TD PG106
tests et couverture

Etat des sources

À ce stade des tds vous devez avoir dans *master* les fichiers suivants :

```
nom_depot/.gitignore   # contient *.o *~ etc...
nom_depot/.clang-format # voir premier TD
nom_depot/README.md
nom_depot/Makefile
nom_depot/prog.c
nom_depot/hash/hash.c
nom_depot/hash/hash.h
nom_depot/hash/list.c
nom_depot/hash/list.h
nom_depot/hash/debug.h
bom_depot/build/.gitignore # contient *
```

Vérifier que vous êtes bien dans cette situation. Normalement, la commande `make` produit des fichiers intermédiaires dans un répertoire `build` (les `.o .so .a`) et trois binaires à la racine du dépôt : `prog prog_dynamic prog_static`.

1 Tests

L'objectif va être d'introduire une cible `test` afin que la commande `make test` exécute l'ensemble des tests (qui ne sont pas encore écrits).

▸**Exercice 1.** *Création d'une branche*
Créer une branche pour le travail sur les tests, vous finirez avec un merge *sur* master *une fois l'environnement de tests opérationnel.*

▸**Exercice 2.** *Environnement de test.*
Ajouter un répertoire appelé `tests`*. Dans ce répertoire, créer un premier fichier* `list_init_tests.c` *qui comportera le code :*

```
#include <assert.h>

void test_list_init1(){
  assert(0 && "my first test!");
}

int main(){
  test_list_init1();
}
```

Ajouter les règles nécessaire dans le `Makefile` *pour qu'un binaire* `build/list_init_tests` *soit produit et exécuter lors de l'appel à* `make test` *(cela implique l'écriture de plusieurs règles, n'oubliez pas le* `.PHONY`*).*

Faire en sorte que vos tests puissent *être lancés avec* `valgrind` *: pour cela vous passerez par une variable makefile (comme* `CFLAGS`*)*

▸**Exercice 3.** *Tests unitaires pour les listes*
Ecrire des tests unitaires pour chacunes des fonctions de la bibliothèque : init, add et action : vous

ferez un binaire pour chaque groupe de tests. Comme vous avez accès aux attributs de la structure, vous pouvez tester leurs valeurs.

Pour les fonctions avec passage de pointeur de fonction (`cellFreeSingle` *et* `cellFreeList`*), cela est un peu plus "technique" : une solution est d'écrire un fonction comme celle-ci :*

```
static int call=0;

void fakeRelease(void *d){
  call+=1;
}
```

Cela nous permet de valider que la fonction a été appelée. On pourra utiliser une version comme celle-ci :

```
static int call=0;
static void *lastFakeReleaseCall=NULL;

void fakeRelease(void *d){
  lastFakeReleaseCall=d;
  call+=1;
}
```

Cela nous permettra de valider que la fonction est appelée et de connaître l'argument passé lors du dernier appel.

Note *: Il n'est pas facile de vérifier que le module list libère bien la mémoire de manière programmatique (via des* `assert` *par exemple), aussi il est tout à fait possible d'avoir un test qui alloue puis libère une liste et que la validation de ce test soit faite via* `valgrind` *lors de son exécution. D'où l'intérêt dans ce cas d'avoir une exécutable par test (si on regroupe tout les tests dans un unique exécutable, lors d'une erreur* `valgrind` *type fuite mémoire, cela demande plus de travail pour en trouver l'origine).*

Pour `cellAdd`*, vérifier bien plusieurs conditions : ajout dans une liste vide, ajout d'un élément déjà présent dans une liste à un seul élément, deux éléments et plus de deux éléments...*

Pour `cellListAction` *: nous pouvons avoir recours à la technique présentée précédemment pour compter le nombre d'appels à la fonction* `action`*.*

Vérifier que vos tests fonctionnent puis fusionner vos modifications dans master*.*

2 Couverture

▸**Exercice 4.** *Couverture*

Les tests, c'est bien mais il est important de connaître la couverture de ces tests : c'est à dire les lignes de code effectivement exécutées durant les tests. Pour cela, il suffit d'ajouter la méta-option `--coverage` *à gcc (pour la compilation et l'édition de lien).*

Ajouter cette option dans le Makefile (sur `CFLAGS` *et* `LDFLAGS`*), puis faire un* `make clean` *et* `make test`*. Regarder le contenu du répertoire de build : il contiendra des fichiers* `*.gcda` *et* `*.gcno`*. Pour analyser ces fichiers, on utilise le programme* `gcov` *: tester la commande* `gcov build/list.o`*. Ajouter l'appel à* `gcov` *après l'execution des tests : vous aurez ainsi un indicateur précieux sur l'exhaustivité de vos tests. Attention cependant : 100% de couverture ne veut pas dire que c'est* complet*, certains enchaînement de configuration ne sont peut-être pas testés. De plus, il est* raisonnable *d'avoir un minimum de 70%, certain cas comme les malloc qui retournent* `NULL` *sont difficiles à mettre en tests.*

3 Profiling

Nous allons faire une analyse des performances en temps des tables de hashage (profiling). Nous pourrions également faire une analyse en utilisation mémoire.

Pour pouvoir faire cette analyse, nous devons faire un grand nombre de fois l'appel aux fonctions pour que le temps passé dans celles-ci soit significatif.

▸**Exercice 5.** *Générateur*
Télécharger le fichier `generator.zip` *sur la page de PG106, vous y trouverez un module implémentant un générateur de chaînes de caractères.*

Écrivez un programme `hash_prof` *qui crée une table de hachage de chaîne de caractères et qui ajoute dans cette table des chaînes générée à l'aide du module ci-dessus. Comme dans le programme d'exemple fournit avec le module, votre programme prendra en paramètre le nombre de chaînes à ajouter ainsi que la longueur de chaque chaîne.*

Vérifier le bon fonctionnement de votre programme (avec `valgrind` *ainsi qu'avec* `-fsanitize=address`*).*

Quelles options de compilation permettent de désactiver les messages de traces et de debug ?

▸**Exercice 6.** *Profiling*
Compiler votre code avec l'option `-pg` *(ainsi que l'option* `-O3`*). À quoi sert cette option ? Exécuter votre programme d'abord avec un petit exemple, puis augmenter les valeurs pour avoir un temps d'exécution qui soit de l'ordre de 5 secondes (utiliser* `time`*). Un fichier est créé après l'exécution du programme : quel est le nom de ce fichier, que contient-il ?*

Analyser les performances avec `gprof` *: dans quelle fonction passe-t-on le plus de temps ? Quelles pourraient être les pistes pour améliorer les performances de la bibliothèque ?*

4 Table de hachage

▸**Exercice 7.** *Compléter vos tests*
Répéter l'écriture des tests pour les tables de hachages .... Bien entendu, vous ferez cela via une branche git !