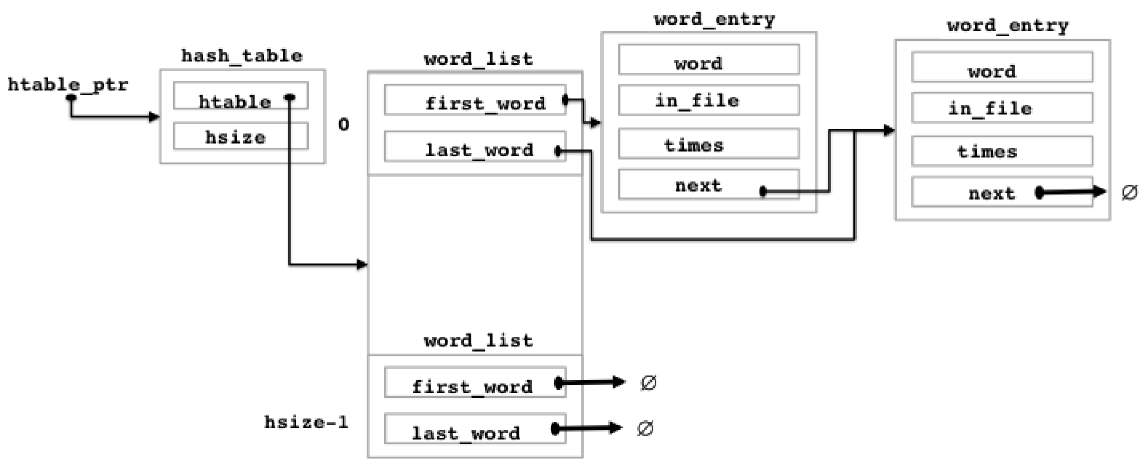
TP5: Word search machine

1 Getting started

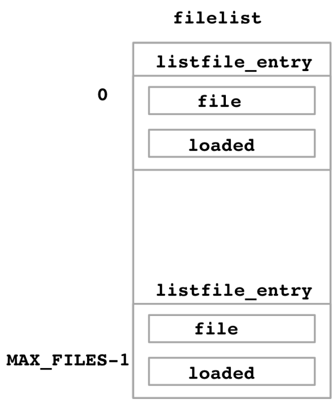
The goal of the project is to implement a word search engine, based on a dictionary. The word search machine has an interface with the user (provided in **main.c**) that allows to :

1. Load a file into the dictionary. The word search machine asks the user to provide the file to be loaded. The user enters the name of the file (e.g. myfile.txt) through the command line. Then, the word search machine executes the function **add_file**. A more detailed description is provided in the next section.
2. Search a word in the dictionary. The word search machine asks the user to provide the word to be searched. The user enters a word (e.g. forest) through the command line and the word search engine machine executes the function **search_word** and informs the user i) in which files this word exists and ii) how many times in has been observed in each file.
3. Remove a file from the dictionary. The word search machine asks the user to provide the file to be removed. The user enters the name of the file (e.g. myfile.txt) through the command line. Then, the word search machine executes the function **remove_file**. Further description is provided in the next section.
4. Print the words stored into the dictionary.
5. Print the list of files, loaded into the dictionary.

The codes for this lab are in the directory **/share/l3info/CUnix/tp5**. The dictionary that stores the words is implemented using a hash table with separate chaining, as the one seen in the course :



An array **filelist** of structures of **listfile_entry** is used to store the names of the files already loaded to the dictionary **file** and their status **loaded**.



A set of test files is provided in the folder **test**.

2 Implementation

The file **types.h** provides the declaration of the required structures (**word_entry, word_list, hash_table, listfile_entry**), the maximum number of files allowed to be loaded (**MAX_FILES**), the maximum length for a filename and a word (**MAX_LENGTH**) and the size of the hash table (**MAX_ENTRIES**). The file **functions.h** provides the declaration of the functions. The source files and the functionalities are described below.

— **main.c** : includes the following functions :
  - — **int main ()** : displays the menu of choices to the user and manages the creation and deletion of the required data structures (see comments in the code).
  - — **int hashcode(char word[], int size)** : returns the hash value of a word, as we have seen during the course. A typical hashcode function sums all the ascii characters of the word and returns the modulo operation of this sum and the size of the hash table.

— **file.c** includes the functions dedicated to the manipulation of a file
  - — **listfile_entry * create_filelist(int maxfiles)** : creates and initializes the table that keeps the information regarding the loaded files.
  - — **int add_file(char filename[], listfile_entry * filelist, hash_table * htable_ptr):** It verifies that the file to be loaded (given by the user) has not already been loaded. In this case, it loads the file. To so that, the array **filelist** is updated by including the name of the new file at its first free position. The array index is the identificator of the file. Then, it reads the file word by word (numbers are excluded) and updates the table correspondingly (see function **update_table**). No difference should be made between capital and small letters, e.g. all words are stored in small letters. When all the words of the file have been loaded to the dictionary, the status of the file in the array **filelist** is set to 1, which means loaded.

  - **int remove_file(char filename[], listfile_entry * filelist, hash_table * htable_ptr):** It verifies that the file to be removed (given by the user) is loaded in the dictionary. In this case, it searches the whole dictionary in order to find all the words that belong to this file and removes them. In case the file is not loaded to the dictionary, it informs the user.
  - **void print_list(listfile_entry * filelist):** Prints the array **filelist** with the names of the loaded files and their status.
  - **void free_filelist(listfile_entry * filelist):** Deallocates the memory for the table that keeps the information about the loaded files.
- **hash.c** includes the functions dedicated to the manipulation of the hash table :
  - **hash_table * create_table():** It creates and initializes the hash table.
  - **int search_word(char word[], listfile_entry *filelist, hash_table *htable_ptr):** It searches the word given by the user in the dictionary and it prints the number of times found in each file. In case the word does not exist, it informs the user.
  - **void update_table(hash_table * htable_ptr, char word[], char filename[], int file_index)** : It looks up for a word and updates the table accordingly. For a word, it verifies if it is the first time that the word is found in this file. In this case, it adds the word to the linked list at the hash table position given by the hashcode function, by creating a new **word_entry**. The word is stored in table **word**, the number of occurrences (**times**) is initialized to 1 and the identificator of the file **in_file** is set to the corresponding index from the array **filelist**. In case the word has already been found before in this file, it has been already inserted in the linked list (same word, same file). Then, only the number of occurrences is increased by one.
  - **void print_table(hash_table *htable_ptr, listfile_entry * filelist):** It prints the non empty positions of the dictionary.
  - **void free_table(hash_table * htable_ptr):** Deallocates the memory used for the hash table

3 Exercise

1. Implement the functions of the word search machine in the files **main.c**, **file.c** and **hash.c**
2. Complete the **Makefile** in order to compile correctly your compilation files.
3. Check your program for correct operation and implementation (memory leak, uninitialized values etc) using Valgrind.
4. You are allowed (and advised) to **add additional sub-functions** to your implementation.

4 Tips

* Program progressively and test each time
* Do not hesitate to insert additional functions

4.1 Version minimal

- hash.c
  - Define the following 3 structures :
    1. word_entry : a list element
    2. word_list : whatever is required to perform the operations over the linked lists
    3. hash_table : a minimum, initial, hash table
  - create_table : create and initialize an empty table
  - update_table : add a word without control at the head of the list
  - print_table : go through the table and print it (without the filename, only the index)
- main.c
  - program hashcode function
  - test from main :
    - create_table,
    - update_table with some values
    - print_table

4.2 Next versions

After the minimal version is tested, we can complete and test little by little our program.

- update_table : complete the function, alongside with print_table and test different cases, such as :
  - missing word
  - word that is found twice in the same file
  - same word that is found in two different files
- search_word : program and test considering different cases, similar to update_table.
- release_table : program and test with valgrind

5 Optional

In order to facilitate the search, it should be possible for the user to enter logical expressions (at least by the operator AND), i.e. word1 AND word2 that will return the files containing both words. More operators (eg OR, NOT) and bracket support are desirable !