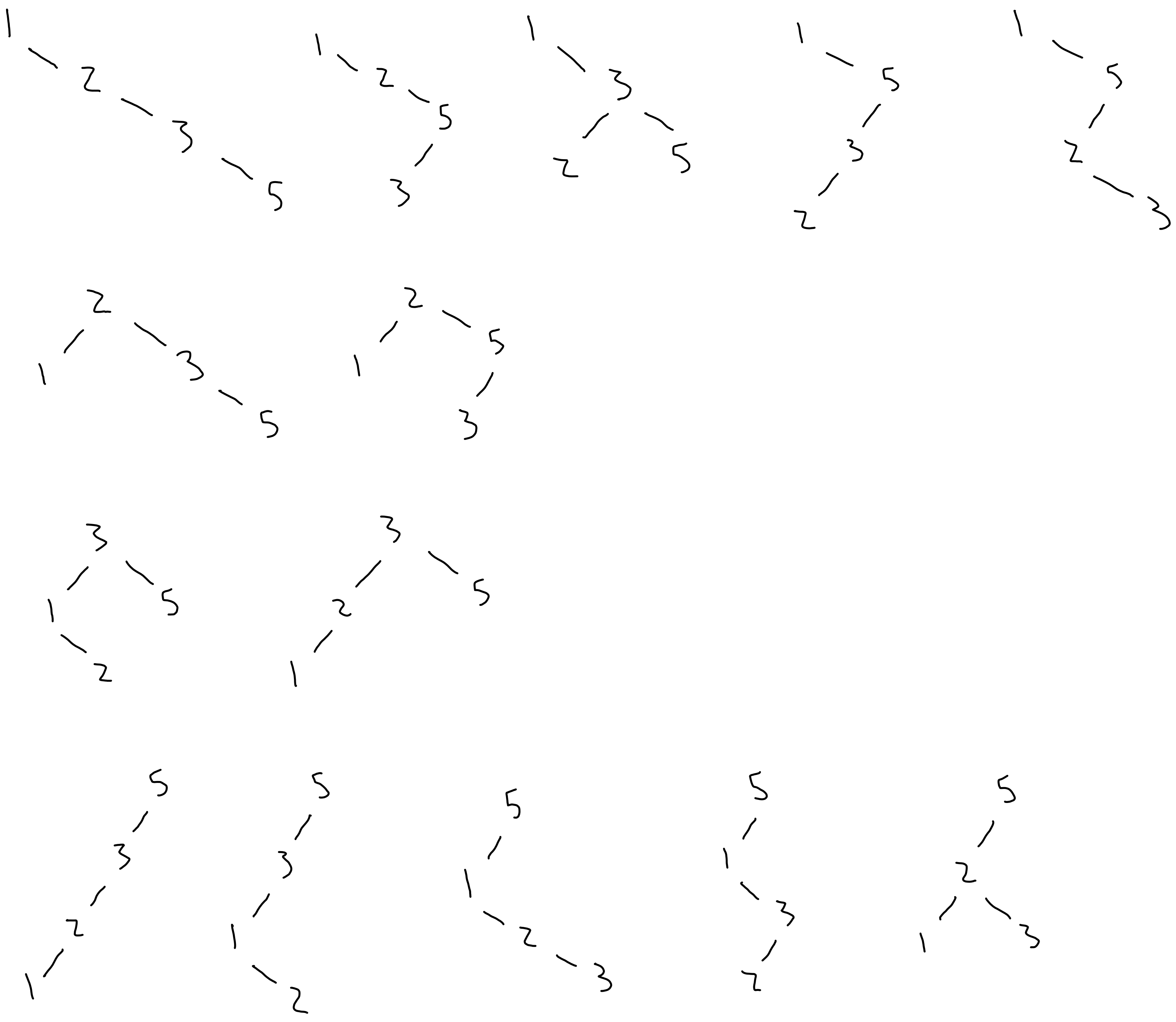
1. [10 points]

Consider a Binary Search Tree.

(a) Draw all valid Binary Search Trees for the four nodes 1, 2, 3, 5.



(b) Look up Catalan Numbers on the internet, and compute the nth Catalan number for n = 4. Compare this number to the number of Binary Search Trees you came up with above.

The nth Catalan number is given by the equation C(n) = (2n)! / ((n+1)!n!). In order to find out how many unique Binary Search Trees we should have in our answer to (a), we plug in n = 4 to this equation C(n).

C(4) = (2*4)! / ((4+1)!4!) = 8! / (5! * 4!) = 40320 / (120 * 24) = 40320 / 2880 = 14.

This checks out, because in part (a) we were able to draw 14 different valid Binary Search Trees.

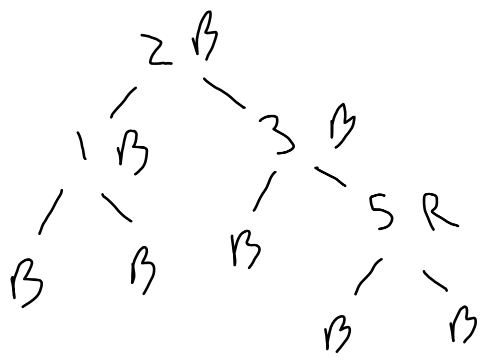
It also makes sense if we consider the fact that, conceptually, in order to find out how many unique BSTs we can make from a set of n numbers, we can simply add up the number of unique trees we can make by choosing a different number from our set as the root each time:

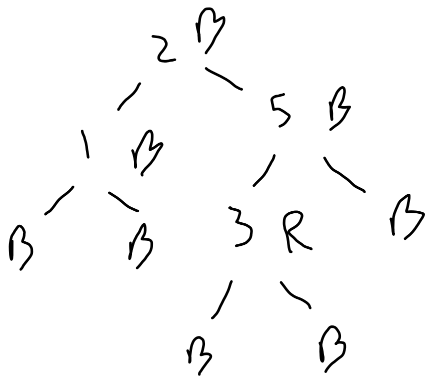
Choosing 1 as the root with n = 4, we can make 5 unique BSTs
Choosing 2 as the root with n = 4, we can make 2 unique BSTs
Choosing 3 as the root with n = 4, we can make 2 unique BSTs
Choosing 5 as the root with n = 4, we can make 5 unique BSTs

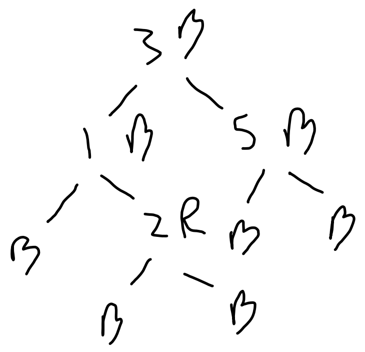
5 + 2 + 2 + 5 = 14 unique BSTs in total for our set of 4 numbers { 1, 2, 3, 5 }.

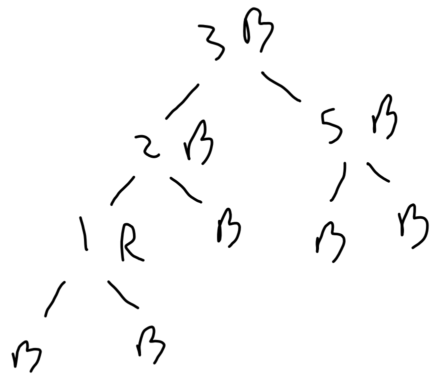
2. [10 points]

(a) Draw all valid Binary Red Black Trees for the four nodes 1, 2, 3, 5. Label each of your nodes R or B for Red/Black. Note the corresponding Black Height of each of the nodes in your Red Black Trees, to help convince yourself it is a valid Red Black tree.



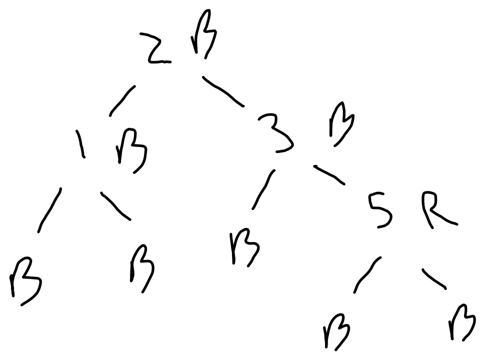






(b) At least for one of your Red Black Trees, give a full explanation of why it is a valid Red Black Tree, and how you determined the Black Height of the nodes.

Considering this Red Black Tree from part (a):



We can conclude that it is a valid Red Black Tree if it satisfies the following properties:

1. **Every node is either red or black**; this is clearly true, since we can see every node has a numerical value as well as a 'B' or 'R' indicating its color

2. **The root is black**; this is also true, since our root is the number 2 and it has label 'B' for black.

3. **Every leaf is black and nil**; this is true, we can see the node 1 has two black leafs with no numerical value indicating that they are nil. Furthermore, node 3 has a single black, nil leaf, and finally node 5 has two black, nil leafs.

4. **If a node is red, then both its children are black**; we only have one red node which is 5, and clearly it has two black children, so this is true.

5. **All paths from node to leaves have same number of black**; we can show that this is true by following every path that leads to a nil leaf and counting the number of black nodes along the way. Starting with the root node, we can follow these paths to get to a leaf:

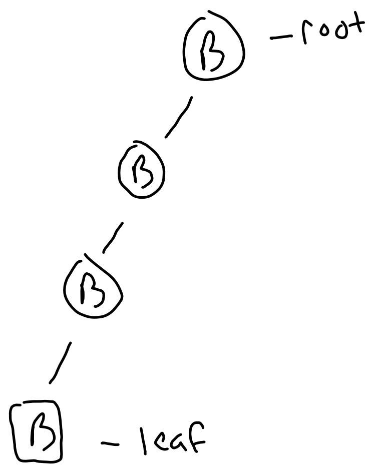
left, left: 2 black nodes encountered
left, right: 2 black nodes encountered
right, left: 2 black nodes encountered
right, right, right: 2 black nodes encountered
right, right, left: 2 black nodes encountered

As we can see, every possible path to a nil leaf has 2 black nodes along the way, so this final property is also true. Since we have shown all 5 properties to be true, this tree must be a valid Red Black Tree.

3. [10 points]

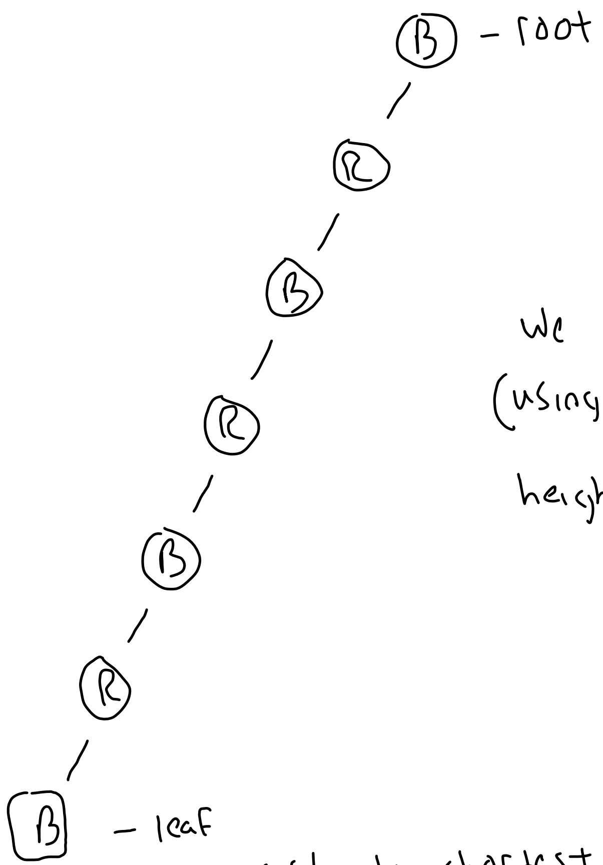
Show schematics of Red Black trees with Black Height 3, that have either the shortest or the longest possible simple path from the root node to a descendant leaf (nil node). What is the maximum ratio between the longest and the shortest possible simple paths?

<u>Shortest</u> <u>path</u> to nil with Black Height 3:



we can see the shortest path (all black) has height 3.

<u>Longest</u> <u>path</u> to nil with Black Height 3:

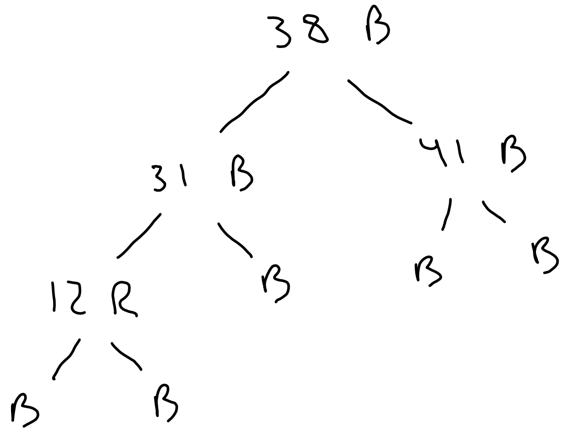


We can see the longest path (using as many red as possible) has height 6.

Therefore, our longest to shortest path ratio is 6 : 3 or 2 : 1. At most, the longest path will be twice as long as the shortest.

4. [10 points]

Consider a Red Black tree that already has 41; 38; 31; 12 inserted. Draw the resulting Red Black tree (you can use for help https://www.cs.usfca.edu/galles/visualization/RedBlack.html). Now consider adding the node 19. Show all the stages of the insert fix up and write down what cases these changes correspond to (from what we learned). Again, you can use the animation for help, but explain what was done in terms of the cases. Finally, explain why the final tree after inserting 19 is a valid Red Black tree.

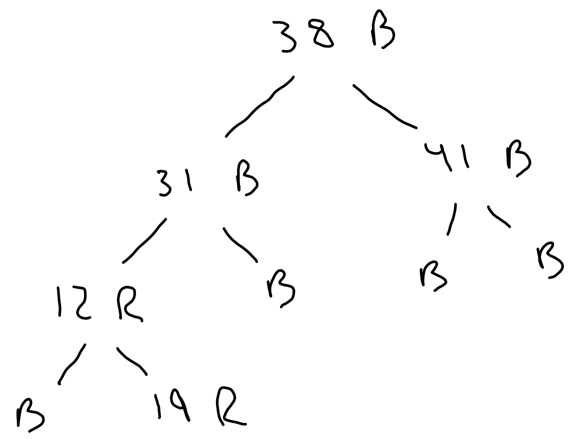


If we want to insert 19 into this tree, first we must find its place using the BST properties that left children are smaller than the parent and right children are bigger:

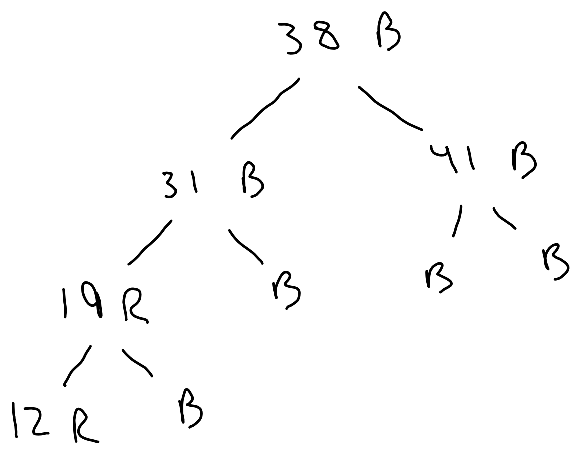
19 is compared with the root 38 and found to be smaller, so we go left.
19 is compared with 31 and found to be smaller, so we go left again.

19 is compared with 12 and found to be bigger, so we go right.

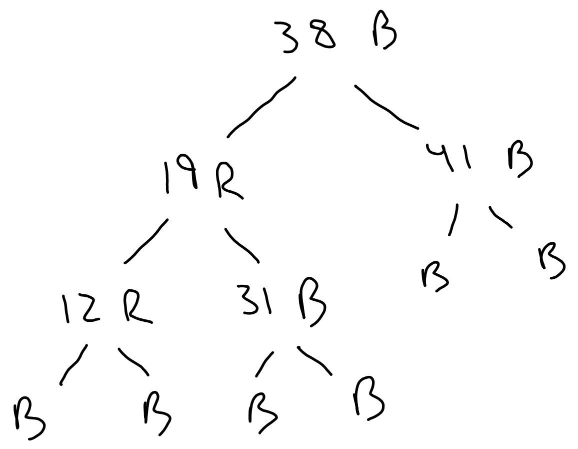
At this point, we have encountered a nil leaf, so we insert 19 with the color red as the right child of 12. Our tree now looks like this:



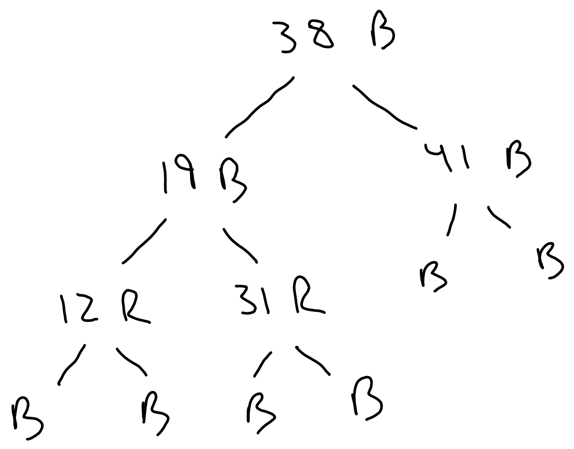
We can see that since 12 and 19 are both red, we have two reds in a row, so we have to fix this to ensure that we have a valid RBT. We will swap the nodes 12 and 19 since our two reds in a row are not both left children (19 is a right child). After the swap, 19 is the parent and 12 is the child. Since 12 is smaller than 19, it will end up as the left child of 19:



But our issue is not solved yet, since we still have two reds in a row. However, since these two red nodes in a row are both left children (19 is left child of 31, 12 is left child of 19), we know we can swap 31 and 19 without violating our tree properties. Our 12 can stay where it is even if we swap 31 and 19 because 12 is smaller than both 31 and 19:



Finally, we will change the color of 19 to black and the color of 31 to red so that we no longer have two reds in a row. The result is a balanced, valid RBT with 19 now inserted.



By following these steps during insertion, we are ensuring all 5 properties of RBTs are still true for this tree:

1. **Every node is either red or black**; this is true, every node is clearly assigned a color (19 was originally inserted with color red).

2. **The root is black**; this is true, we didn't change the root's color at all during insertion.

3. **Every leaf is black and nil**; this is true, all leaf nodes are black and nil (unchanged during insertion).

4. **If a node is red, then both its children are black**; this is true, we ensured this by doing our two swaps and changing 19's color to black and 31's color to red at the end.

5. **All paths from node to leaves have same number of black**; this is true, we ensured this by swapping our two reds in a row (19 and 12) so that they are both left children, and only then swapping 19 with its black parent 31 and doing the color change. This ensures that the Black Height of our new tree after inserting is the same as the Black Height of our original tree. Therefore, if the original tree had the same Black Height at all paths (satisfying this property), then our tree after insertion should also have the same Black Height at all paths.

The result of our insertion is a valid RBT since we took the necessary steps to ensure that both BST and RBT properties are maintained. It is more complicated than inserting into a BST, but it keeps our RBT balanced which is the whole point of using a RBT over a BST (quicker search time due to O(log(n)) height as opposed to BST which has O(n) height).