**CHAPTER 1**

**INTRODUCTION**

* 1. **PROBLEM DEFINITION**

The problem that this program addresses is the need for a spell checker. A spell checker is a software tool that take a single word input and determine if its spelled correctly. Also suggests a correct word if the given word is wrong. This program aims to create a spell checker that can quickly and accurately identify correctly spelled words.

* 1. **OBJECTIVES**

These are the objectives for this mini-project:

1. Take user input in the form of a word and check it against a stored list of words to determine if it is spelled correctly.
2. Suggest few nearest words that the word might be if its spelled incorrectly.
3. The program should be able to process large amounts of words quickly and efficiently.
   1. **METHODOLOGY TO BE FOLLOWED**

* Use of hash tables with efficient hash function of the words to store a large number of words to quickly find if the user's input is in the list.
* Use of Levenshtein Distance algorithm to check for words that are similar to the user's input from the list.
* Use of functions , conditional statements etc to make the code efficient and user friendly.
  1. **EXPECTED OUTCOMES**

1. To determine if the entered word is spelled correctly.
2. To suggest correctly spelled word from the user input .
3. The program should also be able to handle large amounts of words and work efficienly.
   1. **HARDWARE AND SOFTWARE REQUIREMENTS**

**Hardware Requirements**:

* + 1. A Personal Computer/Laptop.
    2. Intel Pentium Processor or later.
    3. RAM 512 MB or more.
    4. Storage of 128GB or more.

**Software requirements**:

1. Windows 7 32-bit/64-bit or above.
2. GCC Compiler, text editor of choice(for ex VS-code, Notepad++),or any IDE(for ex Code::Blocks)
3. XXhash library needs to be installed for using xxhash64() function.
4. A file named “dict.txt” in the same folder as the c file which contains all the correct spelled word.

**CHAPTER 2**

**DATA STRUCTURES**

A data structure is used to store and organize the data. It is a specialized format used for processing, retrieving, organizing and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

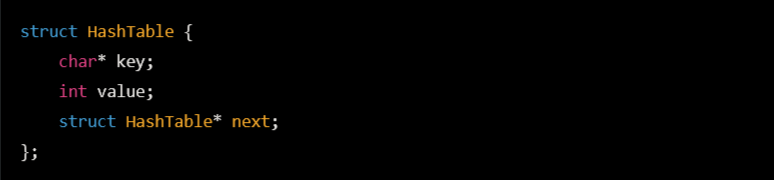
Data structure used in this program are hash tables, linked lists arrays and structures.

* 1. **HASH TABLE**

A hash table is a data structure that stores key-value pairs and uses a hash function to map the key to an index in an array, where the corresponding value can be found or stored. In this program, the keys are the words from the dictionary file and the values are the hash values of the corresponding words.

It is often implemented as an array of linked lists, where each element in the array (also known as a "bucket") represents a linked list of key-value pairs. The position of the key-value pair in the array is determined by a hash function, which takes the key as input and returns an index in the array.

First initialize a structure which contains key, value, and pointer to next in the linked list.When index determined from the hash function is same it leads to collision. To handle collisions, the hash table is implemented as an array of linked lists, where each element in the array (also known as a "bucket") represents a linked list of key-value pairs.



* "key": a pointer to a char array that stores the word
* "hash": a 64-bit hash value of the word, obtained using the XXH64 hash function
* "next": a pointer to the next struct in the linked list at the corresponding index in the hash table array

The next pointer is used to link the key-value pairs in the same bucket together in a linked list. Each key-value pair is represented by a struct HashTable and the next pointer points to the next HashTable struct in the linked list.

**2.1.1 ARRAY OF LINKEDLIST**

An array is a collection of elements that are stored in contiguous memory locations, and each element can be accessed by its index in the array. A linked list, on the other hand, is a collection of elements, called nodes, that are not stored in contiguous memory locations. Each node contains a data element and a pointer to the next node in the list.

In an array of linked lists, each element in the array represents a linked list of key-value pairs. The position of the key-value pair in the array is determined by a hash function, which takes the key as input and returns an index in the array.



**2.2 ARRAYS**

* An array is used to store a collection of variables of the same type. A specific element in an array is accessed by an index.
* The array elements are stored in contiguous memory locations. The first element is stored at the lowest address and the last element is stored at the highest address
  1. **STRUCTURES**
* A structure, also known as a struct, is a user-defined data type in C programming. It allows the programmer to group together different data types into a single composite data type. Structures are similar to records in other programming languages and provide a way to organize data in a logical and meaningful way.
* A struct is defined using the keyword struct, followed by the name of the struct, and a list of variables enclosed in curly braces {}. Each variable in the struct is called a member and has a name and a data type
  1. **LINKED LIST**
* A linked list is a data structure that consists of a sequence of elements, each containing a reference to the next element. Each element in the linked list is called a "node". In this project, the linked list is used to store the words in the dictionary, that have the same index. Each node in the linked list contains the word, its hash value, and a reference to the next element in the list.
* When a new word is inserted into the hash table, it is first hashed using the XXH64 hash function, and the resulting hash value is used to deter-mine the index of the array where the word should be inserted. If there is already a word with the same hash value at that index, the new word is inserted at the front of the linked list, becoming the new head of the list. In this way, the linked list acts as a bucket for storing all the words with the same hash value, allowing for efficient and fast lookups when checking the spelling of a word.

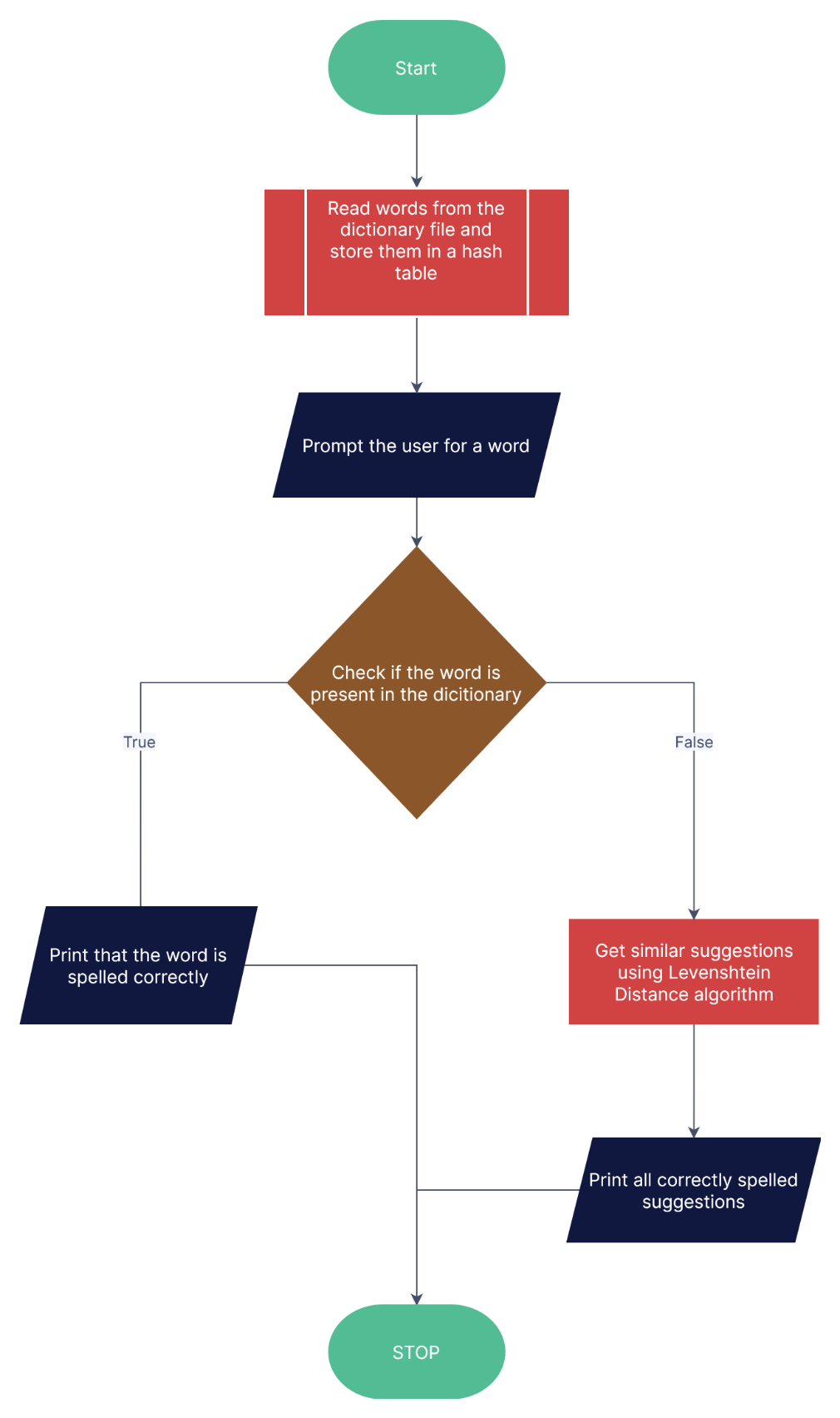
**CHAPTER 3**

**DESIGN**

* 1. **DESIGN GOALS**
* Implement an efficient spell checker using hash table and Levenshtein distance algorithm.
* Optimize performance and reduce collisions and suggestions
* Implement a user-friendly interface.
  1. **ALGORITHM / PSEUDOCODE**

1. Start
2. Read words from the dictionary file and load them to the hash table using xxhash library.
   1. Read the word from dicitionary
   2. Get the hash value from gethash function
   3. Find the index by doing hashvalue % size of hash table.
   4. Add it to the head of the linked list present in that index of the array
3. Prompt the user for a word
4. Check the user’s input against the words present in the hash table
   1. Get the hash value of the given word from gethash function
   2. Find the index from hashvalue % size of hash table.
   3. Compare the given word with all the words of the linked list present in that index of array.
   4. If found then the word is spelled correctly
5. If the user's input is not found in the hash table, use the Levenshtein Distance algorithm to check for similar words present in that linked list.
   1. Assume that similar words have the same hash value so are present in the same linked list or bucket.
   2. Iterate through the linked list an find upto 10 similar words using Levenshtein distance algorithm
   3. Print all the correctly spelled nearest words

**3.3. FLOWCHART**

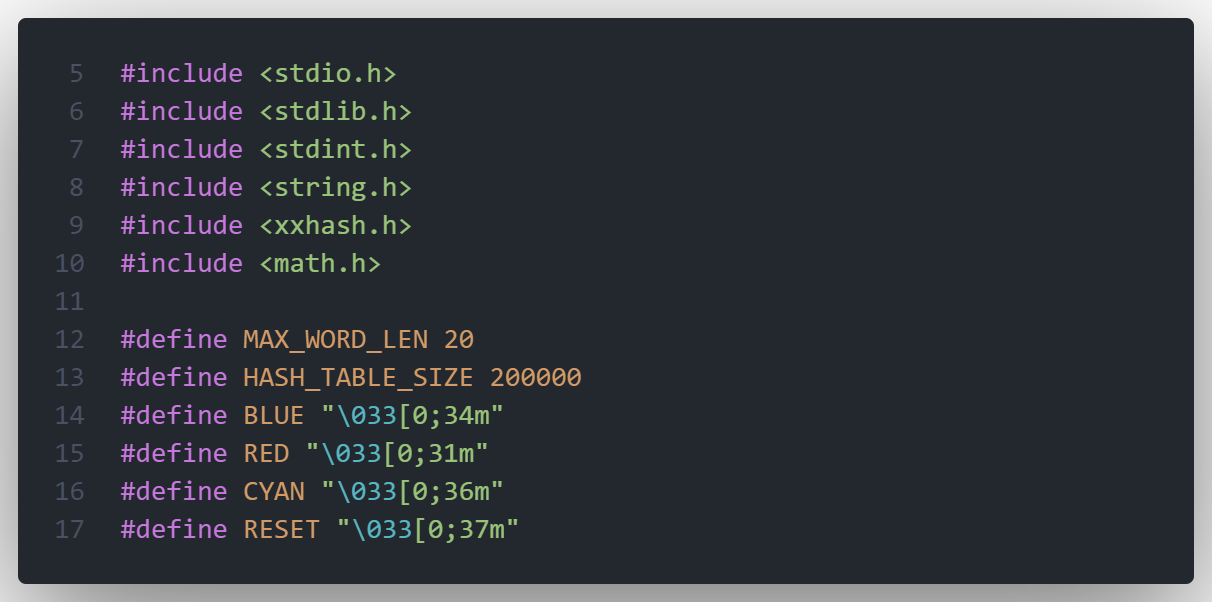


**Fig 3.1: Design flowchart**

**CHAPTER 4**

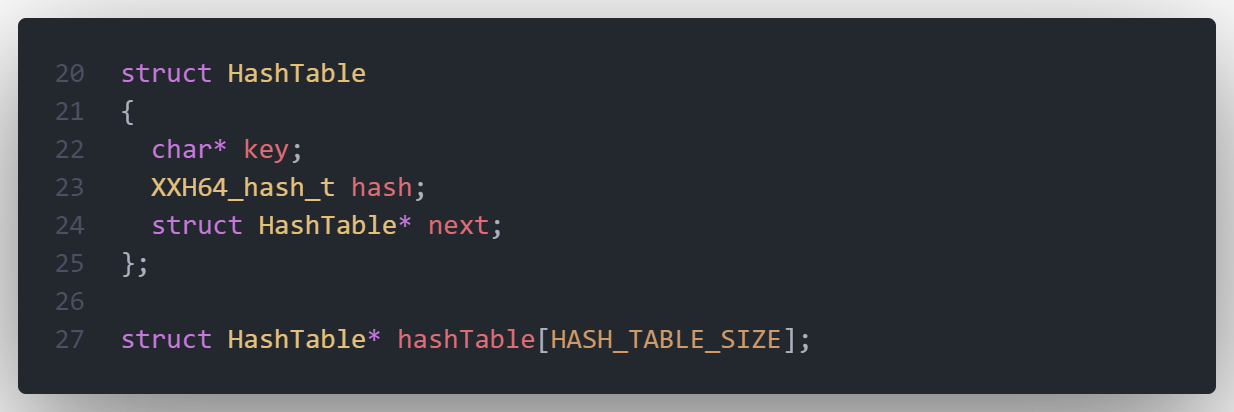
**IMPLEMENTATION**

* 1. **MODULE 1 FUNCTIONALITY**



* Include the necessary C libraries and defining several constants such as the maximum word length, the size of the hash table, and text colors for printing the result.
* Some header files which are necessary to this project are also included like –
  + xxhash.h a library for xxhash function–
    - XXH64() – takes input word, length of the word, and seed to change predictability. Returns 64bit hash value.
    - XXH64\_hash\_t – hash variable
  + math.h
    - fmin – returns minimum of two arguments
  + string.h
    - strcmp() – compares two string returns 0 if true
    - strcpy() – copy string from second argument and stores it to the first argument including null character.
    - strlen() – returns the length of the string.

* 1. **MODULE 2 FUNCTIONALITY**



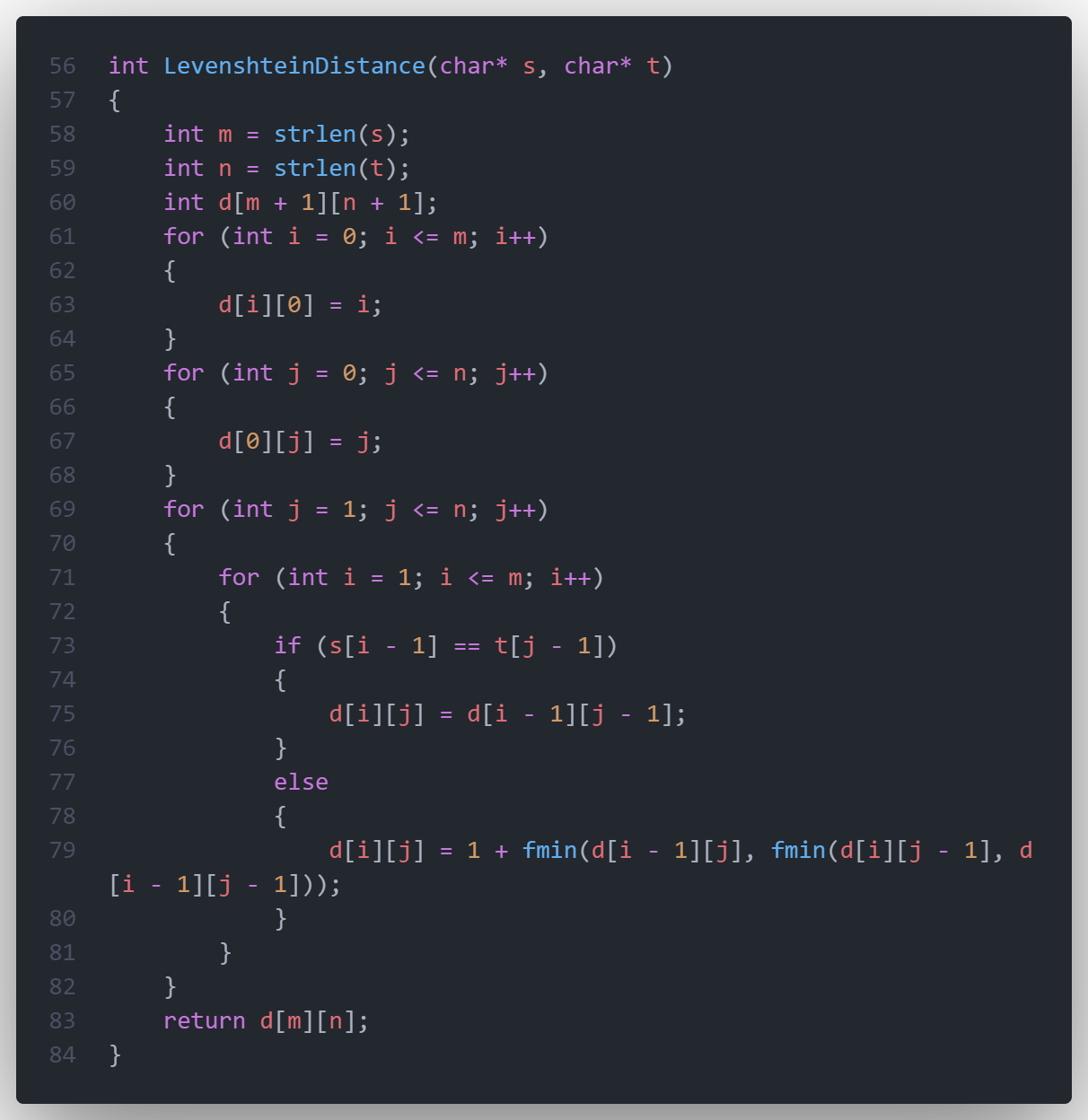
* The struct HashTable is defined to store the words and their hash values. The hash table is implemented as an array of pointers to structs of type HashTable.
  + Key is the word
  + hashstores the hash value of the word
  + next is the pointer pointing to the next variable.
  1. **MODULE 3 FUNCTIONALITY**



* The getHash() function is used to compute the hash value of a word using the very efficient and fast XXH64 library.
  + XXH64() takes the word, length of the word and seed to set the predictability as arguments.
  + Returns the hash value



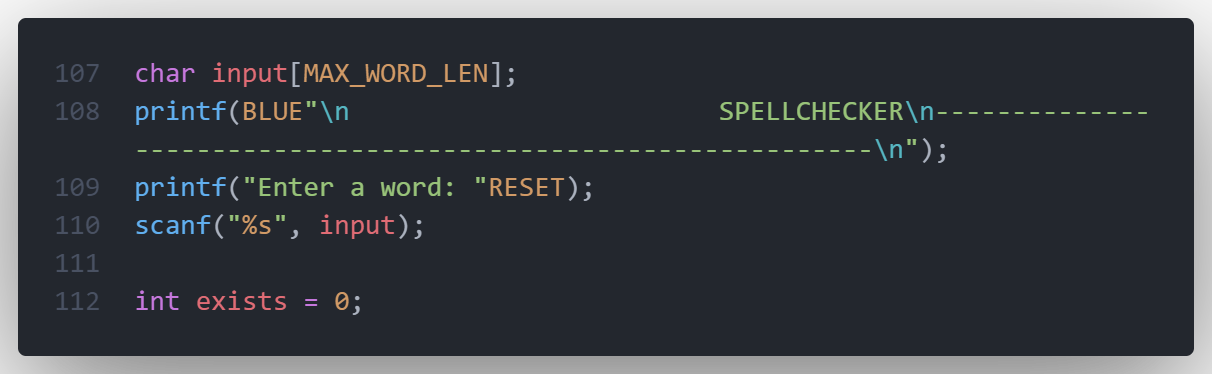
* The insertword() function is used to insert the word to the hash table.
* First get the hash value of the word from gethash() and store it in hash variable.
* Then the index of the hash table (i.e array of linked list) is determined by the modulus of hashvalue and size of hash table.
* Iterate through the linked list at the index and return if the word to be inserted is already present.
* If not create a new node i.e struch Hashtable\* entry , assign the key and hashvalue. Add it to the front of the linked list by pointing “next” value to the previous head/index and change the index to point to the newly created node.



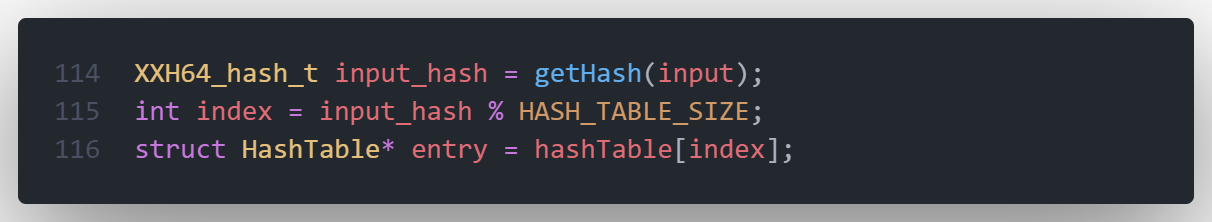
* The LevenshteinDistance() function is used to suggest correct spelling if the input word is misspelled by calculating the minimum number of single-character edits (insertions, deletions, or substitutions) required to change one word into the other using a matrix.
  1. **MODULE 4 FUNCTIONALITY (main function)**



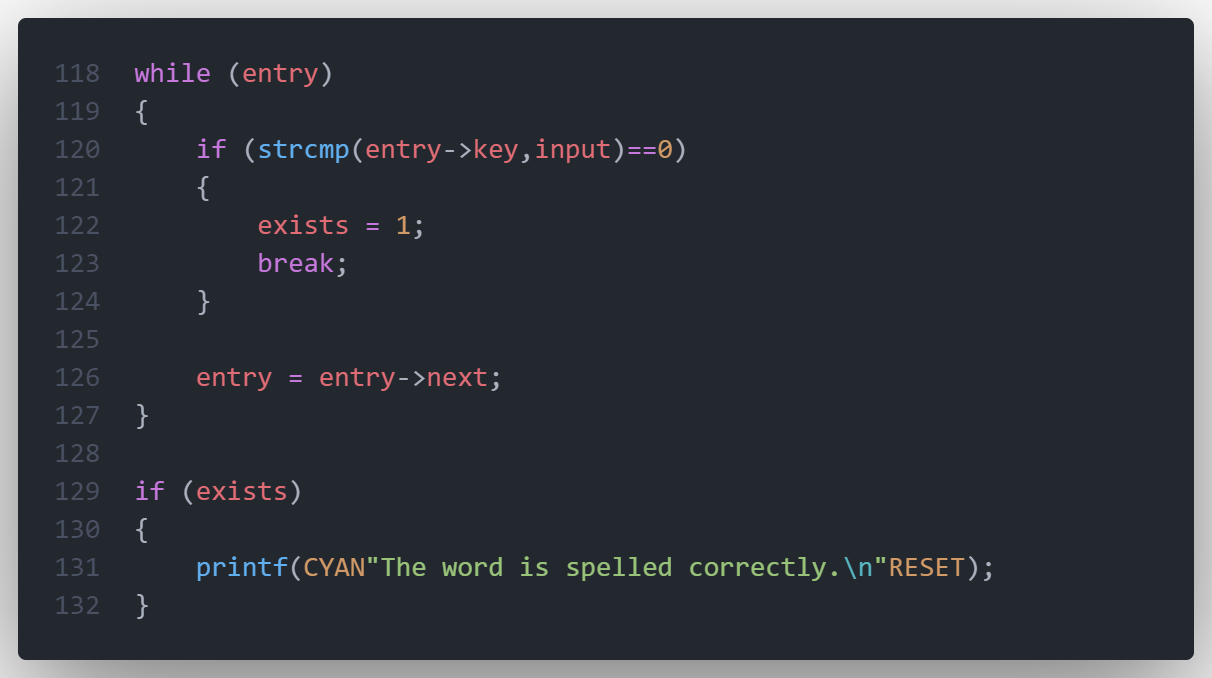
* Open the dict.txt file present in the same directory as the exe file using fopen() in read-only mode by setting the mode to “r”.
* Storing the word temporarily in buffer using fgets() and adding the word into the hashtable using insertword().
* After all the words are loaded into the hashtable the file descriptor is closed using fclose().



* Prompting the user for a word. Store it in “input” variable.



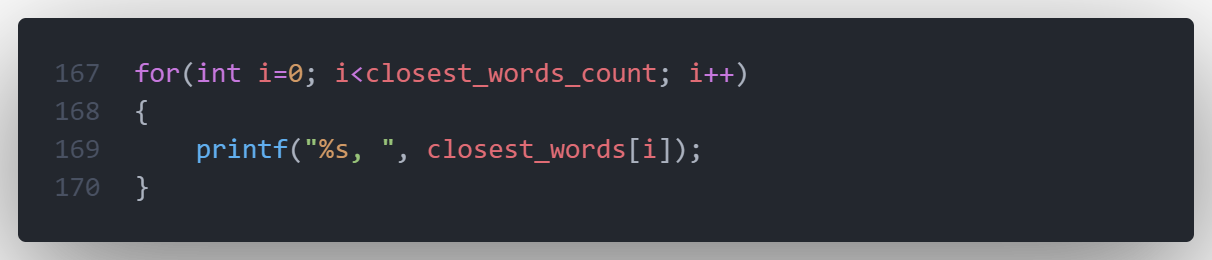
* First computing the hash value of the input word using gethash() function which uses the XXH64 library and returns XXH64\_hash\_t variable.
* Determine the index of the array by calculating the modulus of hashvalue and the hash table size.
* Accessing the linked list or bucket of the index i.e entry to find if the input word is spelled correctly or not.
* Hash tables time complexity is almost O(1). So its very fast. But because of collisions it may be less than O(1).



* Iterate through the linked list using a while loop and comparing key to the input word.
* If it exists break the loop or else go to the next node. The while loop terminates when entry is null
* If the word is found in the linked list then print it out that the word is spelled correctly.
* If the word is not found then print the nearest words using the distance algorithm.
* LevensheteinDistance() takes two words and returns distance integer between them by determining minimum number of changes necessary to get the word.
* First the minimum distance will be the max size of the word
* Similar words will have minimum distance.



* So we will iterate through the hashtable array and each bucket present in that index, compare the distance between the input word and present word and if the distance is similar to minimum distance then it is stored in a separate array.
* If a new word is found with distance smaller then the current minimum distance then the array is reset and the new word is stored.
* Its iterated to the bucket i.e linked list by entry = entry->next. Next is a pointer variable pointing to the next element.
* When while(entry) is false i.e entry is null means there are no more words in that bucket or index and appended to the next index in the hashtable array.

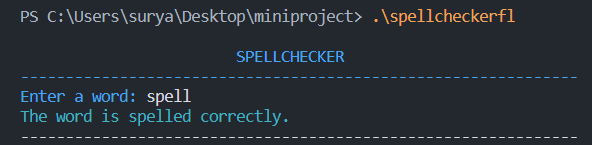


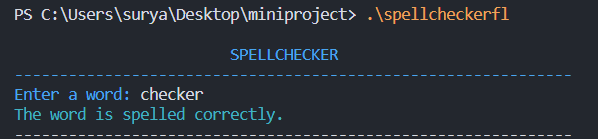
* At the end all the nearest word determined by Levenshtein distance algorithm is printed which were stored in the closest\_words array.
* Return 0 and exit the program

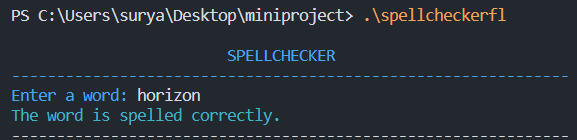
**CHAPTER 5**

**RESULTS**

**5.1 If the word is spelled correctly:**

****

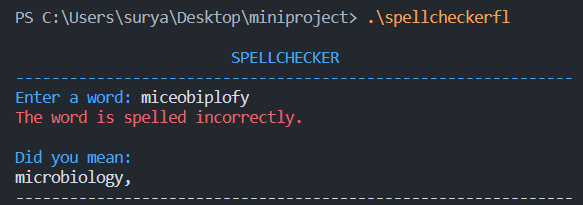
****

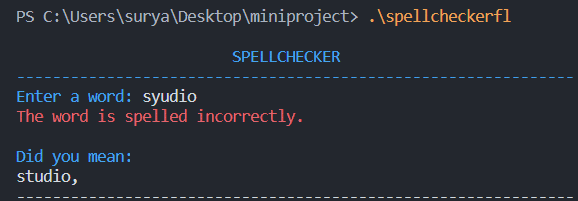
****

**Fig. 5.1: Output if the word is spelled correctly**

Here we can see that whenever a word is entered it checks through the all the predefined words and prints the word is spelled correctly. The dictionary file used in this consist of 194000 english words. So it can easily verify most of the commonly used words.

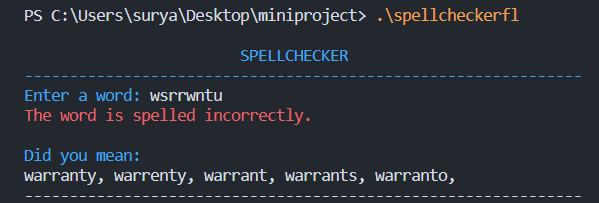
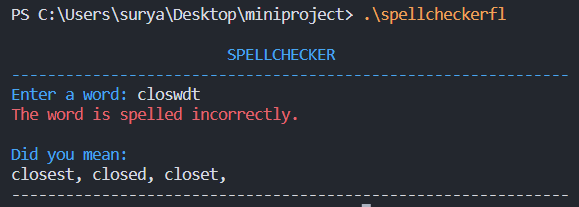
**5.2 If the word is spelled incorrectly:**

****



**Fig 5.2(a) : Suggesting a possible correct word**

Here we can see that when I misspelled a word it identified its incorrect and suggested a possible correct word. The program was able to process large amounts of words quickly and efficiently

**Fig 5.2(b) : Suggesting multiple possible words**

It also gives more then one possible suggestion if the input word is not specific enough . These words all have similar minimum Levenshtein distance from the input word.

**CHAPTER 6**

**CONCLUSION**

* The mini project has successfully accomplished the goals it had set out in the objectives and design sections of this report.
* Users can verify if their word is spelled correctly, can also get suggest correct words
* This can be used as a standalone project but also incorporated to more bigger projects where user inputs are taken.
* It is scalable like increasing the number of word in the dictionary, taking different type of input, like a whole text file instead of words
* In conclusion, this project met the goals and have scalable features in the future

**REFERENCES**

1. [JUST WORDS! Dictionaries and Word Lists (gwicks.net)](http://www.gwicks.net/dictionaries.htm): For the dictionary text file used in this project
2. [GitHub - Cyan4973/xxHash: Extremely fast non-cryptographic hash algorithm](https://github.com/Cyan4973/xxHash) : xxhash library used to get hash function for the timetable
3. [Understanding the Levenshtein Distance Equation for Beginners | by Ethan Nam | Medium](https://medium.com/@ethannam/understanding-the-levenshtein-distance-equation-for-beginners-c4285a5604f0) : Reference for Levenshtein distance algorithm
4. [Calculate the Levenshtein edit distance in C (lemoda.net)](https://www.lemoda.net/c/levenshtein/) : Levenshtein distance program in c