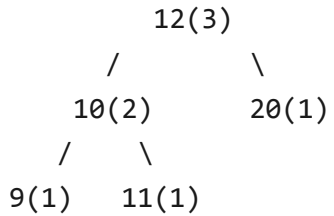
AVL with duplicate keys

Please refer below post before reading about AVL tree handling of duplicates.

How to handle duplicates in Binary Search Tree?

This is to augment AVL tree node to store count together with regular fields like key, left and right pointers.
Insertion of keys 12, 10, 20, 9, 11, 10, 12, 12 in an empty Binary Search Tree would create following.



Count of a key is shown in bracket

Below is implementation of normal AVL Tree with count with every key. This code basically is taken from code for insert and delete in AVL tree. The changes made for handling duplicates are highlighted, rest of the code is same.

The important thing to note is changes are very similar to simple Binary Search Tree changes.

C++

```
// C++ program of AVL tree that
// handles duplicates
#include
#include
```



```
// An AVL tree node
struct node {
int key;
struct node* left;
struct node* right;
int height;
int count;
};

// A utility function to get maximum of two integers
int max(int a, int b);

// A utility function to get height of the tree
int height(struct 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;
}

/* Helper function that allocates a new node with the given key and
NULL left and right pointers. */
struct node* newNode(int key)
{
struct node* node = (struct node*)
malloc(sizeof(struct node));
node->key = key;
node->left = NULL;
node->right = NULL;
node->height = 1; // new node is initially added at leaf
node->count = 1;
return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
struct node* rightRotate(struct node* y)
{
struct node* x = y->left;
struct 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.
struct node* leftRotate(struct node* x)
{
struct node* y = x->right;
struct 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(struct node* N)
{
if (N == NULL)
return 0;
return height(N->left) – height(N->right);
}

struct node* insert(struct node* node, int key)
{
/* 1. Perform the normal BST rotation */
if (node == NULL)
return (newNode(key));

// If key already exists in BST, increment count and return
if (key == node->key) {
(node->count)++;
return node;
}

/* Otherwise, recur down the tree */
if (key < node->key)
node->left = insert(node->left, key);
else
node->right = insert(node->right, key);
```

```
/* 2. Update height of this ancestor node */
node->height = max(height(node->left), height(node->right)) + 1;

/* 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->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. */
struct node* minValueNode(struct node* node)
{
struct node* current = node;

/* loop down to find the leftmost leaf */
while (current->left != NULL)
current = current->left;

return current;
}

struct node* deleteNode(struct 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 {
// If key is present more than once, simply decrement
// count and return
if (root->count > 1) {
(root->count)–;
return;
}
// Else, delete the node

// node with only one child or no child
if ((root->left == NULL) || (root->right == NULL)) {
struct node* temp = root->left ? root->left : root->right;

// No child case
if (temp == NULL) {
temp = root;
root = NULL;
}
else // One child case
*root = *temp; // Copy the contents of the non-empty child

free(temp);
}
```

```
else {
// node with two children: Get the inorder successor (smallest
// in the right subtree)
struct 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(struct node* root)
{
if (root != NULL) {
printf("%d(%d) ", root->key, root->count);
preOrder(root->left);
preOrder(root->right);
}
}

/* Drier program to test above function*/
int main()
```

```
{
struct node* root = NULL;

/* Constructing tree given in the above figure */
root = insert(root, 9);
root = insert(root, 5);
root = insert(root, 10);
root = insert(root, 5);
root = insert(root, 9);
root = insert(root, 7);
root = insert(root, 17);

printf("Pre order traversal of the constructed AVL tree is \n");
preOrder(root);

root = deleteNode(root, 9);

printf("\nPre order traversal after deletion of 9 \n");
preOrder(root);

return 0;
}
```

Java

```
// Java program of AVL tree that handles duplicates
import java.util.*;

class solution {

    // An AVL tree node
    static class node {
        int key;
        node left;
```

```
    node right;
    int height;
    int count;
}

// A utility function to get height of the tree
static int height(node N)
{
    if (N == null)
        return 0;
    return N.height;
}

// A utility function to get maximum of two integers
static int max(int a, int b)
{
    return (a > b) ? a : b;
}

/* Helper function that allocates a new node with the given key and
null left and right pointers. */
static node newNode(int key)
{
    node node = new node();
    node.key = key;
    node.left = null;
    node.right = null;
    node.height = 1; // new node is initially added at leaf
    node.count = 1;
    return (node);
}

// A utility function to right rotate subtree rooted with y
// See the diagram given above.
static 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.
    static 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
    static int getBalance(node N)
    {
        if (N == null)
            return 0;
        return height(N.left) - height(N.right);
    }

    static node insert(node node, int key)
    {
        /*1.  Perform the normal BST rotation */
        if (node == null)
            return (newNode(key));
```

```
        // If key already exists in BST, increment count and return
        if (key == node.key) {
            (node.count)++;
            return node;
        }

        /* Otherwise, recur down the tree */
        if (key < node.key)
            node.left = insert(node.left, key);
        else
            node.right = insert(node.right, key);

        /* 2. Update height of this ancestor node */
        node.height = max(height(node.left), height(node.right)) + 1;

        /* 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.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. */
    static node minValueNode(node node)
    {
        node current = node;

        /* loop down to find the leftmost leaf */
        while (current.left != null)
            current = current.left;

        return current;
    }

    static 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 {
            // If key is present more than once, simply decrement
            // count and return
            if (root.count > 1) {
                (root.count)--;
                return null;
```

```
        }
        // ElSE, delete the node

        // node with only one child or no child
        if ((root.left == null) || (root.right == null)) {
            node temp = root.left != null ? root.left : root.right;

            // 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
    static void preOrder(node root)
    {
        if (root != null) {
            System.out.printf("%d(%d) ", root.key, root.count);
            preOrder(root.left);
            preOrder(root.right);
        }
    }

    /* Drier program to test above function*/
    public static void main(String args[])
    {
        node root = null;

        /* Coning tree given in the above figure */
        root = insert(root, 9);
        root = insert(root, 5);
        root = insert(root, 10);
```

```
        root = insert(root, 5);
        root = insert(root, 9);
        root = insert(root, 7);
        root = insert(root, 17);

        System.out.printf("Pre order traversal of the constructed AVL tree is \n");
        preOrder(root);

        deleteNode(root, 9);

        System.out.printf("\nPre order traversal after deletion of 9 \n");
        preOrder(root);
    }
}
// contributed by Arnab Kundu
```

C#

```
// C# program of AVL tree that
// handles duplicates
using System;

class GFG {

    // An AVL tree node
    class node {
        public int key;
        public node left;
        public node right;
        public int height;
        public int count;
    }

    // A utility function to get
    // height of the tree
    static int height(node N)
    {
        if (N == null)
            return 0;
```

```
        return N.height;
    }

    // A utility function to get
    // maximum of two integers
    static int max(int a, int b)
    {
        return (a > b) ? a : b;
    }

    /* Helper function that allocates a
    new node with the given key and
    null left and right pointers. */
    static node newNode(int key)
    {
        node node = new node();
        node.key = key;
        node.left = null;
        node.right = null;
        node.height = 1; // new node is initially
        // added at leaf
        node.count = 1;
        return (node);
    }

    // A utility function to right
    // rotate subtree rooted with y
    // See the diagram given above.
    static 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.
    static 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
    static int getBalance(node N)
    {
        if (N == null)
            return 0;
        return height(N.left) - height(N.right);
    }

    static node insert(node node, int key)
    {
        /*1. Perform the normal BST rotation */
```

```
        if (node == null)
            return (newNode(key));

        // If key already exists in BST,
        // icnrement count and return
        if (key == node.key) {
            (node.count)++;
            return node;
        }

        /* Otherwise, recur down the tree */
        if (key < node.key)
            node.left = insert(node.left, key);
        else
            node.right = insert(node.right, key);

        /* 2. Update height of this
          ancestor node */
        node.height = max(height(node.left),
                          height(node.right))
                      + 1;

        /* 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.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. */
    static node minValueNode(node node)
    {
        node current = node;

        /* loop down to find the
       leftmost leaf */
        while (current.left != null)
            current = current.left;

        return current;
    }

    static 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 {
            // If key is present more than
            // once, simply decrement
            // count and return
            if (root.count > 1) {
                (root.count)--;
                return null;
            }

            // ElSE, delete the node

            // node with only one child
            // or no child
            if ((root.left == null) || (root.right == null)) {
                node temp = root.left != null ? root.left : root.right;

                // 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
    static void preOrder(node root)
    {
        if (root != null) {
            Console.Write(root.key + "(" + root.count + ") ");
            preOrder(root.left);
            preOrder(root.right);
        }
    }

    // Driver Code
    static public void Main(String[] args)
    {
        node root = null;

        /* Coning tree given in
       the above figure */
        root = insert(root, 9);
        root = insert(root, 5);
        root = insert(root, 10);
        root = insert(root, 5);
        root = insert(root, 9);
        root = insert(root, 7);
        root = insert(root, 17);

        Console.Write("Pre order traversal of "
                      + "the constructed AVL tree is \n");
        preOrder(root);

        deleteNode(root, 9);

        Console.Write("\nPre order traversal after "
```

```
                        + "deletion of 9 \n");
            preOrder(root);
        }
    }

    // This code is contributed by Arnab Kundu
```

Output:

```
Pre order traversal of the constructed AVL tree is
9(2) 5(2) 7(1) 10(1) 17(1)
Pre order traversal after deletion of 9
9(1) 5(2) 7(1) 10(1) 17(1)
```

Thanks to **Rounaq Jhunjhunu Wala** for sharing initial code. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

Recommended Posts:

Find All Duplicate Subtrees

BST to a Tree with sum of all smaller keys

Print BST keys in the given range

Check if a Binary Tree (not BST) has duplicate values

Check if a Binary Tree contains duplicate subtrees of size 2 or more

Iterative approach to check if a Binary Tree is Perfect

Kth node in Diagonal Traversal of Binary Tree

Print all leaf nodes of a binary tree from right to left

Reverse alternate levels of a perfect binary tree using Stack

Recursive Program to Print extreme nodes of each level of Binary Tree in alternate order

Iterative approach to check for children sum property in a Binary Tree

Reverse Clockwise spiral traversal of a binary tree

Spanning Tree With Maximum Degree (Using Kruskal's Algorithm)

Count nodes with two children at level L in a Binary Tree

Improved By : andrew1234, TapanMeena

Article Tags : Tree AVL-Tree

Practice Tags : Tree AVL-Tree

2

To-do Done

2.5

Based on **19** vote(s)

Feedback/ Suggest Improvement Add Notes Improve Article

Please write to us at contribute@geeksforgeeks.org to report any issue with the above content.

Writing code in comment? Please use ide.geeksforgeeks.org, generate link and share the link here.

Load Comments

Share this post!