Task1: AVL insertion Algorithm

1. create a node

2. check if tree is empty or not

3. if tree is empty the inserted node will be the root node.

4. if tree is not empty, do a binary search tree insertion op and also check the balance factor of the node.

5. if the balance factor exceeds 1, we should do rotations on the heavy weighted tree and repeat the insertion from step 4 onwards.

AVL search Algo

1 − Create a node

2 − Check if tree is empty

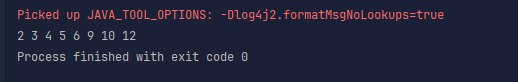
3 − If tree is empty, new node is root node.

4 − not empty, perform Binary Search Tree insertion operation and check balancing factor of the node in the tree.

5 − Suppose balancing factor > apply rotations on node and resume insertion from Step 4.

Task2:

package **Day17**;  
  
class **AVLNode**{  
 int key;  
 int height;  
 **AVLNode** left;  
 **AVLNode** right;  
 public AVLNode(int key){  
 this.key = key;  
 this.height = 1;  
 this.left = null;  
 this.right = null;  
 }  
}  
class **AVLTree**{  
 **AVLNode** root;  
 static int height(**AVLNode** node){  
 if (node == null){  
 return 0;  
 }  
 return node.height;  
 }  
 static int getBalance(**AVLNode** node){  
 if (node==null){  
 return 0;  
 }  
 return *height*(node.left) - *height*(node.right);  
 }  
 static void updateHeight(**AVLNode** node){  
 if (node != null){  
 node.height = 1+ **Math**.*max*(*height*(node.left), *height*(node.right));  
 }  
 }  
 static **AVLNode** rightRotate(**AVLNode** y) {  
 **AVLNode** x = y.left;  
 **AVLNode** T2 = x.right;  
  
 x.right = y;  
 y.left = T2;  
  
 *updateHeight*(y);  
 *updateHeight*(x);  
  
 return x;  
 }  
  
 static **AVLNode** leftRotate(**AVLNode** x) {  
 **AVLNode** y = x.right;  
 **AVLNode** T2 = y.left;  
  
 y.left = x;  
 x.right = T2;  
  
 *updateHeight*(x);  
 *updateHeight*(y);  
  
 return y;  
 }  
  
 static **AVLNode** insert(**AVLNode** node, int key) {  
 if (node == null) {  
 return new AVLNode(key);  
 }  
  
 if (key < node.key) {  
 node.left = *insert*(node.left, key);  
 } else if (key > node.key) {  
 node.right = *insert*(node.right, key);  
 } else {  
 return node;  
 }  
  
 *updateHeight*(node);  
  
 int balance = *getBalance*(node);  
  
 if (balance > 1 && key < node.left.key) {  
 return *rightRotate*(node);  
 }  
  
 if (balance < -1 && key > node.right.key) {  
 return *leftRotate*(node);  
 }  
  
 if (balance > 1 && key > node.left.key) {  
 node.left = *leftRotate*(node.left);  
 return *rightRotate*(node);  
 }  
  
 if (balance < -1 && key < node.right.key) {  
 node.right = *rightRotate*(node.right);  
 return *leftRotate*(node);  
 }  
  
 return node;  
 }  
  
 public void insert(int key) {  
 this.root = *insert*(this.root, key);  
 }  
  
 public **AVLNode** search(int key) {  
 **AVLNode** current = this.root;  
 while (current != null) {  
 if (key == current.key) {  
 return current;  
 } else if (key < current.key) {  
 current = current.left;  
 } else {  
 current = current.right;  
 }  
 }  
 return null;  
 }  
  
 public void inorderTraversal(**AVLNode** node) {  
 if (node != null) {  
 inorderTraversal(node.left);  
 **System**.*out*.print(node.key + " ");  
 inorderTraversal(node.right);  
 }  
 }  
  
}  
public class **Task2** {  
 public static void main(**String**[] args) {  
 **AVLTree** tree = new AVLTree();  
 tree.insert(2);  
 tree.insert(4);  
 tree.insert(3);  
 tree.insert(6);  
 tree.insert(9);  
 tree.insert(10);  
 tree.insert(5);  
 tree.insert(12);  
 tree.inorderTraversal(tree.root);  
 }  
}



Task3:

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 neighbours(parent’s) node is also Red,

then Flip the color of the both neighbour and Parent and Grandparents (If it is not Root Node Otherwise Flip the color of the Parent and neighbour 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