# Cours de programmation séquentielle

Table de hachage

## 1 Préambule

Cette série n'est pas notée. Comme pour le premier semestre, vous pouvez obtenir un bonus de `0.5` sur la note de l'examen en rendant 2/3 des exercices non-notés du semestre. Vous devez mettre le lien de votre repo `git` dans le wiki correspondant sur `Cyberlearn`. **Ne nous rajoutez pas comme reporter!** Vous pouvez à choix laisser le repo public ou alors le mettre en interne.

## 2 Buts

Dans ce travail vous verrez les concepts suivants:

- Créer et utiliser une librairie de table de hachage.
- Utilisation des pointeurs pour la liste chaînée.
- Utilisation de tableaux de pointeurs.
- Création de fonction de hachage simple.

Vous avez vu la théorie sur les tables de hachages avec le Professeur Albuquerque.

## 3 Énoncé

Dans ce travail pratique utilisez la technique de développement pilotée par des tests pour écrire une librairie de table de hachage.

La structure `hm_t` (pour *hashmap*) sera composée comme suit

```
struct hm_t {
    struct entry_t **entries;
    int length;
};
```

Les entrées de la table seront un tableau de liste chaînée dont chaque noeud sera une paire *clé-valeur*, de type `entry_t` (la clé et la valeur sont des chaînes de caractères)

```
struct entry_t {
    char *key;
    char *value;
    struct entry_t *next;
};
```

Ces deux structures seront des types **opaques**: les champs ne pourront pas être accédés directement par l'utilisateur · trice de votre librairie (la déclaration de la structure va dans le fichier `.h` alors que son implémentation se trouve dans le fichier `.c` correspondant).

Les fonctionnalités de votre table de hachage devront être les suivantes:

```
// création d'un pointeur vers une hm
hm_t *hm_create(unsigned int length);
// destruction de la hm et libération de la mémoire
void hm_destroy(hm_t **hm);

// insère la paire key-value dans la hm. si key est déjà présente
// écraser value dans la hm.
hm_t *hm_set(hm_t *hm, const char *const key, const char *const value);
// retourne la valeur associé à la clé, key
char *hm_get(const hm_t *const hm, const char *const key);
// retire une clé de la hm et la retourne
char *hm_rm(hm_t *hm, const char *const key);

// convertit la hm en chaîne de caractères
char *hm_to_str(const hm_t *const hm);
// affiche le contenu de la hm
void hm_print(const hm_t *const hm);
```

---

**Remarque 1** *(Utilitaires)*

Vous devrez probablement écrire d'autres fonctions "utilitaires"" dans votre librairie mais les fonctions ci-dessus sont celles que vous exposerez à l'utilisatrice (p.ex. toutes les fonctions pour créer des `entry_t` et les détruire, ou encore vérifier si la table de hachage possède déjà une clé donnée).

En particulier, il y a la fonction de hachage. Nous vous suggérons d'utiliser la fonction

```
val = 0;

for (int i = 0; i < length(c); ++i) {
    val = 43 * val + c[i];
}
return (val % length);
```

Mais vous êtes libres d'en utiliser une autre si vous voulez (n'importe laquelle de celles vues en cours par exemple, mais évitez les solutions trop compliquées dans un premier temps).

---

---

**Remarque 2** *(Collisions)*

Dans cette implémentation les collisions sont gérées par chaînage: lorsqu'une

clé génère un hash déjà présent dans la table, on insère la clé dans l'alvéole à l'aide d'une liste chaînée. Vous n'utiliserez pas ici d'adressage ouvert (tel que le sondage linéaire ou le double hachage).

---

Vous devrez également écrire un programme qui génère $N$ paires aléatoires de numéros de téléphones (qui seront des chaînes de caractères) et de noms (qui seront juste des lettres aléatoires). Il faut ensuite afficher les paires clé-valeur dans l'ordre dans lequel elles se trouvent dans la table.