**Practical No. 6**

1. Binary Search Tree

Code :

#include <stdio.h>

#include <stdlib.h>

struct node //Creating a node structure

{

int value;

struct node \*l;

struct node \*r;

}\*root = NULL, \*temp = NULL, \*t2, \*t1; //initializing pointers

void insert(); //declaration of functions

void inorder(struct node \*t);

void create();

void search(struct node \*t);

void preorder(struct node \*t);

void postorder(struct node \*t);

int flag = 1;

void main()

{

int ch;

/\*menu driven programs\*/

printf("\nOPERATIONS ---");

printf("\n1 - Insert an element into tree\n");

printf("2 - Inorder Traversal\n");

printf("3 - Preorder Traversal\n");

printf("4 - Postorder Traversal\n");

printf("5 - Exit\n");

while(1)

{

printf("\nEnter your choice : ");

scanf("%d", &ch);

switch (ch)

{

case 1:

insert();

break;

case 2:

inorder(root);

break;

case 3:

preorder(root);

break;

case 4:

postorder(root);

break;

case 5:

exit(0);

default :

printf("Wrong choice, Please enter correct choice ");

break;

}

}

}

/\* To insert a node in the tree \*/

void insert()

{

create();

if (root == NULL) /\*checking the first element in null\*/

root = temp;

else

search(root); /\*otherwise seach and insert\*/

}

/\* To create a node \*/

void create()

{ /\*creating very first node of the element\*/

int data;

printf("Enter data of node to be inserted : ");

scanf("%d", &data);

temp = (struct node \*)malloc(1\*sizeof(struct node));

temp->value = data;

temp->l = temp->r = NULL;

}

/\* Function to search the appropriate position to insert the new node \*/

void search(struct node \*t)

{

if ((temp->value > t->value) && (t->r != NULL)) /\* value more than root node value insert at right \*/

search(t->r);

else if ((temp->value > t->value) && (t->r == NULL))

t->r = temp;

else if ((temp->value < t->value) && (t->l != NULL)) /\* value less than root node value insert at left \*/

search(t->l);

else if ((temp->value < t->value) && (t->l == NULL))

t->l = temp;

}

/\* recursive function to perform inorder traversal of tree \*/

void inorder(struct node \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display");

return;

}

if (t->l != NULL)

inorder(t->l);

printf("%d -> ", t->value);

if (t->r != NULL)

inorder(t->r);

}

/\* To find the preorder traversal \*/

void preorder(struct node \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display");

return;

}

printf("%d -> ", t->value);

if (t->l != NULL)

preorder(t->l);

if (t->r != NULL)

preorder(t->r);

}

/\* To find the postorder traversal \*/

void postorder(struct node \*t)

{

if (root == NULL)

{

printf("No elements in a tree to display ");

return;

}

if (t->l != NULL)

postorder(t->l);

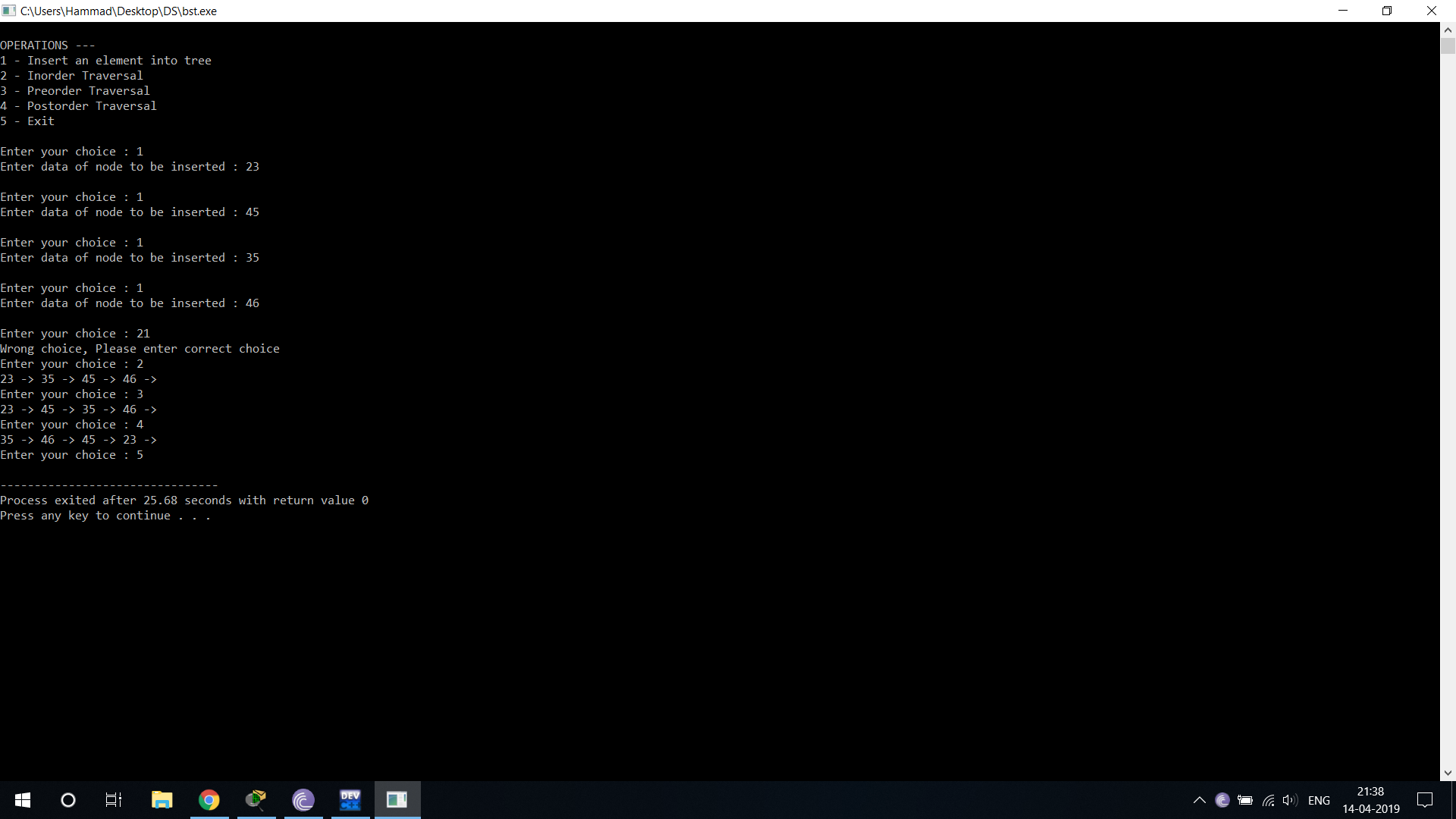
if (t->r != NULL)

postorder(t->r);

printf("%d -> ", t->value);

}

Screenshot :



1. AVL Tree (Adelson-Velskii and Landis) Self Balancing Binary Search Tree

Code :

#include<iostream>

#include<cstdio>

#include<sstream>

#include<algorithm>

#define pow2(n) (1 << (n))

using namespace std;

struct node //creating a node structure

{

int data;

struct node \*left;

struct node \*right;

}\*root;

/\*

\* Class Declaration

\*/

class AVL

{

public: //defining all the functions

int height(node \*);

int diff(node \*);

node \*rr\_rotation(node \*);

node \*ll\_rotation(node \*);

node \*lr\_rotation(node \*);

node \*rl\_rotation(node \*);

node\* balance(node \*);

node\* insert(node \*, int );

void display(node \*, int);

void inorder(node \*);

void preorder(node \*);

void postorder(node \*);

AVL()

{

root = NULL;

}

};

/\*

\* Main Contains Menu

\*/

int main()

{

int choice, item;

AVL avl;

while (1) ///menu driven program

{ cout<<"\n";

cout<<"1.Insert Element into the tree"<<endl;

cout<<"2.Display Balanced AVL Tree"<<endl;

cout<<"3.InOrder traversal"<<endl;

cout<<"4.PreOrder traversal"<<endl;

cout<<"5.PostOrder traversal"<<endl;

cout<<"6.Exit"<<endl;

cout<<"Enter your Choice: ";

cin>>choice;

switch(choice)

{

case 1:

cout<<"Enter value to be inserted: ";

cin>>item;

root = avl.insert(root, item);

break;

case 2:

if (root == NULL)

{

cout<<"Tree is Empty"<<endl;

continue;

}

cout<<"Balanced AVL Tree:"<<endl;

avl.display(root, 1);

break;

case 3:

cout<<"Inorder Traversal:"<<endl;

avl.inorder(root);

cout<<endl;

break;

case 4:

cout<<"Preorder Traversal:"<<endl;

avl.preorder(root);

cout<<endl;

break;

case 5:

cout<<"Postorder Traversal:"<<endl;

avl.postorder(root);

cout<<endl;

break;

case 6:

exit(1);

break;

default:

cout<<"Wrong Choice"<<endl;

}

}

return 0;

}

/\*

\* Height of AVL Tree

\*/

int AVL::height(node \*temp) //To calculate the height of the tree

{

int h = 0;

if (temp != NULL)

{

int l\_height = height (temp->left);

int r\_height = height (temp->right);

int max\_height = max (l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

/\*

\* Height Difference

\*/

int AVL::diff(node \*temp) //to calculate the difference between the node

{

int l\_height = height (temp->left);

int r\_height = height (temp->right);

int b\_factor= l\_height - r\_height; // The difference between heights of left and right subtrees cannot be more tham ome for all nodes

return b\_factor;

}

/\*

\* Right- Right Rotation

\*/

node \*AVL::rr\_rotation(node \*parent)

{

node \*temp;

temp = parent->right;

parent->right = temp->left;

temp->left = parent;

return temp;

}

/\*

\* Left- Left Rotation

\*/

node \*AVL::ll\_rotation(node \*parent)

{

node \*temp;

temp = parent->left;

parent->left = temp->right;

temp->right = parent;

return temp;

}

/\*

\* Left - Right Rotation

\*/

node \*AVL::lr\_rotation(node \*parent)

{

node \*temp;

temp = parent->left;

parent->left = rr\_rotation (temp);

return ll\_rotation (parent);

}

/\*

\* Right- Left Rotation

\*/

node \*AVL::rl\_rotation(node \*parent)

{

node \*temp;

temp = parent->right;

parent->right = ll\_rotation (temp);

return rr\_rotation (parent);

}

/\*

\* Balancing AVL Tree

\*/

node \*AVL::balance(node \*temp)

{

int bal\_factor = diff (temp);

if (bal\_factor > 1)

{

if (diff (temp->left) > 0)

temp = ll\_rotation (temp);

else

temp = lr\_rotation (temp);

}

else if (bal\_factor < -1)

{

if (diff (temp->right) > 0)

temp = rl\_rotation (temp);

else

temp = rr\_rotation (temp);

}

return temp;

}

/\*

\* Insert Element into the tree

\*/

node \*AVL::insert(node \*root, int value)

{

if (root == NULL)

{

root = new node;

root->data = value;

root->left = NULL;

root->right = NULL;

return root;

}

else if (value < root->data)

{

root->left = insert(root->left, value);

root = balance (root);

}

else if (value >= root->data)

{

root->right = insert(root->right, value);

root = balance (root);

}

return root;

}

/\*

\* Display AVL Tree

\*/

void AVL::display(node \*ptr, int level)

{

int i;

if (ptr!=NULL)

{

display(ptr->right, level + 1);

printf("\n");

if (ptr == root)

cout<<"Root -> ";

for (i = 0; i < level && ptr != root; i++)

cout<<" ";

cout<<ptr->data;

display(ptr->left, level + 1);

}

}

/\*

\* Inorder Traversal of AVL Tree

\*/

void AVL::inorder(node \*tree)

{

if (tree == NULL)

return;

inorder (tree->left);

cout<<tree->data<<" ";

inorder (tree->right);

}

/\*

\* Preorder Traversal of AVL Tree

\*/

void AVL::preorder(node \*tree)

{

if (tree == NULL)

return;

cout<<tree->data<<" ";

preorder (tree->left);

preorder (tree->right);

}

/\*

\* Postorder Traversal of AVL Tree

\*/

void AVL::postorder(node \*tree)

{

if (tree == NULL)

return;

postorder ( tree ->left );

postorder ( tree ->right );

cout<<tree->data<<" ";

}

Screenshot :

