Final Project

Data Structures

Project Name: “TrieHards”

Peter Nelson Subrata, Philipus Adriel Tandra, William Jonathan Mulyadi, Wilbert Wirawan Ichwan

Student ID: 2502023562

Class: L1CC

HASH TABLE DICTIONARY

How to run the program;

To run the program just run .cbproj file on C++ builder. Then click on the run with or without debugging in the top pane of the IDE.

Another alternative is to run the .exe file that has been provided in the github.

Function of the program;

The program is a Dictionary implementing a hash-table. The program is supposed to do a bunch of things. Some things that it can do are displaying dictionary data onto the screen in a list box. When the hash table dictionary is run it takes data from the dictionary sheet csv file and then attaches it into arrays that will also insert the values into a hash table taking the entire string as the key and using the hash function to compute the address.

The design of the program;

The hash table starts off as a hash table entry class which contains all of the extra information that we need to store such as the type of the word and the definition of the word. We create another hash map table to store these hash table entries. The hash table entries are made when the constructor to the hash map table is called. The index of the hash map entry is about as big as the number used to modulo divide the hash key.

Inserting;

When inserting data into the hash table there are three parameters that it will take. The word, the type and the definition. The word will be used as the key for the hash function to gain its address. We use linear probing to comply to the collision problem. At the address create a new hash table entry object to replace the old one which has been deleted. We will calculate the runtime of the program of this function as well and write it towards the edit box at the top left corner.

Searching;

Searching is very simple, find the string inputted, send it to a hash function, see if the node present currently is the same as the word or not (if not then proceed to do linear probing). If the word is found then return the index of the word. We will also be calculating runtime for this.

Update;

Updating requires the code to search through the hash table again using a hash function and the hash key. Once the word is found there are two things that can happen depending on the word in the second parameter. If the second parameter asks for “def” return the definition of the word at that hash address. If it is not def then the code will redefine the word into whatever you typed into the update box. Shown in the update element window.

Deletion;

When deleting something from the hash table, we first run the search method to find the address of the key and once found we simply delete the address and make a new one set to NULL.

Sorting;

In this hash map table program, it is proven to be impossible to sort through a hash table by itself. So instead we made a separate data structure that can store values that can easily be sorted using. We used a standard array and for the algorithm we chose to use selection sort. If sorting is clicked then the sorting algorithm runs and the words are sorted alphabetically.

TRIE DICTIONARY

How to run the program;

The program runs very similarly to the previous implementation using a hash table. When the program runs we can also use a bunch of functions such as inserting, updating, deleting and sorting. In the trie implementation we can also do a prefix search which allows for something that is called auto completion of a word when the incomplete strings can be matched with all of the values in the trie and displayed in a drop-down list.

The design of the program;

A trie is just a tree which stores a sequence that can be arranged to form data. For example, a trie made up of strings can be consisted of nodes as long as the alphabet with each node containing one letter from the alphabet. Once you connect the nodes together and reach the leaf node, the word will be found and the search is complete. To make a trie we designate a pointer to the address of the node in the trie and fill it with the entire alphabet length as the width of the tree. In our case, we made 123 nodes for every node in the trie.

Inserting;

When we want to insert into a trie we will call it from the head of the trie and we will first check the first letter of the string that was included to find the first node that we need to get to. When we reach that node we continue to use the second letter of the string as the address to find the second node. We continue to do this until the word length has been exceeded and afterwards whatever node we land on will be turned into a leaf node denoting that there exists a word on this node.

Searching;

When searching through a trie, all we have to do is start from the head then for every character in the string we will send the node to the next node based on the address that the character of that string will return. It will do this until the length of the string runs out. If it runs out then it will check for whether or not it is a leaf node. If it is a leaf node then the word is found. If not then the word does not exist.

Update;

Updating requires us to search through the trie to find the word. Updating the definition is only possible at the leaf of a node and it will replace the old definition with whatever new definition has been inputted.

Deletion;

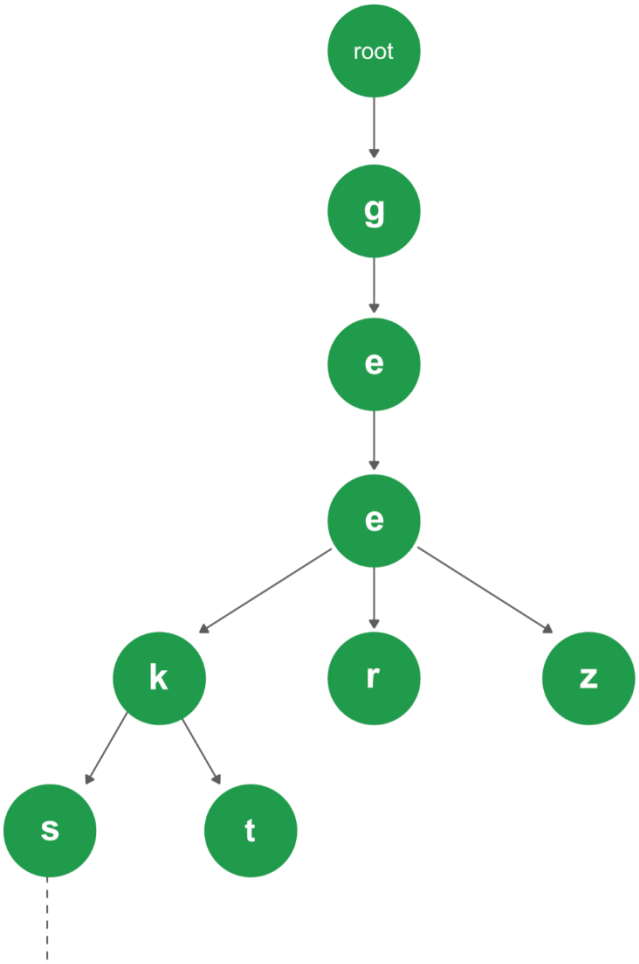
To delete a node in the trie, it will be recursively done. It will recursively call the delete function using the current node as the first parameter and the remainder string without the index 0 letter. Continuing to do this until the leaf node is found will result in the deletion of the word.

Sorting;

Sorting in a trie is very simple, all you have to do is run a pre-order traversal and then it will recursively call from the start of the left most node continuing down and it will print all of the values once the leaf has been reached. Doing this recursively will print out all of the words in the trie alphabetically.

Prefix Search;

Prefix search does something similar to the preorder sort but it starts from a certain string and prints out all of the words in that starting string.


root
g
e
e
k
r
z
s
t