Bahria University,

Karachi Campus



COURSE: CSC-221 DATA STRUCTURES AND ALGORITHM

TERM: FALL 2020, CLASS: BSE- 3 (A)

Submitted By:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(ADIL WAHEED) (65190)

ENROLLMENT NO #:02-131192-082

Submitted To:

Engr. Maam Nazar Mobeen/ Engr. Ramshaa

Signed Remarks: Score:

INDEX

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| SNO | DATE | LAB NO | LAB OBJECTIVE | SIGN |
| 01 | 1-10-2020 | 01 | ONE AND TWO DIMENSIONAL ARRAY |  |
| 02 | 09-10-20 | 02 | Linear Search & Sorting Algorithms |  |
| 03 | 13-10-20 | 03 | Recusrion |  |
| 04 | 30/10/2020 | 04 | Binary Search Algorithm |  |
| 05 | 30/10/2020 | 05 | Merge Sort |  |
| 06 | 30/10/2020 | 06 | Quick Sort |  |
| 07 | 4/11/2020 | 07 | Stack |  |
| 08 | 12/11/2020 | 08 | QUEUE |  |
| 09 | 20/10/2020 | 09 | Doubly Linked List |  |
| 10 | 24/10/2020 | 10 | CIRCULAR Linked List |  |
| 11 | 1/1/2021 | 11 | BUCKET, RADIX SORT & BINARY TREES |  |
| 12 | 4/1/2021 | 12 | BST AND AVL TREE |  |
|  |  |  |  |  |
| SNO | DATE | LAB NO | LAB OBJECTIVE | SIGN |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |

Bahria University,

Karachi Campus



LAB EXPERIMENT NO.

\_\_\_12\_\_\_\_

LIST OF TASKS

|  |  |
| --- | --- |
| TASK NO | OBJECTIVE |
| 1 | **Write a program to implement concept of Binary Search Tree using dynamic trees** |
| 2 | **Implement the dynamic AVL Tree by performing searching.** |
|  |  |
|  |  |
|  |  |

Submitted On:

\_\_\_\_\_\_\_\_\_\_\_\_

(Date: 4/1/2021)

**Task No. 1:**

**Write a program to implement concept of Binary Search Tree using dynamic trees**

Solution:

class Node

{

public Node lchild;

public int info;

public Node rchild;

public Node(int i)

{

info = i;

lchild = null;

rchild = null;

}

}

}

class BinarySearchTree

{

private Node root;

public BinarySearchTree()

{

root = null;

}

public bool IsEmpty()

{

return (root == null);

}

public void Insert(int x)

{

root = Insert(root, x);

}

private Node Insert(Node p, int x)

{

if (p == null)

p = new Node(x);

else if (x < p.info)

p.lchild = Insert(p.lchild, x);

else if (x > p.info)

p.rchild = Insert(p.rchild, x);

else Console.WriteLine(x + "already present in the tree");

return p;

}

public void insert1(int x)

{

Node p = root;

Node par = null;

while (p != null)

{

par = p;

if (x < p.info)

p = p.lchild;

else if (x > p.info)

p = p.rchild;

else

{

Console.WriteLine(x + "already present in the tree");

return;

}

}

Node temp = new Node(x);

if (par == null)

root = temp;

else if (x < par.info)

par.lchild = temp;

else

par.rchild = temp;

}

public bool Search(int x)

{

return (Search(root, x) != null);

}

private Node Search(Node p, int x)

{

if (p == null)

return null; /\*key not found\*/

if (x < p.info)/\*search in left subtree\*/

return Search(p.lchild, x);

if (x > p.info)/\*search in right subtree\*/

return Search(p.rchild, x);

return p; /\*key found\*/

}

public bool Search1(int x)

{

Node p = root;

while (p != null)

{

if (x < p.info)

p = p.lchild; //Move to left child

else if (x > p.info)

p = p.rchild; /\*Move to right child \*/

else /\*x .Found\*/

return true;

}

return false;

}

public void Delete(int x)

{

root = Delete(root, x);

}

private Node Delete(Node p, int x)

{

Node ch, s;

if (p == null)

{

Console.WriteLine(x + "not found");

return p;

}

if (x < p.info) /\*delete from left subtree\*/

p.lchild = Delete(p.lchild, x);

else if (x > p.info) /\*delete from right subtree\*/

p.rchild = Delete(p.rchild, x);

else

{

/\*key to be deleted is found\*/

if (p.lchild != null && p.rchild != null) /\*2 children\*/

{

s = p.rchild;

while (s.lchild != null)

s = s.lchild;

p.info = s.info;

p.rchild = Delete(p.rchild, s.info);

}

else /\*1 child or no child\*/

{

if (p.lchild != null) /\*only left child\*/

ch = p.lchild;

else /\*only right child or no child\*/

ch = p.rchild;

p = ch;

}

}

return p;

}

public void Deletel(int x)

{

Node p = root;

Node par = null;

while (p != null)

{

if (x == p.info)

break;

par = p;

if (x < p.info)

p = p.lchild;

else

p = p.rchild;

} if (p == null)

{

Console.WriteLine(x + "not found in the list");

return;

}

/\*Case C: 2 children\*/

/\*Find morder successor and its parent\*/

Node s, ps;

if (p.lchild != null && p.rchild != null)

{

ps = p;

s = p.rchild;

while (s.lchild != null)

{

ps = s;

s = s.lchild;

}

p.info = s.info;

p = s;

par = ps;

}

/\*Case B and Case A : 1 or no child\*/

Node ch;

if (p.lchild != null) /\*node to be deleted has left child \*/

ch = p.lchild;

else /\*noje to be deleted has right child or no child\*/

ch = p.rchild;

if (par == null) /\*node to be deleted is root node\*/

root = ch;

else if (p == par.lchild)/\*node is left child of its parent\*/

par.lchild = ch;

else

//node is right child of its parent

par.rchild = ch;

}

public int Min()

{

if (IsEmpty())

throw new InvalidOperationException("Tree is empty");

return Min(root).info;

}

private Node Min(Node p)

{

if (p.lchild == null)

return p;

return Min(p.lchild);

}

public int Max()

{

if (IsEmpty())

throw new InvalidOperationException("Tree is empty");

return Max(root).info;

}

private Node Max(Node p)

{

if (p.rchild == null)

return p;

return Max(p.rchild);

}

public int Minl()

{

if (IsEmpty())

throw new InvalidOperationException("Tree is empty");

Node p = root;

while (p.lchild != null)

p = p.lchild;

return p.info;

}

public int Maxl()

{

if (IsEmpty())

throw new InvalidOperationException("Tree is empty");

Node p = root;

while (p.rchild != null)

p = p.rchild;

return p.info;

}

public void Display()

{

Display(root, 0);

Console.WriteLine();

}

private void Display(Node p, int level)

{

int i;

if (p == null)

return;

Display(p.rchild, level + 1);

Console.WriteLine();

for (i = 0; i < level; i++)

Console.Write(" ");

Console.Write(p.info);

Display(p.lchild, level + 1);

}

public void Preorder()

{

Preorder(root);

Console.WriteLine();

}

private void Preorder(Node p)

{

if (p == null)

return;

Console.Write(p.info + " ");

Preorder(p.lchild);

Preorder(p.rchild);

}

public void Inorder()

{

Inorder(root);

Console.WriteLine();

}

private void Inorder(Node p)

{

if (p == null)

return;

Inorder(p.lchild);

Console.Write(p.info + " ");

Inorder(p.rchild);

}

public void Postorder()

{

Postorder(root);

Console.WriteLine();

}

private void Postorder(Node p)

{

if (p == null)

return;

Postorder(p.lchild);

Postorder(p.rchild);

Console.Write(p.info + " ");

}

public int Height()

{

return Height(root);

}

private int Height(Node p)

{

int hL, hR;

if (p == null)

return 0;

hL = Height(p.lchild);

hR = Height(p.rchild);

if (hL > hR)

return 1 + hL;

else

return 1 + hR;

}

}

}

class Program

{

static void Main(string[] args)

{

BinarySearchTree bt = new BinarySearchTree();

int choice, x;

while (true)

{

Console.WriteLine("1.Display List");

Console.WriteLine("2.search");

Console.WriteLine("3.Insert a new node");

Console.WriteLine("4.delete a node");

Console.WriteLine("5.PreOrder Traversal");

Console.WriteLine("6.InOrder Traversal");

Console.WriteLine("7.PostOrder Traversal");

Console.WriteLine("8.Height of tree");

Console.WriteLine("9.Find minimum key");

Console.WriteLine("10.Find maximum key");

Console.WriteLine("11.Quit");

Console.Write("Enter your choice : ");

choice = Convert.ToInt32(Console.ReadLine());

if (choice == 11)

break;

switch (choice)

{

case 1:

bt.Display();

break;

case 2:

Console.Write("Enter the key to be searched : ");

x = Convert.ToInt32(Console.ReadLine());

if (bt.Search(x))

Console.WriteLine("key found");

else

Console.WriteLine("key not found");

break;

case 3:

Console.Write("Enter the key to be inserted : ");

x = Convert.ToInt32(Console.ReadLine());

bt.Insert(x);

break;

case 4:

Console.Write("Enter the key to be deleted: ");

x = Convert.ToInt32(Console.ReadLine());

bt.Delete(x);

break;

case 5:

bt.Preorder();

break;

case 6:

bt.Inorder();

break;

case 7:

bt.Postorder();

break;

case 8:

Console.WriteLine("Heiht of tree is " + bt.Height());

break;

case 9:

Console.WriteLine("Minimum key is " + bt.Min());

break;

case 10:

Console.WriteLine("Maximum key is " + bt.Max());

break;

default:

Console.WriteLine("Wrong choice");

break;

}

Console.WriteLine();

}

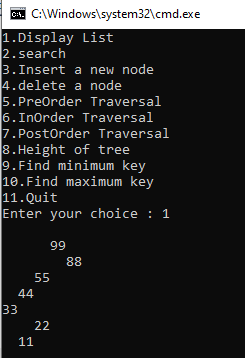
Console.WriteLine("Exiting");

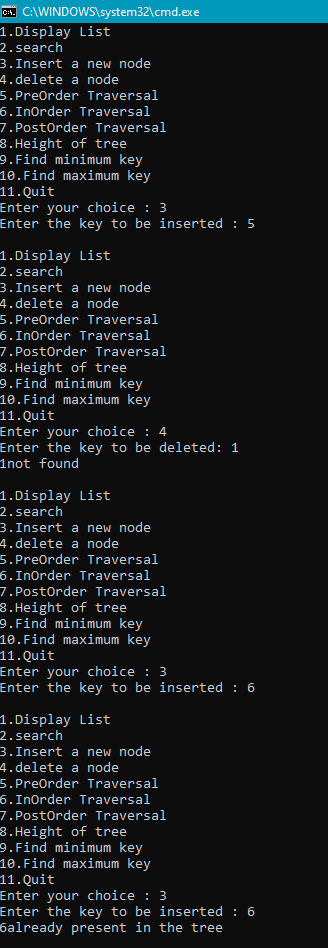
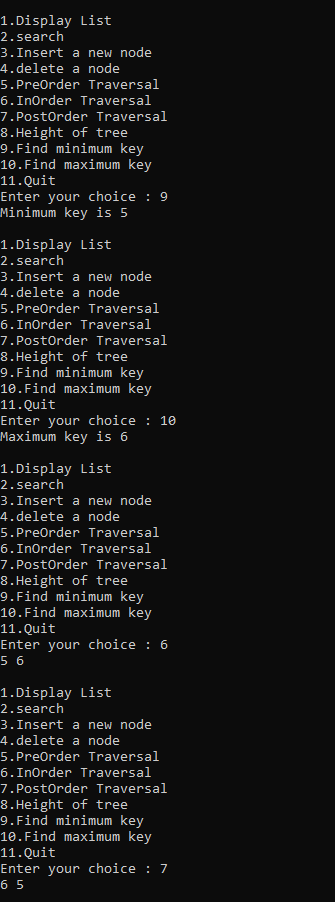
}

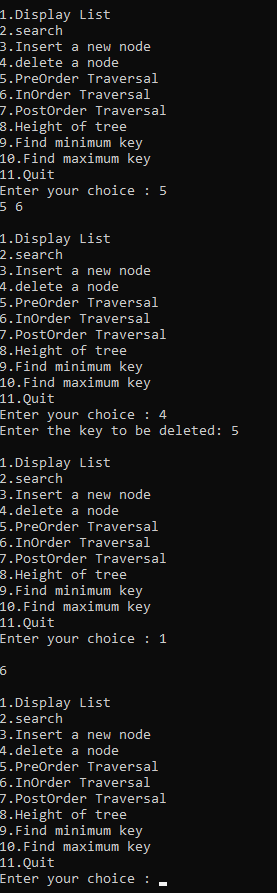
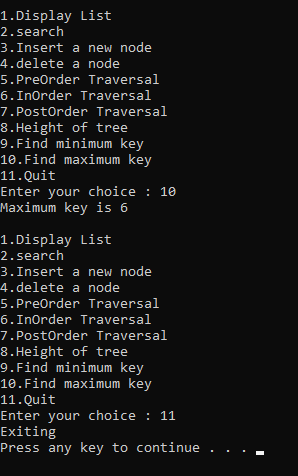
}

}

**OUTPUT:**



**Task No. 2:**

**Implement the dynamic AVL Tree by performing searching.**

Solution:

class AVL

{

class Node

{

public int data;

public Node left;

public Node right;

public Node(int data)

{

this.data = data;

}

}

Node root;

public AVL()

{

}

public void Add(int data)

{

Node newItem = new Node(data);

if (root == null)

{

root = newItem;

}

else

{

root = RecursiveInsert(root, newItem);

}

}

private Node RecursiveInsert(Node current, Node n)

{

if (current == null)

{

current = n;

return current;

}

else if (n.data < current.data)

{

current.left = RecursiveInsert(current.left, n);

current = balance\_tree(current);

}

else if (n.data > current.data)

{

current.right = RecursiveInsert(current.right, n);

current = balance\_tree(current);

}

return current;

}

private Node balance\_tree(Node current)

{

int b\_factor = balance\_factor(current);

if (b\_factor > 1)

{

if (balance\_factor(current.left) > 0)

{

current = RotateLL(current);

}

else

{

current = RotateLR(current);

}

}

else if (b\_factor < -1)

{

if (balance\_factor(current.right) > 0)

{

current = RotateRL(current);

}

else

{

current = RotateRR(current);

}

}

return current;

}

public void Delete(int target)

{//and here

root = Delete(root, target);

}

private Node Delete(Node current, int target)

{

Node parent;

if (current == null)

{ return null; }

else

{

//left subtree

if (target < current.data)

{

current.left = Delete(current.left, target);

if (balance\_factor(current) == -2)//here

{

if (balance\_factor(current.right) <= 0)

{

current = RotateRR(current);

}

else

{

current = RotateRL(current);

}

}

}

//right subtree

else if (target > current.data)

{

current.right = Delete(current.right, target);

if (balance\_factor(current) == 2)

{

if (balance\_factor(current.left) >= 0)

{

current = RotateLL(current);

}

else

{

current = RotateLR(current);

}

}

}

//if target is found

else

{

if (current.right != null)

{

//delete its inorder successor

parent = current.right;

while (parent.left != null)

{

parent = parent.left;

}

current.data = parent.data;

current.right = Delete(current.right, parent.data);

if (balance\_factor(current) == 2)//rebalancing

{

if (balance\_factor(current.left) >= 0)

{

current = RotateLL(current);

}

else { current = RotateLR(current); }

}

}

else

{ //if current.left != null

return current.left;

}

}

}

return current;

}

public void Find(int key)

{

if (Find(key, root).data == key)

{

Console.WriteLine("{0} was found!", key);

}

else

{

Console.WriteLine("Nothing found!");

}

}

private Node Find(int target, Node current)

{

if (target < current.data)

{

if (target == current.data)

{

return current;

}

else

return Find(target, current.left);

}

else

{

if (target == current.data)

{

return current;

}

else

return Find(target, current.right);

}

}

public void DisplayTree()

{

if (root == null)

{

Console.WriteLine("Tree is empty");

return;

}

InOrderDisplayTree(root);

Console.WriteLine();

}

private void InOrderDisplayTree(Node current)

{

if (current != null)

{

InOrderDisplayTree(current.left);

Console.Write("({0}) ", current.data);

InOrderDisplayTree(current.right);

}

}

private int max(int l, int r)

{

return l > r ? l : r;

}

private int getHeight(Node current)

{

int height = 0;

if (current != null)

{

int l = getHeight(current.left);

int r = getHeight(current.right);

int m = max(l, r);

height = m + 1;

}

return height;

}

private int balance\_factor(Node current)

{

int l = getHeight(current.left);

int r = getHeight(current.right);

int b\_factor = l - r;

return b\_factor;

}

private Node RotateRR(Node parent)

{

Node pivot = parent.right;

parent.right = pivot.left;

pivot.left = parent;

return pivot;

}

private Node RotateLL(Node parent)

{

Node pivot = parent.left;

parent.left = pivot.right;

pivot.right = parent;

return pivot;

}

private Node RotateLR(Node parent)

{

Node pivot = parent.left;

parent.left = RotateRR(pivot);

return RotateLL(parent);

}

private Node RotateRL(Node parent)

{

Node pivot = parent.right;

parent.right = RotateLL(pivot);

return RotateRR(parent);

}

}

class Program

{

static void Main(string[] args)

{

Console.WriteLine("The AVL tree is:");

AVL tree = new AVL();

tree.Add(33);

tree.Add(22);

tree.Add(44);

tree.Add(66);

tree.Add(99);

tree.Add(100);

tree.Delete(22);

tree.DisplayTree();

}

}

**OUTPUT:**

