Lab #2 - Tables de hachage.

Objectifs

- Définitions et appels de fonctions manipulant des structures
- Allocation dynamique de mémoire avec `malloc`/`free`
- Manipuler des pointeurs et des tableaux
- Créer de nouveaux types de données
- Programmation modulaire et compilation séparée (`make`)
- Connaître et savoir implémenter une table de hachage
- Savoir évaluer les performances d'une structure de données

Préliminaires

Vous réaliserez ce travail dans un dépôt Git local. Il vous est demandé de commiter **régulièrement** vos contributions et de *pousser* celles-ci sur la plateforme GitLab (https://gitlab.telecomnancy.univ-lorraine.fr) de l'école. Veillez à organiser votre dépôt de la manière suivante :

- un répertoire `src/` dans lequel vous placerez le code source de votre travail
- ne commiterez que le code source `.c` et non les versions compilées de vos programmes. Pour cela ajouter le nécessaire dans le fichier `.gitignore` pour ignorer les fichiers `.o` notamment.

Rendu

Ce travail est à rendre au plus tard le **lundi 16 mai 2022 à 13:00** sur la plateforme GitLab de l'école.

Pré-requis

Pour l'ensemble du module, nous utiliserons l'environnement de développement intégré Visual Studio Code (https://code.visualstudio.com/) avec les extensions (C/C++ Extension et Makefile Tools) et ainsi que le compilateur Clang (https://clang.llvm.org/). Tous ces outils ont déjà été installés et configurés lors des TPs du module de C.

Vous devez maintenant, si ce n'est pas déjà fait, cloner le dépôt du TP, vous placer dans le répertoire ainsi créer. Plus tard, vous devrez ouvrir ce répertoire dans Visual Studio Code (en utilisant la commande `code .` par exemple).

```
1  # en SSH
2  git clone git@gitlab.telecomnancy.univ-lorraine.fr:sd2k22/lab2-festor1.git
3  # ou en HTTPS (ici on insére l'adresse email dans l'URL lors du "clonage" afin de ne pas
       avoir à la re-saisir plus tard -- à chaque "push")
4  git clone https://olivier.festor%40telecomnancy.eu@gitlab.telecomnancy.univ-lorraine.fr/
       sd2k22/lab2-festor1.git
```

Ouvrez dans Visual Studio Code le répertoire de votre dépôt. Vous pouvez utiliser la commande suivante :

```
cd lab2-festor1 # si ce n'est déjà fait
code . # pour ouvrir (en tant que projet) le répertoire courant
```

Tests unitaires

Pour ceux d'entre vous qui souhaiteraient aller un peu plus loin que l'objectif de la séance, nous vous invitons à regarder la librairie de tests unitaires Snow afin d'automatiser vos tests. Cette librairie ne nécessite que d'inclure le fichier `snow/snow.h` et de suivre la syntaxe d'usage de la librairie.

Le site web présente un exemple complet (écriture des tests, assertions que vous pouvez utiliser, compilation et exécution).

C'est parti !

Exercices

L'objectif visé pour ces deux premières séances de TPs est de réaliser un outil basique de vérification orthographique. Dans cette optique, vous devrez concevoir et implémenter les structures de données nécessaires. Dans cette séance, vous vous intéresserez à l'implémentation d'une table de hachage et des fonctions attenantes. La réalisation du dernier élément – le vérificateur qui devra charger un fichier texte et vérifier si chaque mot le composant est présent, ou non, dans le dictionnaire – est laissée aux plus motivé(e)s d'entre vous.

Exercice 1 - Encore une liste

L'objectif de cet exercice est d'écrire (ou de modifier une implémentation que vous possédez) une liste permettant de conserver des éléments de type « élément » (`element_t *`). Dans cette question, cette structure `element_t` contiendra une référence vers deux chaînes de caractères (`char *`). Cette liste servira à implémenter les alvéoles de la table de hachage et ainsi gérer les collisions de hachage.

Question 1.

Écrivez l'implémentation d'une liste dans un fichier `list.c`. Les profils des fonctions attendues sont indiqués dans le fichier d'entête `list.h` fourni.

À titre de rappel, les fonctions attendues doivent permettre de :

- créer une liste : `list_t* list_create()`.
- détruire une liste : `void list_destroy(list_t* one_list)`.
- vérifier si la liste est vide : `bool list_is_empty(list_t* one_list)`.
- ajouter un élément en fin de liste : `void list_append(list_t* one_list, char* one_key, char* one_value)`.
- visualiser un élément en l'affichant sur la sortie standard : `void element_print(element_t* one_element )` selon le format suivant `key: value`.
- visualiser une liste en l'affichant sur la sortie standard : `void list_print(list_t* one_list)`.
- vérifier si une chaîne de caractère est présente dans la liste ou non : `bool list_contains(list_t* one_list, char* one_key)`.

— rechercher un élément dans la liste à partir de la chaîne de caractères `one_key` qu'il contient : `char* * list_find(list_t* one_list, char* one_key)`. La fonction retournera la valeur `NULL` si un tel élément n'est pas présent dans la liste.

Exercice 2 - La table de hachage

Question 2.

À partir du fichier d'entête fourni `table.h`, écrivez dans un fichier `table.c` les fonctions attendues permettant de réaliser l'implémentation d'une table de hachage.

Vous noterez que le code de la fonction de hachage à utiliser est fourni `int hash(char* some_value)`.

À titre de rappel, les fonctions attendues doivent permettre de :

— créer une table : `table_t* table_create(int size)`. Le paramètre `size` permet de spécifier le nombre d'alvéoles de la table.
— détruire la table : `void table_destroy(table_t* one_table)`.
— calculer en utilisant la fonction de hachage l'indice de l'alvéole de la table associée une valeur : `int table_indexof(table_t* one_table, char* one_key)`.
— ajouter une chaîne de caractères dans la table : `bool table_add(table_t* one_table, char* one_key , char* one_value)`. Cette fonction retourne la valeur `false` si une valeur est déjà associée à la valeur `one_key` (dans ce cas, la nouvelle valeur ne sera pas ajoutée).
— vérifier qu'une chaîne est déjà présente ou non dans table : `bool table_contains(table_t* one_table , char* one_key)` – il s'agit d'une comparaison par valeur des chaînes de caractères –.
— obtenir une valeur associée à une clé dans table : `char* table_get(table_t* one_table, char* one_key)`.

Question 3.

Ajouter à votre implémentation (dans des fichiers `dico.h` et `dico.c`) les fonctions permettant de créer une table contenant l'ensemble des mots présents dans un fichier texte. Vous pourrez utiliser le fichier de données fourni (`data/dico_fr_gutenberg.txt`).

Ainsi, il est attendu que vous rajoutiez a minima une fonction dont le profil est : - `table_t* dico_load(char* filename)`

Exercice 3 - Évaluer les performances

Question 4.

Vous évaluerez les performances de remplissage de votre table de hachage en fonction du nombre de sous-tables (alvéoles) choisi au départ. Vous pouvez étudier le temps d'exécution de la fonction de remplissage en fonction de la taille choisie.

Question 5.

Vous évaluerez les performances de la fonction de hachage fournie en exemple. Pour ce faire, vous étudierez la distribution des éléments dans la table de hachage. Pensez-vous que la fonction fournie soit bien choisie ? Pouvez-vous proposer une autre fonction de hachage permettant de corriger les défauts s'il y a lieu ?