Write a 1 or 2-page document describe the design of your code. Use the above examples to illustrate how it works. Specify what data structure you are using and why. Specify the theoritical performance complexity of design with big O notation. Show result from your unit tests to complete the design doc. Argue that it's the fastest way you can come up with.

The data structure I used to solve this problem was a trie. In each node of the trie, I stored each possible letter in each possible word to pop up in the autocomplete as provided in the words list.

The root must be null, or an empty string that has references to its children nodes. Unlike a binary tree structure, a trie node can have multiple children.

I organized this solution into two classes, Trie and TrieNode, where Trie is made up of a bunch of trie nodes. Each trie node has a char that represents the letter in the word, a linked list that points to all its children, a linked list that points to its parent, and a Boolean that says whether or not that it’s the last node (it doesn’t have any children after it).

So by using a trie I can easily traverse all the words, and the possible words that autocompletes the input word is obvious through the path of the children nodes.

In order to store all the possible words in an organized trie, I had to use a function to add each word to the trie.

So autocomplete, you must first find the last node of the prefix, then find all the words associated with that node. Then, use preorder traversal to navigate through the trie. This has a time complexity of O(n) because the worst case scenario is that we have to visit every trie node.

So for the unit tests:

String[] set = { "hello", "high", "seattle", "seatac", "see", "hollow", "how" };

String[] inputs = { "h", "se", "sea", "ho", "xyz" };

These should be the expected values:

String[][] expects = {

{"hello", "high", "hollow", "how"},

{"seattle", "seatac", "see"},

{"seattle", "seatac"},

{"hollow", "how"}

};

Here are the results (shown at bottom of console):

