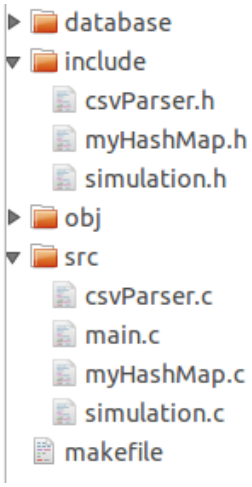
LP25 : Projet
HashMap

Structure des répertoires et fichiers

Par simple convenance personnelle, j'ai décidé de modifier la structure des répertoires et fichiers (par rapport à celle fournie dans le squelette).

Ma nouvelle structure se présente donc comme cela :

- Un répertoire « **database** » qui contient le ou les fichiers CSV de la base de données.
- Un répertoire « **include** » qui contient les .h.
- Un répertoire « **obj** » qui contient les objets générés par GCC en attendant la compilation du binaire complet.
- Un répertoire « **src** » qui contient les sources .c du programme.
- Et le **Makefile** qui se trouve à la racine.



Par ailleurs, j'ai décidé de dissocier la bibliothèque qui s'occupera de traiter le fichier CSV (**csvParser.c**).

Par rapport à la structure donnée, j'ai remonté à la racine le répertoire « **obj** » qui se trouvait dans « **src** », et j'ai renommé le **setup.c** en **main.c**.

Makefile

J'ai réalisé un Makefile qui permet de compiler le projet de manière intelligente : ce qui n'a pas besoin d'être recompilé n'est pas recompilé. La difficulté principale a été de pouvoir gérer le fait que les objets .o se trouvent dans un répertoire différent de celui des sources .c, et des includes .h.

Lorsque l'on fait « `make` », le projet se compile de façon à pouvoir plus facilement être débugué : cela inclut l'option –g à GCC (afin de pouvoir utiliser GDB), mais aussi la constante « DEBUG » qui permet l'affichage des messages d'état qui sont placés un peu partout dans le code afin de savoir ce qu'il se passe lors de l'exécution du programme.

La version « finale » du binaire du projet (c'est-à-dire sans l'option –g et sans les messages de débogage) peut être générée grâce à la commande « `make release` ». Il est préférable d'utiliser le binaire issu de « `make release` » si l'on veut traiter une base de données ayant un grand nombre d'enregistrements car l'affichage de tous les messages dans le terminal prend beaucoup plus de temps.

La commande « `make clean` » permet de supprimer les objets (dans le répertoire « **obj** » donc). Et la commande « `make mrproper` » permet de supprimer les objets et le binaire.

De plus, la commande « `make check-leak` » permet de voir s'il y a ou non des fuites mémoire dans le projet (normalement, il n'y en a pas), et cela grâce à une simple utilisation de l'outil Valgrind.

Par ailleurs, afin de mener des tests davantage poussés, j'ai réalisé un script PHP (c'est le langage script que je maitrise le mieux) qui permet de générer un fichier CSV d'une taille demandée. D'où la commande « `make generate-db` ».

Code

main.c

Ce code, très simple, contient simplement la fonction *main()* qui appelle la fonction *runSimulation()* présente dans le fichier **simulation.c** détaillé ci-dessous.

simulation.c

Ce code aussi très simple contient la fonction *runSimulation().* Cette fonction va créer une HashMap, et la remplir à l'aide de la fonction *parseCSV().* Elle va également afficher le contenu de la HashMap, ses statistiques, effectuer quelques tests, etc. Pour finir, elle va libérer la HashMap, afin de ne pas provoquer notamment de fuites mémoire.

csvParser.c

Cette bibliothèque s'occupe essentiellement du traitement du fichier CSV. Elle possède aussi une fonction : `void parseCSV(char *fileName, HashMap *map)` qui prend donc en premier paramètre le chemin du fichier CSV, et en second paramètre le pointeur de la HashMap dans laquelle on souhaite travailler.
Cette fonction lit le fichier ligne à ligne (avec *fgets()*) et découpe la ligne selon les virgules (grâce à *strtok()*), puis appelle la fonction *hashmapPut()* détaillée plus tard.

myHashMap.c

C'est le cœur du sujet, la HashMap, les opérations que l'on effectue dessus, etc.
Par convenance personnelle, j'ai préféré écrire mon propre code et m'inspirer du squelette fourni plutôt que de le compléter.
Je vais donc commenter, fonction par fonction le rôle de chacune et ce qu'elles font et renvoient. J'ai essayé de les organiser par catégorie.

HashMap (create, expand, free, …)

- `HashMap* hashmapCreate(size_t initialCapacity, int (*hash)(char* key))`
  Permet de créer une HashMap. Elle alloue un espace en mémoire pour la HashMap de la taille de *initialCapacity* buckets passé en paramètre. Elle prend aussi en paramètre un pointeur de fonction *hash* sur la fonction de hachage. Elle retourne le pointeur sur la HashMap créée.

- `void hashmapFree(HashMap* map)`
  Permet de libérer la HashMap. Elle prend en paramètre le pointeur de la HashMap. Elle ne retourne rien.
  Cette fonction va en fait parcourir la HashMap et provoquer un *entryFree(entry)* (développé par la suite) sur chaque entrée. (Une entrée « Entry » est l'équivalent d'un KVContainer dans le sujet). Elle va aussi libérer la mémoire de la structure de la HashMap.

- `size_t calculateIndex(size_t bucketCount, int hash)`
  Cette fonction sert à calculer l'index en fonction du hashage. Elle prend en paramètre le nombre de buckets dans la HashMap, ainsi que le hash généré (qui est un entier). Elle va renvoyer le modulo de *hash* par *bucketCount*, c'est-à-dire l'index.

- `void hashmapExpandTest(HashMap* map)`
  Cette fonction est appelée à chaque ajout d'un élément dans la HashMap. Elle va permettre de vérifier s'il est nécessaire d'agrandir la HashMap. Si c'est nécessaire, elle va l'agrandir. Sinon, elle ne fait rien. Elle prend en paramètre le pointeur de la HashMap. Elle ne retourne rien.
  Dans le cas où il est nécessaire d'agrandir la HashMap (lorsqu'on dépasse le facteur de charge « LOAD_FACTOR »), elle va créer une nouvelle HashMap avec une taille supérieure calculée par la fonction *getNextSize()*. Ensuite, elle va déplacer, (en recalculant l'index) les entrées de l'ancienne HashMap vers la nouvelle. Enfin, elle va libérer la mémoire de la première HashMap.

- `int getNextSize(int number)`
  Cette fonction donnée va calculer la nouvelle taille de la HashMap en fonction de l'ancienne taille donnée en paramètre, grâce à la fonction *naivePrime()* détaillée ci-dessous. Cette fonction renvoie donc un entier.

- `int naivePrime(int number)`
  Cette fonction donnée qui prend en paramètre un nombre entier va renvoyer un nombre entier qui correspond au nombre premier suivant. Les nombres premiers sont préférables pour avoir moins de collisions par la suite, en effet, il ne faut pas oublier que l'on effectuera un modulo pour obtenir l'index…

Records (put, get, remove)

- `Entry* createEntry(char* key, int hash, char* IMEI, char *recordTime, char *providerId, char *techno, double centroidX, double centroidY)`
  Cette fonction va permettre de créer une entrée. Elle est surtout utilisée par la fonction *hashmapPut()* détaillée ci-dessous. Elle prend en paramètre la clé *key*, le *hash* correspondant, l'*IMEI*, le *recordTime*, le *providerId*, la *techno*, le *centroidX*, le *centroidY*. Elle renvoie le pointeur de l'entrée créée.

Il s'agit simplement de faire quelques *malloc()* pour réserver l'espace et les renseigner avec les données fournies en paramètre.

- `void hashmapPut(HashMap* map, char* IMEI, char *recordTime, char *providerId, char *techno, double centroidX, double centroidY)`
  Cette fonction va permette d'ajouter un enregistrement à la HashMap. Elle prend en paramètre le pointeur de la HashMap, l'*IMEI*, le *recordTime*, le *providerId*, la *techno*, le *centroidX*, le *centroidY*. Elle ne renvoie rien.
  Cette fonction va appeler la fonction *createEntry()* pour récupérer le pointeur de la nouvelle entrée, et si à l'index calculé ne se trouve aucune autre entrée, elle y place directement le pointeur. Par contre, s'il existe déjà une ou des entrées, elle va parcourir la liste pour placer la nouvelle entrée en dernier (en changeant le paramètre « *next* » du précédent).

- `Entry* hashmapGet(HashMap* map, char* key)`
  Cette fonction permet de récupérer un enregistrement dans la HashMap. Elle prend en paramètre le pointeur de la HashMap, et la clé dont on souhaite récupérer l'enregistrement. Elle retourne donc l'enregistrement de type *Entry* si l'enregistrement correspond à la clé est trouvé, sinon *NULL*.
  Elle se place directement à l'index calculé et parcourt l'éventuelle liste chainée afin de trouver la clé correspondante.

- `void entryFree(Entry* entry)`
  Cette fonction permet de libérer la mémoire correspondante à une entrée. Elle prend en paramètre l'entrée dont on souhaite libérer la mémoire. Elle ne renvoie rien.
  Il s'agit ici de faire des appels successifs de *free()*.

- `bool hashmapRemove(HashMap* map, char* key)`
  Cette fonction permet de supprimer un élément dans la HashMap. Elle prend en paramètre le pointeur de la HashMap, et la clé dont on souhaite supprimer l'enregistrement correspondant. Elle renvoie *true* si elle a supprimé un enregistrement, *false* sinon.
  On peut comparer cette fonction à la fonction *hashmapGet()*, mais avec un appel sur *entryFree()* pour libérer la mémoire, avec en plus le « raccord » de l'éventuelle liste chainée. (Si on supprime un élément, il faut mettre le pointeur du suivant dans le *next* du précédent).

- `Entry* hashmapPop(HashMap* map, char* key)`
  Cette fonction permet de récupérer et de supprimer un enregistrement. Elle prend en paramètre le pointeur de la HashMap, et la clé dont on souhaite supprimer et récupérer l'enregistrement correspondant. Elle retourne donc l'entrée correspondante si elle a été trouvée, *NULL* sinon.

Il s'agit quasiment du même code que la fonction *hashmapRemove()*, à la différence que l'on ne le supprime pas mais qu'on le retire de l'éventuelle liste chainée, et qu'elle retourne le pointeur de l'enregistrement retiré. Par contre, lorsque l'on utilise cette fonction, il va falloir stocker le pointeur retourné pour ensuite pouvoir effectuer un *entryFree()* dessus afin de libérer la mémoire qu'elle occupe si l'on ne veut pas provoquer de fuites mémoire.

Simple operations

- `int hashmapCountTechno(HashMap* map, char* techno)`
  Cette fonction permet de compter combien il y a d'enregistrements dans la HashMap (donnée en premier paramètre) qui portent la technologie *techno* donnée en second paramètre. Elle renvoi un entier : le nombre trouvé.
  Il s'agit de parcourir l'intégralité de la HashMap et d'incrémenter un compteur lorsque l'on rencontre un enregistrement où la technologie est la même que celle fournie en second paramètre.

Miscellaneous (comparison, hashing)

- `bool equalKeys(char* keyA, int hashA, char* keyB, int hashB)`
  Cette fonction permet de comparer deux enregistrements afin de déterminer s'ils sont ou non égaux. Elle prend en paramètre deux clés et deux hash. Elle renvoi un booléen.
  Si les deux hash sont différents, ce n'est forcément pas la même clé, on renvoie donc *false*. Sinon si les deux clés sont les mêmes, on a affaire au même enregistrement (donc on renvoie *true, false* sinon).

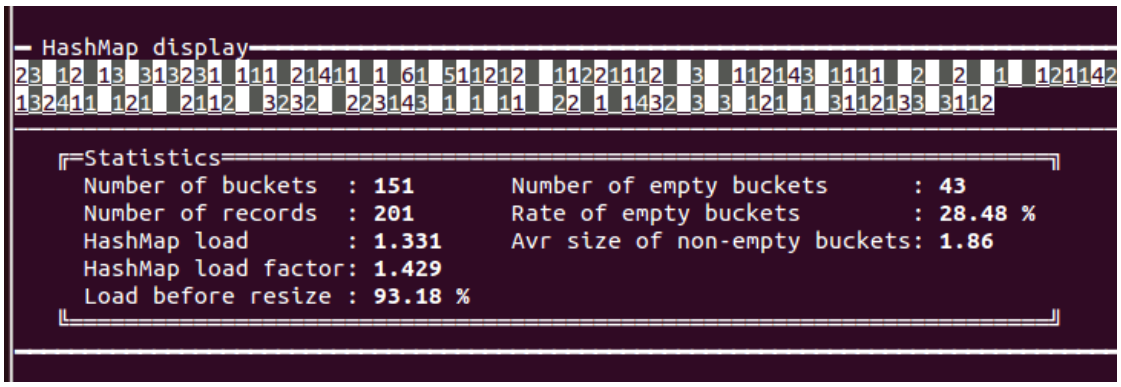
- `int hashmapIntHash(char* key)`
  Cette fonction n'est pas à négliger puisque c'est la fonction de hachage. Elle prend en paramètre une chaine de caractères, la clé *key*, dont on souhaite calculer son hash correspondant. Elle retourne donc un entier, le hash.
  Au début, pour tester, j'avais pris comme hash la somme des codes ASCII des caractères présents dans la chaine. Cette solution donnait des résultats plutôt mauvais. Je me suis alors mis à la recherche d'une fonction qui donnait de meilleurs résultats. J'ai aussi essayé de combiner ces fonctions. J'ai donc sélectionné la fonction de hachage de Paul Hsieh qui donne de bons résultats. J'ai pris soin de commenter les autres fonctions testées et non-utilisées (la somme des codes ASCII, celle de Robert Jenkins', et Knuth).

Display functions (display HashMap: graph & statistics, entry)

- `void entryDisplay(Entry* entry, int index)`
  Cette fonction permet d'afficher une entrée à l'écran, avec son contenu. Elle prend en paramètre le pointeur de l'entrée, et l'index. L'index donné, s'il est supérieur ou égal à 0, permet juste d'être affiché (cela sert notamment pour la fonction *hashmapDisplay()* ci-dessous). Si l'on souhaite juste afficher une entrée sans l'index (car on ne le connaît pas forcément), on peut mettre l'index à -1 pour désactiver la fonctionnalité. Par exemple, elle affiche l'entrée sous la forme :
  - Si on connaît l'index (≥ 0):
    ```
    [147]: key=00020a7a63ac276de7218725d365598e,11:21:41,
    hash=563744906 ==> IMEI=00020a7a63ac276de7218725d365598e,
    time=11:21:41, provId=31788, tech=DCS 1800, cX=575825.0,
    cY=2422675.0
    ```
  - Si l'on ne connaît pas l'index (< 0):
    ```
    [00020a7a63ac276de7218725d365598e,11:21:41]: hash=563744906 ==>
    IMEI=00020a7a63ac276de7218725d365598e, time=11:21:41,
    provId=31788, tech=DCS 1800, cX=575825.0, cY=2422675.0
    ```

- `void hashmapDisplay(HashMap* map)`
  Cette fonction permet d'afficher la HashMap donnée en paramètre. Elle écrit directement sur la sortie standard et ne renvoie rien.
  Elle parcourt l'intégralité de la HashMap et appelle à chaque entrée la fonction *entryDisplay()* détaillée ci-dessus.

- `void hashmapDisplayGraph(HashMap* map)`
  Cette fonction permet d'afficher de manière graphique le contenu de la HashMap, et délivre aussi quelques statistiques sur celle-ci.
  Par exemple :



Chaque bloc blanc ou gris représente un bucket, le nombre à l'intérieur donne le nombre d'enregistrements dans celui-ci. S'il n'y a pas de nombre, c'est que le bucket est vide.

Tests de performances

Après avoir effectué de multiples tests, notamment en faisant varier le nombre d'enregistrements ainsi que la constante « LOAD_FACTOR », je suis arrivé au résultat (par dichotomie) que le facteur de charge le plus conciliant pour cette application était de 1/0,45 :

Pour une base de données contenant 250 000 enregistrements.

- Avec un facteur de charge « par défaut » de 1/0,7 ≈ 1,427 :

```
Statistics
  Number of buckets   : 175211    Number of empty buckets      : 42243
  Number of records   : 250000    Rate of empty buckets        : 24.11 %
  HashMap load        : 1.427     Avr size of non-empty buckets: 1.88
  HashMap load factor: 1.429
  Load before resize : 99.88 %
```

  En se fixant pour objectif de baisser le taux de buckets vides (actuellement de 24,11 %) et d'augmenter le nombre d'enregistrements par bucket non-vide (actuellement à 1,88) pour atteindre environ 2,5.

- Avec un facteur de charge de 1/0,5 = 2,0 :

```
Statistics
  Number of buckets   : 125029    Number of empty buckets      : 16835
  Number of records   : 250000    Rate of empty buckets        : 13.46 %
  HashMap load        : 2.000     Avr size of non-empty buckets: 2.31
  HashMap load factor: 2.000
  Load before resize : 99.98 %
```

  Nous obtenons une moyenne de 2,31 enregistrements par bucket non-vide, avec seulement 13,46 % de buckets vides, soit environ 11 points de moins que précédemment.

- Avec un facteur de charge de 1/0,4 = 2,5 :

```
Statistics
  Number of buckets   : 100019    Number of empty buckets      : 8234
  Number of records   : 250000    Rate of empty buckets        : 8.23 %
  HashMap load        : 2.500     Avr size of non-empty buckets: 2.72
  HashMap load factor: 2.500
  Load before resize : 99.98 %
```

  Le taux de buckets vides baisse encore, et nous atteignons 2,72 en moyenne du nombre d'enregistrements par bucket non-vide.

- Avec un facteur de charge de 1/0,45 ≈ 2,222 :

```
Statistics
  Number of buckets   : 112741    Number of empty buckets      : 12379
  Number of records   : 250000    Rate of empty buckets        : 10.98 %
  HashMap load        : 2.217     Avr size of non-empty buckets: 2.49
  HashMap load factor: 2.222
  Load before resize : 99.79 %
```

Ici, nous arrivons à une moyenne du nombre d'enregistrements par bucket non-vide de 2,49 et notre taux de buckets vides est presque de 11 %. Ce ratio de 1/0,45 semble donc être convenable.