Affan Dhankwala

March 14, 2020

Design for Autocomplete

Autocomplete is a class that will—in essence—complete whatever word you were writing. It works like the Google search engine but it is rather simplified to only account for one word. Before I began this project, I thought about using a hash-map and mapping each word with its specific characters into the hash-map like the mini-search engine but I quickly found out that since the mini-search engine has hash-map was a recursive design, it would be extremely inefficient—assuming I coded it correctly—if I had many words. After listening to the lecture and learned that the correct data type to incorporate would be a Trie, I decided to create a Trie-like structure for my code. For my Trie structure I decided I would need three classes. One for my Trie node, one for my actual usage of my Trie nodes, and the final class for my tests. The help I used for my solution are these two links: <https://medium.com/@daetam/trie-autocomplete-8dd23ddd3846> and https://www.lavivienpost.com/autocomplete-with-trie-code/

In my Trie node class, since the structure resembles that of a tree, I decided to incorporate the basic tree class variables and that meant it needed to hold a character value and it needed to point to the next node. After some time I realized that I needed more instance variables. I needed to hold the current word that the Trie node was on by incorporating a LinkedList (through java.util) and this instance variable would store the word until the current node. To simplify the backtracking process, I also reversed the next pointer and called it parent because I wanted to easily go back up the tree without recursion or it would not be as efficient. Now I needed a way to indicate that I had reached the end of a word. For example, searching up you should autocomplete to “you” and “youtube” even though you is already in youtube. To approach this issue, I decided that instead of adding a hyphen or a special character node, I would just include a boolean in each node that would indicate whether it reached the end of a word. In terms of space, instead of allocating space to create nodes every time a word finishes, I slightly increase the space that each node takes in memory. In terms of time, I no longer have to traverse that extra node and could just check the boolean operation in each node to figure out if the word has ended. Now the methods that this class has is getChild method that takes in a character and finds the node associated with that character. This would be useful for when I need to search and see if a node with a certain character exists. The final function that I incorporate in my Trie node class is the getWords function. The getWords function traverses the whole linkedList of nodes and per word and when it reaches the end of a tree it will go back and recurse through the list for more words. It will only stop when the boolean condition returns true, meaning the word has ended and it should move to another word.

In my Trie class, I have three functions: insert, search, and autocomplete. The autocomplete uses the search and the insert serves the same purpose as the add function of the ArrayDictionary class. Insert works in that it initially should check if there is any other word that exists in the Trie that matches whatever was passed. If that returns true then the insert will do nothing and just return because there is no point in inserting a word that is already present in the Trie. It will call the search function that I will discuss later. If the word, however, is not present in the Trie, then the insert function will iterate through the Trie in the correct spot by calling getChild until it stumbles upon a null node—which is where it will add the new characters of the word that it is adding to the Trie. At the end of the insert function, we also set the boolean to true to indicate that a word has ended. The search function, which was referenced earlier, will simply look through each character and find how many overlapping characters a word has with the Trie and then it will return false and the assign the last node to the whatever variable was assigned to search. The search will also return true if the last character of the word has a boolean set to true which means that the word has ended and we have successfully searched for a word. Now finally, we may discuss what the autocomplete does. The autocomplete takes in a word and then assigns a node to the root case of our Trie. This is the initial position and if nothing is overlapping in any of the Trie elements and the current searched term then it will return the empty Trie node. Now, we iterate through each character of our word and then we call getChild for each iteration so that we may create a list of all the words that share the current characters of the search term. There is also a terminating condition inside the loop that if the list no longer contains any of the characters—meaning none of the words begin with the search term—then it will just return a null ArrayList and that way it is not wasting time by continuing the loop. After the loop, has finished, the completed list will have all the nodes that begin with the search term and we return them in a form of ArrayList to simplify the testing process.

I would like to briefly describe the tests used to validate this class to ensure accurate results before jumping into the time complexity of my algorithm. My test cases include the required test case that checks for a search term in a list of words and also checks for a search term that doesn’t exist in the list. I have include a test that passes a search term but it should only return the words that begin with the search term and not merely contain it. This is to make sure that if one is searching “tub”, “Youtube” doesn’t appear as a search suggestion. A final test that I added was having an empty Trie and then searching words and searching null and I expected to see nothing in both.

In regards to the time complexity of this solution, if we exclude the insertion as part of the autocomplete then all other external functions—those that are not autocomplete—are the ones that autocomplete calls. That would be the function of getChild. GetChild iterates through each node in the LinkedList of nodes and that increases if the word is larger so the complexity of getChild is O(N). Autocomplete calls getChild for every character in the word that is passed through it and that in itself is an O(N) complexity. Together this whole algorithm is an O(N2) complexity. This is the quickest way to do this solution because it looks at each character in the Trie once and returns whether the search term is present at the beginning of the collection of nodes. I am not sure of whether there are quicker methods out there but if they are I would assume they don’t repeatedly look into the Trie for words that have similar beginnings but rather they store these findings somewhere. But since my coding skills are currently not that advanced, I stuck to this solution and found it to be efficient.

Thank you