : 3 occurences lignes: 14   8   2
            dejavu.o : 2 occurences lignes: 23  22
                  -p : 2 occurences lignes: 10   9
                  || : 2 occurences lignes: 37  34
              indent : 2 occurences lignes: 17   8   2
                   = : 9 occurences lignes: 15  12  11  10   9   6   5   4   3
               $(RM) : 2 occurences lignes: 37  34
               statt : 4 occurences lignes: 37  34
              main.o : 2 occurences lignes: 20  19
                  de : 2 occurences lignes: 14  14
            libhte.a : 3 occurences lignes: 23  22
116 mots dont 80 différents
```

Pour répondre à cette question, il va vous falloir:

- Définir un type de liste chainée pour le stockage des numéros de ligne.

  Ce type contient deux champs: un champ vers l'élément suivant et un entier représentant le numéro de ligne.

- Changer la structure hte_data_s afin d'ajouter un pointeur sur une liste chainée pour le stockage des numéros de lignes.
- Créer un nouvel élément de numéro de ligne lors du parcours du fichier (fonction count)
- Parcourir les listes créées pour les afficher (fonction result_count)

Vous donnerez également des statistiques sur l'usage des tables de hachage, telles que le nombre moyen de comparaisons nécessaires lors de la recherche d'un mot.

Compte-Rendu

Vous rédigerez un compte-rendu avec la réponse aux questions posées et une explication des évolutions démandées. Pour ce tme, comme pour les suivants, vous serez interrogez **individuellement** devant machine.

Nous vous demandons également de renommer le fichier attachment:tool.1 (il se trouve dans le répertoire man1) en statt.1. Ce fichier 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. Vous devez écrire la page de manuel du programme statt. Le but de cette opération est de vous montrer que l'écriture d'un man ne pose pas de problème de forme. Pour utiliser ce man ajouter, le répertoire `.` à la variable d'environnement `MANPATH`:

```
export MANPATH=.:$MANPATH
```

Ainsi, vous pouvez taper la commande `man statt` dans le répertoire tme2 pour voir votre manuel.