COL106 : Data Structures and Algorithms, Semester II 2024-25
Practice Programming Questions on AVL Trees & Red Black Trees

February 2025

Instructions

- Please use the following questions as practice questions for AVL Trees and Red Black Trees.
- The questions with * next to them should be attempted during the lab sessions, and your solutions must be uploaded on moodlenew. Note that your submissions will not be evaluated, but will be used as a mark of your attendance. We will filter out all the submissions that are not from the lab workstations. So do not use your laptops for submitting the programs.

Questions

1. *** Check if a BST is an AVL Tree.**

   Implement a function to check if a BST rooted at `TreeNode root` is an AVL Tree in $\mathcal{O}(N)$ time complexity, where N is the number of nodes in the BST. Implement helper functions as needed.

```java
public class TreeNode {

    int val;
    TreeNode left;
    TreeNode right;

    /**
     * Constructor of TreeNode
     * Time Complexity: O(1)
     */

    TreeNode(int ele);

    /**
     * Checks if BST rooted at this TreeNode is an AVL Tree.
     * Time Complexity: O(N)
     * where N is the number of nodes in the BST.
     */

    public boolean check();

}
```

Listing 1: TreeNode implementation in TreeNode.java

2. *** Implement an AVL Tree from scratch.**

Implement the given methods of the class signature to obtain a self-balancing AVL tree. Feel free to make use of your implementation in Q1. Implement helper functions as needed.

```java
public class AVLTree {

    TreeNode root;

    /**
     * Initializes an empty AVL Tree.
     * Time Complexity: O(1)
     */
    public AVLTree();

    /**
     * Searches an element in the tree.
     * Time Complexity: O(log N)
     * where N is the number of elements in the tree
     */
    public boolean search(int ele);

    /**
     * Inserts a new element into the tree, maintaining the AVL property.
     * Assume that the element does not exist in the tree.
     * Time Complexity: O(log N)
     * where N is the number of elements in the tree
     */
    public void insert(int ele);

    /**
     * Deletes a specific element from the tree, maintaining the AVL property.
     * Assume that the element exists in the tree and is distinct.
     * Time Complexity: O(log N)
     * where N is the number of elements in the tree
     */
    public void delete(int ele);

}
```

Listing 2: AVLTree implementation in AVLTree.java

3. **Construct an AVL Tree from a sorted array.**

Use constructor overloading in `AVLTree` to construct an AVL Tree from a sorted array in $\mathcal{O}(N)$ time complexity, where N is the number of elements in the sorted array. You may debug using the function implemented in Q1.

Constructor overloading, in simple terms, is used to instantiate an object of a class with different input arguments. Hence, here, the function signature would be `public AVLTree(int[] sorted_arr)` inside `AVLTree` class.

Hint: Inorder insertion of elements of sorted array into an empty AVL Tree will violate the imposed time complexity bound.

4. **Sort an array using an AVL Tree.**

Modify the constructor `public AVLTree(int[] sorted_arr)` inside `AVLTree` class to `public AVLTree(int[] arr, boolean sorted)` to construct AVLTree if `sorted` is `true` from a sorted array as in Q3 else from

an unsorted array. Implement `public int[] sorted_arr()` inside `AVLTree` class that returns the sorted array.

Notice that if the same logic were to be used for sorting via an ordinary BST, the worst case time complexity would be $\mathcal{O}(N^2)$ where N is the number of nodes in the unsorted array. Sorting by AVL Trees bounds the worst case time complexity to $\mathcal{O}(N \log N)$.

Hint: Here is where inorder insertion comes into play. But, which traversal should be used to get the elements back, that too, in order? ;)

5. *** Implementation of Red Black trees.**

Implement the RedBlackTree class as described below. Feel free to define helper methods as needed.

```
/**
 * Color enum for Red-Black Tree
 */
enum Color {
    RED, BLACK
}

/**
 * Node class for the Red-Black Tree
 */
class Node {
    int data;        // Key value stored in the node
    Node parent;     // Reference to the parent node
    Node left;       // Reference to the left child
    Node right;      // Reference to the right child
    Color color;     // Color of the node (Color.RED or Color.BLACK)
}

/**
 * Red-Black Tree implementation
 */
public class RedBlackTree {
    private Node root;   // Root of the tree

    /**
     * Constructor to initialize an empty Red-Black Tree
     * runtime O(1)
     */
    public RedBlackTree() {

    }


    /**
     * Searches for a node with the given key in the tree
     *
     * k The key to search for
     * return The node containing the key if found,
     otherwise TNULL
      * runtime O(log N) N is number of nodes
     */
    public Node searchTree(int k) {
        // Implementation here
        return null;
```

```
    }



    /**
     * Inserts a new node with the given key into the tree
     * and maintains the Red-Black Tree properties
     *
     * key The key to insert
      * runtime O(log N) N is number of nodes
     */
    public void insert(int key) {
        // Implementation here
    }


    /**
     * Deletes the node with the given key from the tree
     * and maintains the Red-Black Tree properties
     *
     * data The key to delete
      * runtime O(log N) N is number of nodes
     */
    public void deleteNode(int data) {
        // Implementation here
    }


}
```

Listing 3: Red Black tree implementation

6. *** Check if given tree is Red Black or not.**

   Given the root of a tree with nodes of `class Node` as defined in previous problem, check if the tree satisfies all properties of a Red Black tree or not. Implement a method `public boolean checkRedBlackProperties(Node node)` in the same `RedBlackTree class` defined in previous problem.

   Time Complexity : $O(N)$ where N is number of nodes in tree.

7. **Join two Red Black trees.**

   Given the roots of two Red Black trees $T1$ and $T2$, such that all node values of $T1$ are smaller than all node values of $T2$, write a method `public static RedBlackTree join(RedBlackTree T1, RedBlackTree T2)` in `RedBlackTree class` (defined in Problem 5) to join both of them into a single tree T which is also a Red Black tree.

   Time Complexity : $O(h1 + h2)$ where $h1, h2$ are heights of $T1, T2$ respectively.