# TP n$^o$ 1 (SDD Avancées- Automne 2025)

*Table de Hachage*

On se propose dans ce premier TP de coder la gestion de table de hachage. Il est demandé ici de compléter les fonctions du fichier hash.c vue en cours avec les tests correspondants à chaque fonction. Les fonctions de gestion des listes chaînées ainsi que les Cmake pour la construction du projet global sont fournis.

**Consignes** :
**La programmation se fera en C sous Linux**
**Le TP à rendre le vendredi de la semaine prochaine .**

Le répertoire globale contient les répertoires et fichiers suivants

```
    -----------------------------------
  hash_tp
       debug
             hash
            list
       src
              hash
                     CMakeLists.txt
                     HachConfig.cmake.in
                     HashConfig.cmake.in
                     hash.c        //A COMPLETER
                     hash.h
                     hash.inc
                     test-hash.c
             list
                  CMakeLists.txt
                  ListConfig.cmake
                  ListConfig.cmake.in
                  list.c
                  list.h
                  list.pc.in
                  test-list.c
            tmp
             include
             lib

 -----------------------------------
```

```
 Pour la compilation,
a) Positionnez vous dans le repertoire debug/list
b) lancez les commandes suivantes

  -cmake ../../src/list/ -DCMAKE_INSTALL_PREFIX=../tmp -DCMAKE_BUILD_TYPE=Debug
  -make
```

```
-make install
```

pour l'installation de la librairie liste afin qu'elle puisse être utilisée par d'autres programmes Si aucune erreur, refaire la même chose en vous positionnant dans le répertoire Debug / Hash

Pour la table de hashage on utilisera la structure suivantes (vue en cours) :

```
typedef struct {
    size_t              (*hash) (const void *, size_t);
    int                 (*compare) (const void *, const void *);
    size_t              length;
    size_t              nb;
    size_t              size;
    List                list[1];
}HashTable;
```

Complétez l écriture du codes des fonctions suivantes et testez les sur un programme à créer test-hash.c

```
Certaines fonctions sont déjà données.

HashTable*
hashtable_new(size_t size, size_t(*hash) (const void *, size_t),
int (*compare) (const void *, const void *));

void
hashtable_apply(HashTable * table, void (*func) (void *, void *), void *extra_data);
void

hashtable_print(HashTable * table, void (*print) (void *, FILE *), FILE * stream);
void

hashtable_delete(HashTable * table, void (*delete) (void *));
int

hashtable_insert(HashTable ** ptable, void *data, void (*delete) (void *));

void              *
hashtable_lookup(HashTable * table, void *data);

int
hashtable_retract(HashTable ** ptable, void *data, void (*delete) (void *));
```

**Les fonctions (données) de manipulation des listes et sont les suivantes.**

Vous n ?utiliserez que les fonctions dont vous aurez besoin pour la table de hachage.

```
 // function creating a list
```

```
List list_new (void);

/* function removing the first element of a list.
  TRUE is returned in the case of a success, FALSE otherwise */

int list_remove_first (List * plist, void (*delete) (void *));

/*
 * function removing the last element of a list.
 * TRUE is returned in the case of a success, FALSE otherwise
 */
int list_remove_last (List * plist, void (*delete) (void *));

/*
 * function removing the nth element of a list.
 * TRUE is returned in the case of a success, FALSE otherwise
 */
int list_remove_nth (List * plist, size_t nth, void (*delete) (void *));

/*
 * function deleting a list entirely
 */
void list_delete (List * plist, void (*delete) (void *));

/*
 * function computing the length of a list
 */
size_t
list_length (List list);

/*
 * function returning the test: is list empty?
 */
int list_is_empty (List list);

/*
 * function prepending an element to a list.
 * TRUE is returned in the case of a success, FALSE otherwise
 */
int list_prepend (List * plist, const void *data, size_t size);

/*
 * function appending an element to a list
 * TRUE is returned in the case of a success, FALSE otherwise
 */
int list_append (List * plist, const void *data, size_t size);

/*
 * function inserting an element in the nth position.
 * TRUE is returned in the case of a success, FALSE otherwise
 */
int list_insert (List * plist,
 const void *data,
 size_t size,
 size_t nth);

/*
 * function inserting an element in a sorted list
 * the compare function has the same meaning in the standard qsort function
 */
int list_insert_sorted (List * plist,
const void *data,
size_t size,
int (*compare) (const void *, const void *));

/*
 * function to give the pointer to the first data
 * NULL is returned if the list is empty
 */
void *list_first_data (List list);

/*
 * function giving the pointer to the last data
 * NULL is returned if the list is empty
 */
void *list_last_data (List list);

/*
 * function giving the pointer to the nth data
 * NULL is returned if the case of an error
 */
void *list_nth_data (List list, size_t nth);


/*
 * function giving the last sublist of a list.
 * NULL is returned if the list is empty
 */
List
list_last (List list);

/*
 * function giving the nth sublist of a list.
 * NULL is returned if the case of an error
 */
List
list_nth (List list, size_t nth);

/*
 * function finding the pointer to a data.
 * NULL is returned if the case of an error: the data does not exist
 * in the list
 */
void *list_find_data (List list,
      const void *data,
      int (*compare) (const void *, const void *));

/*
 * function giving the index of a data. -1 is returned if the data
 * is not in the list
 */
int list_index_data (List list,
     const void *data,
     int (*compare) (const void *, const void *));

/*
 * function applicating an operation to the entire list
 * extra_data is passed as the second argument of the func function for each
 * element of the list
 */
void list_foreach (List list, void (*func) (void *, void *), void *extra_data);


/*
 * function concatening the list add to the list pointed by plist
 */
void list_concat (List * plist, List add);

/*
 * function splitting a list into two list
 * *pbefore will contain all elements "inferior or equal" to data
 * *pafter will contain all elements "superior" to data
 */
void list_split (List * plist,
 List * pbefore,
 List * pafter,
 const void *data,
 int (*compare) (const void *, const void *));

/*
 * function sorting a list
 */
void list_sort (List * plist, int (*compare) (const void *, const void *));

/*
 * function merging two list. The list *plist will be sorted according
 * to the function compare
 */
void list_merge (List * plist,
 List add,
int (*compare) (const void *, const void *));
```