

Dans ce TP, on souhaite produire un programme capable de compter le nombre d'occurrences de chaque mot présent dans un texte donné.

L'objectif est de travailler différents aspects de la programmation en C vus l'année dernière dont la compilation, la modularisation, l'écriture de code propre et maintenable, les entrées/sorties, l'allocation dynamique et le débogage.

*Ce sujet est dérivé du sujet de L2 portant sur le même projet mais on s'intéresse plus spécifiquement à l'implémentation d'une table de hachage pour l'algorithmique.*

Dans un premier temps, on dira qu'un mot est une suite de lettre minuscule/majuscule de la table ASCII, on ignore donc les accents et les nombres.

Pour cela, on va construire une table de correspondance permettant d'associer à un mot son nombre d'occurrences. En python on utilisera assez naturellement un dictionnaire avec le code suivant :

```
1  # Chargement du contenu du fichier
2  filename = 'input.txt'
3  with open(filename) as file:
4      content = file.read()
5
6  # Découpage en un tableau de "mots"
7  words = content.strip().split()
8
9  # On compte chaque mot
10 counts = {}
11 for word in words:
12     if word not in counts:
13         counts[word] = 1
14     else:
15         counts[word] += 1
16
17 # Affichage du résultat
18 print(len(counts), "mots")
19 for word, count in counts.items():
20     print("-", word, count)
```

La structure de ce programme se retrouvera dans la fonction principale en C. Cependant les lignes 10 à 15 exploitent la structure de dictionnaire de Python, absente en C. On commence par ce point.

Les fonctionnalités dont on souhaite disposer pour la table de correspondance sont les suivantes :

- Créer d'une nouvelle table (ligne 10)
- Tester si une clé est présente dans la table (ligne 12)
- Associer une valeur à une clé (ligne 13 et 15)
- Récupérer la valeur associée à une clé (ligne 15)
- Obtenir le nombre d'entrées dans la table (ligne 18)
- Itérer sur les éléments (ligne 19)

Une structure de donnée fréquemment utilisée pour ce besoin est la table de hachage présentée en cours d'[algorithmique et structures de données](#). Pour rappel une table de hachage est une manière de réaliser une table de correspondance (ou tableau associatif) se basant sur un tableau dans lequel on place des paires (clé, valeur), et une fonction de hachage permettant de calculer la position dans le tableau d'une paire à partir de sa clé.

Dans un monde idéal, pour deux paires dont les clés sont différentes, les positions obtenues par la fonction de hachage sont aussi différentes. Dans la réalité, cela peut arriver (en fonction de la qualité de la fonction de hachage), on appelle ça une collision. Pour les gérer, il existe plusieurs solutions, on ne s'intéresse ici qu'à l'une d'entre elles : le hachage externe par l'utilisation d'une liste chaînée. Chaque case de notre tableau contient une liste (chaînée) des paires de (clé, valeur) ayant la même position par notre fonction de hachage.

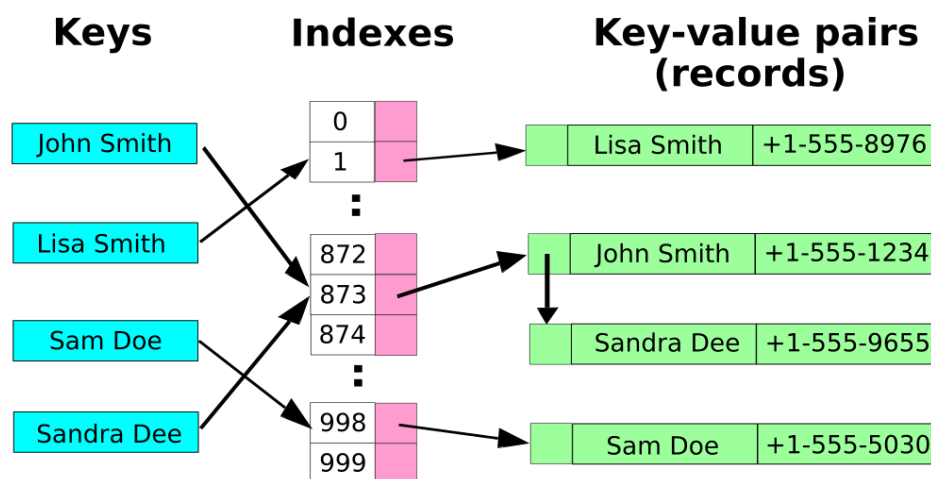


Figure 1: Gestion des collisions <sup>1</sup>

Par exemple sur la [fig:collision], les clés “John Smith” et “Sandra Dee” obtiennent la même position (873) dans le tableau et leurs entrées forment une liste chaînée.

On propose la structure de donnée suivante :

```
typedef struct Table Table;
struct Table {
    struct Entry **entries; // Tableau de listes chaînées
    int size;               // Taille du tableau
    int count;              // Nombre d'entrées dans la table
};

typedef struct Entry Entry;
struct Entry {
    struct Item item; // Paire clé/valeur
    struct Entry *next; // Pointeur vers l'élément suivant dans la liste chaînée
};

typedef struct Item Item;
struct Item {
    char *key; // Clé
```

<sup>1</sup>Source Wikipedia <https://commons.wikimedia.org/wiki/File:HASHTB32.svg>

```
int value;    // Valeur associée (ici un entier)
};
```

et l'interface suivante pour la table de hachage :

```
// Initialise une nouvelle table
int table_init(Table *table, int size);

// Tester si une clé est présente dans la table
int table_contains(Table *table, char *key);

// Associer une valeur à une clé
int table_set(Table *table, char *key, int value);

// Récupérer la valeur associée à une clé
int table_get(Table *table, char *key, int *value);

// Obtenir le nombre d'entrées dans la table
int table_size(Table *table);

// Remplit le tableau `items` des paires (clé,valeur) de la table
// Attention: les clés ne sont pas dupliquées
int table_items(Table *table, Item items[], int size);

// Itérer sur les éléments
int table_foreach(Table *table, int (*fun)(char *key, int value, void *data), void *data);
```

Comme le C ne dispose pas de gestion automatique de la mémoire on ajoute la fonction suivante à l'interface.

```
// Libère la mémoire associée à la table
int table_free(Table *table);
```

Les fonctions précédentes seront à gestion d'erreur lorsque c'est pertinent.

La fonction de hachage sera celle présentée en cours d'algorithmique et structures de donnée (*64-bit FNV-1a*).

## 1 Exercice 1 : Table de hachage

1. Fournir un code de base à chaque fonction.

*Note : cela se réduit souvent à un simple `return`*

2. Écrire un Makefile afin de compiler le programme de test. Tester la compilation.
3. Équiper chaque fonction de pré-assertions.
4. Ajouter les options de debugage suivante à votre makefile :

- `-g -fsanitize=address,undefined` à la compilation des fichiers C en fichier objet.
- `-fsanitize=address,undefined` à l'édition des liens (voir [la fiche suivante](#))

- 
5. Implémenter les fonctions précédentes. Tester au fur et à mesure avec le programme fourni.

*Note : Faire un schéma pour l'ajout d'une nouvelle entrée*

## 2 Exercice 2 : Compteur de mots

Dans cette partie, on s'intéresse au programme principale. Il n'y est pas nécessaire d'apporter de modification au module `table`.

1. Implémenter le programme principal lisant les mots depuis l'entrée standard et reproduisant le code python précédent.
2. Modifier le programme afin de lire un fichier dont le nom est passé en argument du programme. Le programme s'il est présent et l'entrée standard sinon.
3. Modifier le programme afin que le découpage en mot ne se fasse plus uniquement par les espaces mais aussi par la ponctuation. Pour simplifier, on considèrera que les mots ne sont composés que de lettres sans accent et on ne fera pas de différence entre les minuscules et les majuscules.
4. Ajouter des options l'affichage des résultats :
  - par ordre alphabétique
  - par ordre sur le nombre d'occurrences
  - croissant/décroissant sur les ordres précédents
  - limiter le nombre de résultats affichés
5. Ajouter des options pour modifier ce qui est compté :
  - par  $n$ -uplet de mots ( $n$  étant une valeur en argument)
  - seulement les mots précédant un mot donné
  - seulement les mots suivant un mot donné

*Note : Tout ce qui n'est pas un mot sera ignoré, par exemple dans "d'un rond-point", les mots "d" et "un", et "rond" et "point" se suivent malgré l'apostrophe et le trait d'union.*

*Note 2 : Il peut être utile d'introduire une structure/module intermédiaire afin de gérer une suite de mots.*

## 3 Exercice 3 : Test de modularisation

1. Échanger votre module de table de hachage avec celui d'autres étudiants. Votre programme compile-t-il toujours ? Est-il encore fonctionnel ?
2. Parmi les structures de la table, quelles sont celles qui doivent être dans l'interface (car utilisées par d'autres modules) et celles qui sont propres à l'implémentation ?

*Note : En C, il est possible de manipuler des pointeurs vers des types structurés qui ne sont pas encore définis (ils sont dits incomplets). C'est par exemple le cas pour les types récurifs comme les listes chaînées : dans le type `struct Entry`, on utilise un pointeur de `struct Entry` qui n'est pas encore défini. Le type a besoin d'être défini quand on accède à la donnée pointée ou lorsque qu'on a besoin de la taille du type (`sizeof` ou arithmétique des pointeurs).*

3. Modifier `table.h` et `table.c` en conséquence et vérifier que le programme compile toujours.
4. Peut-on faire mieux en modifiant légèrement l'interface du module `table` ?

---

## 4 Exercice 4 : Debogage développeur

1. Ajouter des statistiques de collisions sur la table de hachage.
2. Afficher des messages de debug sur l'utilisation de la table de hachage pendant son utilisation et/ou à la libération de cette dernière.
3. À l'aide de macro(s), permettre d'activer/désactiver cette fonctionnalité à la compilation.

## 5 Exercice 5 : Quelques fonctionnalités supplémentaires sur la table de hachage

1. Écrire une fonction `int table_remove(Table *table, char *key)` permettant de supprimer une entrée d'une table.
2. Proposer une méthode pour conserver l'ordre dans lequel sont ajoutés les clés à la table. Cela modifie-t-il l'interface du module `table` ? Si oui, est-ce possible de l'éviter ?
3. L'implémenter et ajouter une fonction `table_foreach_ordered` parcourant la table dans l'ordre d'insertion.
4. Écrire la fonction `table_resize` qui redimensionne la table de hachage. Cela est notamment pratique quand on ignore le nombre d'éléments que l'on va y placer à la création.
5. Implémenter un redimensionnement automatique dès que la table atteint un certain taux de remplissage (par exemple, deux fois plus d'entrées que de case, c'est-à-dire avec une moyenne de deux collisions par case).

Vérifier que le programme fonctionne toujours.

## 6 Exercice 6 : Hachage interne

Une autre méthode pour gérer les collisions consiste à placer une entrée à la première position libre après la position où elle devrait se trouver. C'est le *hachage interne* (c'est aussi celle qui est présentée dans le cours du S3).

Chaque case ne contient alors au plus une entrée, et, en cas de suppression d'entrées, un marqueur indiquant que l'entrée a été supprimé (cela est nécessaire pour la recherche et l'ajout).

1. Après avoir relu les sildes du cours d'[algorithmique et structures de données](#), donner les changements à réaliser dans les modules `main` et `table` pour passer à ce type de table de hachage.
2. Créer un nouveau module `table_interne` et implementer la table de hachage interne.