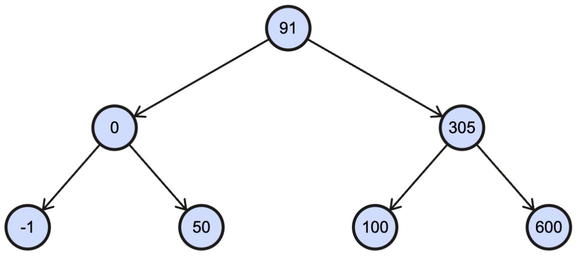
CS 61B MT2 Exam Review Session 2
Spring 2023 March 9, 2022

1 Common Ancestor

Given a binary search tree, we want to find the closest common ancestors of two given nodes `p` and `q` in the BST. The closest common ancestor is defined as the lowest in the tree that has both `p` and `q` as descendants. By this definition, a node is also considered a descendant of itself.

For example, the root node of the following BST:



```
lowestCommonAncestor(root, -1, 50) == 0
lowestCommonAncestor(root, -1, 0) == 0 // a node is also its own descendant
lowestCommonAncestor(root, 100, 305) == 305
lowestCommonAncestor(root, -1, 305) == 91
```

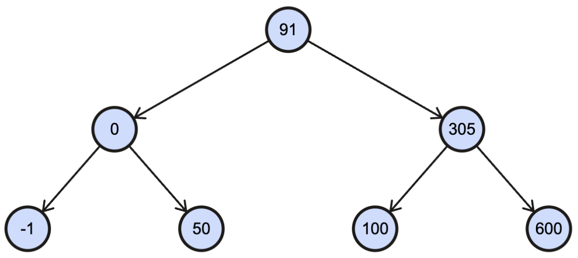
```
public class CommonAncestor {
    public class Node {
        int val;
        Node left, right;
        public Node(int val) { this.val = val; }
    }

    public Node closestCommonAncestor(Node root, Node p, Node q) {
        if (____________________________________________) {
            return null;
        }
        int minVal = Math.min(p.val, q.val);
        int maxVal = ____________________________________________;
        if (root.val < minVal) {
            ____________________________________________________;
        } else if (_______________________________________) {
            ____________________________________________________;
        }
        ____________________________________________________;
    }
}
```

2 All About Balance

Given a sorted array `arr`, fill out the function `makeBalancedBST` that converts the sorted array into a perfectly balanced binary search tree and returns the root node of the resulting tree. A binary search tree is balanced when, for a total of `N` nodes, the height of the tree is roughly $\log N$.

For example, given the input `arr = [-1, 0, 50, 91, 100, 305, 600]`, a call to `makeBalancedBST(arr)` would return the root node of the following BST:



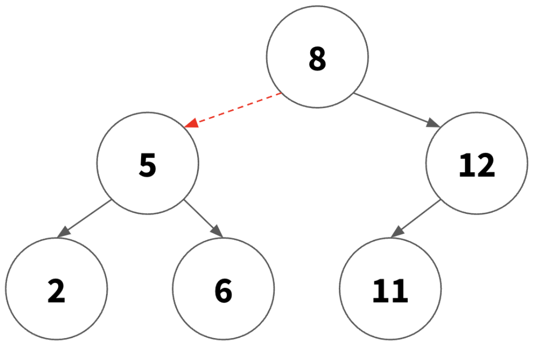
```
public class AllAboutBalance {
    public class Node {
        int val;
        Node left;
        Node right;
        public Node(int val) { this.val = val; }
    }

    public Node makeBalancedBST(int[] arr) {
        return ___________________________________________;
    }

    public Node makeBalancedBSTHelper(int[] arr, int lo, int hi) {
        if (_____________________________________________) {
            return null;
        }
        int mid = ________________________________________;
        Node curr = ______________________________________;
        __________________________________________________;
        __________________________________________________;
        return curr;
    }
}
```

3 LLRBs and 2-3 Trees

Given the LLRB (left-leaning red black tree) below, draw the corresponding 2-3 tree. Perform the following inserts on both the LLRB and 2-3 tree. For inserts to the LLRB, write the sequence of operations used to maintain LLRB properties (rotateLeft, rotateRight, or colorFlip). Assume that inserts automatically set the root color to black after all operations are performed (like in Homework 8).



(a) Draw the corresponding 2-3 tree.

(b) Insert 3 into the 2-3 tree you found above and draw the resulting tree.

(c) Now insert 4 and draw and resulting tree. Then, convert this 2-3 tree into its corresponding LLRB Tree.

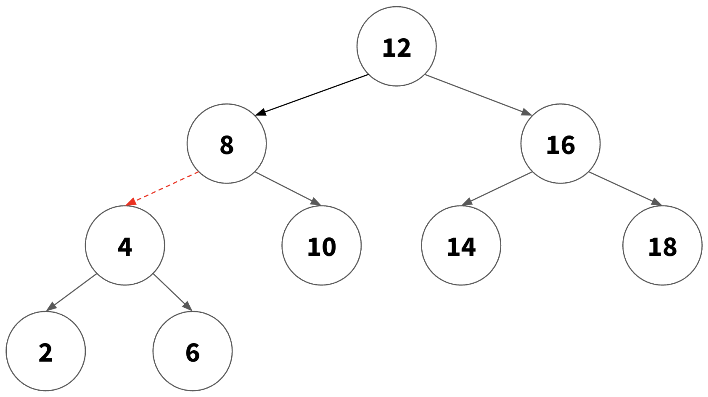
4 LLRB Operations

Perform the following insertions on the Left Leaning Red Black Tree (LLRB) given below. For each insertion, give the fix up operations needed. Recall a fix up operation is one of the following:

- `rotateLeft`
- `rotateRight`
- `colorFlip`
- change the root node to black.

Note that insertions are **dependent**. If only two operations are necessary, pick "None" for the third operation. If only one operation is necessary, pick "None" for the second and third operation. If no operations are necessary, pick "None" for all three operations.

If you put "None" for the "Operation applied", **leave the "Node to apply on" blank.**



a) Insert 17

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	

b) Insert 15. Note that insertions are dependent, so insert 15 into the state of the LLRB after the insertion of 17.

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ None	

c) Insert 13. Note that insertions are dependent, so insert 13 into the state of the LLRB after the insertion of 15.

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	

d) Insert 19. Note that insertions are dependent, so insert 19 into the state of the LLRB after the insertion of 13.

	Operation applied	Node to apply on
1st operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	
2nd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	
3rd operation	◯ `rotateLeft()` ◯ `rotateRight()` ◯ `colorFlip()` ◯ change root to black ◯ `None`	

5 Fill in the Blanks

Fill in the following blanks related to min-heaps. Let `N` is the number of elements in the min-heap. For the entirety of this question, assume the elements in the min-heap are **distinct**.

1. `removeMin` has a best case runtime of __________________ and a worst case runtime of __________________.
2. `insert` has a best case runtime of __________________ and a worst case runtime of __________________.
3. A __________________ or __________________ traversal on a min-heap *may* output the elements in sorted order. Assume there are at least 3 elements in the min-heap.
4. The fourth smallest element in a min-heap with 1000 elements can appear in __________________ places in the heap.
5. Given a min-heap with $2^N - 1$ distinct elements, for an element
   - to be on the second level it must be less than __________________ element(s) and greater than __________________ element(s).
   - to be on the bottommost level it must be less than __________________ element(s) and greater than __________________ element(s).

   Hint: A complete binary tree (with a full last-level) has $2^N - 1$ elements, with N being the number of levels.

6 Summer 2021 Final Q6

(a) Suppose we have the min-heap below (represented as an array) with **distinct** elements, where the values of A and B are unknown. Note that A and B aren't necessarily integers.

`{1, A, 3, 5, 9, 11, 13, 10, B}`

What can we say about the relationships between the following elements? Put >, <, or `?` if the answer is not known.

A ○ > ○ < ○ ? 1

A ○ > ○ < ○ ? 3

B ○ > ○ < ○ ? 10

A ○ > ○ < ○ ? B

(b) Note for both parts below, the values of `A` and `B` should **not** violate the min-heap properties. Put `-inf` or `inf` if there isn't a lower or upper bound, respectively. If the bound for `B` depends on the value of `A`, or vice versa, you may put the variable in the bound, e.g. `A < B`.

Considering **one `removeMin`** call, put **tight** bounds on `A` and `B` such that:

- We perform the **maximum** number of swaps.

  ______ < **`A`** < ______

  ______ < **`B`** < ______

- We perform the **minimum** number of swaps.

  ______ < **`A`** < ______

  ______ < **`B`** < ______