P31 Développement efficace - épisode 6

1 - Dictionnaires et hachage

Un dictionnaire est une structure de données qui permet de stocker des couples `(clé, valeur)` et de retrouver la valeur associée à une clé donnée.

Une manière *très simple*, mais *très peu efficace*, de le mettre en oeuvre est d'utiliser un `vector` de couples `(clé, valeur)`. Retrouver la valeur associée à une clé donnée nécessite alors de parcourir les éléments du `vector` les uns après les autres et de comparer la clé recherchée avec les clés des couples stockés.

Une autre approche consiste à ranger ces couples `(clé, valeur)` dans un arbre binaire de recherche. Les éléments sont stockés dans l'arbre de sorte que pour chaque noeud, tous les noeuds de son sous-arbre de gauche possèdent des clés inférieures à la sienne et tous les noeuds de son sous-arbre de droite des clés supérieures (il faut donc une relation d'ordre sur les clés). Retrouver la valeur associée à une clé donnée se réalise alors en parcourant l'arbre à partir de la racine et en descendant à gauche ou à droite en fonction du résultat de la comparaison de la clé recherchée avec les clés des noeuds rencontrés. Cette approche est *bien plus efficace* que la précédente, mais nécessite de maintenir un arbre *équilibré* pour rester efficace.

Si les clés sont des entiers, et que ces entiers sont compris entre 0 et un maximum `m` donné, l'approche la plus efficace (et imbattable) consiste évidemment à utiliser un simple `vector` de taille `m` au sein duquel on stocke uniquement les valeurs. Les clés n'ont plus besoin d'être stockées explicitement et la valeur correspondant à la clé `i` se trouve simplement dans la case d'indice `i` du `vector`, qui est directement accessible.

Si on ne se sert que de quelques valeurs entières entre 0 et `m` en tant que clé, cette approche représente cependant un énorme gaspillage de mémoire : tout l'espace utilisé par les cases correspondant aux clés inutilisés est réservé pour rien. De plus si `m` est très grand, cette approche n'est pas envisageable.

Le **hachage** consiste à transformer une clé donnée en un entier compris entre 0 et `p`, `p` étant plus petit que la taille de l'espace des clés. On se contente alors d'un `vector` de taille `p` pour stocker les valeurs. Si `t` est la fonction de transformation de clé, alors la valeur associée à une clé `k` est stockée dans la case d'indice `t(k)`. Gain intéressant, la nature des clés n'a pas d'importance : elles peuvent être des entiers, des chaînes de caractères, des structures, ... du moment que l'on peut écrire une fonction `t` qui les transforme en un entier compris entre 0 et `p`.

Dans le cas où l'on connaît à l'avance le nombre d'éléments que l'on va stocker ainsi que leurs clés, on peut choisir `p` comme étant égal à ce nombre, et il est possible de construire une fonction *parfaite* et *minimale* qui transforme chacune de ces `p` clés en un entier unique dans l'intervalle `[0, p]`.

Évidemment, on ne connait pas toujours à l'avance le nombre d'éléments que l'on va stocker et leurs clés. Dans ce cas, on peut choisir `p` au mieux comme une estimation de ce nombre. Quand le nombre d'éléments à stocker est supérieur au `p` choisi, même la meilleure fonction ne peut plus éviter les conflits : deux clés différentes seront transformées en le même entier. On parle de **collision** et il faut prévoir un mécanisme pour les gérer.

Le mécanisme le plus simple pour gérer les collisions est le **chaînage** : chaque case du `vector` ne contient plus simplement la valeur correspondant à la clé, mais une liste chaînée de couples `(clé, valeur)`. Quand on cherche la valeur associée à une clé donnée, on transforme la clé pour calculer l'indice de la case du `vector` dans laquelle elle devrait être stockée, puis on parcourt la liste chaînée contenue dans cette case jusqu'à trouver la clé recherchée.

2 - Mise en oeuvre

On va mettre en oeuvre une table de hachage avec chaînage pour stocker des couples `(clé, valeur)` où la clé est une chaîne de caractères et la valeur un pointeur vers une valeur de type arbitraire.

On considère généralement les fonctions de transformation de clé `t` comme agissant en deux étapes :

- une *fonction de hachage* `h` hache la clé `k` en une valeur entière `h(k)`. Cette valeur est généralement exprimée dans un espace relativement grand (par exemple `[0, 2^32 - 1]`). On attend d'une fonction de hachage qu'elle répartisse le plus uniformément possible les valeurs dans cet espace.
- une *fonction de réduction* `r` réduit cette valeur de manière à ce qu'elle soit comprise dans l'intervalle `[0, p]`.

La fonction `t` est donc la composition de deux fonctions `h` et `r` : `t(k) = r(h(k))`. Au final, on pourra utiliser :

- pour la fonction `h` : la fonction `str_hash` suivante, que vous pourrez déclarer et définir dans les fichiers `utils.h` et `utils.c`
- pour la fonction `r` : l'opérateur `%`
- pour comparer des chaînes de caractères : la fonction `strcmp` de la bibliothèque standard

```c
uint32_t str_hash(char const *str)
{
    uint32_t h = 0;
    for (char const *p = str; *p != '\0'; p++)
    {
        h = (h << 4) + (uint32_t)(*p);
        uint32_t g = h & 0xF0000000L;
        if (g != 0)
            h = h ^ (g >> 24);
        h = h & ~g;
    }
    return h;
}
```

> ***Créer les fichiers `containers/hash_map.h` et `containers/hash_map.c`.***
> ***Définir les structures `hash_map` et `key_value_pair` ainsi que l'ensemble des fonctions suivantes :***

- initialisation et libération :

```c
// initialise un vector de taille nb_buckets dont chaque case contient
// une liste chaînée de key_value_pair
void hash_map_init(hash_map *hm, size_t nb_buckets, size_t value_size);
// il est probablement plus prudent d'écrire cette fonction après avoir écrit
// les fonctions de manipulation des données
void hash_map_free(hash_map *h);
```

- accès aux propriétés courantes :

```c
size_t hash_map_size(hash_map const *h);
size_t hash_map_value_size(hash_map const *h);
```

- manipulation des données :

```c
// retourne vrai si la clé key est présente dans la table de hachage h, faux sinon
bool hash_map_contains(hash_map const *h, char const *key);
// après avoir vérifié que la clé key n'est pas déjà présente dans la table de
// hachage h, copie les clé et valeur dans un nouveau key_value_pair et insère
// ce dernier en tête de la liste chaînée correspondant à la clé
void hash_map_insert(hash_map *h, char const *key, void const *value);
// copie la valeur contenue dans la key_value_pair correspondant à la clé key
// à l'adresse pointée par value
// (suppose que la clé key est présente dans la table de hachage h)
void hash_map_get(hash_map const *h, char const *key, void *value);
// copie la mémoire pointée par value dans la key_value_pair correspondant à la clé key
// (suppose que la clé key est présente dans la table de hachage h)
void hash_map_set(hash_map *h, char const *key, void const *value);
// supprime la key_value_pair correspondant à la clé key
// (suppose que la clé key est présente dans la table de hachage h)
void hash_map_erase(hash_map *h, char const *key);
```

Tester les différentes fonctions écrites en déclarant et en utilisant des tables de hachage dans votre main.

3 - Parcourir les éléments

Dans une table de hachage, les éléments ne sont pas stockés de manière ordonnée. Il reste cependant possible de parcourir les éléments contenus dans la table.

Déclarer et écrire la fonction suivante, qui parcourt une table de hachage h et appelle la fonction f pour chaque paire (clé, valeur) contenue :

```
void hash_map_foreach(hash_map const *h,
                      void (*f)(char const *key, void const *value));
```

On pourra tester cette fonction en l'utilisant pour afficher les éléments d'une table de hachage. Il faudra bien sûr écrire tout d'abord une fonction d'affichage dédiée au type des valeurs stockées dans la table (dans les fichiers `utils.h` et `utils.c`).

4 - Analyse de texte

On va utiliser une table de hachage pour compter le nombre d'occurrences de chaque mot dans un texte.

Créer les fichiers `algos/text_analysis.h` et `algos/text_analysis.c`, puis y définir la fonction suivante :

```
// La hash_map h contient des données de type int
void text_analysis(char const *filename, hash_map *h);
```

Cette fonction ouvre en lecture le fichier dont le nom est reçu en paramètre. Elle parcourt ensuite le contenu du fichier mot par mot et pour chaque mot rencontré :

- si le mot fait moins de 4 caractères, elle ignore le mot
- sinon, elle transforme le mot en minuscules, puis
  - si le mot est déjà présent dans la table de hachage, elle incrémente le compteur associé
  - sinon, elle insère le mot dans la table de hachage avec un compteur initialisé à 1

On pourra utiliser la fonction de la bibliothèque standard `fscanf` avec le format `%s` pour lire les mots du fichier, et la fonction `tolower` pour transformer les caractères en minuscules.

Exécuter cette fonction sur les fichiers `victor_hugo.txt` et `john_keats.txt` fournis. Par curiosité, faire varier le nombre de buckets de la table de hachage, et afficher la taille des différentes listes contenues dans ces buckets.

On veut maintenant pouvoir afficher les mots lus, avec leur nombre d'occurrences, du plus fréquent au moins fréquent, tous les mots ayant le même nombre d'occurrences étant affichés dans l'ordre alphabétique.

Comme on l'a déjà mentionné, les valeurs contenues dans une table de hachage ne sont pas ordonnées (on ne peut pas tout avoir). Pour effectuer ce tri, on peut utiliser l'algorithme le plus efficace que l'on connaisse jusqu'à présent : le tri par tas.

Dans les fichiers `utils.h` et `utils.c`, définir une structure de données `word_count` qui contient un mot et son nombre d'occurrences.
Écrire la fonction de comparaison suivante qui compare deux `word_count` par leur nombre d'occurrences, et si ces derniers sont égaux, par leur mot.

```
int compare_word_count(void const *a, void const *b);
```

On peut maintenant déclarer un tas de `word_count` avec la fonction `compare_word_count` pour définir la relation d'ordre entre ses éléments.

Il nous reste à parcourir la table de hachage et à insérer chaque `word_count` dans le tas. La fonction `hash_map_foreach` écrite précédemment pourrait nous aider. Cependant, la fonction que l'on fournit à `hash_map_foreach` est appelée uniquement avec des pointeurs vers la clé et la valeur de chaque élément de la table de hachage. On n'aura donc pas accès, dans cette fonction, au tas dans lequel on souhaite insérer les `word_count`.

Une solution consiste à déclarer le tas dans une variable globale. Bof… code smell… (effets de bord, pas de réutilisabilité, …)

On va plutôt opérer une modification (classique) sur la fonction `hash_map_foreach` : on va lui fournir en argument supplémentaire un pointeur générique `context`, qu'elle passera à son tour en paramètre à la fonction `f`, en plus des pointeurs de clé et de valeur, à chaque appel.

Faire cette modification dans la fonction `hash_map_foreach`.
Dans les fichiers `utils.h` et `utils.c`, écrire la fonction suivante, que l'on passera à `hash_map_foreach` afin d'insérer un `word_count` dans le tas passé en contexte pour chaque élément de la table de hachage.

```
void insert_word_count_in_heap(char const *key, void const *value, void *context);
```

Il ne nous reste plus qu'à extraire les `word_count` du tas un par un pour les afficher dans l'ordre souhaité.