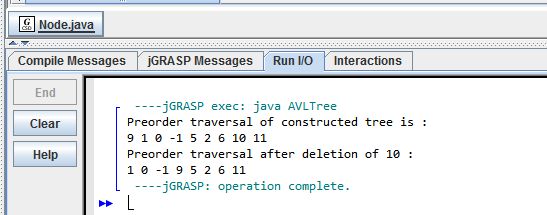
CSCI 321 Computer Science III Summer 2019

Assignment 5

**AVL Tree**

1. Fill in the missing part in the insert method in **Code 1** (attached in the next page).
2. Fill in the missing part in the deleteNode method in **Code 1**.



// Java program for AVL Tree   
  
class Node   
{   
 int key, height;   
 Node left, right;   
  
 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   
 // See the diagram given above.   
 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   
 // See the diagram given above.   
 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   
 unbalanced \*/  
 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.left.key)  
 return rightRotate(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.left.key)  
 {  
 node.right = rightRotate(node.right);  
 return rightRotate(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);   
 }   
 }   
  
 public static void main(String[] args)   
 {   
 AVLTree tree = new AVLTree();   
  
 /\* Constructing tree given in the above figure \*/  
 tree.root = tree.insert(tree.root, 9);   
 tree.root = tree.insert(tree.root, 5);   
 tree.root = tree.insert(tree.root, 10);   
 tree.root = tree.insert(tree.root, 0);   
 tree.root = tree.insert(tree.root, 6);   
 tree.root = tree.insert(tree.root, 11);   
 tree.root = tree.insert(tree.root, -1);   
 tree.root = tree.insert(tree.root, 1);   
 tree.root = tree.insert(tree.root, 2);   
  
 /\* The constructed AVL Tree would be   
 9   
 / \   
 1 10   
 / \ \   
 0 5 11   
 / / \   
 -1 2 6   
 \*/  
 System.out.println("Preorder traversal of "+   
 "constructed tree is : ");   
 tree.preOrder(tree.root);   
  
 tree.root = tree.deleteNode(tree.root, 10);   
  
 /\* The AVL Tree after deletion of 10   
 1   
 / \   
 0 9   
 / / \   
 -1 5 11   
 / \   
 2 6   
 \*/  
 System.out.println("");   
 System.out.println("Preorder traversal after "+   
 "deletion of 10 :");   
 tree.preOrder(tree.root);   
 }   
}

**Splay Tree**

1. Assume that you have inserted a sequence of keys into a Splay Tree:

9, 2, 90, 53, 4, 64, 95, 59

Draw the step-by-step workflow after searching for key 4. You need to start by showing the splay tree containing all the keys above.

maxDepth()

1. If tree is empty then return 0

2. Else

(a) Get the max depth of left subtree recursively i.e.,

call maxDepth( tree->left-subtree)

(a) Get the max depth of right subtree recursively i.e.,

call maxDepth( tree->right-subtree)

(c) Get the max of max depths of left and right

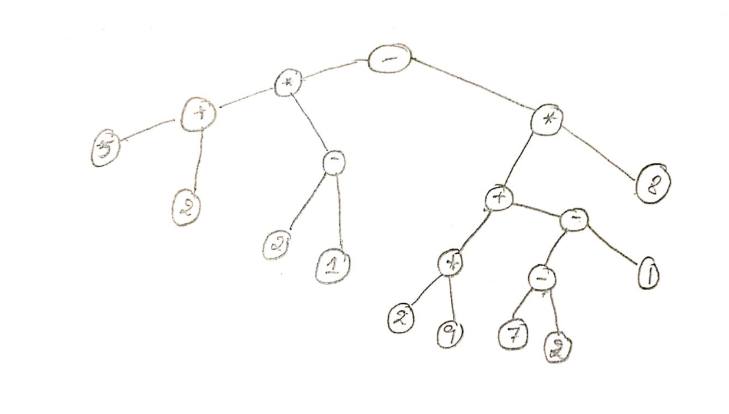
subtrees and add 1 to it for the current node.

max\_depth = max(max dept of left subtree,

max depth of right subtree)

+ 1

(d) Return max\_depth



**Code 1**

// Java program for AVL Tree

class Node

{

int key, height;

Node left, right;

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

// See the diagram given above.

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

// See the diagram given above.

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

unbalanced \*/

int balance = getBalance(node);

// If this node becomes unbalanced, then

// there are 4 cases

// Left Left Case

**Add your code here**

// Right Right Case

**Add your code here**

// Left Right Case

**Add your code here**

// Right Left Case

**Add your code here**

/\* 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

**Add your code here**

// Left Right Case

**Add your code here**

// Right Right Case

**Add your code here**

// Right Left Case

**Add your code here**

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);

}

}

public static void main(String[] args)

{

AVLTree tree = new AVLTree();

/\* Constructing tree given in the above figure \*/

tree.root = tree.insert(tree.root, 9);

tree.root = tree.insert(tree.root, 5);

tree.root = tree.insert(tree.root, 10);

tree.root = tree.insert(tree.root, 0);

tree.root = tree.insert(tree.root, 6);

tree.root = tree.insert(tree.root, 11);

tree.root = tree.insert(tree.root, -1);

tree.root = tree.insert(tree.root, 1);

tree.root = tree.insert(tree.root, 2);

/\* The constructed AVL Tree would be

9

/ \

1 10

/ \ \

0 5 11

/ / \

-1 2 6

\*/

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

"constructed tree is : ");

tree.preOrder(tree.root);

tree.root = tree.deleteNode(tree.root, 10);

/\* The AVL Tree after deletion of 10

1

/ \

0 9

/ / \

-1 5 11

/ \

2 6

\*/

System.out.println("");

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

"deletion of 10 :");

tree.preOrder(tree.root);

}

}