Trie Data Structure

Michael Sayers, Vanessa Melgar, Linda Tran, Shina Adewumi

University of Rhode Island

CSC 212 - 0001: Data Structures and Abstractions

Professor Jonathan Schrader

December 5, 2022

INTRODUCTION

The purpose of this project is to implement the trie data structure to a scrabble wordfinder written in c++ programming language. The original scrabble board game consists of forming words from lettered titles. Furthermore, the trie (keyword tree) data structure consists of storing strings that can be visualized. The trie is also known to be the prefix tree. The word “trie” comes from the word “retrieval”. When moving down to the tree starting from the root, each node will illustrate a portion of that word.

Some important key aspect of a trie for a set of strings is that the root node is always the null node. Second, each node can have a maximum of 26 children ranging from ‘A’ to ‘Z’. Each node except the root can store a letter of the alphabet. The basic components of the trie that is implemented are the insert, search, and deletion of a node. It is important to understand that the insertion works by having every character of the input key inserted as an individual node in the tree and the key character is the index to the array of the children (Geeks, 2022). The removal implementation starts by checking the trie for the word, then deletes each node as long as the node is not necessary as the prefix of another key (Geeks, 2022). The search works by comparing the characters as it moves down the trie tree. The search will terminate if it’s at the end of the string or lacks a key (Geeks, 2022).

There are pros and cons with the trie data structure, for example, some of the strengths are sometimes space-efficient. Typically, tries are space-inefficient and compared to storing strings in a set (Cake Labs, 2022). Despite this, the data structure is useful when it comes to prefix queries, since it’s able to store lots of words that begin with a similar prefix (Cake Labs, 2022).

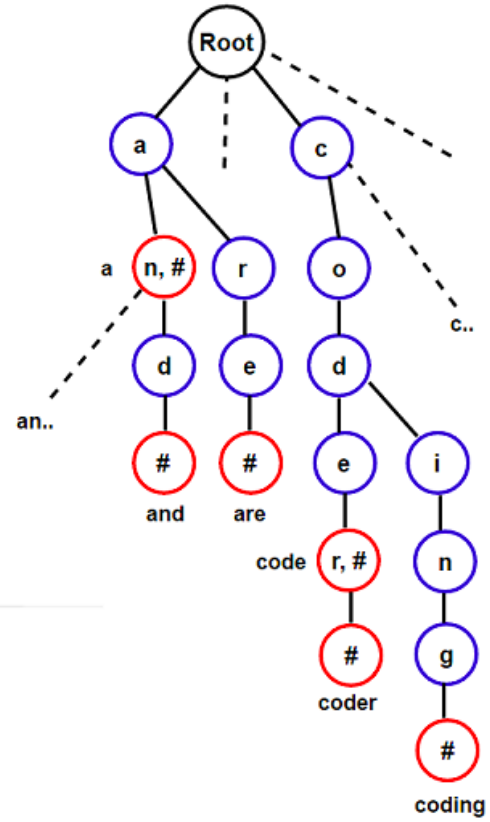

Figure 1. Each key is shown in the trie as a path from an internal node or leaf, *credit*.

METHODS

We started with the Trie class and a Node class, to begin with. Inside the Node class, we used a simple data structure of the “children”, “key”, “numChildren”, count, and “isWord” flag. We added the “numChildren” towards the middle of development to support the remove function when we would want to remove a word. With the Node class built, we could start with the Trie class. The only data member we needed within the Trie class was root which was a pointer to the top node. We choose to include the following functions for our Trie class: recursiveInsert, insert, search, ascend, descend, remove and destroy. The ascend and descend were chosen to replace the postorder, inorder, and preorder. The decision was made since multiple children simply stating which direction you were listing the children was sufficient to list all the nodes of the Trie.

The trie has two implementations of the insert function, an iterative approach and a recursive one. Both implementations are void, seeing as they are only meant to update the trie, not return any value. Each implementation has a public function that takes a string of the word being inserted as an argument and a private function of the same name that takes the same string along with a pointer to the root node of the trie. The public function calls its private counterpart.

The basic algorithm the iterative insert follows is to begin by creating a temporary node that points to the root and to iterate through each character of the string, inserting the letters that are not yet in the trie and updating the number of children each node has. If the word has not already been inserted into the trie, the function will add a new child to the second to last letters' children to represent the full word, set its "isWord" flag to true, and increment the "count" counter. If it had been inserted already, the function will only increment the count representing the number of times that particular word appears in the trie.

The recursive insert function is very similar to the iterative approach explained above in an algorithmic sense, except that the function calls itself in order to insert each new node to the trie. This function also takes an extra argument that points to the current location of the string it's inserting. The decision to make both methods came about when attempting to compare the efficiency of the trie program with different solutions put in place.

The remove function removes the given key from the trie entirely. It has both a public function and a private function of the same name and boolean type. The public function takes the string representing the word in question as an argument. It then calls the private function with the same string along with a pointer to the root node of the trie. Essentially, this function will return false when the trie is empty, the key given is not present in the trie or the key's final node is not

marked a word. In any other case, the function will return true. In other words, each function returns true to represent a successful removal of a word or false to represent otherwise.

After making sure the trie isn't empty, the removal algorithm starts by creating a temporary vector of pointers of a size one greater than the length of the key's string. The first node would point to the root and each subsequent node would point to the next letter in the key. As long as the key is present and marked as a word, the final node representing the completion of the word is deleted from the trie and the number of children of the second to the last node is updated as necessary. The function will then loop through the word backward. If a node does not have children, it is not needed in the trie and is also deleted. Otherwise, removal was successful and the function outputs true.

The "ascend" runs through the tree starting from the head of the tree and prints out each child. A temporary node initially stores the root of the tree and then stores the current child with each iteration and prints it out. We used this in place of the other tree traversals inorder, preorder, and post-order.

The "descend" function, like the, ascend function, runs through the tree and prints out the children, but in this case, the tail of the linked list is the starting point. It prints out all the children up until the head of the linked list. It can be considered a reverse of the ascend function.

The "search" function is to go through the entire trie tree and it will print out the count of the word.

The destroy function empties the Trie with the exception of the root node. It walks Trie through visiting each child going to the bottom of the Trie before doing anything else. It uses

pop_back to remove the word nodes and sets each letter child to nullptr. It destroys each child and then moves either the next child at that level, and then up the Trie. The destroy function uses recursion to ensure that all the children are visited and destroyed before destroying the parent.

IMPLEMENTATION

Our research gave us a few choices to choose from to showcase the data structure, Trie. One use of the Trie is for autocomplete to speed up the search of the word and to be able to display all possible words. (Singh, 2020) Another use of the Trie is to use it in networking within the router to look up network maps. (Singh, 2020) To implement our Trie data structure, we decided to use the search function to validate Scrabble words. We would query the user for a word. The word would be searched on in the Trie. Which would benefit from the Trie data structure since it would have a time complexity which is based on the length of the word. (Agarwal, 2021) If the word was found on the Trie then the word was a scrabble word. In addition to the Scrabble verification, we created a visualization to be able to output a dot file of the data structure. We choose to use the memory location to identify each node. We would list out each node with a label. Then output the parent-child relationship. With the insert function and recursive insert functions, we were able to compare the times for each function and graph it. This led to the conclusion that the insert function for the word lists that we were using and lists up to 1 million, would be better served with the iteration insert vice the recursive insert function.

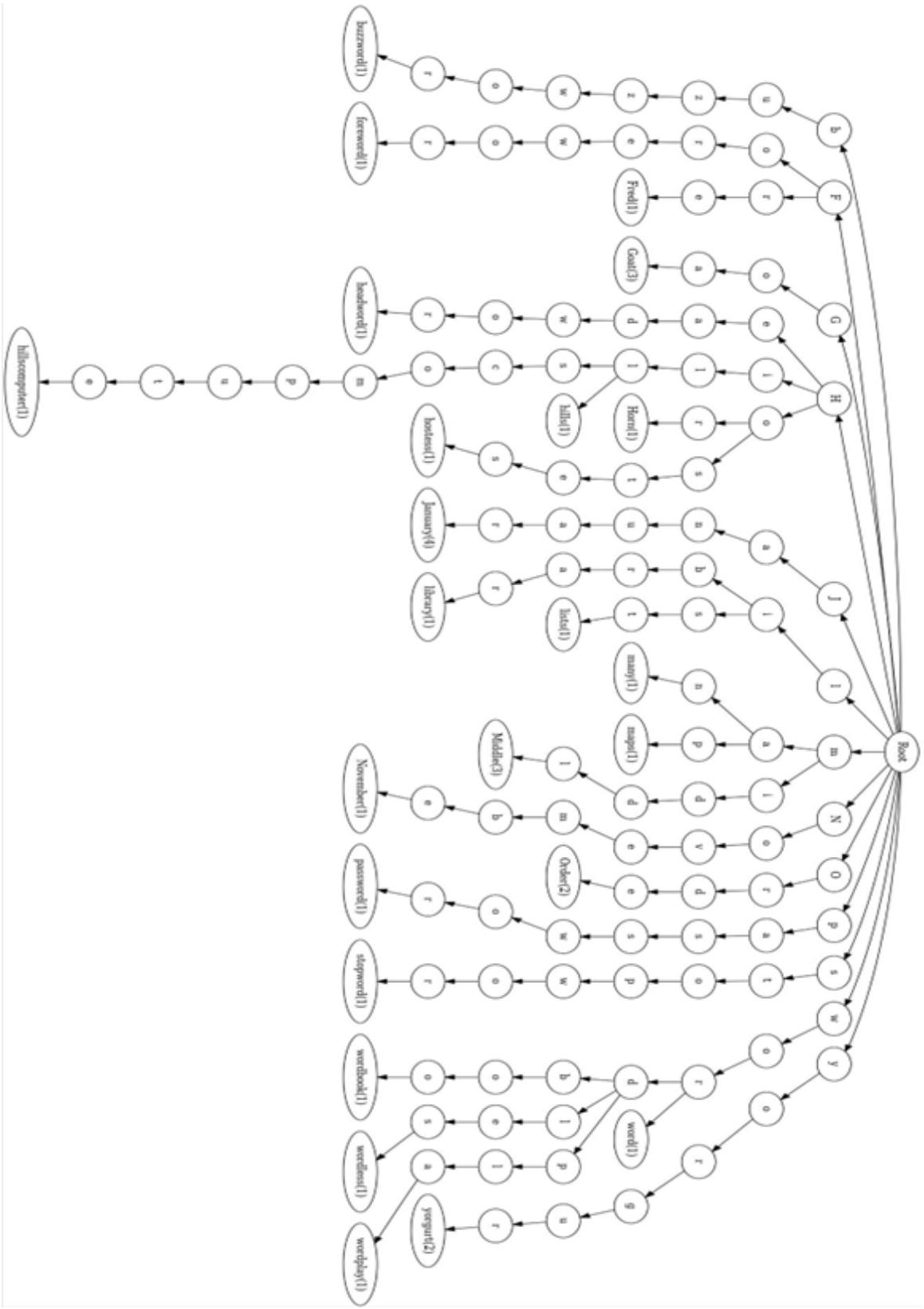

Figure 2. Tree of the Trie Data Structure for the Visualization Word List.

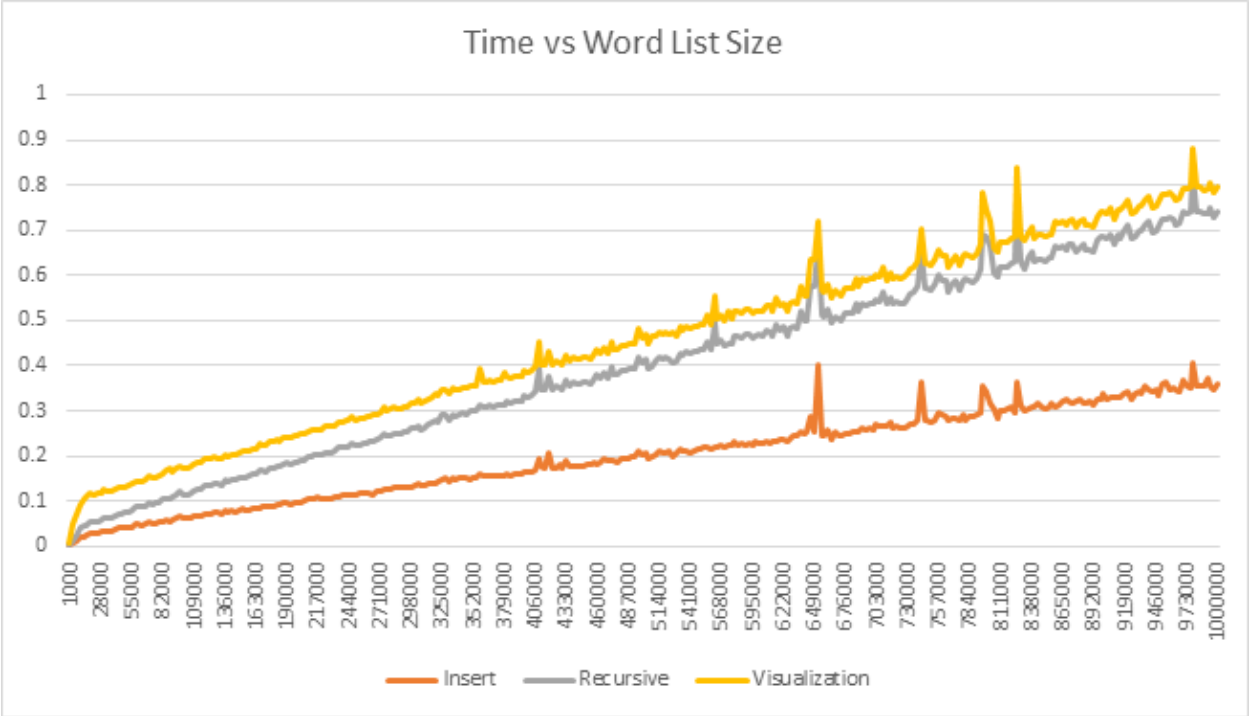

Figure 3. Using Timing Function For Insert, Recursive Insert, and Visualization

CONTRIBUTIONS

Michael Sayers concentrated on the main file, the node class, and the overall integration of everything. He thought of using the Scrabble dictionary for word verification. Michael researched and obtained several different word lists to compare times, eventually choosing the wordle list to repeat to generate the data for a graph of time versus word list size. Each data entry is every 3000 words starting at 1000 words. The data for the graph goes up to 1 million words. Michael also researched and implemented the visualization of the graph which assisted in the final development of the remove function. Vanessa Melgar implemented the insertion and removal functions of the trie. She also created a second version of the trie that used unordered maps instead of vectors to test whether such an implementation was to be more time and/or memory efficient. Shina Adewumi worked on the ascend function and the descend functions. Linda Tran worked on the search function and helped implement the timing function. Each member of the group was responsible for writing a section in the paper.

References

Agarwal, U. (2021, December 14). *Trie Data Structure - Underrated Data Structures and Algorithms*. Medium. https://medium.com/underrated-data-structures-and-algorithms/trie-data-structure-fd9a2aa6fcb8

"Trie: (Insert and Search)." *GeeksforGeeks*, 10 Nov. 2022, https://www.geeksforgeeks.org/trie-insert-and-search/.

Singh, S. (2020b, April 2). *Applications of Trie Data Structure*. OpenGenus IQ: Computing Expertise & Legacy. https://iq.opengenus.org/applications-of-trie/

Trie data structure - javatpoint. www.javatpoint.com. Retrieved December 5, 2022, from https://www.javatpoint.com/trie-data-structure

Trie Data Structure: Interview cake. Interview Cake: Programming Interview Questions and Tips. Retrieved December 5, 2022, from https://www.interviewcake.com/concept/java/trie