Trie Data Structure

Team 12:

Michael Sayers msayers1, Vanessa Melgar vmelgar9272, Linda Tran lindaqtran, Shina Adewumi fa-adewumi

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, ascend, descend, destroy. The ascend and descend were chosen to replace the postorder, inorder, and preorder. The decision was made since with multiple children simply stating which direction you were listing the children was sufficient to list all the nodes of the Trie.

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, 2020b) Another use of the Trie is to use it in networking within the router to lookup network maps. (Singh, 2020b) 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 off 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.

Michael Sayers concentrated on the main file, the node class, and the overall integration of everything. He thought of using the Scrabble dictionary for a 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 verse 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.

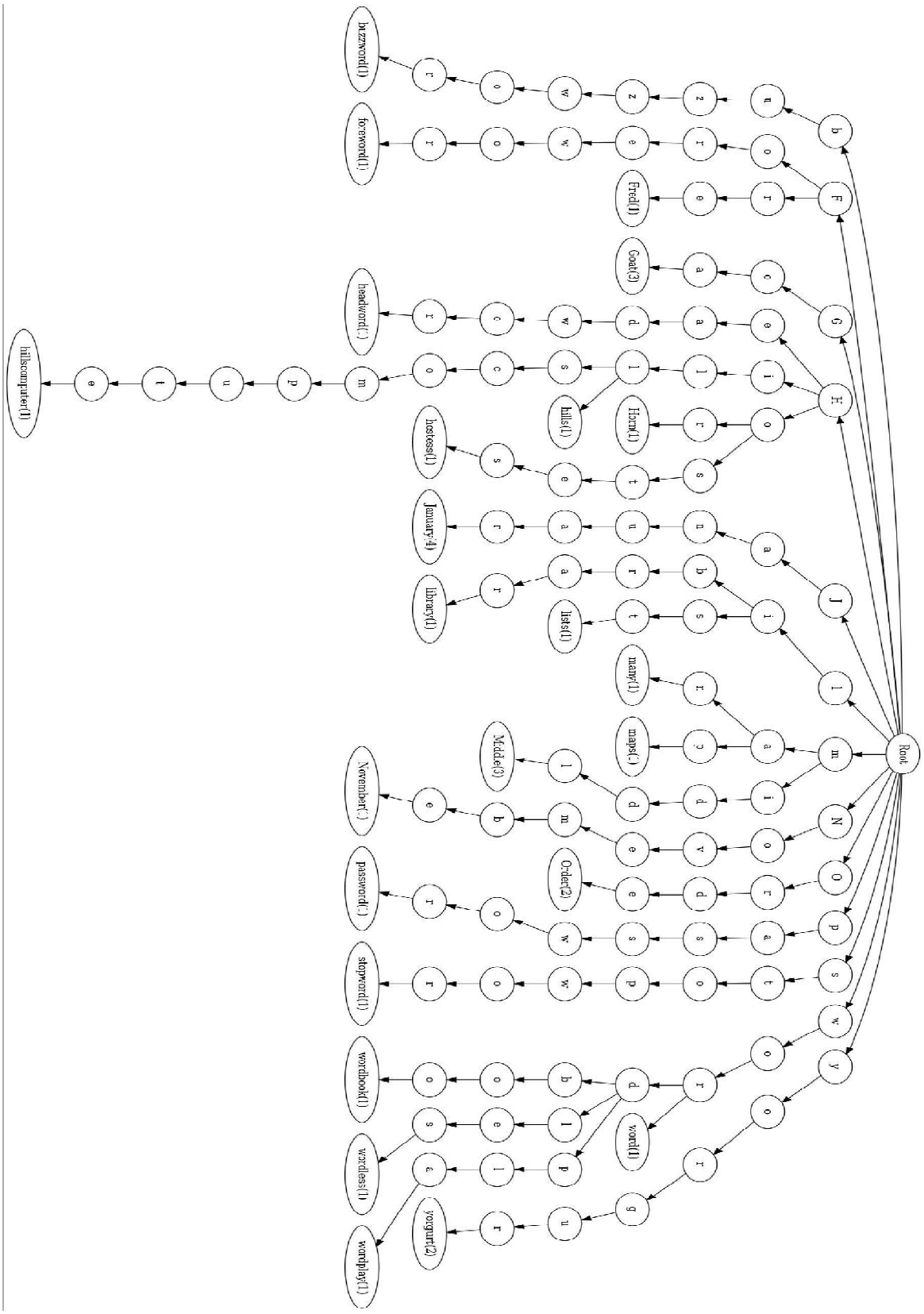
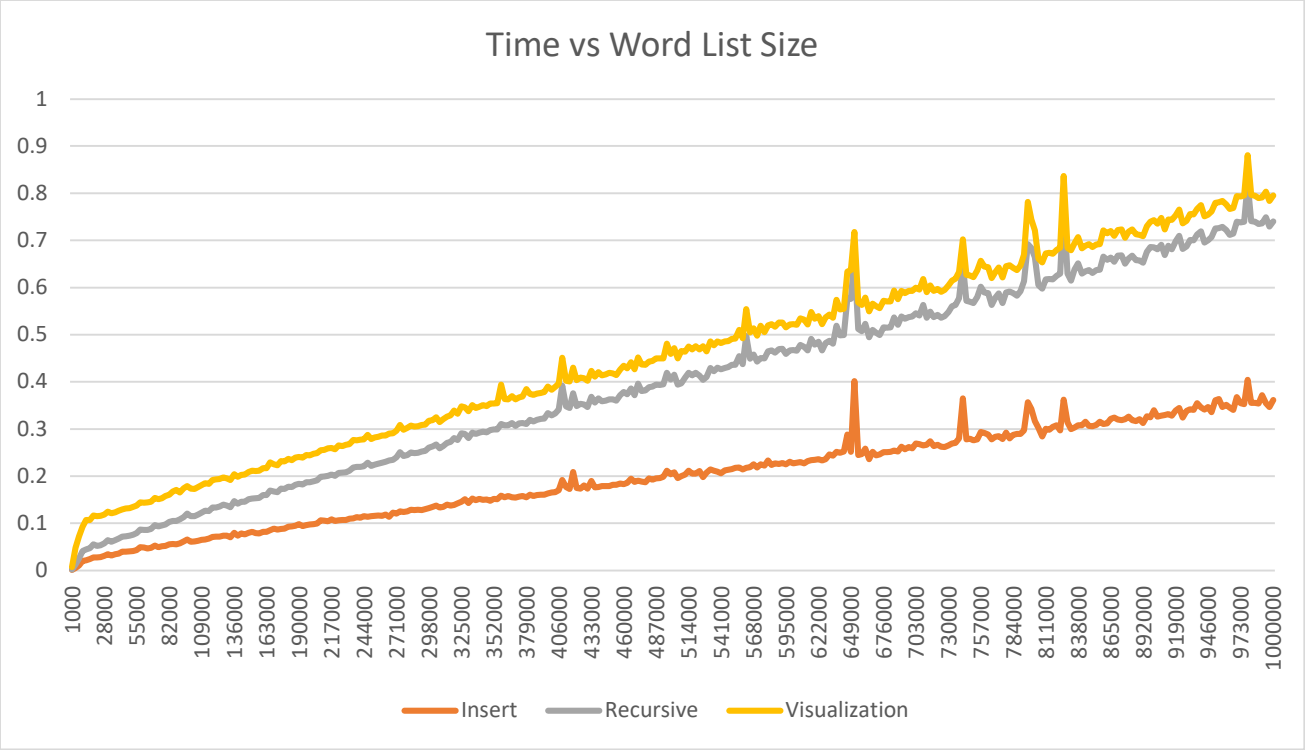


Figure 1. Tree of the Trie Data Structure for the Visualization Word List.


Time vs Word List Size
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
1000
28000
55000
82000
109000
136000
163000
190000
217000
244000
271000
298000
325000
352000
379000
406000
433000
460000
487000
514000
541000
568000
595000
622000
649000
676000
703000
730000
757000
784000
811000
838000
865000
892000
919000
946000
973000
1000000
Insert
Recursive
Visualization

References

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

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