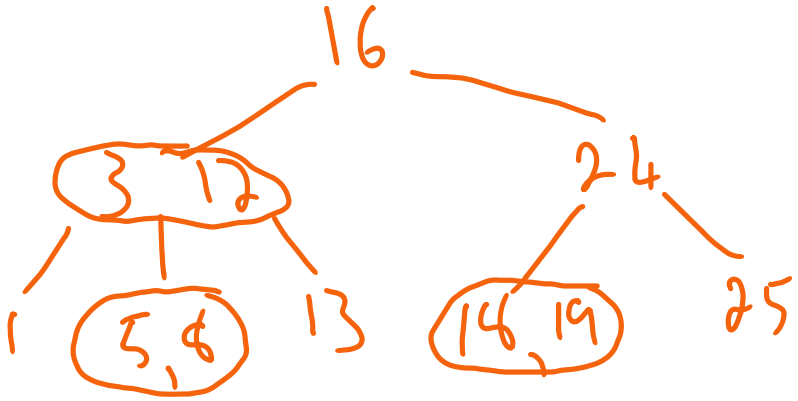
COMPSCI 2C03 – Week 7 Exercises

Sample solutions and notes on sample solutions for this week's exercises. Corrected Nov 9, 2023.

Lecture 1: 2-3 Trees

1. Exercise 3.3.2: Draw the 2-3 tree that results when you insert the keys 25, 12, 16, 13, 24, 8, 3, 18, 1, 5, 19 in that order into an initially empty tree.



2. Exercise 3.3.3: Find an insertion order for the keys 19, 5, 18, 3, 8, 24, 13 that leads to a 2-3 tree of height 1.

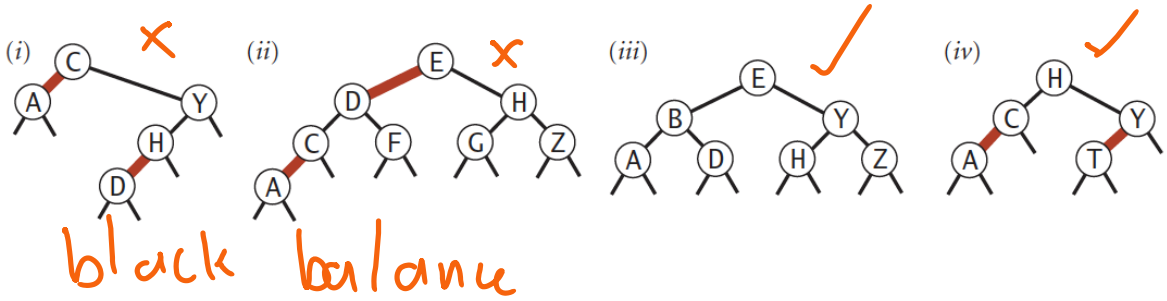
18, 19, 24, 5, 8, 3, 13

There are others as well. Start by drawing a height 1 tree that works, then reverse engineer the insertions so that you get the keys you want in the root node, then insert the rest.

Supplementary question – it's actually a bit more of a challenge to find an insertion order for a height 2 tree. Can you find one?

Lecture 2: Left-Leaning Red-Black Trees

3. Exercise 3.3.8: Which of the following are left-leaning red-black BSTs?



4. Exercise 3.3.33a: Write a pseudocode function called **is_23** to check that no node is connected to two red links and that there are no right-leaning red links.

```
Note – the color field of a node is true if it is red, false if it is black.

is_23(node):                                    # wrapper to present a nicer interface
    return is_23_work(node, false)


is_23_work(node, parent_isred):
    if node == null:
            return true       # empty tree is a red-black tree
    if parent_isred and node.color:
            return false      # parent and this node are red
    if node.right != null and node.right.color:
            return false      # right leaning red link – covers case of left and right red links
    if not is_23_work(node.right, node.color):
            return false      # stop the recursion as soon as we find it's not a 2-3 tree
    if not is_23_work(node.left, node.color):
            return false
    return true               # everything checks out. It's a 2-3 tree.
```

5. Exercise 3.3.33b: Write a pseudocode function called **is_balanced** to make sure every leaf node has the same black height. Slight fix to this code – black height is the height to each null link rather than to leaf nodes.

   It's possible there is a more elegant way to do this. The approach here requires two helper functions. The first one returns the black height of the minimum value because it's easy to find. The second helper accepts a target black height and traverses the entire tree looking for null links. If it finds a null link that doesn't match the target black height, it returns false.

```
is_balanced(node):
        if node == null:
                return true
        # find black height of min value in tree
        black_height = leftmost _black_height(node, -1)
        # use it to check balance
        return is_bal_work(node, -1, black_height)


leftmost_black_height(node, current_black_height):
        if node == null:                            # base case
                return current_black_height + 1 # FIXED to count NULL Links

        if not node.color:                          # it's black. Increase black height.
                current_black_height ← current_black_height + 1

        return leftmost_black_height(node.left, current_black_height)



is_bal_work(node, current_black_height, black_height):
        if node == null
                return current_black_height + 1 == black_height      # FIXED to check NULL Links

        if not node.color:                                           # increase black height
                current_black_height ← current_black_height + 1

        if not is_bal_work(node.left, current_height, black_height):      # recursive calls
                return false

        if not is_bal_work(node.right, current_height, black_height):
                return false

        return true
```

6. Exercise 3.3.38: *Fundamental Theorem of Rotations.* Show that any BST can be transformed into any other BST on the same set of keys by a sequence of left and right rotations.

   Any BST can be rotated right from the root until the minimum element is at the root. At this point, the tree will look like this:

   This can be repeated on the right subtree, then on its right subtree, and so on until every node in the tree has only a right child and we have a worst case BST structure.

   Consider BST A and B with the same set of keys. To transform BST A into BST B:

   Perform the right rotations described above on BST B to transform it into a tree in which every node has only a right child. Keep track of the number of right rotations performed at each node position.

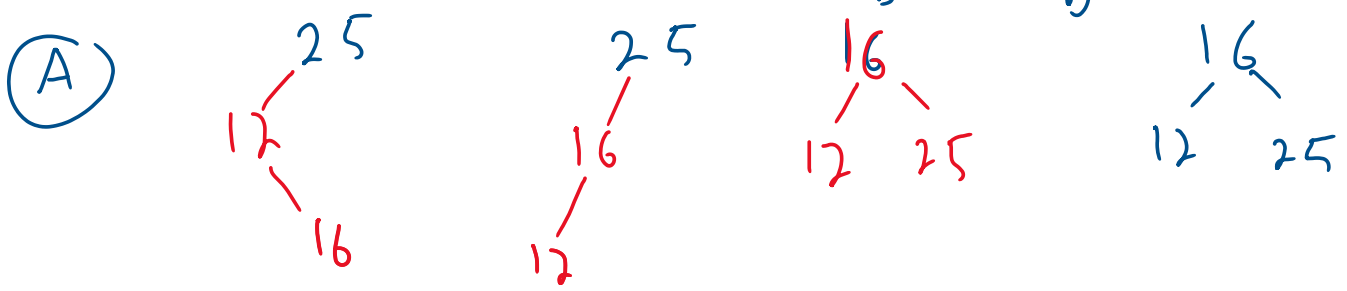
   Then preform the necessary right rotations on BST A to transform it into a tree with only right children. At this point BST A and B are in the same form.

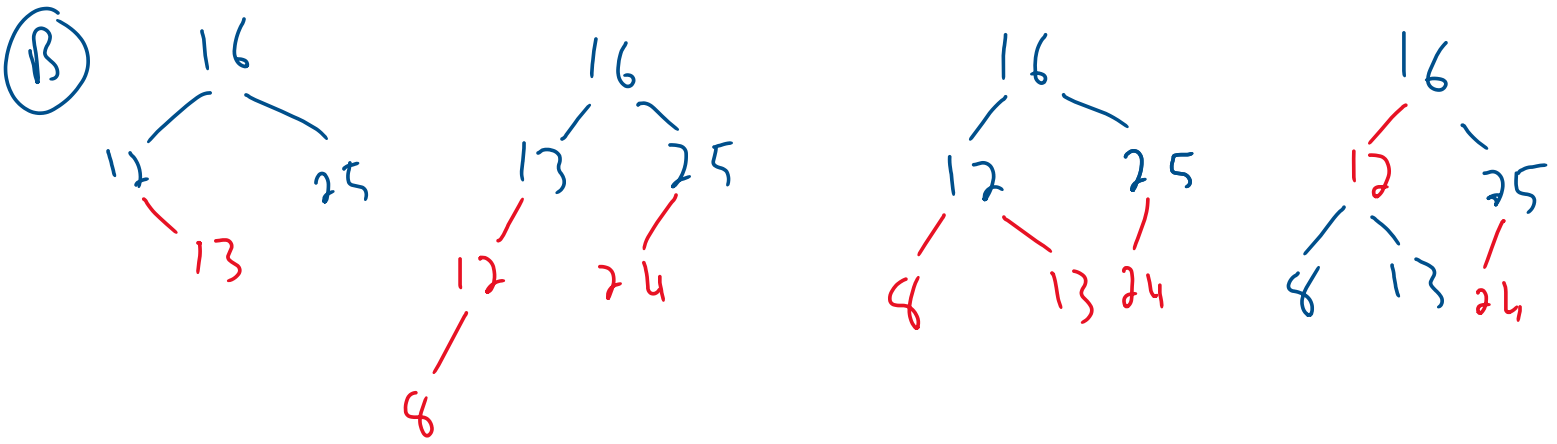
   Now start at the bottom of the tree and perform the same number of left rotations at each node, moving up the tree. This will reverse the right rotations that transformed BST B so at the end of this process, both trees will be in the form of the original BST B

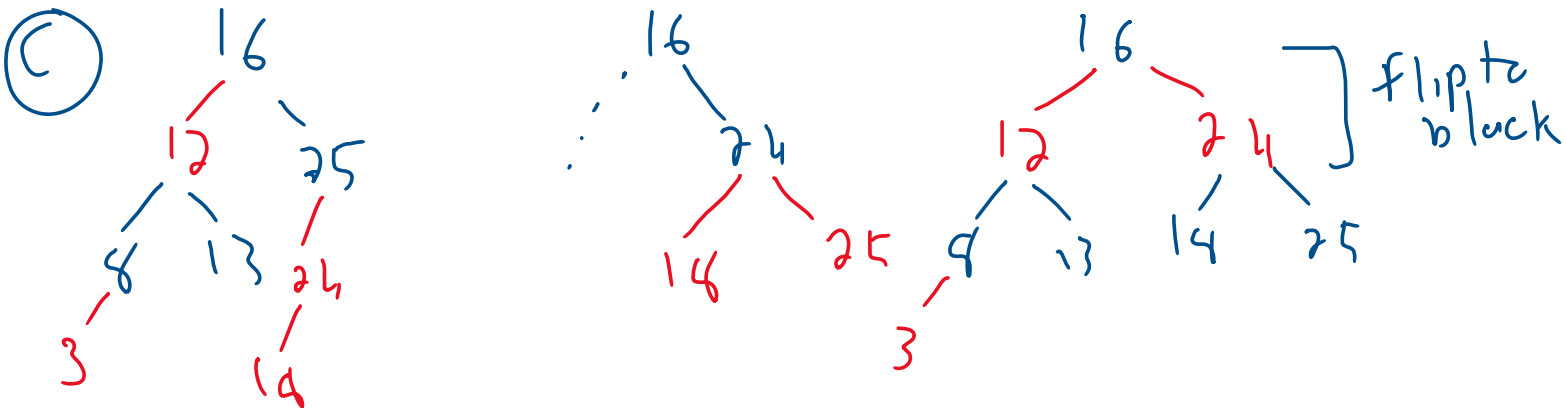
Lecture 3: Left-Leaning Red-Black Insert

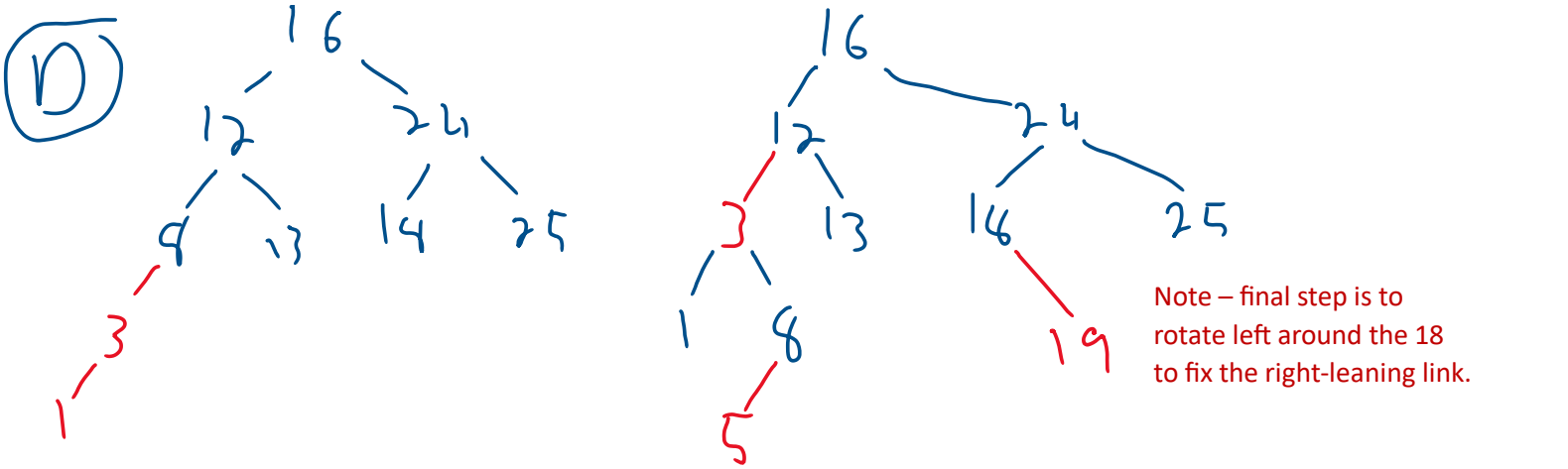
7. Exercise 3.3.10: Simulate the insertion of the keys 25, 12, 16, 13, 24, 8, 3, 18, 1, 5, 19 in that order into an initially empty tree left-leaning red-black BST.

A B C D









Note – final step is to rotate left around the 18 to fix the right-leaning link.

8. Exercise 3.3.13: If you insert keys in increasing order into a left-leaning red-black BST, does the tree height increase monotonically? Does your answer change if you insert keys in decreasing order?

   Height increases monotonically in increasing order but not decreasing order: https://algs4.cs.princeton.edu/33balanced/ (scroll down to the exercise solutions 14 and 15 for a video)

   Note: If you insert in descending order, the 7th insert causes tree height to decrease by 1. Try it!

9. Exercise 3.3.14: Draw the left-leaning red-black BSTs that result when you insert numbers 1 through 10 in order into an initially empty tree. What happens in general when trees are built by insertion of keys in ascending order? How about in descending order?

   This is a very general question so it can be answered in a number of ways. Here are some possible points you could raise…

   - Whether you add keys in increasing order or decreasing order to an initially empty RBT, the height of the RBT is balanced after each insert operation, and is always O(log N). Of course, both of these cases would result in worst case structure for a plain BST.
   - Height increases monotonically in increasing order but not decreasing order: https://algs4.cs.princeton.edu/33balanced/ (scroll down to the exercise solutions 14 and 15 for a video)
   - Another thing of note is that inserting in increasing order never causes any right rotations. Inserting in decreasing order never causes any left rotations.
   - What else happens “in general”? 😊