# ENEE 140 Project 2: Finding a Needle in the Haystack

**Posted**: Tuesday, 2 April 2024
**Due**: This project has two deadlines:

- A partial implementation is due on Friday, 19 April 2024 at 11:59 PM
- A complete implementation is due on Monday, 29 April 2024 at 11:59 PM

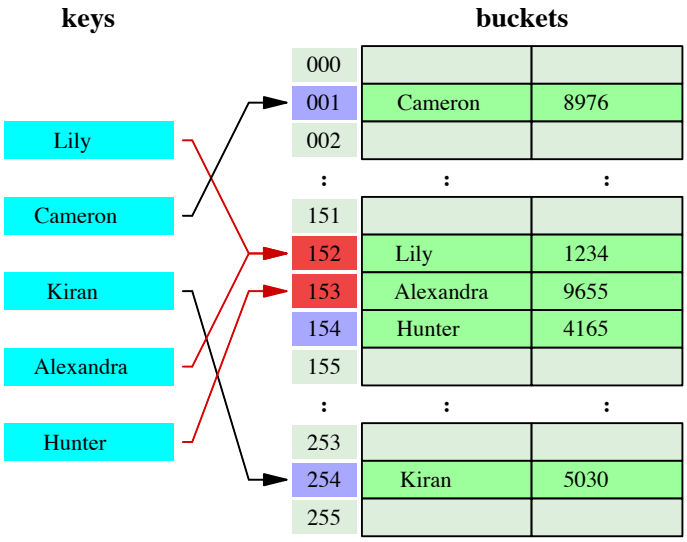
## Project objectives

1. Declare and manipulate arrays and strings.
2. Learn to avoid reading/writing beyond array and string bounds.
3. Cement your understanding of conditional statements, loops and functions.
4. Learn how to complete a complex programming project in C by splitting the functionality into several code and header files.
5. Understand how to implement a hash table for fast lookups.

## Project description

In this project, you will write a program that reads a large text file (for example, a file that contains the text from all the plays written by William Shakespeare) and then allows the user to search for several words to determine if they are present in the file. For each word that can be found in the file, the program should also print the number of times the word appears.

This functionality can be implemented using a data structure known as a *hash table*, which maps *keys* to *values*. In your program, the keys will be C strings (corresponding to the distinct words from the input file) and the values will be integers (corresponding to the number of occurrences). The hash table allows you to store a <key, value> pair, to look up a key, and to retrieve the value that corresponds to a key stored in the table. The diagram below (adapted from Wikipedia) illustrates a hash table.




An entry in the hash table is called a *bucket* and must be able to store a key and a value. This can be achieved by defining a C `struct` that contains an integer and a string. You will then declare an array of buckets, which has a pre-determined number of elements (256, in the diagram). The bucket where a <key, value> pair will be stored is determined by applying a *hash function* to the key. A hash function maps a string to an integer, which corresponds to the index of a bucket. For example, in the diagram the string "Lily" is mapped to 152, and the key-value pair <"Lily", 1234> is stored in bucket 152. A good hash function will distribute the key strings uniformly among the available buckets.

However, sometimes two different strings will map to the same value; in the diagram, the output of the hash function for "Alexandra" is also 152. This is called a *collision*. When you are trying to store a key-value pair in the hash table and you find that the corresponding bucket is already occupied (because of a hash collision), you must find some other place to store they key and the value. One possible strategy to handle collisions is to continue scanning the bucket array until you find an open bucket. In the diagram, the pair <"Alexandra", 9655> is therefore stored in bucket 153. Similarly, when searching for a key in the hash table, compute the initial bucket number by applying the hash function to the key, and then determine if the key is stored in that bucket. If you did not find the key, continue probing down the array until you either find the key or you hit an empty bucket; if you ran into an empty bucket, this means that the value is not stored in the hash table. This strategy is called *closed hashing with linear probing*.

Note you may encounter collisions even for a key that has a unique hash; for example, in the diagram "Hunter" is the only string that maps to 153. However, after inserting "Lily" and "Alexandra" in the hash table, buckets 152 and 153 are occupied and the pair <"Hunter", 4165> must be stored in bucket 154.

With this hash table design, the number of buckets available corresponds to the maximum number of key-value pairs you can store. The *load factor* is the ratio of the number of key-value pairs stored to the number of available buckets. As the load factor approaches 1, the performance of the hash table degrades, as collisions become more frequent and you must probe a long chain of buckets during each store and search operation. A good rule of thumb is that the load factor of a closed

hash table should not exceed 0.7. Another important statistic is the *longest collision chain*.

### Program structure

The functionality of the program will be implemented in three files. `enee140_hashtable.h` is a header file that includes function prototypes for your hash table's advanced programming interface (API). In this header file, you may also declare some constants you use in your program, such as the maximum number of buckets and the maximum size of a key string. `enee140_hashtable.c` is a C file where you define (implement) these functions, as well as any other helper functions you may find useful. `enee140_lookup.c` is the C file that includes the `main()` function of your program, as well as some additional functions for processing the lines and the words read from standard input. In the `main()` function, you will extract each word from the input and you will invoke (call) the functions declared in `enee140_hashtable.h` to store and search words in the hash table.

## User interface

In this project, you will write a program that reads a large text file and counts the frequency of occurrence for each word. To do this, you will store the words and corresponding counts in a hash table and you will read the file sequentially. Whenever you encounter a word that you've seen before, you will increment its count in the hash table.

### 1 Command line arguments

Your first task is to read the name of the text file from a parameter provided on the command line. The program will be launched as follows:

```
./enee140_lookup filename
```

where `filename` is the name of the text file that you will analyze.

### 2 Line input

Your second task is to initialize the hash table by invoking the function described in Step 5. Then, open the input file for reading, using the following code:

```c
FILE *file;

// Check command-line arguments
if (argc < 2) {
    fprintf(stderr, "Usage: %s filename \n", argv[0]);
    return -1;
}

file = fopen(argv[1], "r");
if (file == NULL) {
    printf ("Could not open the %s file.\n", argv[1]);
    exit (-1);
}
```

This code snippet will open the `filename` file from the same directory as the program. Read the file line-by-line, keeping track of the line count, until you encounter EOF. You can read a complete line from the file using `fgets(line, MAX_LINE, file)`. From each line read in this way, extract the words (separated by whitespace and punctuation marks), as described in Step 12. If the word is already present in the hash table, retrieve the corresponding value, using the function described in Step 8, increment it, and re-insert the word with the new value by invoking the function described in Step 6. If the word is not already present, insert it with a value of 1.

### 3 Main menu

Your third task is to print out the following menu on the screen:

```
Welcome to the ENEE140 word lookup program!
1. Print hash table statistics
2. Look up a word
3. Exit
```

Then prompt the user for a choice by printing out:

```
Enter your choice (1-3):
```

You may assume that the user will enter an integer. If the input is a valid number between 1 and 3, you should proceed to the next step. Otherwise, you should print out the following error message and ask user to re-enter a choice:

```
Sorry, that is not a valid option
Enter your choice (1-3):
```

After the user provides a user option, proceed to the next step.

### 4 Menu options

Depending on the selected option, do the following:

1. Print the load factor, computed using the function from Step 9.
2. Prompt the user for a word:

   ```
   Enter a word:
   ```

   Read the word provided by the user and determine if the word was included in the input you processed in Step 2 and the frequency of occurrence, by invoking the function from Step 8. Print either the word count or a message stating that the word was not found.
3. Exit the program.

## Hash table functionality

Before you start implementing functions for storing and searching data in your hash table, you must declare the arrays that you will use to store the keys and values, and you must implement a hash function, to map a key to an index in these arrays. In `enee140_hashtable.h`, define two

constants, which specify the number of buckets available and the longest string that can be stored in the hash table, as follows:

```
#define NBUCKETS        50021   // prime number, for better hash uniformity
#define MAX_STRING      80
```

It is a good idea to choose a prime number for the number of buckets, as this will result in a more uniform distribution for the outputs of the hash function, which will reduce the number of collisions. You may use the following hash function, which is simple and reasonably effective when the keys are strings:

```
unsigned
hash_function(char s[])
{
    unsigned hashval, i;

    for (hashval=0, i=0; s[i] != '\0'; i++) {
        hashval += s[i] + 31*hashval;
    }

    return hashval % NBUCKETS;
}
```

This function returns a positive integer between 0 and `NBUCKETS`, which can be used as an index into the key and the value arrays. For example, `hash_function("Tudor")` returns 31687 and `hash_function("Dumitras")` returns 48160. Implement the hash function in `enee140_hashtable.c`. In the same file, declare the storage of the hash table as follows:

```
typedef struct _bucket {
    char key[MAX_STRING];
    int value;
} Bucket;

typedef Bucket Hashtable[NBUCKETS];

static Hashtable my_hash_table;
static int used_buckets;
```

`my_hash_table` and `used_buckets` are static variables, which may be read and modified in all the functions implemented in `enee140_hashtable.c`, but are not visible in other files. `my_hash_table` is an array of `struct`s; each element of the array has two members, `key` (a string) and `value` (an integer), which correspond to the key-value pair stored in that bucket. For example, considering the hash table shown on page 2, `my_hash_table[0].keys` contains an empty string (bucket 0 is unoccupied), while `my_hash_table[1].key` contains the string "Cameron" and `my_hash_table[1].value` is 8976.

You are now ready to start implementing the functions from your hash table's public API (Steps 5–9). These functions should be declared in `enee140_hashtable.h` and implemented in `enee140_hashtable.c`.

## 5 Initialize the hash table

Implement a function with the following declaration:

```
void    hash_initialize();
```

In this function, clear the content of the hash table by ensuring that each key is an empty string. An empty string indicates that the corresponding bucket is not occupied and can be used to store a key and a value. In this function you should also reset all the variables that hold various statistics for the hash table (e.g. `used_buckets`). You must invoke this function before performing any other operations on the hash table.

**Hint:** You could also define a helper function that determines if a bucket is empty. This function will come in handy when implementing the functionality described in the following sections.

## 6 Store a key-value pair in the hash table

Implement a function with the following declaration:

```
int     hash_put(char key[], int value);
```

This function stores a key-value pair in the hash table. First, get the index of a bucket by invoking the hash function on the key. If the bucket is empty, copy the key string in the key field of the bucket and the value in the value field. Otherwise, scan down in the bucket array from that index until you either find the key or you hit the first open bucket. If the key is already present in the hash table, update the value; otherwise store the key and the value in the empty bucket you have identified. If you reach the end of the bucket array, resume scanning from the beginning of the array. If you successfully stored the key-value pair in the hash table, increment the number of buckets in use and return 1. If the operation was unsuccessful, return 0.

**Hint:** If you have used all the available buckets, you could keep scanning the array forever. In this case, the key-value pair cannot be stored because the hash table is full. Make sure you add a check to prevent this problem.

## 7 Look up a key in the hash table

Implement a function with the following declaration:

```
int     hash_lookup(char key[]);
```

This function should return 1 if key is stored in the hash table, and 0 otherwise. Start by getting the index of a bucket by invoking the hash function on the key. If the bucket is occupied, compare its key member with the `key` parameter passed to the function. If they are identical, then the key is present in the hash table. If not, continue comparing the `key` parameter to the keys stored in the next buckets down the array (resuming from the beginning if you've reached the end of the array). Stop probing when you either find the key (in which case the key is stored in the hash table) or you find an empty bucket (in which case the key is not stored in the hash table).

**Hint:** You may use the `strncmp` function from `string.h` for comparing two strings.

### 8 Retrieve the value that corresponds to a key in the hash table

Implement a function with the following declaration:

```
int     hash_get(char key[]);
```

Search for the `key` parameter in the hash table, as described in Step 7. If you find the key, return the value stored in the corresponding bucket. If you do not find it, return 0.

**Hint:** As the `hash_get` and `hash_lookup` functions probe the array of buckets in the same manner, you may define a helper function that implements this probing functionality and returns the array index where the key was found or -1 if the search failed. You can then invoke this helper function from both `hash_get` and `hash_lookup`, in order to implement the required functionality.

### 9 Load factor of the hash table

Implement a function with the following declaration:

```
float   hash_load_factor();
```

This function should return the load factor of the hash table, computed by dividing the number of used buckets by the number of available buckets. The return value should be a fractional number between 0 and 1 (inclusive). If the hash table is empty (e.g., right after invoking `hash_initialize()`), the load factor is 0. If the hash table is full (there are no buckets available for storing a new key), the load factor is 1.

### 10 5 Bonus Points: Collision chain statistics

Maintain additional statistics about the hash table. Implement a function that returns the maximum length of a collision chain. Also implement a function that returns the average length of a collision chain. Invoke these functions in Step 4 if the user requests option 1.

### 11 10 Bonus Points: Remove a key-value pair from the hash table

Implement a function to remove a key-value pair from the hash table. The function should receive a single parameter (the key to be removed), and it should return 1 if the key was removed successfully or 0 if the key was not present in the hash table.

**Hint:** This operation is not trivial with the hash table design described in this project. If you simply set the key as the null string, you may break the collision chain of some other key. One way to remove a key-value pair safely is to check if the next bucket (after the key you just removed) is empty. If it is not, delete that key-value pair and re-insert it by invoking `hash_put`. Repeat these operations down the collision chain, until you reach an empty bucket.

## String manipulation

The function described in Step 12 is used for extracting words from the lines of text read in Step 2. This function should be defined and implemented in `enee140_lookup.c`.

### 12 Extract words from lines of text

Implement a function with the following declaration:

```
int next_word(const char line[], char word[], int size);
```

This function receives three parameters: `line[]`, a C string ending in `'\n'`; `word[]`, a C string, consisting of characters that are not whitespace (spaces, tabs, newlines, etc.), to be extracted from `line[]`; and `size`, the maximum number of characters that can be copied into `word[]`. The function should copy characters one-by-one from `line[]` into `word[]`, stopping at whitespace, punctuation characters (e.g. `.`, `;`, `[`) or after writing `size` characters (remember that valid C strings must end in `'\0'`).

If the function is invoked again with the same `line[]` parameter, it should extract the next word from the line. In other words, the function should resume copying where the last copy stopped. If the function is invoked with a new `line[]` parameter, it should start copying from the beginning of the line.

The function should return 1 if some characters were copied and 0 if no characters were copied and the end of `line[]` was reached.

**Hint:** You may use the functions `isspace()` and `ispunct()` from `ctype.h` to determine if a character is whitespace.

## Testing

To test this program, you will need the `shakespeare.txt` file. You can find this file in the GRACE class public directory, at `public/projects/project2`.

You can test if your program produces the correct answers using a combination of UNIX commands. For example, you can find how many times the word "Prospero" appears as follows:

```
cat shakespeare.txt | tr '[:space:]' '\n' | tr '[:punct:]' '\n' |
    tr '[:upper:]' '[:lower:]' |
    grep -c '^prospero$'
146
```

## Project requirements

1. You must program in C and name your program files `enee140_hashtable.c`, `enee140_hashtable.h` and `enee140_lookup.c`. Templates for these programs are included at the end of this document (you do not have to use them, but they may provide some hints).
2. Your programs must compile on the GRACE UNIX machines using

   ```
   gcc enee140_hashtable.c enee140_lookup.c
   ```
3. Your programs must be readable to other programmers (e.g. the TAs).
4. You must first submit a partial implementation by **Friday, 19 April 2024 at 11:59 PM**, then a complete implementation by **Monday, 29 April 2024 at 11:59 PM**.

### Partial implementation

Your partial implementation must correctly implement the functions from your hash table's public API (Steps 5–9). Create a .zip archive containing `enee140_hashtable.h` and `enee140_hashtable.c`, then log into Elms, click on Gradescope in the course menu, and go to Project 2 (partial) to submit your work.

### Complete implementation

Your complete implementation must implement all the steps described above correctly. Create a .zip archive containing the programs you wrote, then log into Elms, click on Gradescope in the course menu, and go to Project 2 (complete) to submit your work.

## Grading criteria

| | |
|---|---|
| Correctness: | 80% |
| Good coding style and comments: | 20% |
| Late submission penalty: | -40% for the first 24 hours<br>-100% for more than 24 hours |
| Program that does not compile on GRACE: | -100% |
| Wrong file names:<br>(other than `enee140_lookup.tar.gz`, `enee140_hashtable.c`, `enee140_hashtable.h`, and `enee140_lookup.c`) | -100% |
| Bonus (collision chain statistics): | 5% |
| Bonus (key removal): | 10% |

## Program templates

You can start from the following templates (also available in the GRACE class public directory, at `public/projects/project2`).

Template for `enee140_hashtable.h`

```c
/*
 * enee140_hashtable.h
 *
 *  Header file for a hash table library.
 */

#ifndef ENEE140_HASHTABLE_H_
#define ENEE140_HASHTABLE_H_




/*
 * Define the parameters of a hashtable that will store English words.
 * Provision the capacity of the hashtable considering the fact that
 * Shakespeare's works include about 30,000 unique words and that the
 * lookup performance tends to decrease with a load factor > 0.7.
 */
#define NBUCKETS        50021   // prime number, for better hash uniformity
#define MAX_STRING      80




/* Function prototypes for the public hashtable API. */

void    hash_initialize();

float   hash_load_factor();

int     hash_put(char key[], int value);

int     hash_get(char key[]);

int     hash_lookup(char key[]);


#endif /* ENEE140_HASHTABLE_H_ */
```

Template for `enee140_hashtable.c`

```c
/*
 * enee140_hashtable.c
 *
 *  Implementation of the hash table operations.
 */

#include "enee140_hashtable.h"
#include <string.h>


/*
 * Define the bucket and hashtable data types.
 */
typedef struct _bucket {
    char key[MAX_STRING];
    int value;
} Bucket;

typedef Bucket Hashtable[NBUCKETS];



/*
 * Hashtable main storage
 */
static Hashtable my_hash_table;
static int used_buckets;



/*
 * Internal library functions
 */

unsigned
hash_function(char s[])
{
    unsigned hashval, i;

    for (hashval=0, i=0; s[i] != '\0'; i++) {
        hashval += s[i] + 31*hashval;
    }

    return hashval % NBUCKETS;
```

```
}


/*
 * Main hashtable API
 */

float
hash_load_factor()
{
}

/*
 * Initialize the hash table by clearing its content.
 */
void
hash_initialize()
{
}

/*
 * Insert a <key, value> pair in the hash table.
 * Return 1 if the insert was successful, and 0 if
 * the key could not be inserted because the hash
 * table if full.
 */
int
hash_put(char key[], int value)
{
}

/*
 * If key is stored in the hashtable, return the corresponding
 * value. Otherwise, return 0.
 */
int
hash_get(char key[])
{
}

/*
 * Returns 1 if key is stored in the hashtable, and 0 otherwise.
 */
int
hash_lookup(char key[])
```

```
{
}
```

Template for `enee140_lookup.c`

```c
/*
 * enee140_lookup.c
 *
 *  Read the input line-by-line, store the words in a
 *  hash table, and allow the user to look up several words.
 */

#include "enee140_hashtable.h"
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdlib.h>


#define MAX_LINE 1000


int next_word(const char line[], char word[], int size);

int prompt_and_check(int max_option);
void lowercase_string(char word[]);


int
main(int argc, char *argv[])
{
    char word[MAX_STRING], line[MAX_LINE];
    FILE *file;

    // Check command-line arguments
    if (argc < 2) {
        fprintf(stderr, "Usage: %s filename \n", argv[0]);
        return -1;
    }

    file = fopen(argv[1], "r");
    if (file == NULL) {
        printf ("Could not open the %s file.\n", argv[1]);
        exit (-1);
    }

    // Initialize the hashtable
    hash_initialize();
```

```
    // Read file line-by-line
    while (fgets(line, MAX_LINE, file) != NULL) {

        // Extract each word from line and add it to the hash table
        while (next_word(line, word, MAX_STRING)) {
        }
    }



    // Print menu and implement project functionality

    return 0;
}
```