Name: WILL NIGEL C. DE JESUS

1. For this problem, use the red-black tree visualization of David Galles (https://www.cs.usfca.edu/ galles/visualization/RedBlack.html)

   (a) (5 points) Let's consider an insertion pattern that is the worst case of insertion for a binary search tree (e.g. keys are inserted in sorted order)

   Starting from an empty red-black tree, insert the values $1, 2, \ldots, 22$ in order. At each stage, record the maximum depth in the tree. Provide a list of these maximum depths, in order (i.e., $0, 1, 1, 2, \ldots$).

   You can use the "Animation Speed" slider at the bottom of the tool to speed up the animation. You may need to use "Change Canvas Size" to be able to see the entire tree.

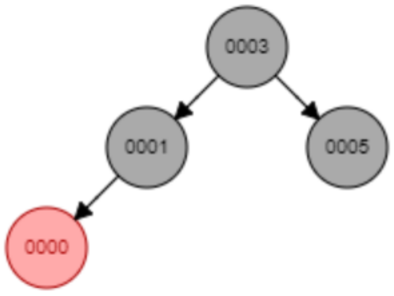
   Solution:
   0,1,1,2,2,3,3,3,3,4,4,4,4,5,5,5,5,5,5,5,5,6

   (b) (5 points) What do you notice about the left side of tree after all 22 keys have been inserted that prevents the root-to-rightmost-node path from getting too large?

   Solution:
   Whenever the height of the right side tree exceeds the height of the left side tree **twice**, the tree performs rotations to balance the left and right sides.

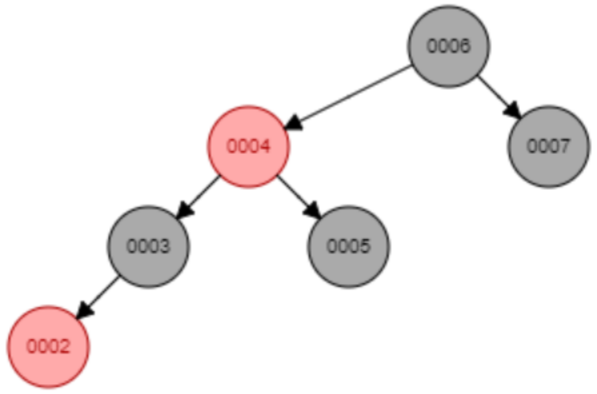
   (c) (5 points) Start with a new tree and then try a series of insertions and deletions. Your goal is to get a tree that has at least one root-to-leaf path of 3 nodes that has no branching. That is, there is some path from the root to some node a to some other node b, such that b is the only child of a, and b itself has no children. Take a screenshot of your final tree and put it as the answer here. No explanation is needed.

   Solution:



(d) (5 points) Try to repeat the previous taks to get a non-branching root-to-leaf path of 4 nodes rather than 3. You will find that this cannot be done. Using the rules of a red-black tree, give an explanation about why this is impossible. Your explanation only needs to handle the 4-node case, but notice that it generalizes to longer lengths and helps explain why the tree cannot become too imbalanced.

Solution:



In order to reach 4 node root-to-leaf path, the rbt will already perform rotations to balance out the depth of the tree intended, which will result to branching out.

2. (10 points) Recall the AVL tree that was introduced in CMSC 123. Prove that an AVL tree with n nodes has height $O(\log n)$. (*Hint*: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the h-th Fibonacci number.)

Solution:
Inspection:
The height of an AVL tree of h has height $O(\log n)$
For $n(1) = 1$ and $n(2) = 2$
For $n > 2$, an AVL tree is of height h is composed of the root node, one $n-1$ height subtree and one $n-2$ height subtree
$n(h) = 1 + n(h-1) + n(h-2)$
Proof by induction:
IH: The height of an AVL tree of h has height $O(\log n)$
BC: For $k \leq 2$, $n(1) = 1$ and $n(2) = 2$
IS: For $k > 2$
$T(n) = T(n-1) + T(n-2) + 1$
$T(n) = 2T(n-2)$
$T(k) = 2T(k-2)$
$T(k) = 4T(k-4)$

$T(k) = 8T(k-6)$
...
$T(k) = 2^k T(k-2k)$
$T(k) = 2^{\frac{k}{2}-1}$
$T(k) = 2logk$
Conclusion:
AVL Tree of h has hieght $O(\log n)$

3. (30 points) **CODING** Nearly all modern editors include a spellcheck feature which identifies incorrect words and suggests possible words which the user might have meant to type. While modern spellcheck algorithms typically involve machine learning, we can create a simple spellchecker by suggesting words which are an **edit distance** of 1 from the incorrect word.

   We say that a word is an edit distance of m from another words if we can transform one of the words into the other by using a total of m operations. Possible operations are deleting a character, insertin a character, replacing a character, and switching two adjacent characters. As m increases, finding words which are an edit distance of m apart becomes exponentially harder. However, it turns out, around 80% of spelling errors are an edit distance of just 1 from the correct word! Thus, we will be able to create a decent spellchecker by simply suggesting words which are an edit distance of 1 from the incorrect word.

   Task
   Using the starter code, implement a simple spellchecker that takes in an input word and returns a list of valid words which are an edit distance of 0 or 1 from the input word.

   - You can assume that the input word will consist only of lowercase alphabetic characters. This means you do not need to consider uppercase characters, spaces, punctuation, numbers, etc.
   - The list of valid words are provided in the starter code. You should not return any words which are not in the list of valid words.
   - You do not need to consider whether the input word is spelled correctly. Your function should still return words which are within an edit distance of 0 or 1 of the input word regardless.
   - A template for a ternary search tree data structure (an extension of the binary search tree) is provided. You should fill in the functions `insert` and `contains` within this data structure. Your spellchecker should make use of this ternary search tree data structure to store the list of valid words and to check whether a string is among the stored valid words.

- Both your `insert` and `contains` function should have worst-case running time of $O(n + k)$, where n is the number of words in the ternary search tree and k is the number of characters in the word being inserted or searched for.
- You do not need to implement your own edit distance function. It's already provided for you within the function `getNearbyStrings`. The function `getNearbyStrings` takes in an input word and returns a list of strings which are an edit distance of at most 1 away from the input word. It is your task to use the ternary search tree data structure to check whether each of the strings in the list is among the stored valid words.

Ternary Search Trees
In the lecture, we discussed how we can store and search for values efficiently using a binary search tree. In a binary search tree, each node has (up to) two children, one on the left and one on the right. The left child has a smaller valud that its parent node and the right child has a larger value that its parent node. Binary search trees are great for storing and searching data efficiently; however, if we want to store a huge amount of data, binary search tree might become quite large and take up a lot of memory. What improvements can we make?

If we want to store a lot of dictionary words, the sheer amout of words and the number of bits required to store each character of each words, a binary search tree can be quite memory-inefficient. However, in the set of dictionary words, there is a lot of *repeated information*. Many words share common prefixes but in a binary search tree we store all of these words separately and thus have to store the full bit-length of each of these words. What if we only had to store each prefix once?

This is where ternary search trees come in. In a ternary search tree, each node has *three* child nodes instead of two: a left child, a middle child, and a right child. The left child contains a character value less that its parent node, and the right child contains a character value that is greater than its parent node. The middle child, on the other hand, contains the next characters of a stored word.

How do we insert and search for words? You can visualize how they work with this visualization: https://www.cs.usfca.edu/ galles/visualization/TST.html. Try using the "Insert" function to insert different words in a ternary search tree, including words with and without common prefixes and words with different starting letters, and try using "Find" function to search for words that may or may not be in your ternary search tree. One helpful set of words to try is ["she", "sells", "sea", "shells", "by", "the", "sea", "shore"]. As you go through the visualization, read through the following step-by-step descriptions.

Insert When we insert a word in a ternary search tree, we insert one characters at a time. We first check the middle child node. If none exists, we insert the character, traverse down that new node, and move on to the next characters in our word. If a middle child node exists and its value is equal to our character then we traverse down the middle child node and move on to the next characters in our word.

If a middel child exists but its value is not equal to our characters, then we either traverse down the left child node if our characters has value less than the middle child node or traverse down the right child node if our character has value greater than the middel child node. If no such left or right child node exists, we create it and insert the characters there, and then store the remaining characters in our word in successive middle child nodes.

Additionally, upon inserting the last character of our word, we mark the node containing the last character as an end node. This is important for searching.

Search When searching for a word in a ternary search tree, we compare one character at a time. We first check the middle child node. If the middle child's value is equal to our character, then we traverse down it and move one to the next character in our word. If the middle child's value is less than our character, then we traverse down the left child node. If it is equal, we move on to the next character in our word; otherwise, we traverse down the left or right child of that node and repeat. If the middle child's value is greater than our character, then we traverse down the right child node. Similarly, if it is equal, we move on to the next character in our word; otherwise, we traverse down the left or right child of that node and repeat.

If any of the child nodes we check do not exist, then we return that the word does not exist in the tree. Or if we reach the last character of our word and its node is not marked as an end node, then we also return that the word does not exist in the tree. If we reach the last character of our word and its node is marked as an end node, only then do we return that the word exists in tree.