TD PG106
Mise au propre d'un code source

Important : Pour ce TD, vous **devez** utiliser un éditeur de texte basique (vi, vim ou emacs). L'utilisation de Atom, vscode ... ou chatgpt copilot ... est **interdite**. Merci de suivre cette consigne pédagogique !

▸**Exercice 1.** *clone :*

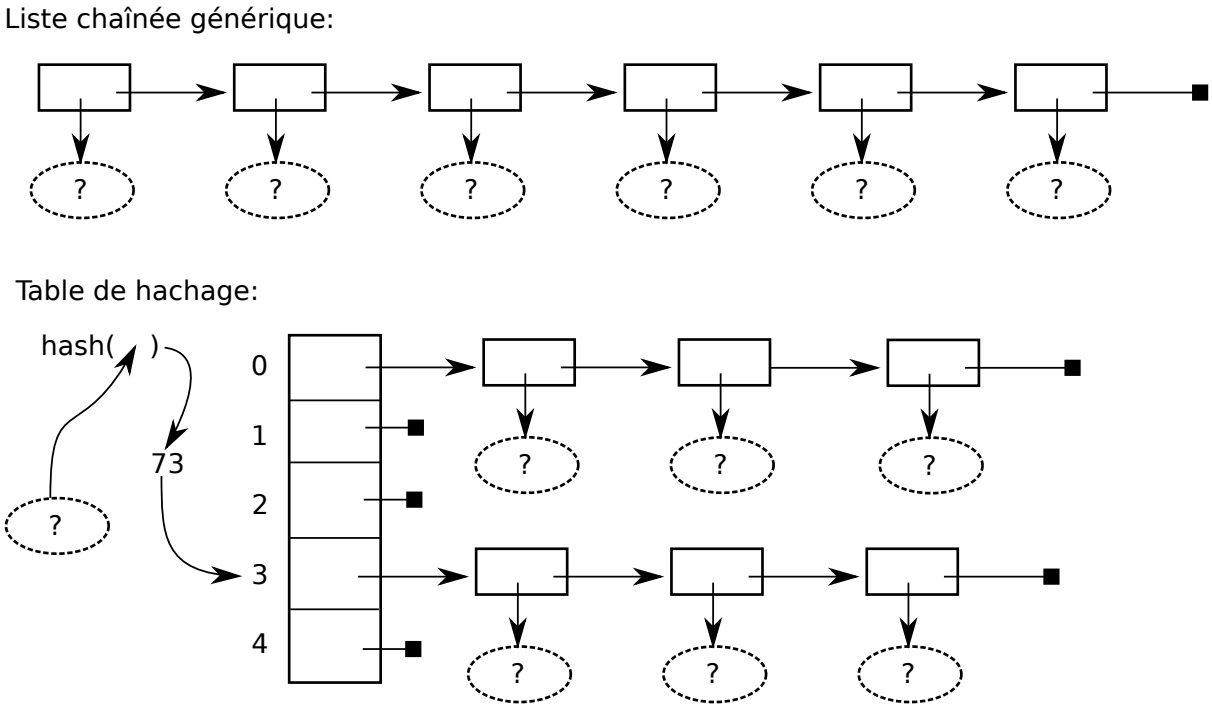
Rendez-vous sur `thor` *dans la page associé au module PG106/Votre groupe. Cloner le dépôt* `git` *associé à votre login (il y a un dépôt par élève). Dans ce dépôt, vous trouverez le fichier* `hash.c` *sur lequel nous allons travailler par la suite.*

1 Introduction

▸**Exercice 2.** *hash.c*

A la racine vous disposez du fichier `hash.c`.

Ce source est une implémentation des tables de hachage. Le principe est de stocker des éléments dans la table de manière efficace de façon à ce que la recherche d'éléments se fasse rapidement.



Pour cela, une table de hachage de type Separate Chaining est composée de N listes chainées. On ne connait pas la nature exacte des éléments de la table mais on dispose d'une fonction qui renvoie un entier associé à un élément (deux éléments différents peuvent être associés au même entier). Cet entier détermine la liste dans laquelle l'élément doit être inséré. On dispose également d'une fonction qui permet la comparaison de deux éléments. Ainsi, les listes chainées sont ordonnées selon l'ordre croissant des éléments.

En parcourant rapidement le fichier source `hash.c` *:*

— *Quelle structure encode la liste chainée ?*
— *Quelle est le type de liste chainée ?*
— *À quoi correspondent les paramètres de* `hash_init` *?*
— *Quelle est la nature de* `dbg` *?*

2 Reformatage : Convention de code

Une "convention de codage et nommage" (coding style et naming convention) est un document qui décrit un certain nombre de régles quand à l'écriture de programmes. Ce document fixe par exemple la langue (français/anglais), le nommage des structures (majuscule, minuscule, utilisation de l'underscore), des macros, des fonctions, des variables locales et globales. Egalement, on y trouve des régles générales : type de retour de fonction, gestion des erreurs, position des accolades ...

▸**Exercice 3.** `clang-format` *:*

À quoi sert le programme `clang-format` *? Quels sont les conventions supportées nativement par cet outil ?*

Parmis ces conventions, donner au moins 2 différences entre elles. (nous reviendrons par le suite sur cet outil).

Tester `clang-format` *avec différent styles appliqués au fichier* `hash.c` *et observer les différences.*

Expliquer comment faire en sorte que la tabulation corresponde à 4 espaces et que les lignes ne fassent pas plus de 60 caractères.

Créer un fichier `.clang-format` *basé sur la norme* `WebKit` *avec* `TabWidth` *à 4,* `ColumnLimit` *à 80 et* `PointerAlignment` *à "Right".*

Appliquer ce style à `hash.c`.

▸**Exercice 4.** *Mise en place d'un répertoire de projet :*

Par la suite, on appelera racine de votre projet le répertoire racine de votre dépôt `git`*. Dans ce répertoire créer un fichier REAME.md.*

Le fichier REAME.md sera rédigé au format Markdown (chercher sur internet). Dans ce fichier écrire une section

- *Authors avec votre nom et adresse email*
- *About vide pour l'instant*
- *Coding style dans laquelle indiquer la norme de nommage et codage. Vous préciserez également que le projet (code et documentation) est en anglais.*
- *Compiling vide pour l'instant*

3 Remaniement de source

Nous allons maintenant, étape par étape, mettre au propre ce code source.

▸**Exercice 5.** *Mise en forme de code syntaxique.*

Utiliser clang-format *pour la mise en forme syntaxique du programme (ce qui a déjà été fait à l'exercice 2 normalement).*

▸**Exercice 6.** *Homogénéisation du nommage des fonctions.*

Commencer par renommer les fonctions en respectant votre convention de codage (voir WebKit).

Ajouter des commentaires devant chaque fonction avec une description en une ligne de ce que fait la fonction.

Après chaque renommage,vérifier que le code compile toujours.

▸**Exercice 7.** *Découpage.*

Le source contient :

- *des macro de débuggage assez génériques*
- *une implémentation de liste chaînée*
- *une implémentation de table de hachage*
- *un programme d'exemple avec des structures.*

Etape par étape :

— *créer un fichier* `debug.h` *dans lequel vous placerez les macro concernées*
— *vérifier que ça compile toujours (*`gcc -I. hash.c` *doit produire un exécutable)*
— *déplacer les listes chaînées dans* `liste.h` *et* `liste.c`*. Vous metterez les prototypes de fonctions dans le header ainsi que la déclaration de la structure.*
— *vérifier que ça compile toujours (*`gcc -I. liste.c hash.c` *doit produire un exécutable)*
— *déplacer la table de hachage dans* `hash.h` *et* `hash.c` *: concernant la macro* `HASH_SIZE`*, est-ce que celle-ci va dans le fichier .h ou .c ?*
— *et laisser le reste dans un fichier* `prog.c`
— *vérifier que ça compile toujours (*`gcc -I. prog.c liste.c hash.c` *doit produire un exécutable)*

4 Pré-compilation

▸**Exercice 8.** *Les asserts :*
Ajouter dans le programme `prog.c` *un appel à la fonction d'ajout d'un élément dans la table de hachage avec* `NULL` *comme premier paramètre.*
— *Compiler (en une seule commande à partir des fichiers sources .c) et exécuter : que se passe-t-il ?*
— *Compiler à nouveau en ajoutant l'option* `-DNDEBUG` *au compilateur. Executer : que ce passe-t-il ?*
— *Consulter la documentation de la fonction assert.*
En utilisant l'option `-E`*, observer le résultat de la pré-compilation du fichier* `hash.c` *avec et sans l'option* `-DNDEBUG`*.*

▸**Exercice 9.** *Macros :*
Faire en sorte que la macro `dbg` *n'affiche rien si la macro* `NDEBUG` *est définie.*
Faire en sorte que la macro `trace` *n'affiche rien si la macro* `NTRACE` *est définie.*
Faire en sorte que l'on puisse redéfinir la valeur `HASH_SIZE` *via les options de compilation lors de la compilation du fichier* `hash.c`

5 Compilation

Créer un fichier `compile.sh`, ce fichier sera un script shell dans lequel vous metterez les commandes de compilations au fur et à mesure des exercices suivants (une commande par ligne). Le début du script doit être comme ci-dessous

```
#!/bin/bash
set -xe
echo "Compilation:"
```

Positionner les droits en exécution sur ce script (`chmod`) et vérifier qu'il fonctionne.

▸**Exercice 10.** *La compilation en objets :*
Mettre dans le fichier `compile.sh` *les commandes qui permettent de compiler les sources en objet puis de liés ces objets en binaire.*
Afin de pouvoir facilement activer/désactiver les options de compilation, ajouter deux variables : `CFLAGS` *qui sera utilisée lors des phases de pré-compilation et compilation et* `LDFLAGS` *qui sera utilisée lors de l'édition de lien.*

▸**Exercice 11.** *Analyse :*
A l'aide des commandes `objdump` *et* `nm`*, lister le contenu des fichiers objets et du binaire produits.*

▸**Exercice 12.** *Bibliothèque statique :*

Compiler les modules list et hash sous la forme d'une bibliothèque statique libhash.a.
Ajouter la commande permettant de générer le programme `program_static` *à partir de cette bibliothèque statique.*
À l'aide de la commande `ldd`*, afficher les dépendences en bibliothèque des exécutables produits jusqu'à présent.*

▸**Exercice 13.** *Bibliothèque dynamique :*

Compiler list.c et hash.c en des objets valides pour la production d'une bibliothèque dynamique.
Regrouper ces objets au sein de la bibliothèque `libhash.so`.
Ajouter la commande permettant de générer le programme `program_dynamic` *à partir de cette bibliothèque. À l'aide de la commande* `ldd`*, afficher les dépendences en bibliothèque de ce nouvel exécutable.*

▸**Exercice 14.** *Modification :*
Modifier la fonction d'initialisation des tables de hachage pour qu'elle affiche une message sur la sortie standard.
Recompiler **uniquement** *les fichiers objets et les deux bibliothèque puis, sans les recompiler, tester les executables précédents.*
Supprimer les deux bibliothèques et, sans les recompiler, tester les executables précédents.

6 Bonus

Les exercices ci-dessous sont **optionnels** et permettent de s'entrainer à la programmation.

▸**Exercice 15.** *rehash :*
Ecrire une fonction `hashResize(struct HashTable *,int newSize)` *qui ré-organise la table de hachage pour que sa table interne soit de taille* `newSize`.

▸**Exercice 16.** *Age :*
Ajouter un champs `age` *à la structure* `Person` *et écrivez une nouvelle fonction de comparaison basée sur l'age.*
Tester cela en créant une nouvelle table de hachage contenant des personnes et triée selon l'age.

▸**Exercice 17.** *Ordered Action :*
Du fait de la répartition dans les listes de la table de hachage, la fonction `hashAction` *ne suit pas l'ordre croissant total des éléments (elle le fait selon l'ordre croissant liste par liste).*
Ecrire une fonction `hashOrdredAction` *qui appelle la fonction* `action` *passée en argument en suivant un ordre croissant total.*

▸**Exercice 18.** *Merge : Ajouter la fonction* `listMerge` *au module* `list` *qui fusionne deux listes chaînées en concervant l'ordre des éléments.*