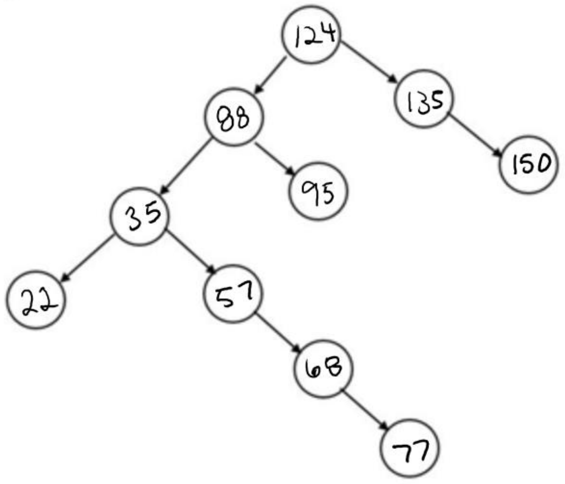
Ivy Truong
4/5/20
CSCI 3412
Homework 5

1) a). Keys ***57, 135, 22, 35,68, 124, 77, 150, 95, 88*** assigned to binary search tree below.

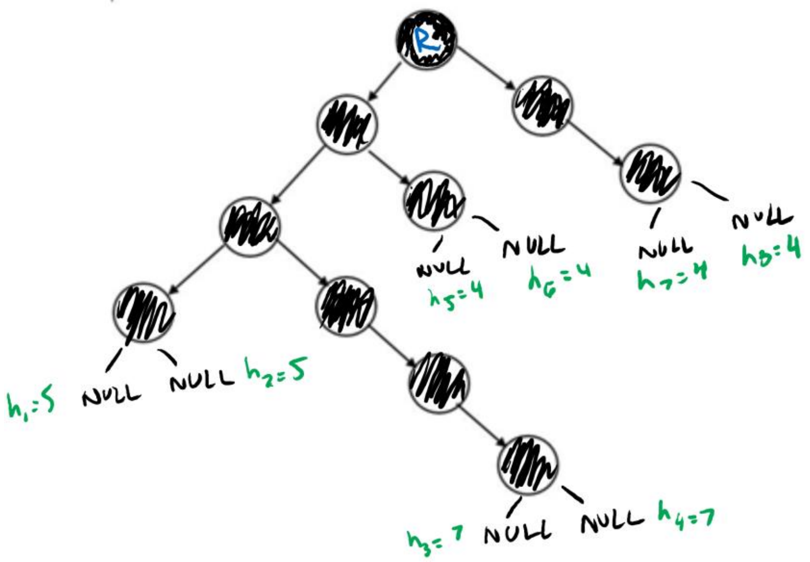


b). Consider an array A of random numbers with a size of 10. To fill the tree given a random ordered list of any set of keys, we would first sort the list of keys in ascending order. Next, we would find the third largest value in the list, which is at A[7] and make that the root of the tree. Then, we would input the values into the left side of the tree in this order: insert the value in A[5] as the left child of value in A[7], insert the value in A[1] as the left child of value in A[5], insert the value in A[0] as the left child of value in A[1], insert the value in A[2] as the right child of value in A[1], insert the value in A[3] as the right child of value in A[2], insert the value in A[4] as the right child of A[3], and insert the value in A[6] as the right child of value in A[5]. Lastly, we would input the values in the right side of the tree, where the value in A[8] will be the right child of the root (A[7]) and the value in A[9] will be the right child of the value in A[8]. By following the algorithm above, we would be inserting the nodes from the sorted array in a preorder traversal, starting from the parent, then the left child, then the right child.

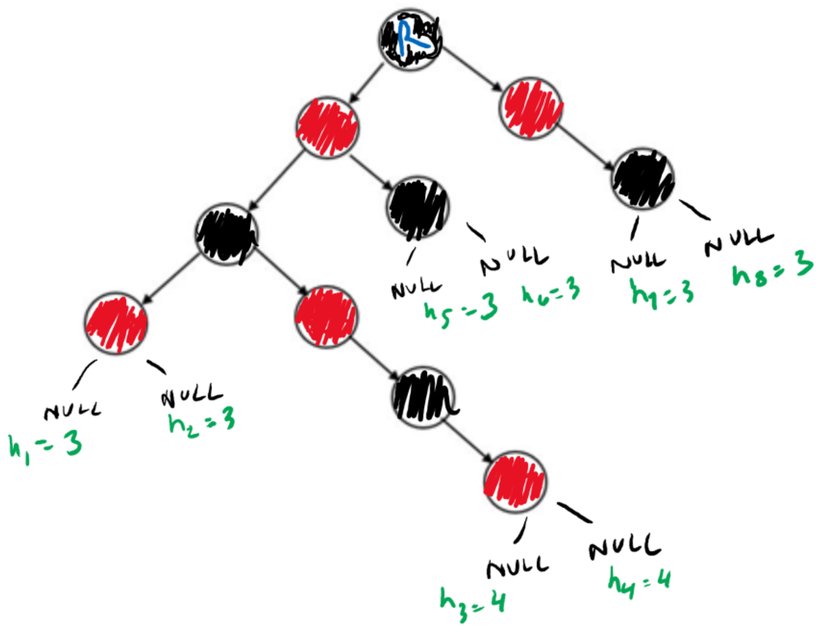
c). i. The ordered list of keys which can be sequentially inserted into the tree starting at the root node is 124, 88, 35, 22, 57, 68, 77, 95, 135, 150.

ii. To get the tree from 1a and 1b, you would have to insert the values in the list from 1c part i using preorder traversal and adding some other modifications to get the same properties of the tree. Through preorder traversal (parent, left, right), the keys in the list from 1c) i. can be inserted correctly starting at the root node.

d). This specific binary search tree in 1a and 1b cannot be colored to form a red black tree because it would violate property 3 or 4. Property 4 states that all simple paths from any node to a descendant leaf have the same number of black nodes. For example, let's say all the nodes in the tree from problem 1a and 1b were black. The tree would still follow the other three properties because (1) a node is either red or black, (2) the root and descendant nodes (NULL) are black, and (3) if the node is red its parent is black. Let's calculate the black height of the tree when we start at root node. The image below shows that the black height starting from the root node to each of its descendant nodes is not equal, violating property 4.

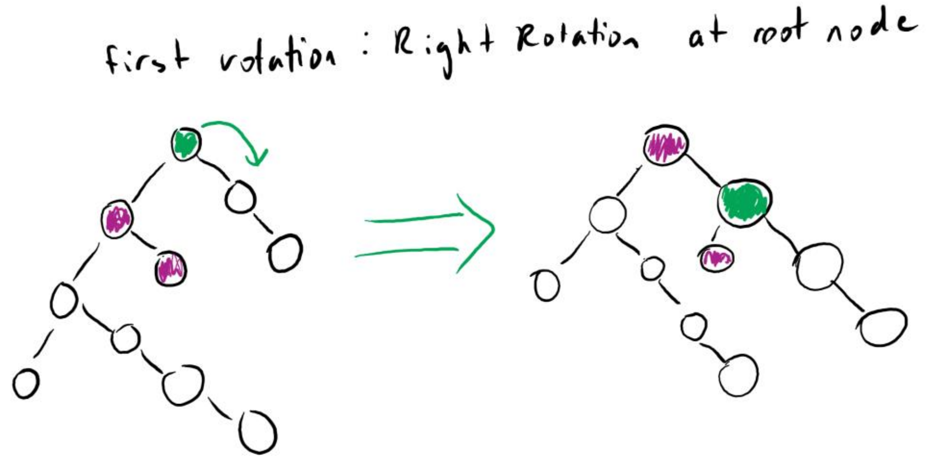


Below is another example if red nodes were included in the tree. In the example below, even if we changed one of the black nodes to a red node in order to satisfy property 4, the tree would still violate property 3, where a red node's parent is black. Changing one of the black nodes below would cause two red nodes to be next to each other, violating property 3.

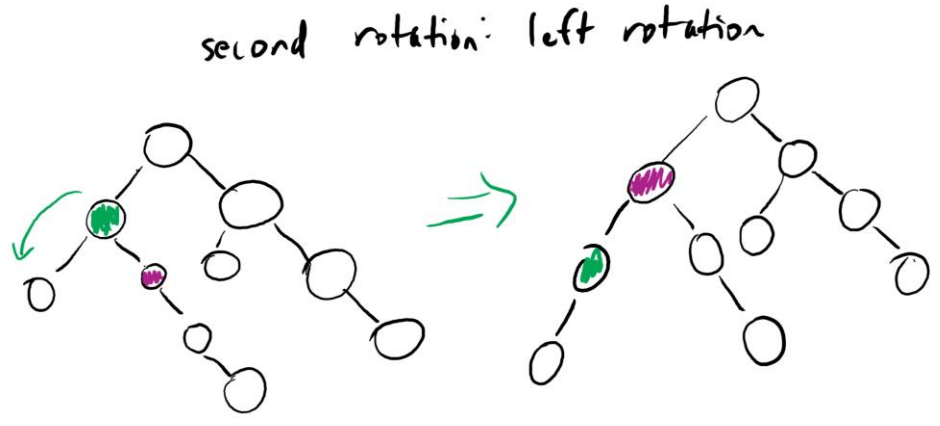


Another reason why the tree cannot be colored into a red black tree is because red and black trees are considered height balanced trees. Therefore, this means that the height from any given node to the leaf node on the left subtree should have a height that is equal to or has a difference of 1 from the right subtree. Formally, if L= left height of tree and R = right height of tree, the height of the tree must be L=R, L=R-1, or L = R+1. If the tree is not balanced, then the red black tree will have at least 1 extra black node in the tree that would violate property 4.

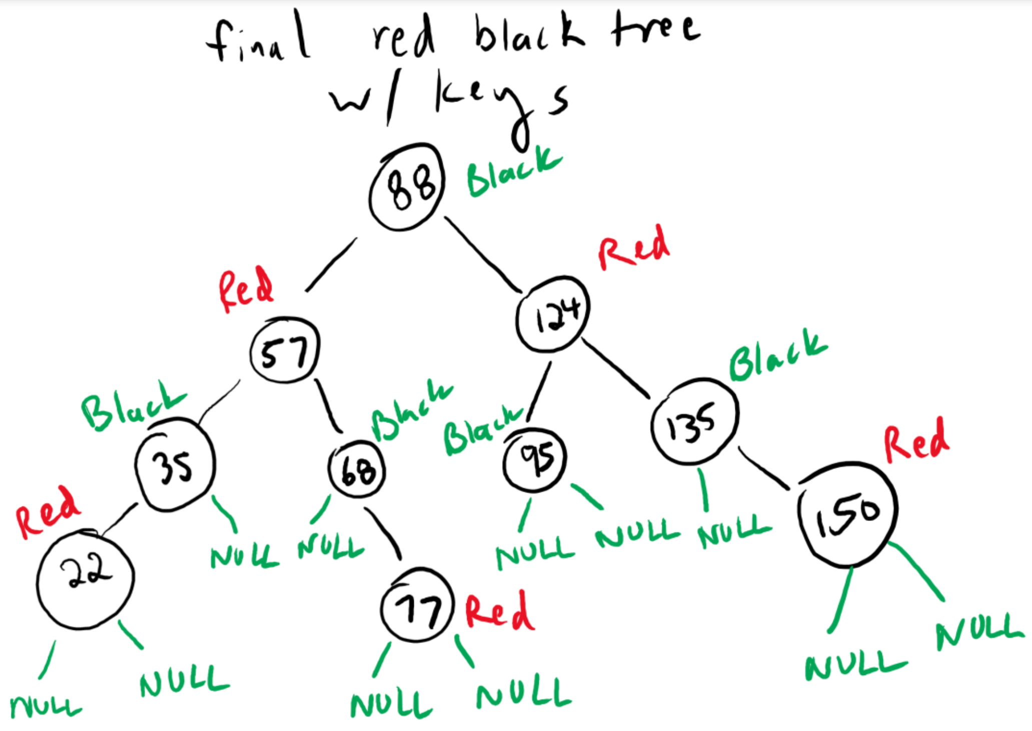
e). The first rotation done was a left rotation on the root node. This rotation was done so that there were more nodes on the right side and fewer nodes on the left side to balance the height of the tree. To do the left rotation, the root node became the right child of its left child and the left child became the new root node. Since the (old) root node's left child already has a right child before the rotation, that node's right child becomes the left child of the (old) root node.



For the second rotation, the rotation was a left rotation done on the new root node's left child. The rotation was done to balance out the height of the left subtree of the new root. To do the left rotation, the new root's left child became the left child of its right child. This right child became the new left child the new root and the parent of the old left child new root.



Below is the final red black tree with all the keys inserted and with their corresponding colors.



2) The time efficiency test results for LCS recursion and LCS memoisation was 32 minutes 40 seconds for recursion and 1 minute 20 seconds for memoisation.

3) e. From the bar graph generated, the threshold number would be around 4500. The threshold of number of recursive calls for this graph is around 4500 because the graph shows that between 4000 and 5000 recursive calls, the frequency of recursive calls has reached the max collisions. After that threshold, the number of collisions for each number of recursive calls starts to decrease.

Summary of what each file in submission represents:

- outRec1.txt: output of LCS numbers with recursion and the time efficiency of the LCS recursion function
- outMem.txt: output of LCS numbers with memoisation and the time efficiency of the LCS memoisation function
- outModRec.txt: output of LCS numbers and recursive calls tuple using the modified LCS recursion function and the time efficiency of the function
- LCStuple.txt: same as outModRec.txt file, but only contains the tuples of LCS numbers and recursive calls; used to test the hash function in 3c/d
- hash.txt: list of all the values hashed by the number of recursive calls
- hashLen.txt: list of unique keys and the number of collisions/frequency of values with the same key
- hashPlt.png: image of the bar graph of values in hashLen.txt (bar graph without modification)

- hashPlot.html: html file of bar graph from hashPlt.png (bar graph without modification)
- moddedhash.txt: list of all values hashed by the number of recursive calls and use of secondary hash when number of recursive calls exceeded 10,000
- moddedHashLen.txt: list of unique keys and the number of collisions/frequency of values with the same key
- modHashPlt.png: image of the bar graph values in moddedhash.txt (bar graph with modification)
- modHashPlot.html: html file of bar graph from modHashPlt.png (bar graph with modification)