**UIT1412**

**Programming and Data Structures Lab**

**IT-Section A**

**NAME: Bhuvaneswar A S**

**REG NO: 195002023**

**EX: End semester assignment.**

1. Implement the concept of Inheritance by creating a base class for Fruits and inherit it to be applied for Apple and Orange sub classes.

**Program:**

#include<iostream>

#include<stdio.h>

using namespace std;

class fruits{

public:

static int fcounter;

fruits(){fcounter++;}

};

class apples: public fruits

{

public:

static int acounter;

apples():fruits(){acounter++;}

};

class oranges:public fruits

{

public:

static int ocounter;

oranges():fruits(){ocounter++;}

};

int fruits::fcounter = 0;

int apples::acounter = 0;

int oranges::ocounter = 0;

int main()

{

apples a1,a2,a3,a4;

oranges o1,o2,o3,o4,o5;

fruits f1,f2; //other fruits than mango and apples

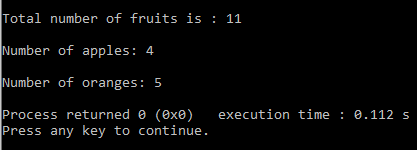
cout<<"\nTotal number of fruits is : "<<fruits::fcounter<<endl;

cout << "\nNumber of apples: " << apples::acounter << endl;

cout << "\nNumber of oranges: " << oranges::ocounter << endl;

}

**Output:**



2.Create exception to handle stack overflow

**Program:**

#include <iostream>

#include <exception>

#define maxe 5

using namespace std;

struct MyException : public exception {

const char \* what () const throw () {

return "stack overflow";

}

};

class stacks{

int a[5],top;

public:

stacks(){top=-1;}

void insert()

{

int k;

cout<<"\nEnter the value to insert :";

cin>>k;

try{

if(top==maxe-1)

throw MyException();

else

{

a[++top]=k;

}

}catch(MyException& e) {

cout << "exception class executed " <<endl;

cout << e.what() <<endl;

}}

};

int main() {

stacks a;

a.insert();

a.insert();

a.insert();

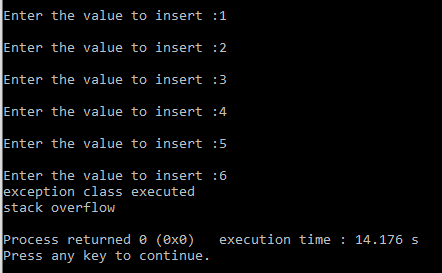
a.insert();

a.insert();

a.insert();

}

**Output:**



3.Implement Banking application with required Object-oriented concepts with respect to SSNCoin transfer similar to Bitcoins

**Program:**

#include<iostream>

using namespace std;

class ssncoins

{

string name;

double account\_no;

int coins1,coinsb;

double amount1,amountb;//backup to track details amountb

public:

ssncoins(){coins1=5;coinsb=5;amount1=5000,amountb=5000;}

void getcoins(ssncoins \*a1,ssncoins \*a2,int n)

{

try{

if(a2->coins1>=n){

a1->coins1+=n;

a2->coins1-=n;

a2->amount1-=n\*420;

}

else

throw "not suffficient balance int account check account balance";

}catch(const char \*msg){cerr<<endl<<msg<<endl;

}

}

void transfered\_amt(ssncoins\*a1)

{

int n=a1->coins1-a1->coinsb;

a1->amount1+=(n\*420);

cout<<"\ntotal amount of transfered :"<<a1->amount1;

}

void check\_details()

{

cout<<"\ndetails of the account holder: "<<endl;

cout<<"name: "<<name<<endl;

cout<<"account number: "<<account\_no<<endl;

cout<<"coins: "<<coins1<<endl;

cout<<"amount: "<<amount1<<endl;

}

void getdetails()

{

cout<<"\nenter the details of the account holder: "<<endl;

cout<<"name: ";cin>>name;cout<<endl;

cout<<"account number: ";cin>>account\_no;cout<<endl;

cout<<"coins: ";cin>>coins1;cout<<endl;

}

void sucess()

{

coinsb=coins1;

amountb=amount1;

}

void calculate\_amount()

{

amount1=amountb=coins1\*420;

}

};

int main()

{

int n;

ssncoins a1,a2,dummy;

cout<<"1:"<<endl;

a1.getdetails();cout<<"2:"<<endl;a2.getdetails();

a1.calculate\_amount(),a2.calculate\_amount();

cout<<"\n enter the number of coins to be transfered: ";cin>>n;

dummy.getcoins(&a1,&a2,n);

dummy.transfered\_amt(&a1);

cout<<"\ncheck details :";a1.check\_details(),a2.check\_details();

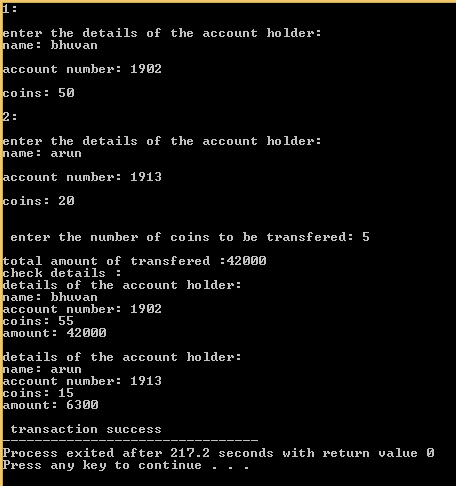
cout<<"\n transaction success";

a1.sucess();//transaction sucess so updtae backup

a2.sucess();

}

**Output:**

****

4. Make a template for Matrix class, test it on different data types, and add a generic member function.

**PROGRAM:**

#include<iostream>

using namespace std;

template<typename t>

class matrix

{

t a[20][20];

int rows,columns;

public:

void create();

};

template<typename t>

void matrix<t>::create()

{

int i,j=0;

cout<<"\n enter the number of rows and columns:";

cin>>rows>>columns;

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

{

for(j=0;j<columns;j++)

{

cout<<"\n enter the value :";

cin>>a[i][j];

}

}

cout<<"\n printing the matrix :\n";

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

{

for(j=0;j<columns;j++)

{

cout<<a[i][j]<<"\t";

}

cout<<"\n";

}

}

int main()

{

matrix<int>p;

p.create();

matrix<float>q;

q.create();

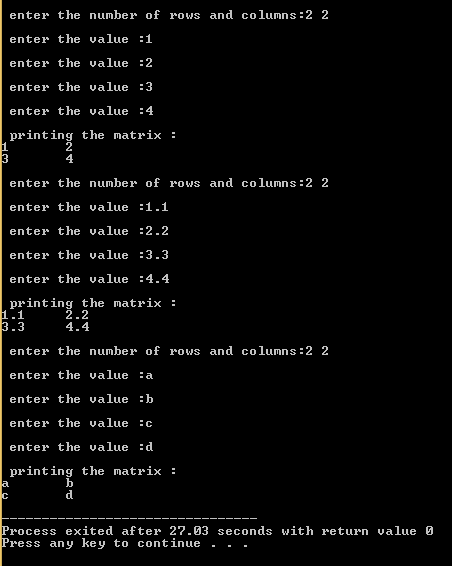
matrix<char>r;

r.create();

return 0;

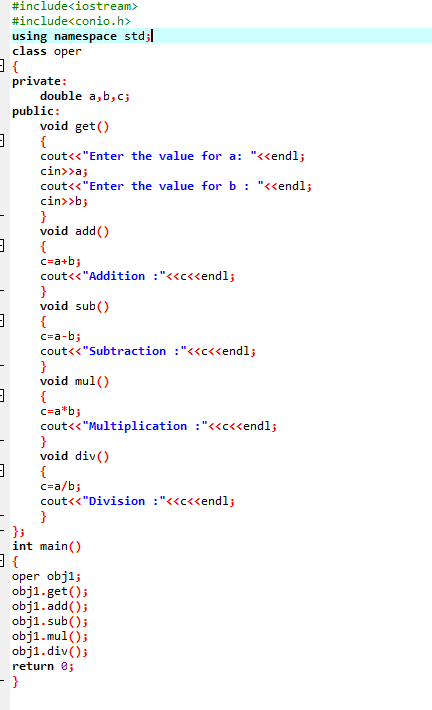
}

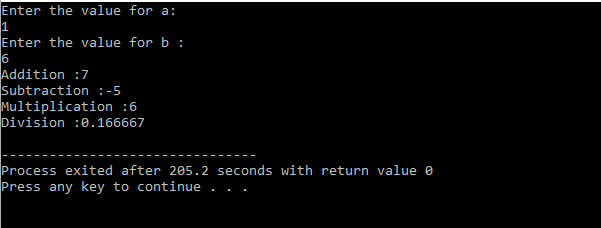
**OUTPUT:**

****

**5.** Write simple code that uses the +, -, \*, and / operators for doubles.

**PROGRAM:**



**Output:**

**7.** Implement AVL tree and compare its time complexity with respect to Binary Search Tree.

**PROGRAM:**

#include <iostream>

#include <queue>

using namespace std;

struct node {

struct node \*left;

int data;

int height;

struct node \*right;

};

class AVL

{

public:

struct node \* root;

AVL(){

this->root = NULL;

}

int calheight(struct node \*p){

if(p->left && p->right){

if (p->left->height < p->right->height)

return p->right->height + 1;

else return p->left->height + 1;

}

else if(p->left && p->right == NULL){

return p->left->height + 1;

}

else if(p->left ==NULL && p->right){

return p->right->height + 1;

}

return 0;

}

int bf(struct node \*n){

if(n->left && n->right){

return n->left->height - n->right->height;

}

else if(n->left && n->right == NULL){

return n->left->height;

}

else if(n->left== NULL && n->right ){

return -n->right->height;

}

}

struct node \* llrotation(struct node \*n){

struct node \*p;

struct node \*tp;

p = n;

tp = p->left;

p->left = tp->right;

tp->right = p;

return tp;

}

struct node \* rrrotation(struct node \*n){

struct node \*p;

struct node \*tp;

p = n;

tp = p->right;

p->right = tp->left;

tp->left = p;

return tp;

}

struct node \* rlrotation(struct node \*n){

struct node \*p;

struct node \*tp;

struct node \*tp2;

p = n;

tp = p->right;

tp2 =p->right->left;

p -> right = tp2->left;

tp ->left = tp2->right;

tp2 ->left = p;

tp2->right = tp;

return tp2;

}

struct node \* lrrotation(struct node \*n){

struct node \*p;

struct node \*tp;

struct node \*tp2;

p = n;

tp = p->left;

tp2 =p->left->right;

p -> left = tp2->right;

tp ->right = tp2->left;

tp2 ->right = p;

tp2->left = tp;

return tp2;

}

struct node\* insert(struct node \*r,int data){

if(r==NULL){

struct node \*n;

n = new struct node;

n->data = data;

r = n;

r->left = r->right = NULL;

r->height = 1;

return r;

}

else{

if(data < r->data)

r->left = insert(r->left,data);

else

r->right = insert(r->right,data);

}

r->height = calheight(r);

if(bf(r)==2 && bf(r->left)==1){

r = llrotation(r);

}

else if(bf(r)==-2 && bf(r->right)==-1){

r = rrrotation(r);

}

else if(bf(r)==-2 && bf(r->right)==1){

r = rlrotation(r);

}

else if(bf(r)==2 && bf(r->left)==-1){

r = lrrotation(r);

}

return r;

}

void levelorder\_newline(){

if (this->root == NULL){

cout<<"\n"<<"Empty tree"<<"\n";

return;

}

levelorder\_newline(this->root);

}

void levelorder\_newline(struct node \*v){

queue <struct node \*> q;

struct node \*cur;

q.push(v);

q.push(NULL);

while(!q.empty()){

cur = q.front();

q.pop();

if(cur == NULL && q.size()!=0){

cout<<"\n";

q.push(NULL);

continue;

}

if(cur!=NULL){

cout<<" "<<cur->data;

if (cur->left!=NULL){

q.push(cur->left);

}

if (cur->right!=NULL){

q.push(cur->right);

}

}

}

}

struct node \* deleteNode(struct node \*p,int data){

if(p->left == NULL && p->right == NULL){

if(p==this->root)

this->root = NULL;

delete p;

return NULL;

}

struct node \*t;

struct node \*q;

if(p->data < data){

p->right = deleteNode(p->right,data);

}

else if(p->data > data){

p->left = deleteNode(p->left,data);

}

else{

if(p->left != NULL){

q = inpre(p->left);

p->data = q->data;

p->left=deleteNode(p->left,q->data);

}

else{

q = insuc(p->right);

p->data = q->data;

p->right = deleteNode(p->right,q->data);

}

}

if(bf(p)==2 && bf(p->left)==1){ p = llrotation(p); }

else if(bf(p)==2 && bf(p->left)==-1){ p = lrrotation(p); }

else if(bf(p)==2 && bf(p->left)==0){ p = llrotation(p); }

else if(bf(p)==-2 && bf(p->right)==-1){ p = rrrotation(p); }

else if(bf(p)==-2 && bf(p->right)==1){ p = rlrotation(p); }

else if(bf(p)==-2 && bf(p->right)==0){ p = llrotation(p); }

return p;

}

struct node\* inpre(struct node\* p){

while(p->right!=NULL)

p = p->right;

return p;

}

struct node\* insuc(struct node\* p){

while(p->left!=NULL)

p = p->left;

return p;

}

~AVL(){

}

};

int main(){

AVL b;

int c,x;

do{

cout<<"\n1.Display levelorder on newline";

cout<<"\n2.Insert";

cout<<"\n3.Delete\n";

cout<<"\n0.Exit\n";

cout<<"\nChoice: ";

cin>>c;

switch (c)

{

case 1:

b.levelorder\_newline();

break;

case 2:

cout<<"\nEnter no. ";

cin>>x;

b.root = b.insert(b.root,x);

break;

case 3:

cout<<"\nWhat to delete? ";

cin>>x;

b.root = b.deleteNode(b.root,x);

break;

case 0:

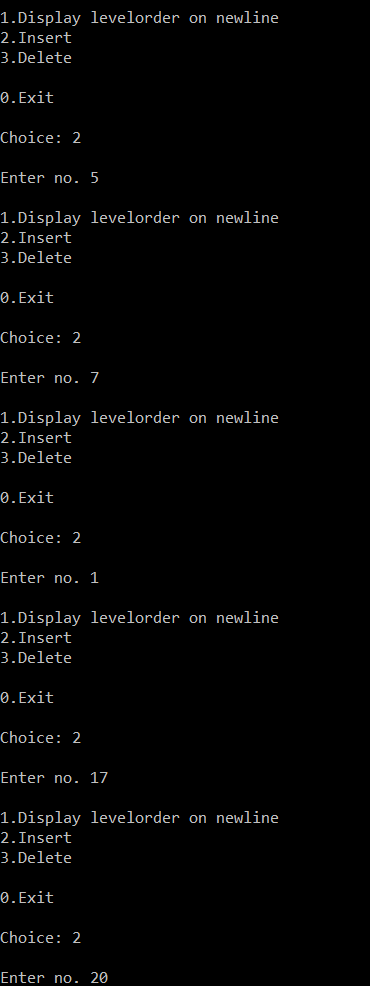
break;

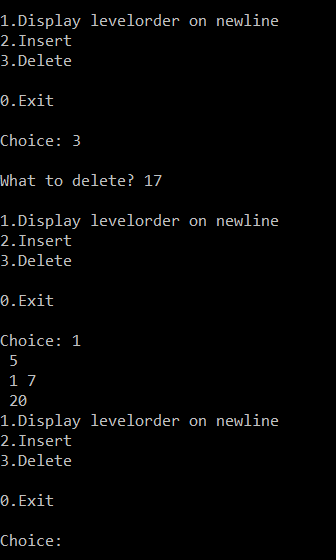
}

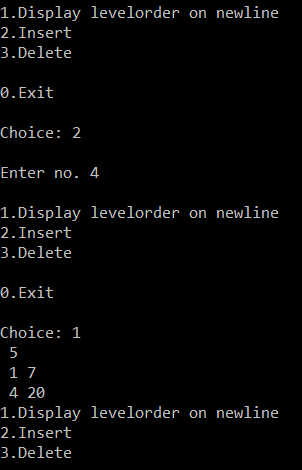
} while(c!=0);

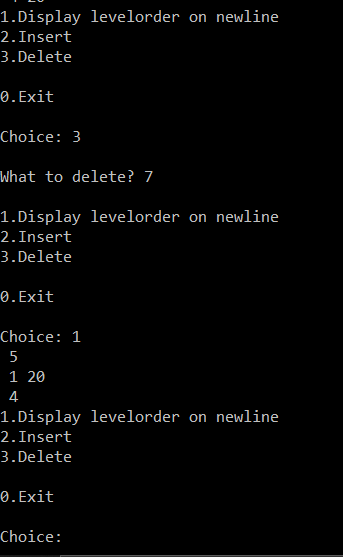
}

**OUTPUT:**









**TIME COMPLEXITY:**

**BINARY SEARCH TREE:**

In average cases, properties like insert, search and deletion operations in O(log n) time where n is the number of nodes in the tree. However, the time complexity for these operations is O(n) in the worst case when the tree becomes unbalanced.

**AVL TREE:**

Due to the balancing property, the insertion, deletion and search operations take O(log n) in both the average and the worst cases. Therefore, AVL trees give us an edge over Binary Search Trees which have an O(n)time complexity in the worst case scenario.

**Time complexity of BST:**

**Insertion:** worst case time complexity is O(n). In general, the time complexity would be O(h).

**Deletion:** worst case time complexity is O(n). In general, the time complexity would be O(h).

**Time complexity of AVL:**

**Insertion:** In general, the time complexity would be O(log n).

**Deletion:** In general, the time complexity would be O(log n).

**8.** Print a given tree in 3 traversal methods.

a. pre-order

b. post-order

c. in-order

**PROGRAM:**

#include<iostream>

#include<cstdio>

#include<cstdlib>

using namespace std;

struct avl {

int d;

struct avl \*l;

struct avl \*r;

}\*r;

class avl\_tree {

public:

int height(avl \*);

int difference(avl \*);

avl \*rr\_rotat(avl \*);

avl \*ll\_rotat(avl \*);

avl \*lr\_rotat(avl\*);

avl \*rl\_rotat(avl \*);

avl \* balance(avl \*);

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

void show(avl\*, int);

void inorder(avl \*);

void preorder(avl \*);

void postorder(avl\*);

avl\_tree() {

r = NULL;

}

};

int avl\_tree::height(avl \*t) {

int h = 0;

if (t != NULL) {

int l\_height = height(t->l);

int r\_height = height(t->r);

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

h = max\_height + 1;

}

return h;

}

int avl\_tree::difference(avl \*t) {

int l\_height = height(t->l);

int r\_height = height(t->r);

int b\_factor = l\_height - r\_height;

return b\_factor;

}

avl \*avl\_tree::rr\_rotat(avl \*parent) {

avl \*t;

t = parent->r;

parent->r = t->l;

t->l = parent;

cout<<"Right-Right Rotation";

return t;

}

avl \*avl\_tree::ll\_rotat(avl \*parent) {

avl \*t;

t = parent->l;

parent->l = t->r;

t->r = parent;

cout<<"Left-Left Rotation";

return t;

}

avl \*avl\_tree::lr\_rotat(avl \*parent) {

avl \*t;

t = parent->l;

parent->l = rr\_rotat(t);

cout<<"Left-Right Rotation";

return ll\_rotat(parent);

}

avl \*avl\_tree::rl\_rotat(avl \*parent) {

avl \*t;

t = parent->r;

parent->r = ll\_rotat(t);

cout<<"Right-Left Rotation";

return rr\_rotat(parent);

}

avl \*avl\_tree::balance(avl \*t) {

int bal\_factor = difference(t);

if (bal\_factor > 1) {

if (difference(t->l) > 0)

t = ll\_rotat(t);

else

t = lr\_rotat(t);

} else if (bal\_factor < -1) {

if (difference(t->r) > 0)

t = rl\_rotat(t);

else

t = rr\_rotat(t);

}

return t;

}

avl \*avl\_tree::insert(avl \*r, int v) {

if (r == NULL) {

r = new avl;

r->d = v;

r->l = NULL;

r->r = NULL;

return r;

} else if (v< r->d) {

r->l = insert(r->l, v);

r = balance(r);

} else if (v >= r->d) {

r->r = insert(r->r, v);

r = balance(r);

} return r;

}

void avl\_tree::show(avl \*p, int l) {

int i;

if (p != NULL) {

show(p->r, l+ 1);

cout<<" ";

if (p == r)

cout << "Root -> ";

for (i = 0; i < l&& p != r; i++)

cout << " ";

cout << p->d;

show(p->l, l + 1);

}

}

void avl\_tree::inorder(avl \*t) {

if (t == NULL)

return;

inorder(t->l);

cout << t->d << " ";

inorder(t->r);

}

void avl\_tree::preorder(avl \*t) {

if (t == NULL)

return;

cout << t->d << " ";

preorder(t->l);

preorder(t->r);

}

void avl\_tree::postorder(avl \*t) {

if (t == NULL)

return;

postorder(t ->l);

postorder(t ->r);

cout << t->d << " ";

}

int main() {

int c, i;

avl\_tree avl;

while (1) {

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

cout << "2.show 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 >> c;

switch (c) {

case 1:

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

cin >> i;

r = avl.insert(r, i);

break;

case 2:

if (r == NULL) {

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

continue;

}

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

avl.show(r, 1);

cout<<endl;

break;

case 3:

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

avl.inorder(r);

cout << endl;

break;

case 4:

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

avl.preorder(r);

cout << endl;

break;

case 5:

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

avl.postorder(r);

cout << endl;

break;

case 6:

exit(0);

break;

default:

cout << "Wrong Choice" << endl;

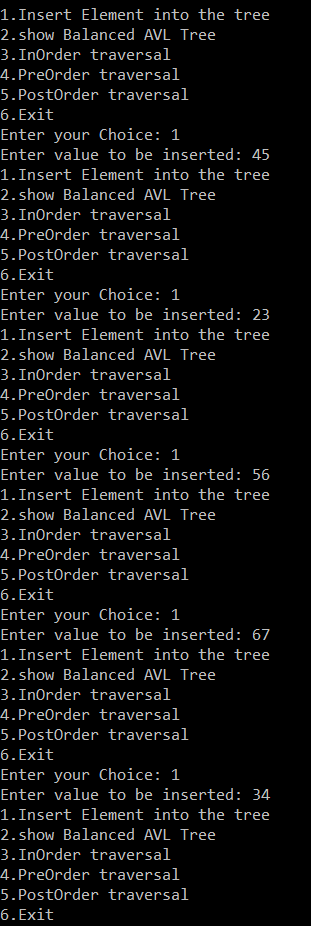
}

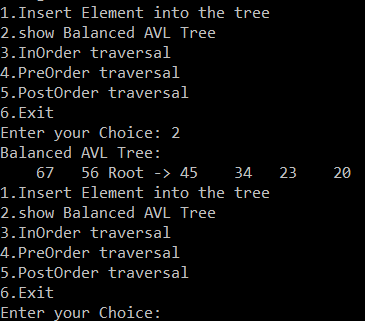
}

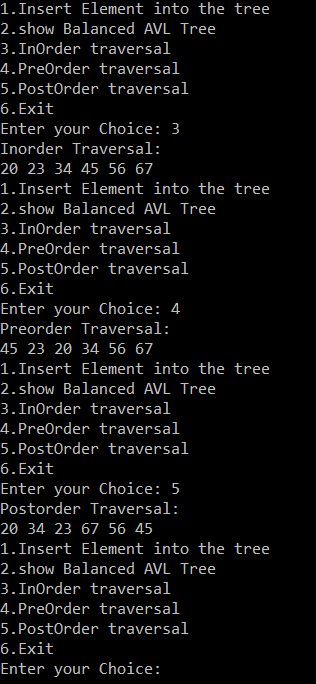
return 0;

}

**OUTPUT:**







**9.** Implement B-tree and compare its time complexity with respect to Binary Search Tree and AVL Tree.

**Source code:**

#include<iostream>

using namespace std;

using namespace std::chrono;

class Btreenode

{

int \* keys;

Btreenode\*\*c;

int t;

int n;

bool leaf;

public:

Btreenode(int \_t,bool leaf);

void insert\_non(int k);

void splitchild(int i,Btreenode\*y);

void traverse();

friend class Btree;

};

class Btree

{

Btreenode\*root;

int t;

public:

Btree(int \_t)

{

t=\_t;

}

void traverse()

{

if(root!=NULL)

{

root->traverse();

}

}

void insert(int k);

};

Btreenode ::Btreenode(int t,bool leaf)

{t=\_t;

leaf=leaf\_;

keys=new int[2\*t-1];

c=new Btreenode\*[2\*t];

n=0;}

void Btree::insert(int k)

{

if(root==NULL)

{

root=new Btreenode(t,true);

root->keys[0]=k;

root->n=1;

}

else

{

if(root->n==2\*t-1)

{

Btreenode\*s=new Btreenode(t,false);

s->c[0]=root;

s->splitchild(0,root);

int i=0;

if(s->keys[0]<k)

{

i++;

}

s->c[i]->insert\_non(k);

root=s;

}

else

root->insert\_non(k);

}

}

void Btreenode::splitchild(int i,Btreenode\*y)

{

Btreenode\*z=new Btreenode(y->t,y->leaf);

z->n=t-1;

for(int j=0;j<t-1;j++)

{

z->keys[j]=y->keys[j+t];

}

if(y->leaf==false)

{

for(int j=0;j<t;j++)

{

z->c[j]=y->c[j+t];

}

}

y->n=t-1;

for(int j=n;j>=i+1;j--)

{

this->c[j+1]=this->c[j];

}

this->c[i+1]=z;

for(int j=n-1;j>=i;j--)

{

this->keys[j+1]=this->keys[j];

}

this->keys[i]=y->keys[t-1];

n++;

}

void Btreenode::insert\_non(int k)

{

int i;

i=n-1;

if(leaf==true)

{

while(i>=0 && keys[i]>k)

{ keys[i+1]=keys[i];

i--;}

keys[i+1]=k;

n++;}

else

{

while(i>=0 && keys[i]>k)

{

i--;

}

if(c[i+1]->n==2\*t-1)

{

splitchild(i+1,c[i+1]);

if(keys[i+1]<k)

i++;

}

c[i+1]->insert\_non(k);

}}

void Btreenode::traverse()

{int i;

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

{

if(leaf==false)

{

c[i]->traverse();}

cout<<" "<<keys[i]<<" ";

}

if(leaf==false)

{

c[i]->traverse();

}

}

int main()

{

Btree t(3);

auto start = high\_resolution\_clock::now();

t.insert(10);

t.insert(20);

t.insert(30);

t.insert(40);

t.insert(50);

t.insert(25);

t.traverse();

auto stop = high\_resolution\_clock::now();

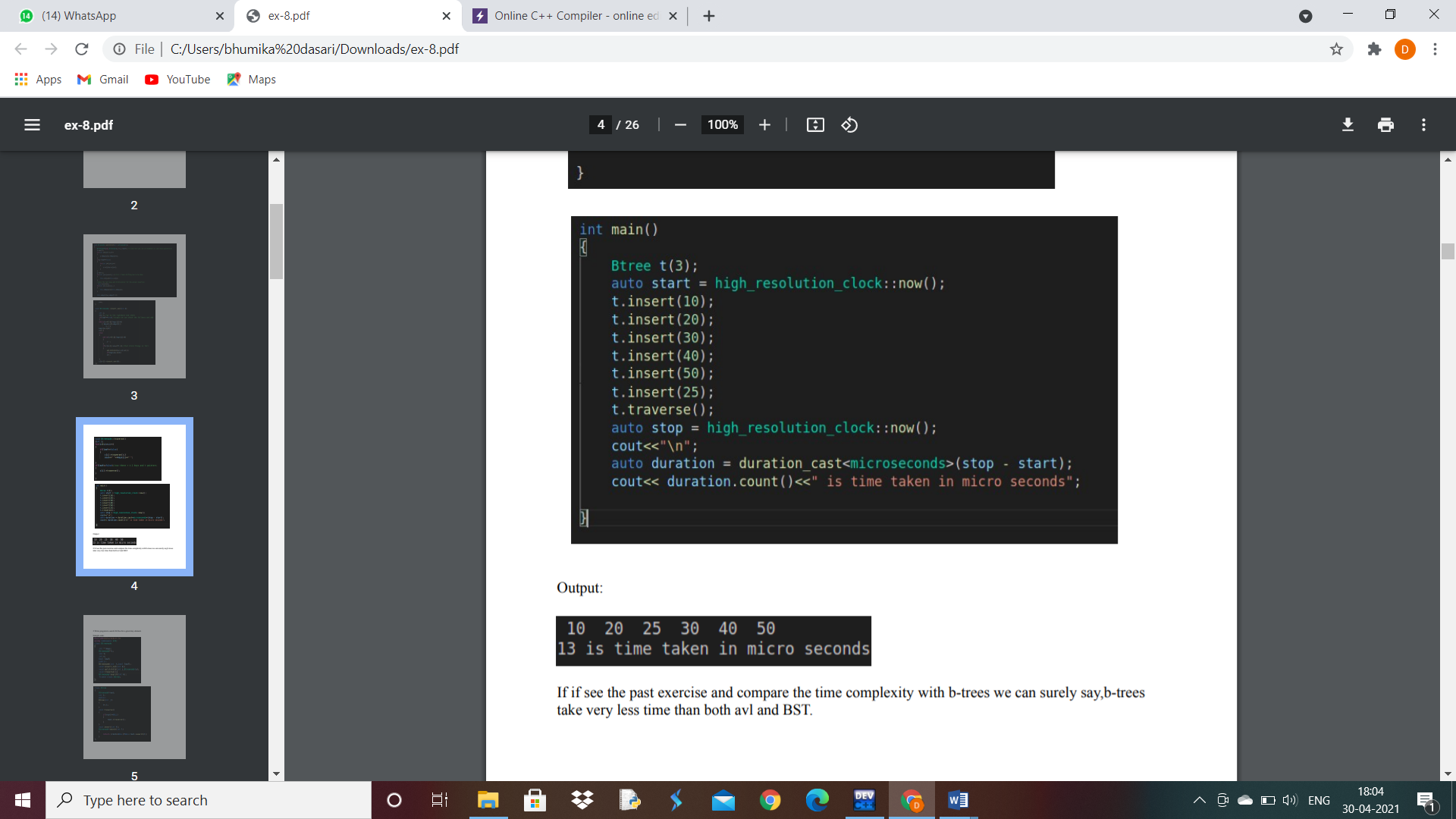
cout<<"\n";

auto duration = duration\_cast<microseconds>(stop - start);

cout<< duration.count()<<" is time taken in micro seconds";

}

**Output:**

**\**

**TIME COMPLEXITY:**

Comparing the time complexity with AVL-trees done in the past exercise, we can say that b-tree takes very less time than both AVL and BST.

**10.** Write program to search B-Tree for a given key element.

**Source code:**

#include<iostream>

using namespace std;

class Btreenode

{

int \* keys;

Btreenode\*\*c;

int t;

int n;

bool leaf;

public:

Btreenode(int \_t,bool leaf);

void insert\_non(int k);

void splitchild(int i,Btreenode\*y);

void traverse();

Btreenode\*search(int k);

friend class Btree;

};

class Btree

{

Btreenode\*root;

int t;

public:

Btree(int \_t)

{

t=\_t;

}

void traverse()

{

if(root!=NULL)

{

root->traverse();

}

}

void insert(int k);

Btreenode\*search(int l)

{

return (root==NULL)?NULL:root->search(l);

}

};

Btreenode ::Btreenode(int t,bool leaf)

{t=\_t;

leaf=leaf\_;

keys=new int[2\*t-1];

c=new Btreenode\*[2\*t];

n=0;}

void Btree::insert(int k)

{

if(root==NULL)

{

root=new Btreenode(t,true);

root->keys[0]=k;

root->n=1;

}

else

{

if(root->n==2\*t-1)

{

Btreenode\*s=new Btreenode(t,false);

s->c[0]=root;

s->splitchild(0,root);

int i=0;

if(s->keys[0]<k)

{

i++;

}

s->c[i]->insert\_non(k);

root=s;

}

else

root->insert\_non(k);

}

}

void Btreenode::splitchild(int i,Btreenode\*y)

{

Btreenode\*z=new Btreenode(y->t,y->leaf);

z->n=t-1;

for(int j=0;j<t-1;j++)

{

z->keys[j]=y->keys[j+t];

}

if(y->leaf==false)

{

for(int j=0;j<t;j++)

{

z->c[j]=y->c[j+t];

}

}

y->n=t-1;

for(int j=n;j>=i+1;j--)

{

this->c[j+1]=this->c[j];

}

this->c[i+1]=z;

for(int j=n-1;j>=i;j--)

{

this->keys[j+1]=this->keys[j];

}

this->keys[i]=y->keys[t-1];

n++;

}

void Btreenode::insert\_non(int k)

{

int i;

i=n-1;

if(leaf==true)

{

while(i>=0 && keys[i]>k)

{ keys[i+1]=keys[i];

i--;}

keys[i+1]=k;

n++;}

else

{

while(i>=0 && keys[i]>k)

{

i--;

}

if(c[i+1]->n==2\*t-1)

{

splitchild(i+1,c[i+1]);

if(keys[i+1]<k)

i++;

}

c[i+1]->insert\_non(k);

}}

void Btreenode::traverse()

{int i;

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

{

if(leaf==false)

{

c[i]->traverse();}

cout<<" "<<keys[i]<<" ";

}

if(leaf==false)

{

c[i]->traverse();

}

}

Btreenode\* Btreenode::search(int l)

{

int i=0;

while(i<n && keys[i]<l)

{

i++;

}

if(keys[i]==l)

return this;

if(leaf==true)

return NULL;

return c[i]->search(l);

}

int main()

{

Btree t(3);

for(int i=100;i<110;i++)

{

t.insert(i);

}

t.traverse();

int l;

cout<<"\nenter the element to be searched :";

cin>>l;

t.search(l)!=NULL?cout<<"Key present": cout<<"key not available";

}

**Output:**

