**DAY17 – July 12th**

**AVL TREE**

**AVL Insertion Algorithm**

1. Create a Node
2. Check if the tree is empty or not
3. If the tree is empty, the inserted node will be the root node.
4. If the tree is not empty, do a binary search tree insertion Operation.
5. Update the height of the ancestor node
6. Calculate the balance factor of each node
7. If unbalanced, perform one of the 4 rotations:

* LL (Left-Left) → Right Rotation
* RR (Right-Right) → Left Rotation
* LR (Left-Right) → Left Rotation + Right Rotation
* RL (Right-Left) → Right Rotation + Left Rotation

**Code for AVL Tree**

class Node15 {  
 int key, height;  
 Node15 left, right;  
  
 Node15(int d) {  
 key = d;  
 height = 1; // New node is initially added at leaf  
 }  
}  
public class AVL\_Insertion  
{  
 Node15 root;  
 int height(Node15 N)  
 {  
 if (N == null) return 0;  
 return N.height;  
 }  
 // Get max of two numbers  
 int max(int a, int b)  
 {  
 return (a > b) ? a : b;  
 }  
 // Right rotate  
 Node15 rightRotate(Node15 y) {  
 Node15 x = y.left;  
 Node15 T2 = x.right;  
  
 // Rotation  
 x.right = y;  
 y.left = T2;  
  
 // Update heights  
 y.height = max(height(y.left), height(y.right)) + 1;  
 x.height = max(height(x.left), height(x.right)) + 1;  
  
 // Return new root  
 return x;  
 }  
 Node15 leftRotate(Node15 x) {  
 Node15 y = x.right;  
 Node15 T2 = y.left;  
  
 // Rotation  
 y.left = x;  
 x.right = T2;  
  
 // Update heights  
 x.height = max(height(x.left), height(x.right)) + 1;  
 y.height = max(height(y.left), height(y.right)) + 1;  
  
 // Return new root  
 return y;  
 }  
 // Get Balance factor  
 int getBalance(Node15 N)  
 {  
 if (N == null) return 0;  
 return height(N.left) - height(N.right);  
 }  
 Node15 insert(Node15 node, int key) {  
 // 1. Perform normal BST insertion  
 if (node == null)  
 return new Node15(key);  
  
 if (key < node.key)  
 node.left = insert(node.left, key); // if key is less than current node insert at left subtree.  
 else if (key > node.key)  
 node.right = insert(node.right, key);// if key is greater than current node insert at right subtree.  
 else // Duplicate keys not allowed  
 return node;  
  
 // 2. Update height  
 node.height = 1 + max(height(node.left), height(node.right));  
  
 // 3. Get balance factor  
 int balance = getBalance(node);  
  
 // 4. Balance the tree if unbalanced  
  
 // LL Case  
 if (balance > 1 && key < node.left.key)  
 return rightRotate(node);  
  
 // RR Case  
 if (balance < -1 && key > node.right.key)  
 return leftRotate(node);  
  
 // LR Case  
 if (balance > 1 && key > node.left.key) {  
 node.left = leftRotate(node.left);  
 return rightRotate(node);  
 }  
  
 // RL Case  
 if (balance < -1 && key < node.right.key) {  
 node.right = rightRotate(node.right);  
 return leftRotate(node);  
 }  
  
 return node;  
 }  
  
 // Inorder traversal  
 void inorder(Node15 node) {  
 if (node != null) {  
 inorder(node.left);  
 System.*out*.print(node.key + " ");  
 inorder(node.right);  
 }  
 }  
  
 // Main method  
 public static void main(String[] args) {  
 AVL\_Insertion tree = new AVL\_Insertion();  
  
 int[] Values = {10, 20, 50, 40, 30, 25};  
  
 for (int key : Values) {  
 tree.root = tree.insert(tree.root, key);  
 }  
  
 System.*out*.println("Inorder traversal of AVL tree:");  
 tree.inorder(tree.root);  
 }  
}

**Output**

Inorder traversal of AVL tree:

10 20 25 30 40 50

**NOTES: When search is frequent, Insertion and Deletion are rare we use AVL Tree.**

**RED – BLACK TREES**

**TASK 3 – Algorithm for Red-Black tree**

Insert an Element - Red Black Tree −

1. Check tree is empty. If empty, then insert new node - color Black. (Because Root Node - Black in color)

2. else if Tree - not empty then insert new node as leaf node to the end and color - Red.

3. If parent of new node is Red and its neighbour’s(parent’s) node is also Red,

then Flip the color of the both neighbor and Parent and Grandparents (If it is not Root Node Otherwise Flip the color of the Parent and neighbor only) i.e., Black.

4. If parent of new node is Red and its neighbours(parent’s) node is empty or NULL,

then Rotate (either Left-Left or Left-Right rotation) the new node and parent.

5. we have two types of rotation

- Left Left Rotation and

- Left Right Rotation.

6. we apply Rotation in some conditions only.

The conditions are −

- If parent of new node is Red and neighbour node is empty or NULL, then rotate left or right rotation.

- In Left-Left Rotation flip the color of the parent and grandparent.

Make the parent as Grandparent and grandparent as child

**TASK 4 – Code for Red-Black Trees**

class Day17\_Red\_BlackTree {  
 private static final boolean *RED* = true;  
 private static final boolean *BLACK* = false;  
  
 // Node structure  
 class Node {  
 int key;  
 Node left, right, parent;  
 boolean color;  
  
 Node(int key) {  
 this.key = key;  
 this.color = *RED*;  
 }  
 }  
  
 private Node root;  
  
 // Left Rotate  
 private void rotateLeft(Node x) {  
 Node y = x.right;  
 x.right = y.left;  
 if (y.left != null) y.left.parent = x;  
  
 y.parent = x.parent;  
  
 if (x.parent == null) root = y;  
 else if (x == x.parent.left) x.parent.left = y;  
 else x.parent.right = y;  
  
 y.left = x;  
 x.parent = y;  
 }  
  
 // Right Rotate  
 private void rotateRight(Node y) {  
 Node x = y.left;  
 y.left = x.right;  
 if (x.right != null) x.right.parent = y;  
  
 x.parent = y.parent;  
  
 if (y.parent == null) root = x;  
 else if (y == y.parent.left) y.parent.left = x;  
 else y.parent.right = x;  
  
 x.right = y;  
 y.parent = x;  
 }  
  
 // Insert key  
 public void insert(int key) {  
 Node node = new Node(key);  
 root = bstInsert(root, node);  
 fixViolation(node);  
 }  
  
 // Standard BST insert  
 private Node bstInsert(Node root, Node node) {  
 if (root == null) return node;  
  
 if (node.key < root.key) {  
 root.left = bstInsert(root.left, node);  
 root.left.parent = root;  
 } else if (node.key > root.key) {  
 root.right = bstInsert(root.right, node);  
 root.right.parent = root;  
 }  
 return root;  
 }  
  
 // Fix red-black tree violations  
 private void fixViolation(Node node) {  
 Node parent = null, grandparent = null;  
  
 while (node != root && node.color == *RED* && node.parent.color == *RED*) {  
 parent = node.parent;  
 grandparent = parent.parent;  
  
 // Parent is left child of grandparent  
 if (parent == grandparent.left) {  
 Node uncle = grandparent.right;  
  
 // Case 1: Uncle is RED  
 if (uncle != null && uncle.color == *RED*) {  
 grandparent.color = *RED*;  
 parent.color = *BLACK*;  
 uncle.color = *BLACK*;  
 node = grandparent;  
 } else {  
 // Case 2: Right child  
 if (node == parent.right) {  
 node = parent;  
 rotateLeft(node);  
 }  
  
 // Case 3: Left child  
 parent.color = *BLACK*;  
 grandparent.color = *RED*;  
 rotateRight(grandparent);  
 }  
 } else { // Parent is right child of grandparent  
 Node uncle = grandparent.left;  
  
 if (uncle != null && uncle.color == *RED*) {  
 grandparent.color = *RED*;  
 parent.color = *BLACK*;  
 uncle.color = *BLACK*;  
 node = grandparent;  
 } else {  
 if (node == parent.left) {  
 node = parent;  
 rotateRight(node);  
 }  
  
 parent.color = *BLACK*;  
 grandparent.color = *RED*;  
 rotateLeft(grandparent);  
 }  
 }  
 }  
 root.color = *BLACK*;  
 }  
  
 // Utility: Preorder traversal for debugging  
 public void preOrder(Node node) {  
 if (node != null) {  
 System.*out*.print(node.key + "(" + (node.color ? "R" : "B") + ") ");  
 preOrder(node.left);  
 preOrder(node.right);  
 }  
 }  
  
 public Node getRoot() {  
 return root;  
 }  
  
 public static void main(String[] args) {  
 Day17\_Red\_BlackTree tree = new Day17\_Red\_BlackTree();  
 int[] values = {10, 20, 30, 15, 25, 5};  
  
 for (int val : values) {  
 tree.insert(val);  
 }  
  
 System.*out*.println("Pre-order traversal of Red-Black Tree:");  
 tree.preOrder(tree.getRoot());  
 }  
}

**Output**

Pre-order traversal of Red-Black Tree:

20(B) 10(R) 5(B) 15(B) 30(R) 25(B)