Project 1 – Tries (God, I’m trying!)

CSE 503 – Summer 2024

Richard Douglas

Project 1 – Tries (God, I’m trying!)

Assignment Description

Part I (45 points):

Part II (45 points):

Part III (10 points):

Logic Employed

Overview

The trie node

The trie class

Constructors, destructors, and data members

Insert

Search

Remove

Insertion from file

Levenstein distance

Find nearest words

I/O

Screenshots

Conclusions

Assignment Description

Project Details:

This project will be three parts. Part I is to construct a Trie using a dictionary file provided. Part II is to implement a command-line search auto complete interface. Please note that you need to implement your own version of Trie. Part III is to implement a recommendation search. You cannot use existing C++ library Trie or use an implementation online. Points will be taken off otherwise.

Part I (45 points):

Dictionary.txt is provided to you to construct the Trie. Each line contains a valid search query. Your task is to insert these queries into your Trie. Implement trie classes for part 1.

Part II (45 points):

Using the Trie class completed in Part I, create a C++ program that takes an user input and output auto completion options. The interface should be similar to the following:

$> Please type search queries:
$> binary sea
$> Your options are:
$> binary search
$> binary search tree
$> binary search tree java

Implement a search.cpp file to complete part II

Part III (10 points):

Implement a recommendation search. When you search a term that is not in the Trie, recommend the top 3 most similar entries in the Trie:

$> Please type search queries:
$> hopr
$> Do you mean:
$> hope
$> hot
$> hyte

Important! : Please DO NOT change file names or add files on your own. I will compile your code using these file names. If your code does not compile you will get an automatic 0 point on this project.

Logic Employed

Overview

Alrighty, yall... this one deserves a breakdown. I will explain each of the following pieces of the puzzle in turn: The trie node class, the trie class and its various members, the insertion from file function, Levenstein distance function, find nearest words function.

The trie node

This node is similar to those we have created for linked lists and binary search trees. In this implementation, I simply used the unordered_map class to create pairs containing the value and a pointer to the nodes child nodes. There is also a boolean data member which acts as a flag to signify whether the node represents the end of a word. What a clever solution to storing search terms this way!

The trie class

Constructors, destructors, and data members

This implementation uses fairly basic constructors and destructors. The destructor does call a custom function to delete all of the nodes. This deletion uses recursion to reach the leaf nodes, then deletes that node, and returns back to each ancestor in turn and deleting that too.

Insert

The insertion process is one of iterating through each character of a word to check whether that letter is in the child node using the built in find method for unordered lists. If the character does not exist in the nodes children, we create a child node, and keep iterating through the characters and stepping through the nodes to check them for the character. Finally, once we iterate through all of the characters of the insertion term, we set the boolean value to mark the end of the word as true!

Search

To search for all of the possible results from a search term is tricky. We traverse the tree similarly to insertion – by iterating through all of the characters in the term and checking successive child nodes. However, if we reach a character which is not contained in any child nodes, we return null. If we are able to iterate through all of the characters, then clearly we have found the search term. However, Our objective was to find all of the results subsequent to the search term. To do this, it is helpful to implement a helper function to return all of the words that are ancestors of the search terms found node. First, we initialize a vector (remember, vectors can be dynamically resized, making this convenient) and pass it, along with the current node and the search term itself.

The helper function recursively checks every node of the subtree, starting at the search term's node, to see whether the flag labeling that node as the end of a word is set to true. If the flag is set to true, the result is appended to our results vector. The results vector is modified by reference, nullifying the need for a return value from this helper function.

Remove

The public member function of this class simply calls a private member function. This is to encapsulate- we need to pass some set values into this private function, which shouldn't be passed by the public member function, like the root memory address and current depth in the trie.

Basically, what happens in the private remove function is a recursive call is made to search for an end of word node containing the word searched to be removed from the trie. If there is no end of word node indicating that word is in the trie, the function simply returns false. Otherwise, the node's end of word flag can be set to false, removing that word from the trie.... If the node has no children, it can be deleted.

Insertion from file

We use ifstream to create a file object, using the "Dictionaries.txt" file, as per the instructions. If it does not successfully open, we print out an error message and return from the function. Otherwise, we iterate through each line of the file and insert the string value from that line into the trie, using the insertion function.

Levenstein distance

This algorithm is a bit tricky. The gist of it is that we create a M x N dimensional matrix, where M is the length of word 1 and N is the length of word 2. We then compare the two words at each of their indices for equality. If they are equal, we add zero to the score, otherwise, we add 1. The value in the matrix at index [M][N] is the levenshtein distance. This is used in this project to determine the nearest results if a search term is not found in the trie.

Find nearest words

In this function, we instantiate a vector, thin populate it with all of the words from the trie (by calling the "collectAllWords" function). Then we instantiate a vector of pairs of types string and int to hold all of the distances. We calculate the levenstein distance from the search term for each of the words in the trie, then add a pair of that word, and the distance into our vector of pairs. Next, we sort the distances vector, ascending by distance. Finally, we instantiate one more vector to hold the top three string values in our sorted distance vector, and return that.

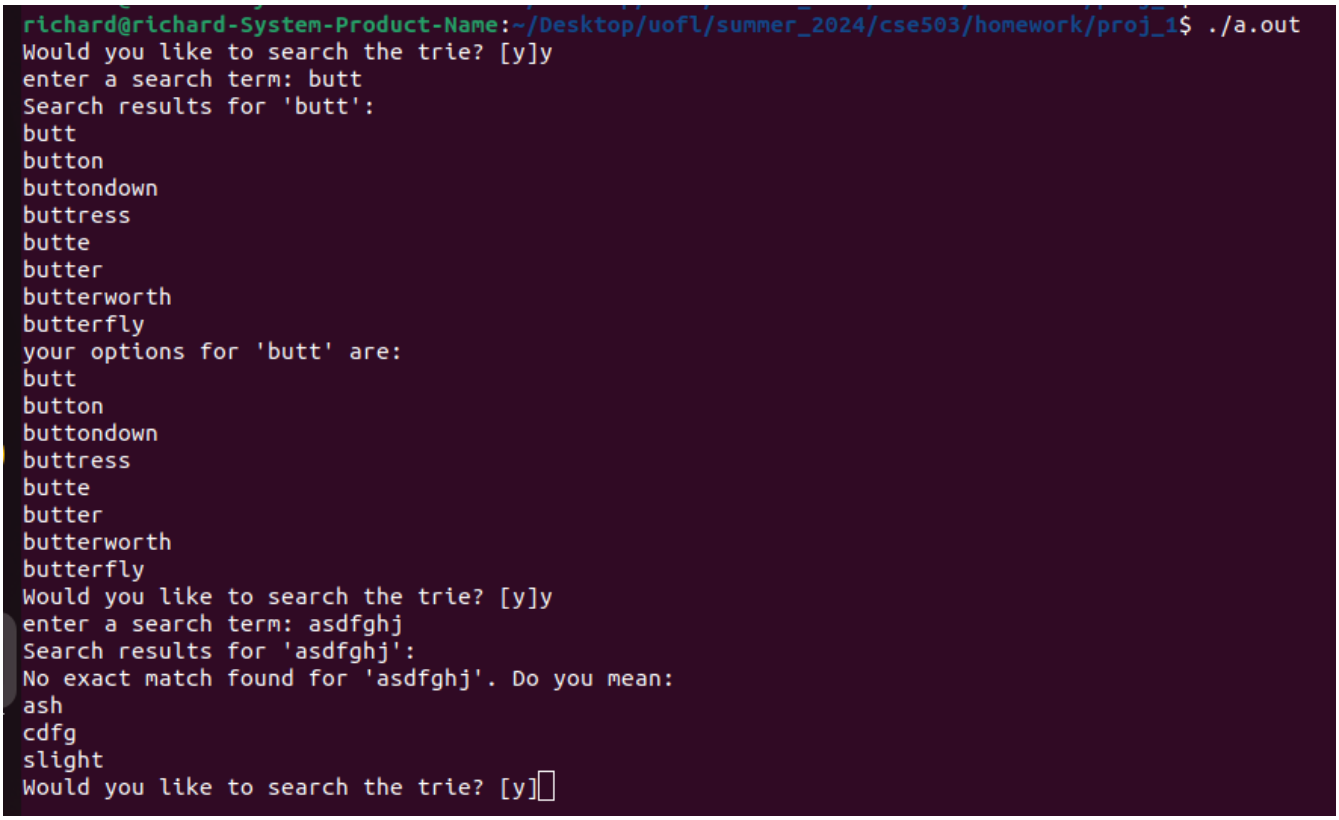
I/O

To populate the trie, we were provided a large text file called Dictionary.txt.

In the main function of the C++ file, I created an infinite loop where the user is solicited to search the trie, and then the results are printed for them. This is repeated until the user enters something other than ‘y’ or ‘Y’ when asked if they want to search the trie.

Screenshots

The following is the output run in my terminal.



Conclusions

This exercise was difficult but impressively demonstrates the power of trees, specifically tries. It also was great practice implementing recursion. I had fun... but my eyes are tired now and I must away.