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.

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 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 amount 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 last node is updated as necessary. The function will then loop through the word backwards. 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.

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. 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 in order to test whether such an implementation were to be more time and/or memory efficient.

Diagram

Description automatically generated

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

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