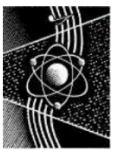

THE UNIVERSITY of York

Department of Electronics

C Programming Assignment Report

Data Structure and Algorithms(ELE00007I)

Examination Number: **Y3834764**

Content

I. Introduction

In regard to a program, both execution and efficiency are important factors to justify the quality of this program. However, in order to finish off a program quickly, programmers tend to focus of the execution, making a program developing faster but slower when it actually runs. Correctly chosen appropriate data structure will help both in code formatting and overall program efficiency.

II. Program Overview

The main feature of this program is a demonstration of a text prediction function, which is deployed on modern smart phone operation system. Program will read the input from the user, then search through a list of words and display all the words that start with that user input accordingly, Multiple data structures and searching and sorting algorithm are implemented in this particular program. In order to maximize program efficiency, appropriate structure and algorithm is selected for different part of this program. These will be explained in the next section. Following diagram shows the general structure of this program.

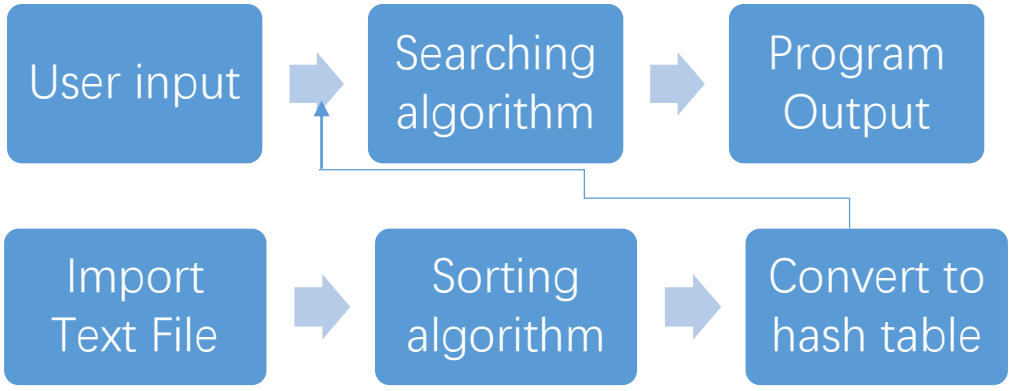

Fig 2.1 – Overall structure of the program.

III. Data Structure Selection

1. List

The use of List data structure is quite simple. It will import the text file into the whole program, acting like a dictionary. So the rest of the program can directly use the content in the list without reading the file again. This structure contains the capacity of the list, the number of entities in the list and an array that contains the words in it.

2. Hash Element and Hash Table

After sorting is completed, each single word will be stored in a hash element. The structure hash element has the content of the word, the previous and the next entity of

this hash element. Each hash element has a hash value which is defined by its first letter. For instance, 'word' has the first letter of 'w' whose ASCII code is 119. Then using the hashFunction it will return the value of 23 as its hash value. Elements that share the same hash value will be inserted into the same bucket in the hash table. Every time a new element is inserted, it will link its previous and next entity for further use.

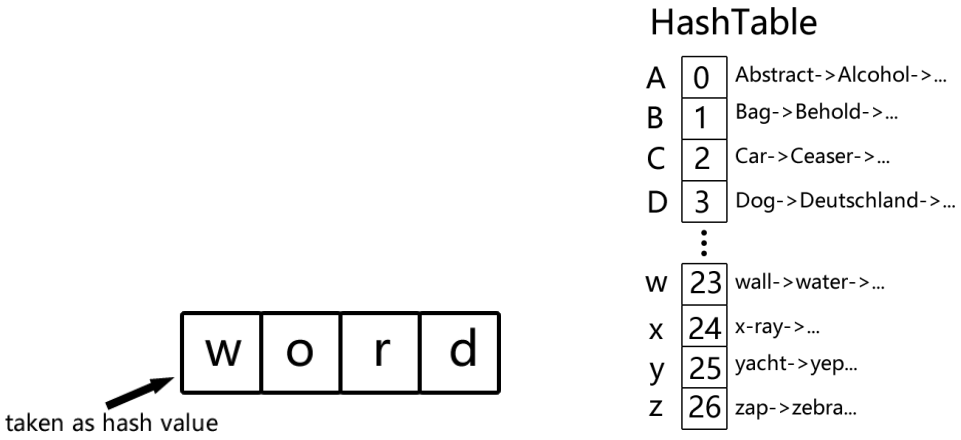

Fig 3.2-1 Demonstration of a single hash element and hash table

3. Stack

Once searching in the hash table is finished, all matched elements will be stacked into a stack one by one. The stack contains the value of its length (i.e. the maximum number of words that it can contains), the current pointer to the next empty slot, and the array of words. The reason I'm using a separate structure instead of printing out the element directly is to avoid walking through a single bucket of the hash table, which, in regard of the code, is not the best choice. Meanwhile, providing a way to mark all the words in the stack by using the array's index, which is used in word selection function later on.

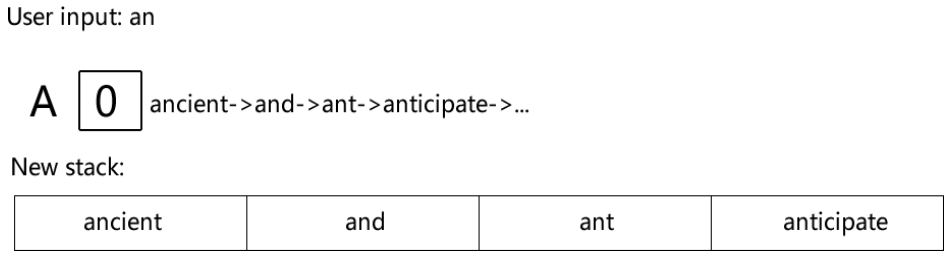

Fig 3.3-1 Relationship between user input, hash table bucket and stack.

IV. Algorithm Selection

1. Sorting

Quick sort is implemented and is used only once right after the data file is read by the program. Considering there are 25,000+ entities in the file, which are not sorted at all, this algorithm has an average-case performance, $O(n \log(n))$, compared with bubble sort's $O(n^2)$ (in the worst case) & insertion sort's $O(n^2)$ (in average case)

2. Binary Search & Linear Search

Searching through a sorted list, in this case, a bucket of a hash table, linear search has $O(n)$ average case performance while binary search has $O(\log(n))$. Surely binary search algorithm is more efficient than linear search. I modified the structure of my hash element by adding a 'prev' element, which is linked to the previous element of the current one in a given bucket. The direct searching result of binary search is not guaranteed the very first matched entity in the bucket, but very close to it.

After searching is done, using the linked previous entity of each hash element, we can find out the first element that matches the user input. Marking it as a start point, adding all matched entities into the stack one by one until there's no more matched entity.

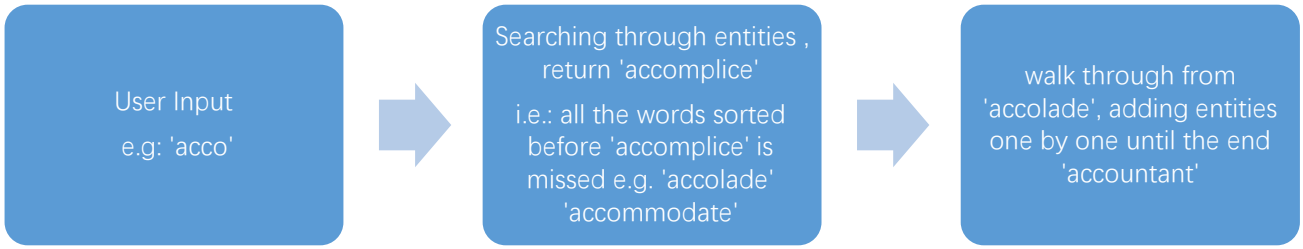

Fig 4.2-1 Improved process of binary search

3. String Comparison

One of the most algorithm of this program is to compare two given words, returning the result of there are identical or not. If not, identify which one should be placed first. In C's string.h, a given function called strcmp (stands for string comparison) will do the requirement above. However, in this particular program, we only need to compare part of the string as the program is 'predicting' user's input. So I implemented an individual function called strparcmp (stands for string partial comparison) which will only compare part of the letters. The 'part' is decided by the shortest word in comparison. (e.g.: user input: bak, target word: balcony, this function will only compare

the first three letters of these two words.) So that, for instance, ‘bag’ and ‘baggage’ will be considered as ‘matched’ entities, which complete the ‘predict’ functionality.

V. Testing

To simplify and conduct the testing completely, I created an individual text file which contains several words:

```
Heaven
Terrain
industry
Inferno
Lambda
torque
silver
gold
global
elite
Guardian
```

1. List implementation

This function takes the text file name as a parameter, return a list structure. Then the list will be displayed by a list display function.
It should have the following output (exactly the same as what it shows in the text file.)

```
Heaven
Terrain
industry
Inferno
Lambda
torque
silver
gold
global
elite
Guardian
```

2. Sorting

This function takes a list structure as parameter, using quick sort algorithm, it should have an expected output like this:

```
Guardian Heaven Inferno Lambda Terrain elite global gold industry
silver torque
```

3. List and hash table conversion

This function will convert the list to hash table, the expected output should be a categorized list of words.

```
G: Guardian
H: Heaven
I: Inferno
L: Lambda
T: Terrain
e: elite
g: global gold
i: industry
s: silver
t: torque
```

4. String partial comparison

This function will take two strings as parameter, return the result of the comparison, 0 for matched, -1 for the first string is sorted before the second string, 1 for opposite.

Expected output:

Parameter 1	**Parameter 2**	**Return**
tor	torque	0
glo	gold	-1
g	silver	-1
ind	Inferno	1

5. Final Test

At this point, I substitute the real text file instead of the temporary file. As user input a string, expected output should be all the words in the file that starts with user's input. When there's no result that matched user's input, it will ask user to input again.

User input: acco
Output: accolade accommodate accompaniment accompanist accompany … accost account accountant

User input: Bon
Output: Bonaparte Bonaventure Boniface Bonn Bonneville Bonnie

User input: collect
Output: collect collectible collector

User input: klo
Output: No result found. Please try again.

VI. Conclusion

The main object of this assignment is to build a text prediction system that implemented in modern smartphone. Clearly what actually implemented in the smartphone is way more complicated than just comparing string. It also contains user's habit, word usage and et cetera. The way people communicate nowadays requires rapid response. Of course, on the aspect of hardware, is fast enough. However, on the software aspect, you don't want this system to execute for seconds just to show which word the user is going to input. That could frustrate people.

Completing sorting and automatic selecting in a blink of an eye requires the implementation of suitable and efficient data structure and algorithms. Not only this tiny functionality, but also massive program, like games, require efficient way of program implementation as well. You don't want the game become frozen just because it's trying to calculate how many damage you have taken.

On the contrary, right tool for the right object is important as well. Using quick sort for just 10 words is just a waste of hardware. Therefore, except functionality, which defines what the program should do, suitable data structure, efficient algorithm, define how the program is going to do what it should do. Only by combining those three elements can we build a program to perfect as close as possible.