Tutorial 5 - Autocompletion

What we learn in this tutorial:

- Dynamic allocation of memory
- Working with dynamic lists

```
Autocompletion
Loaded 22729 words
> rio
riot
rioter
rioting
riotous
riots
> deri
derides
derision
deriv
derivation
derivative
derive
derived
derives
```

The goal is to build a program that provides an "autocompletion" service. The program provides a list of words that match the beginning of a word given as input.

This is done "learning" the words from a large text file and building a dictionary whose words are compared to the user input. The program asks the user to input a string containing the beginning of a word and lists all the words in its dictionary that match that beginning.

The listing on the left shows an expected output when using the file "shakespeare.txt" as input. It might take a while to load all the words from this file, so you want to debug the program with a shorter file first.

We define a word as a group of at least 3 alphabetic characters bounded by non-alphabetic characters. Thus, to identify words in the text, we read character by character the text file from the beginning. As we meet an alphabetic character (the function *isalpha()* from *ctype.h* can be used to match uppercase and lowercase characters), we append it at the end of a temporary buffer. We keep collecting characters in the buffer until a non-alphabetic character is meet. If at least three alphabetic characters are present in the buffer, we try insert the word in the dictionary. If the word is already present in the dictionary, we simply increase the counter associated to that word.

1) Write a program that:

1. Declares a file handler (FILE *fp) and open the file "shakespeare.txt" in the local directory for reading.
2. Declares the following typedef definition for an entry in the dictionary (the dictionary is implemented as a linked list):

```
typedef struct entry {
    char* word;
    int qnt;
    struct entry* next;
} entry;
```

3. Scans the characters in the file using a while loop until EOF:
    a. If the character is alphabetic, add the character buffer[] incrementing an index
    b. If the character is not alphabetic, complete the string with the terminator (NULL) and call the function entry() with the following prototype.

```
entry* insert(char* s, entry* dict);
```

4. Implement the function insert() to do a sorted insertion in the list:
    a. Scan the list to find an insertion point in alphabetic order. To compare two strings in alphabetic order, use the function strcmp() from string.h
    b. Check if the loop scan was ended because the word already existed. If this is the case increase the word count of that word.
    c. If the word was not present, allocate the new entry (using *malloc()* from *stdlib.h*), allocate memory for the string within the entry too, and copy the input string in the entry.
    d. Update the list head if the insertion occurred at the beginning and return the new head (the head should be returned regardless an update to head happened)

5. Implement the function find() with the following prototype:

```
void find(char* s, entry* dict)
```

   The function scans the array to check that the string "s" input by the user matches the beginning of the dictionary entry. The function *strncmp()* can be used to compare only n entry (as opposed to strcmp() that compares the entire string). The function prints the corresponding word when a match is found.

2) Modify the program so that only the first 5 most common entries (the ones with the largest number of occurrences) are listed when the user inputs a search string. This requires sorting the entries once found.