```
#ifndef __TYPES_H__
#define __TYPES_H__

#define MAX_LENGTH 50    // maximum word length of an entry
#define MAX_FILES 20     // maximum number of files
#define MAX_ENTRIES 1023 // capacity of hash table

// elements of the word list
typedef struct word_entry
{
  char word[MAX_LENGTH];
  int in_file; // index of file in file table
  int times;   // how many times does the word exist
  struct word_entry *next;
} word_entry;

// simple linked list of word entries
typedef struct
{
  word_entry *first_word;
  word_entry *last_word;
} word_list;

// a hash table is an array of word_list + maximum number of elements in the arr
ay
typedef struct
{
  word_list *htable;
  int hsize; // capacity of array
} hash_table;

// names of files loaded in the hash table + loaded status
typedef struct
{
  char filename[MAX_LENGTH];
  int loaded; // true if file loaded
} listfile_entry;

#endif // __TYPES_H__
```

```c
#ifndef __FUNCTIONS_H__
#define __FUNCTIONS_H__

#include "types.h"

// ------------------------------------------------------------------------
// file.c
// ------------------------------------------------------------------------

// create and initialize file table of capacity maxfile
listfile_entry *
create_filelist(int maxfiles);

// add words from file to table
int add_file(char filename[],
                                                  listfile_entry *filelist,
                                                  hash_table *htable_ptr);

// remove file from file table
int remove_file(char filename[],
                                                                 listfile_entry *
filelist,
                                                                 hash_table *htab
le_ptr);

// print file table
void print_list(listfile_entry *filelist);

// free file table
void free_filelist(listfile_entry *filelist);

// ------------------------------------------------------------------------
// hash.c
// ------------------------------------------------------------------------

// create hash table
hash_table *
create_table();

// search a word in table and print it
// returns : true if found, false otherwise
int search_word(char word[],
                                                                 listfile_entry *
filelist,
                                                                 hash_table *htab
le_ptr);

// add/update a word in table
void update_table(hash_table *htable_ptr,
                                                                         char wor
d[],
                                                                         char fil
ename[],
                                                                         int file
_index);

// print table contents
void print_table(hash_table *htable_ptr,
                                                                  listfile_entry
*filelist);

// free hash table
void free_table(hash_table *htable_ptr);

// ------------------------------------------------------------------------
// main.c
// ------------------------------------------------------------------------

// compute hash value for word
```

```c
// returns : N;/ 0 <= N < size
int hashcode(char word[], int size);

#endif // __FUNCTIONS_H__
```

```c
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "../include/types.h"
#include "../include/functions.h" // extern functions declarations

// ---------------------------------------------------------------------------
// inner functions declarations
// ---------------------------------------------------------------------------
void delete_words_from_table(int j, listfile_entry *filelist, char filename[], h
ash_table *htable_ptr);
int file_handler(char s[], char word[], char filename[], int indexToAdd, hash_ta
ble *htable_ptr);
int is_file_on_table(listfile_entry *filelist, char filename[]);
int search_space_in_file_table(listfile_entry *filelist);
int search_index_of_file(listfile_entry *filelist, char filename[]);

//---------------------------------------------------------------------------
// global functions definitions
//---------------------------------------------------------------------------

/**
   Create and initialize file table of capacity maxfiles

   parameters :
   maxfiles : capacity of file table

   returns : pointer to table or NULL in case of error
 */
listfile_entry *create_filelist(int maxfiles)
{

  listfile_entry *file_list = malloc(sizeof(listfile_entry) * maxfiles);
  if (file_list == NULL)
  {
    return NULL;
  }
  for (int i = 0; i < maxfiles; i++)
  {
    file_list[i].loaded = 0;
    memset(file_list[i].filename, ' ', MAX_LENGTH);
  }
  return file_list;
}

/**
   add words from file to table
   - checks if the file has already been loaded
   - updates the file table (if file not already loaded)
   - reads every word in the file (idem)
   - updates the hash table (idem)

   parameters :
   filename   : name of file :)
   filelist   : pointer to table of files
   htable_ptr : pointer to hash table

   returns :
    1 if file already present in table
    2 if no space left in filelist
   -1 if file doesn't exist or can't be read
   -2 if allocation error
    0 if everything ok
*/
int add_file(char filename[],
             listfile_entry *filelist,
             hash_table *htable_ptr)
```

```c
{
  if (is_file_on_table(filelist, filename) != -1)
  {
    return 1;
  }

  int indexToAdd = search_space_in_file_table(filelist);
  if (indexToAdd == -1)
  {
    return 2;
  }

  strcpy(filelist[indexToAdd].filename, filename);
  filelist[indexToAdd].loaded = 1;

  char word[MAX_LENGTH];
  char s[MAX_LENGTH] = "test/";
  strcat(s, filename);

  return file_handler(s, word, filename, indexToAdd, htable_ptr);
}

/**
   remove file from file table

   parameters :
   filename   : name of file to remove
   filelist   : pointer to table of files
   htable_ptr : pointer to hash table

   returns :
   -1 if file not in table
    0 if file removed
*/
int remove_file(char filename[],
                listfile_entry *filelist,
                hash_table *htable_ptr)
{
  int file_index = search_index_of_file(filelist, filename);
  if (file_index == -1)
  {
    fprintf(stderr, "File is not in the table.\n");
    return -1;
  }

  delete_words_from_table(file_index, filelist, filename, htable_ptr);
  return 0;
}

/*
  print file table (only loaded files)

  parameters :
   filelist : pointer to table of files
*/
void print_list(listfile_entry *filelist)
{
  printf("Files loaded: \n");
  for (int i = 0; i < sizeof(filelist); i++)
  {
    if (filelist[i].loaded == 1)
    {
      printf("\t- %s [%d]\n", filelist[i].filename, i);
    }
  }
}

/**
   free file table
```

```
parameters :
   filelist   : pointer to table of files
*/
void free_filelist(listfile_entry *filelist)
{
  free(filelist);
}

// ***********************************************************************
// inner functions
// ***********************************************************************

/**
 * Delete words from the table for a specific file
 *
 * parameters:
 *  file_index : index of the file to remove
 *  filelist : pointer to table of file to remove
 *  filename : name of the file to remove
 *  htable_ptr : pointer to hash table
 */
void delete_words_from_table(int file_index, listfile_entry *filelist, char file
name[], hash_table *htable_ptr)
{
  for (int i = 0; i < htable_ptr->hsize; i++)
  {
    if (htable_ptr->htable[i].first_word != NULL)
    {
      word_list *word_list_to_delete = &htable_ptr->htable[i];
      word_entry *current = word_list_to_delete->first_word;

      while (current != NULL)
      {
        if (current->in_file == file_index)
        {
          word_list_to_delete->first_word = current->next;
          free(current);
          current = word_list_to_delete->first_word;
        }
        else
        {
          current = current->next;
        }
      }
    }
  }
  strcpy(filelist[file_index].filename, "");
  filelist[file_index].loaded = 0;
  printf("File %s got removed.\n", filename);
}

/**
 * Handle the file manipulation to add the content of this file to the hash tabl
e
 *
 * parameters:
 *  s : Relative path of the file
 *  word : word to read in the file (loop through all words)
 *  filename : name of the file to add
 *  indexToAdd : gives the index where we should load the file
 *  htable_ptr : pointer to hash table
 *
 * returns:
 *  -1 if the files doesn't exist or can't be read
 *  -2 if allocation error
 *  0 if all fine
 */
int file_handler(char s[], char word[], char filename[], int indexToAdd, hash_ta
```

```
ble *htable_ptr)
{
  FILE *fp;
  fp = fopen(s, "r");
  if (fp == NULL)
  {
    fprintf(stderr, "File doesn't exist or can't be read.\n");
    return -1;
  }

  int i = 0;
  while (fscanf(fp, "%s", word) == 1)
  {
    if (i > htable_ptr->hsize)
    {
      fprintf(stderr, "\n[ALLOCATION ERROR] – There are too many words in %s\nThe file has been lo
aded with just %d words.\n", filename, i);
      return -2;
    }
    update_table(htable_ptr, word, filename, indexToAdd);
    i++;
  }
  fclose(fp);
  return 0;
}

/**
 * Check if the file already exist
 *
 * parameters:
 *  filelist : pointer to table of file
 *  filename : name of the file
 *
 * returns:
 *  1 if file already exist
 *  -1 if file doesn't exist
 */
int is_file_on_table(listfile_entry *filelist, char filename[])
{
  for (int i = 0; i < MAX_FILES; i++)
  {
    if (strcmp(filelist[i].filename, filename) == 0)
    {
      fprintf(stderr, "File is already present in table.\n");
      return i;
    }
  }
  return -1;
}

/**
 * Check if there is space to add a file in the filelist and returns the index w
here it can be added
 *
 * parameters:
 *  filelist : pointer to table of file
 *
 * returns:
 *  i if there is space on filelist
 *  -1 if there is no space left
 */
int search_space_in_file_table(listfile_entry *filelist)
{
  for (int i = 0; i < MAX_FILES; i++)
  {
    if (filelist[i].loaded == 0)
    {
      return i;
    }
  }
  fprintf(stderr, "No space left in filelist.\n");
  return -1;
}

/**
 * Search for a file in filelist and return the index
 *
 * parameters:
 *  filelist : pointer to table of file
 *  filename : name of the file
 *
 * returns:
 *  i if file is found on filelist
 *  -1 if is not found
 */
int search_index_of_file(listfile_entry *filelist, char filename[])
{
  for (int i = 0; i < MAX_FILES; i++)
  {
    if (strcmp(filelist[i].filename, filename) == 0 && filelist[i].loaded == 1)
    {
      return i;
    }
  }
  return -1;
}
```

```c
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../include/functions.h" // global functions declarations

// ------------------------------------------------------------------------
// inner functions declarations
// ------------------------------------------------------------------------

word_entry *create_word_entry(int fileIndex, char word[]);
int search_word_in_table(word_list *word_list_to_search_in, listfile_entry *file
list, char word[]);
void update_handler(word_list *word_list_to_add_in, int file_index, char word[])
;
void print_table_index(hash_table *htable_ptr, listfile_entry *filelist, int i);
char *clean_word(char word[]);

//------------------------------------------------------------------------
// global functions definitions
//------------------------------------------------------------------------

/**
   create and initialize hash table

   returns : pointer to table or NULL if creation error
*/
hash_table *create_table()
{
   hash_table *hash_t = (hash_table *)malloc(sizeof(hash_table));
   if (hash_t == NULL)
   {
      return NULL;
   }
   hash_t->hsize = MAX_ENTRIES;
   hash_t->htable = (word_list *)malloc(sizeof(word_list) * hash_t->hsize);
   if (hash_t->htable == NULL)
   {
      return NULL;
   }
   for (int i = 0; i < hash_t->hsize; i++)
   {
      hash_t->htable[i].first_word = NULL;
      hash_t->htable[i].last_word = NULL;
   }
   return hash_t;
}

/**
   search a word in table ; print word if found, with number of occurrences
   and file where word is found

   parameters :
   word : the word to look for
   filelist   : pointer to table of files
   htable_ptr : pointer to hash table

   returns : true if found, false otherwise
*/
int search_word(char word[],
                listfile_entry *filelist,
                hash_table *htable_ptr)
{
   int hCode = hashcode(word, strlen(word));
   word_list *word_list_to_search_in = &htable_ptr->htable[hCode];
   return search_word_in_table(word_list_to_search_in, filelist, word);
}

/**
```

```c
   lookup for word in table and update table accordingly

   parameters :
   htable_ptr : pointer to hash table
   word : word to be added into the hash table
   filename   : filename from where the word was read
   file_index:  the position where the filename has been stored
*/
void update_table(hash_table *htable_ptr,
                  char word[],
                  char filename[],
                  int file_index)
{
   if (htable_ptr == NULL)
   {
      htable_ptr = create_table();
   }
   word = clean_word(word);
   int hCode = hashcode(word, strlen(word));
   word_list *word_list_to_add_in = &htable_ptr->htable[hCode];
   update_handler(word_list_to_add_in, file_index, word);
}

/**
   print table contents

   parameters :
   htable_ptr : pointer to hash table
   filelist   : pointer to table of files
*/
void print_table(hash_table *htable_ptr,
                 listfile_entry *filelist)
{
   printf("Words in the table: \n");
   for (int i = 0; i < htable_ptr->hsize; i++)
   {
      print_table_index(htable_ptr, filelist, i);
   }
}

/**
   free hash table

   parameters :
   htable_ptr : pointer to hash table
*/
void free_table(hash_table *htable_ptr)
{
   for (int i = 0; i < htable_ptr->hsize; i++)
   {
      word_entry *to_free = htable_ptr->htable[i].first_word;
      while (to_free != NULL)
      {
         word_entry *aux = to_free;
         to_free = to_free->next;
         free(aux);
      }
   }
   free(htable_ptr->htable);
   free(htable_ptr);
}

// ------------------------------------------------------------------------
// inner functions definitions
// ------------------------------------------------------------------------

/**
 * Create a new word entry
 *
 * parameters:
 *  fileIndex : index of the file where the word is
 *  word : to add to the word entry
 *
 * returns:
 *  NULL if there is allocation error
 *  to_add if the word entry is well created
 */
word_entry *create_word_entry(int fileIndex, char word[])
{
   word_entry *to_add = (word_entry *)malloc(sizeof(word_entry));
   if (to_add == NULL)
   {
      return NULL;
   }
   strcpy(to_add->word, word);
   to_add->in_file = fileIndex;
   to_add->times = 1;
   to_add->next = NULL;
   return to_add;
}

/**
 * Search for a word in the table
 *
 * parameters:
 *  word_list_to_search_in : word list needed to find the word in
 *  filelist : pointer to table of file
 *  word : word to search
 *
 * returns:
 *  0 if the word does not exist in the table
 *  1 if the word was found
 */
int search_word_in_table(word_list *word_list_to_search_in, listfile_entry *file
list, char word[])
{
   while (word_list_to_search_in->first_word != NULL)
   {
      word_entry *to_search = word_list_to_search_in->first_word;
      if (strcmp(word_list_to_search_in->first_word->word, word) == 0)
      {
         printf("The word exist %d times.\nHe was found in file %s at index %d.\n", to_search->time
s, filelist[to_search->in_file].filename, to_search->in_file);
         return 1;
      }
      word_list_to_search_in->first_word = word_list_to_search_in->first_word->n
ext;
   }
   fprintf(stderr, "The word does not exist in table.\n");
   return 0;
}

/**
 * Update the table
 *
 * parameters:
 *  word_list_to_search_in : word list needed to add the word in
 *  file_index : index of the file
 *  word : word to add
 */
void update_handler(word_list *word_list_to_add_in, int file_index, char word[])
{
   if (word_list_to_add_in->first_word == NULL)
   {
      word_list_to_add_in->first_word = create_word_entry(file_index, word);
   }
   else
   {
```

```
      word_entry *to_add = word_list_to_add_in->first_word;
      word_entry *prev = to_add;
      while (to_add != NULL)
      {
         if (strcmp(to_add->word, word) == 0 && to_add->in_file == file_index)
         {
            to_add->times++;
            return;
         }
         prev = to_add;
         to_add = to_add->next;
      }
      word_entry *new_word = create_word_entry(file_index, word);
      prev->next = new_word;
   }
}

/**
 * print the table index passed in paremeters (i)
 *
 * parameters:
 *  htable_ptr : pointer to hash table
 *  filelist : pointer to table of file
 *  i : index of the table to print
 */
void print_table_index(hash_table *htable_ptr, listfile_entry *filelist, int i)
{
   if (htable_ptr->htable[i].first_word != NULL)
   {
      word_list *word_list_to_print = &htable_ptr->htable[i];
      word_entry *to_print = word_list_to_print->first_word;
      while (to_print != NULL)
      {
         char *filename_to_print = filelist[to_print->in_file].filename;
         char *word_to_print = to_print->word;
         int times = to_print->times;
         if (filelist[to_print->in_file].loaded == 1)
         {
            printf("Filename: %s || Times in file: %d || Word: %s\n", filename_to_print, times,
 word_to_print);
         }
         to_print = to_print->next;
      }
   }
}

/**
 * Clean a word to make it to lower case and avoid having to deal with punctuati
on, space, or digit
 *
 * parameters:
 *  word : word to clean
 *
 * returns:
 *  word without punctuation
 */
char *clean_word(char word[])
{
   for (int i = 0; word[i]; i++)
   {
      if (word[i] == ',' || word[i] == '.' || isspace(word[i]) || isdigit(word[i]
) || word[i] == '!' || word[i] == ';')
      {
         word[i] = ' ';
      }
      else
      {
         word[i] = tolower(word[i]);
      }
```

```
    }
    return word;
}
```

```c
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../include/types.h"
#include "../include/functions.h"

//--------------------------------------------------------------------------

int main()
{
  // create hash table
  hash_table *htable_ptr;
  htable_ptr = create_table();

  // create filelist array
  listfile_entry *filelist;
  filelist = create_filelist(MAX_FILES);

  // display menu
  while (1)
  {
    int nbchoices = 0;
    fprintf(stderr, "\nChoisir une action\n");
    fprintf(stderr, "%d. Load a file in dictionary\n", ++nbchoices);
    fprintf(stderr, "%d. Search a word in dictionary\n", ++nbchoices);
    fprintf(stderr, "%d. Remove file from dictionary\n", ++nbchoices);
    fprintf(stderr, "\n");
    fprintf(stderr, "%d. Print dictionary\n", ++nbchoices);
    fprintf(stderr, "%d. Print file list\n", ++nbchoices);
    fprintf(stderr, "\n0. Quit\n");
    int choice;
    while (1)
    {
      fprintf(stderr, "Your choice ? ");
      scanf("%d", &choice);
      if (choice >= 0 && choice <= nbchoices)
      {
        break;
      }
      fprintf(stderr, "\nError %d is an incorrect choice\n", choice);
    }
    if (choice == 0)
    {
      break;
    }

    fprintf(stderr, "-----------------------------------------------\n");

    char file_str[30];
    char word[MAX_LENGTH];
    switch (choice)
    {
      // Load a file in dictionary
    case 1:
      printf("Which file do you want to load ?\n");
      scanf("%s", file_str);
      add_file(file_str, filelist, htable_ptr);
      break;

    // Search a word in dictionary
    case 2:

      printf("Which word do you want to search ?\n");
      scanf("%s", word);
      search_word(word, filelist, htable_ptr);
      break;

    // Remove file from dictionary
```

```c
    case 3:

      printf("Which file do you want to remove ?\n");
      scanf("%s", file_str);
      remove_file(file_str, filelist, htable_ptr);
      break;

      // Print dictionary
    case 4:
      print_table(htable_ptr, filelist);
      break;

    // Print file list
    case 5:
      print_list(filelist);
      break;
    }
    fprintf(stderr, "-----------------------------------------------\n");
  }

  // the end : free allocated memory
  free_filelist(filelist);
  free_table(htable_ptr);

  return 0;
}

// compute hash value for word
// returns : N ; 0 <= N < size
int hashcode(char word[], int size)
{
  int N = 0;
  while (*word != '\0')
  {
    N += *word++;
  }
  return (N % size);
}
```