**PROGRAM:**

**import** java.util.\*;

**import** BinaryST.Node;

**class** Node

{

**int** key, height;

Node left, right;

**public** **int** data;

Node(**int** d)

{

key = d;

height = 1;

}

}

**class** AVLTree

{

Node root;

// A utility function to get height of the tree

**int** height(Node N)

{

**if** (N == **null**)

**return** 0;

**return** N.height;

}

// A utility function to get maximum of two integers

**int** max(**int** a, **int** b)

{

**return** (a > b) ? a : b;

}

// A utility function to right rotate subtree rooted with y

Node rightRotate(Node y)

{

Node x = y.left;

Node T2 = x.right;

// Perform 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;

}

// A utility function to left rotate subtree rooted with x

Node leftRotate(Node x)

{

Node y = x.right;

Node T2 = y.left;

// Perform 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 of node N

**int** getBalance(Node N)

{

**if** (N == **null**)

**return** 0;

**return** height(N.left) - height(N.right);

}

Node insert(Node node, **int** key)

{

/\* 1. Perform the normal BST rotation \*/

**if** (node == **null**)

**return** (**new** Node(key));

**if** (key < node.key)

node.left = insert(node.left, key);

**else** **if** (key > node.key)

node.right = insert(node.right, key);

**else** // Equal keys not allowed

**return** node;

/\* 2. Update height of this ancestor node \*/

node.height = 1 + max(height(node.left),

height(node.right));

/\* 3. Get the balance factor of this ancestor

node to check whether this node became

Wunbalanced \*/

**int** balance = getBalance(node);

// If this node becomes unbalanced, then

// there are 4 cases Left Left Case

**if** (balance > 1 && key < node.left.key)

**return** rightRotate(node);

// Right Right Case

**if** (balance < -1 && key > node.right.key)

**return** leftRotate(node);

// Left Right Case

**if** (balance > 1 && key > node.left.key)

{

node.left = leftRotate(node.left);

**return** rightRotate(node);

}

// Right Left Case

**if** (balance < -1 && key < node.right.key)

{

node.right = rightRotate(node.right);

**return** leftRotate(node);

}

/\* return the (unchanged) node pointer \*/

**return** node;

}

/\* Given a non-empty binary search tree, return the

node with minimum key value found in that tree.

Note that the entire tree does not need to be

searched. \*/

Node minValueNode(Node node)

{

Node current = node;

/\* loop down to find the leftmost leaf \*/

**while** (current.left != **null**)

current = current.left;

**return** current;

}

Node deleteNode(Node root, **int** key)

{

// STEP 1: PERFORM STANDARD BST DELETE

**if** (root == **null**)

**return** root;

// If the key to be deleted is smaller than

// the root's key, then it lies in left subtree

**if** (key < root.key)

root.left = deleteNode(root.left, key);

// If the key to be deleted is greater than the

// root's key, then it lies in right subtree

**else** **if** (key > root.key)

root.right = deleteNode(root.right, key);

// if key is same as root's key, then this is the node

// to be deleted

**else**

{

// node with only one child or no child

**if** ((root.left == **null**) || (root.right == **null**))

{

Node temp = **null**;

**if** (temp == root.left)

temp = root.right;

**else**

temp = root.left;

// No child case

**if** (temp == **null**)

{

temp = root;

root = **null**;

}

**else** // One child case

root = temp; // Copy the contents of

// the non-empty child

}

**else**

{

// node with two children: Get the inorder

// successor (smallest in the right subtree)

Node temp = minValueNode(root.right);

// Copy the inorder successor's data to this node

root.key = temp.key;

// Delete the inorder successor

root.right = deleteNode(root.right, temp.key);

}

}

// If the tree had only one node then return

**if** (root == **null**)

**return** root;

// STEP 2: UPDATE HEIGHT OF THE CURRENT NODE

root.height = max(height(root.left), height(root.right)) + 1;

// STEP 3: GET THE BALANCE FACTOR OF THIS NODE (to check whether

// this node became unbalanced)

**int** balance = getBalance(root);

// If this node becomes unbalanced, then there are 4 cases

// Left Left Case

**if** (balance > 1 && getBalance(root.left) >= 0)

**return** rightRotate(root);

// Left Right Case

**if** (balance > 1 && getBalance(root.left) < 0)

{

root.left = leftRotate(root.left);

**return** rightRotate(root);

}

// Right Right Case

**if** (balance < -1 && getBalance(root.right) <= 0)

**return** leftRotate(root);

// Right Left Case

**if** (balance < -1 && getBalance(root.right) > 0)

{

root.right = rightRotate(root.right);

**return** leftRotate(root);

}

**return** root;

}

// A utility function to print preorder traversal of

// the tree. The function also prints height of every

// node

**void** preOrder(Node node)

{

**if** (node != **null**)

{

System.***out***.print(node.key + " ");

preOrder(node.left);

preOrder(node.right);

}

}

//find a key

**public** **static** **boolean** search(Node root, **int** key) {

**if**(root==**null**) { //no tree

**return** **false**;

}

**else** **if**(root.data==key){ // root is te key

**return** **true**;

}

**else** **if**(key<root.data) { //key is less tn root

**return** *search*(root.left,key);

}

**else**

**return** *search*(root.right,key); // key is greatr then root

}

**public** **static** **void** main(String[] args)

{

Scanner sc = **new** Scanner(System.***in***);

AVLTree tree = **new** AVLTree();

**int** arr[]=**new** **int**[9];

System.***out***.println("enter elemnts in tree");

**for**(**int** i=0;i<arr.length;i++) {

arr[i]=sc.nextInt();

}

//insertin values in te tree

**for**(**int** i=0;i<arr.length;i++) {

tree.root=tree.insert(tree.root,arr[i]);

}

/\* The constructed AVL Tree would be

40

/ \

30 60

/ \ / \

25 35 50 78

/ \

20 28

\*/

System.***out***.println("Preorder traversal of "+

"constructed tree is : ");

tree.preOrder(tree.root);

System.***out***.println();

//deletion

System.***out***.println("enter the elemnt to be deleted:");

**int** del1=sc.nextInt();

tree.root = tree.deleteNode(tree.root, del1); //call delete function

/\* The AVL Tree after deletion of 28

40

/ \

30 60

/ \ / \

25 35 50 78

/

20

\*/

System.***out***.println("Preorder traversal after "+

"deletion of "+del1+ ":");

tree.preOrder(tree.root);

System.***out***.println();

System.***out***.println("enter the elemnt to be deleted:");

**int** del2=sc.nextInt();

tree.root = tree.deleteNode(tree.root, del2); //call delete function

/\* The AVL Tree after deletion of 60

40

/ \

30 78

/ \ /

25 35 50

/

20

\*/

System.***out***.println("");

System.***out***.println("Preorder traversal after "+

"deletion of "+del2+ " :");

tree.preOrder(tree.root);

System.***out***.println();

//finding a key

System.***out***.println("enter the key to be searched:");

**int** key = sc.nextInt();

**if**(*search*(tree.root,key)) {

System.***out***.println("key found");

}

**else**

System.***out***.println("key not found");

}

}

**OUTPUT:**

enter elemnts in tree

35

50

40

25

30

60

78

20

28

Preorder traversal of constructed tree is :

40 30 25 20 28 35 60 50 78

enter the elemnt to be deleted:

28

Preorder traversal after deletion of 28:

40 30 25 20 35 60 50 78

enter the elemnt to be deleted:

60

Preorder traversal after deletion of 28 :

40 30 25 20 35 78 50

enter the key to be searched:

60

key not found