

Algorithms Lab LAB MANUAL CSE 2261

**CSE – Semester IV**

**List of Programs**

|  |  |  |  |
| --- | --- | --- | --- |
| **S. No** | **Program Name** | **Page No.** | **Faculty Sign** |
| 1 | Write a program to implement Binary Search Tree construction and traversal | 3 |  |
| 2 | Write a program to implement Doubly Linked List | 8 |  |
| 3 | Write a program to implement Bubble Sort | 12 |  |
| 4 | Write a program to find the GCD of two numbers using the Consecutive Integer Method | 14 |  |
| 5 | Write a program to find the GCD of two numbers using Euclid Method | 15 |  |
| 6 | Write a program to implement Selection Sort | 16 |  |
| 7 | Write a program to implement Breadth First Search of a Graph | 17 |  |
| 8 | Write a program to implement Knapsack Algorithm | 19 |  |
| 9 | Write a program to implement Insertion Sort | 21 |  |
| 10 | Write a program to find the Diameter of a Tree | 22 |  |
| 11 | Write a program to find the Height of a Tree | 25 |  |
| 12 | Write a program to implement Quick Sort | 27 |  |
| 13 | Write a program to determine the efficiency of Binary Search Tree Traversals | 30 |  |
| 14 | Write a program to construct AVL Tree | 36 |  |
| 15 | Write a program to construct a heap using the bottom-up approach | 39 |  |

# Write a program to implement Binary Search Tree construction and traversal CODE:

# include <iostream>

# include <cstdlib> using namespace std;

struct node

{

int info;

struct node \*left; struct node \*right;

}\*root ;

void insrt(node \*tree, node \*newnode)

{

if (root == NULL)

{

root = new node;

root->info = newnode->info; root->left = NULL;

root->right = NULL;

cout<<"Root Node is Added"<<endl; return;

}

if (tree->info == newnode->info)

{

cout<<"Element already in the tree"<<endl; return;

}

if (tree->info > newnode->info)

{

if (tree->left != NULL)

{

insrt(tree->left, newnode);

}

else

{

tree->left = newnode; (tree->left)->left = NULL;

(tree->left)->right = NULL; cout<<"Node Added To Left"<<endl; return;

}

}

else

{

if (tree->right != NULL)

{

insrt(tree->right, newnode);

}

else

{

tree->right = newnode; (tree->right)->left = NULL;

(tree->right)->right = NULL; cout<<"Node Added To Right"<<endl; return;

}

}

}

void preorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl; return ;

}

if (ptr != NULL)

{

cout<<ptr->info<<" "; preorder(ptr->left); preorder(ptr->right);

}

}

void inorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl; return;

}

if (ptr != NULL)

{

inorder(ptr->left); cout<<ptr->info<<" "; inorder(ptr->right);

}

}

void postorder(node \*ptr)

{

if (root == NULL)

{

cout<<"Tree is empty"<<endl; return;

}

if (ptr != NULL)

{

postorder(ptr->left); postorder(ptr->right); cout<<ptr->info<<" ";

}

}

int height(node\* node)

{

if (node == NULL) return 0;

else

{

int lheight = height(node->left); int rheight = height(node->right);

if (lheight > rheight) return(lheight + 1);

else return(rheight + 1);

}

}

void printGivenLevel(node\* root, int level)

{

if (root == NULL) return;

if (level == 1)

cout << root->info << " "; else if (level > 1)

{

printGivenLevel(root->left, level-1); printGivenLevel(root->right, level-1);

}

}

void levelorder(node \*ptr)

{

int h = height(root); int i;

for (i = 1; i <= h; i++) printGivenLevel(root, i);

}

void display(node \*ptr, int level)

{

int i;

if (ptr != NULL)

{

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

if (ptr == root) cout<<"Root: ";

else

{

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

}

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

}

}

int main()

{

int n,i=0;

struct node \*temp;

cout<<"Enter number of nodes: "; cin>>n;

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

{

cout<<"Enter node "<<i+1<<": "; temp = new node;

cin>>temp->info; insrt(root, temp); cout<<"\n";

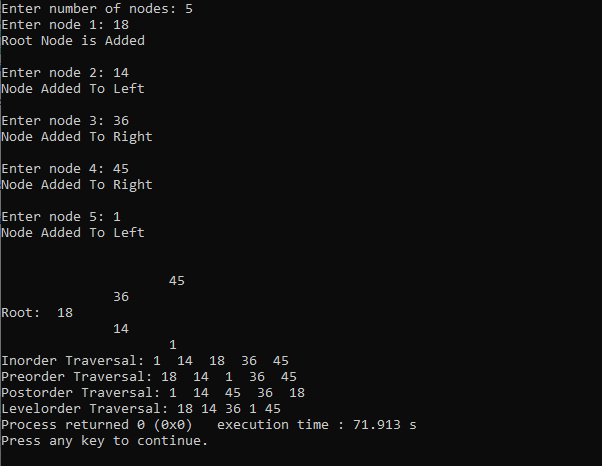
}

display(root,1); cout<<"\nInorder Traversal: "; inorder(root); cout<<"\nPreorder Traversal: "; preorder(root); cout<<"\nPostorder Traversal: "; postorder(root);

cout<<"\nLevelorder Traversal: "; levelorder(root);

}

# OUTPUT:



1. **Write a program to implement Doubly Linked List**

# CODE:

#include<iostream> #include<cstdlib> using namespace std;

struct node

{

int num;

struct node \*right; struct node \*left;

}\*f,\*r,\*temp; int insrt()

{

temp = new node; cout<<"Enter number "; cin>>temp->num;

temp->left=NULL; temp->right=NULL;

}

int isEmpty()

{

if(f == NULL && r==NULL)

{

return true;

}

return false;

}

int insrtbeg()

{

insrt();

if(isEmpty())

{

f=temp; r=temp;

}

else

{

temp->right=f; f->left = temp; f=temp;

}

cout<<"Node inserted at beginning"<<endl;

}

int insrtend()

{

insrt();

if(isEmpty())

{

f=temp; r=temp;

}

else

{

r->right = temp; temp->left = r; r= temp;

}

cout<<"Node inserted at end"<<endl;

}

void deletebeg()

{

if(isEmpty())

{

cout<<"List Empty"<<endl; return;

}

if(f->right == NULL)

{

cout<<f->num<<" Deleted"<<endl; r = NULL;

}

else

{

cout<<f->num<<" Deleted"<<endl; f->right->left=NULL;

}

f = f->right;

}

void display()

{

if(isEmpty())

{

cout<<"List Empty"<<endl;

}

else

{

temp=f;

while(temp != NULL)

{

cout<<temp->num<<" "; temp=temp->right;

}

}

}

int main()

{

int choice; char op; do

{

cout<<"Enter option: \n1.Insert at the beginning \n2.Insert at the end \n3.Delete from beginning \n4.Display"<<endl;

cin>>choice; switch(choice)

{

case 1:

insrtbeg(); break;

case 2:

insrtend(); break;

case 3:

deletebeg(); break;

case 4:

display(); break;

default:

cout<<"Invalid Entry"<<endl; break;

}

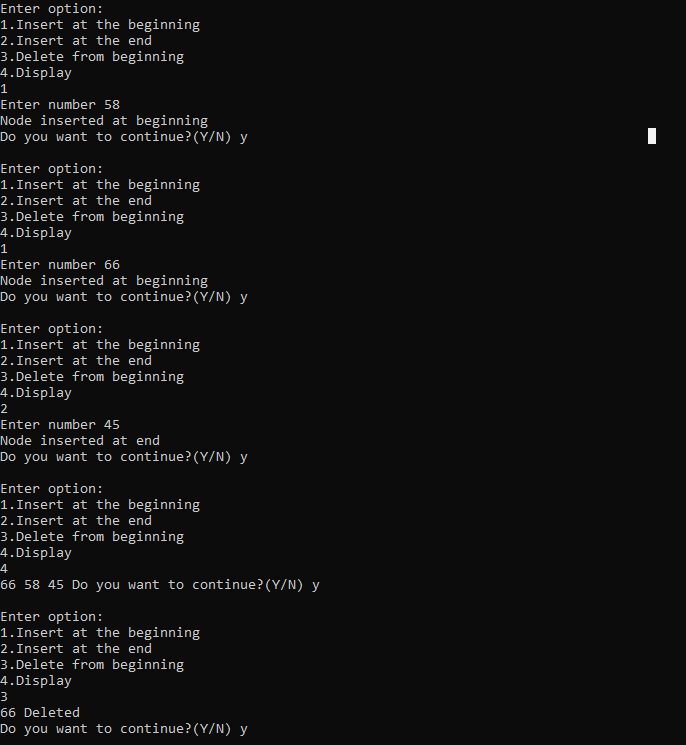
cout<<"Do you want to continue?(Y/N) "; cin>>op;

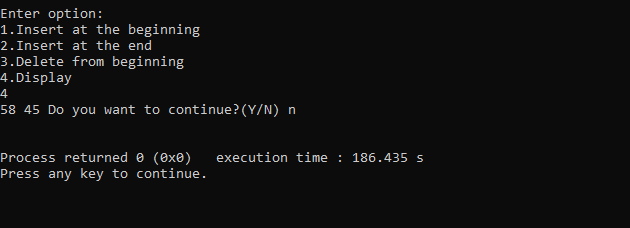
cout<<endl;

}while(op == 'y' || op == 'Y');

}

# OUTPUT:





1. **Write a program to implement Bubble Sort**

# CODE:

#include<iostream> #include<ctime> #include <stdlib.h>

using namespace std;

int bubble\_sort(int a[],int n)

{

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

{

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

{

if(a[j]>a[j+1])

{

swap(a[j],a[j+1]);

}

}

}

cout<<"\nSorted Array is:\n"; for(int i=0; i<n; i++)

cout<<a[i]<<" "; cout<<endl;

}

int main()

{

int n;

cout<<"Enter the no of elements: "; cin>>n;

int a[5];

for(int i=0;i<5;i++)

{

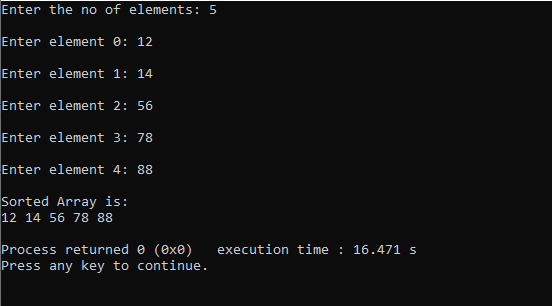
cout<<"\nEnter element "<<i<<": "; cin>>a[i];

}

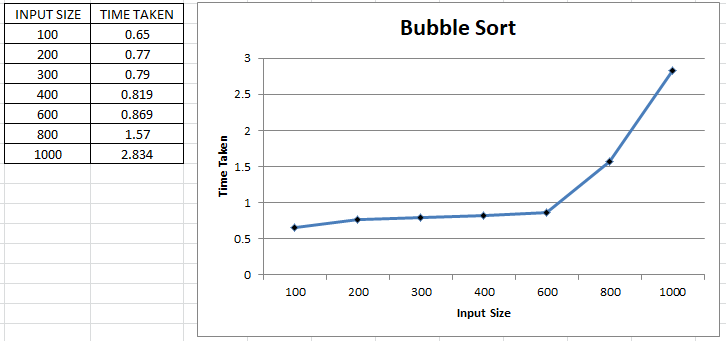
bubble\_sort(a,5);

}

# OUTPUT:



**EFFICIENCY:**



# Write a program to find the GCD of two numbers using the Consecutive Integer Method

**CODE:**

#include<iostream> using namespace std;

int gcd3(int a, int b, int t)//Consecutive Integer checking method

{

if(a==0 && b!=0)

{

return b;

}

else if(a!=0 && b==0)

{

return a;

}

else

{

while(t > 0)

{

if(a%t==0 && b%t==0) return t;

t=t-1;

}

}

}

int main()

{

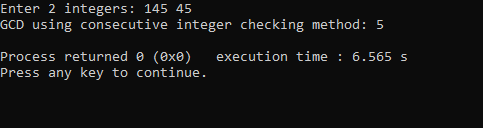
int a,b;

cout<<"Enter 2 integers: "; cin>>a>>b;

cout<<"GCD using consecutive integer checking method: "<<gcd3(a,b,min(a,b))<<endl;

}

# OUTPUT:



1. **Write a program to find GCD of Two numbers using the Euclid Method CODE:**

#include<iostream> using namespace std;

int gcd1(int a, int b)//Euclids A=B.Q+R (Simplest)

{

int r=0;

if(a !=0 && b!=0)

{

/\*r = a%b; a=b; b=r;\*/

gcd1(b,a%b);

}

if(a==0 && b!=0)

{

return b;

}

if(a!=0 && b==0)

{

return a;

}

}

int main()

{

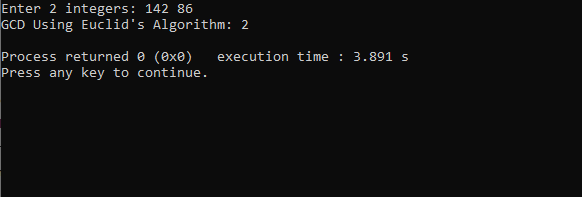
int a,b;

cout<<"Enter 2 integers: "; cin>>a>>b;

cout<<"GCD Using Euclid's Algorithm: "<<gcd1(a,b)<<endl;

}

# OUTPUT:



1. **Write a program to implement Selection Sort**

# CODE:

#include<iostream> #include<ctime> #include <stdlib.h>

using namespace std;

int selection\_sort(int a[], int n)

{ int sml=0;

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

{

sml = i;

for(int j=i+1; j<n; j++)

{

if(a[sml]>a[j])

{

sml= j;

}

}

swap(a[i],a[sml]);

}

cout<<"\nSorted Array is:\n"; for(int i=0; i<n; i++)

cout<<a[i]<<" "; cout<<endl;

// time\_req = clock() - time\_req;

// cout<<"Time taken: "<<time\_req<<"seconds"<<endl;

}

int main()

{

int n=5,i;

//cout<<"Enter the no of elements";

//cin>>n; int a[10];

for(i=0;i<5;i++)

{

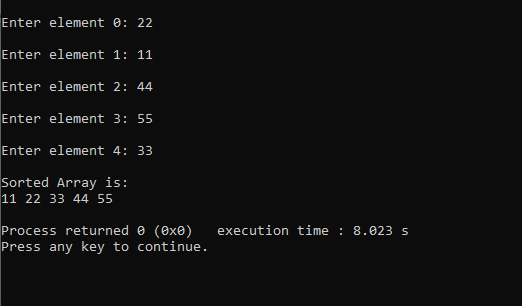
cout<<"\nEnter element "<<i<<": "; cin>>a[i];

}

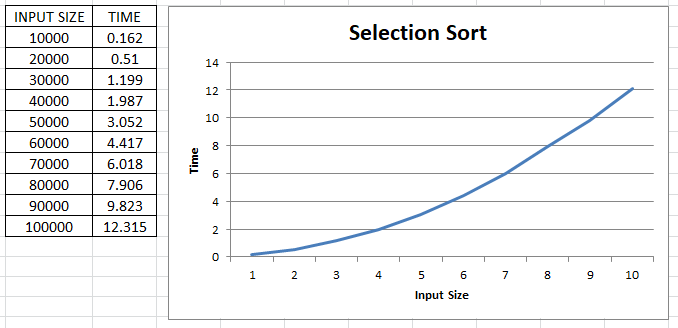
selection\_sort(a,5);

}

# OUTPUT:



**EFFICIENCY:**



# Write a program to implement Breadth First Search of a Graph CODE:

#include<iostream> #include <list>

using namespace std; class Graph

{

int vertices; list<int> \*adj;

public:

Graph(int V);

void add\_edge(int src, int des); void BFS(int src);

};

Graph::Graph(int V)

{

this->vertices = V;

adj = new list<int>[V];

}

void Graph::add\_edge(int src, int des)

{

adj[src].push\_back(des);

}

void Graph::BFS(int src)

{

bool \*visited = new bool[vertices]; for(int i=0; i<vertices; i++)

{

visited[i] = false;

}

list<int> queue;

visited[src] = true; queue.push\_back(src); list<int>::iterator i;

do

{

int temp=queue.front(); cout<<temp<<" "; queue.pop\_front();

for(i = adj[src].begin(); i!=adj[src].end(); i++)

{

if(!visited[\*i])

{

visited[\*i] = true; queue.push\_back(\*i);

}

}

}while(!queue.empty());

}

int main()

{

Graph g(4); g.add\_edge(0, 1);

g.add\_edge(0, 2);

g.add\_edge(1, 2);

g.add\_edge(2, 0);

g.add\_edge(2, 3);

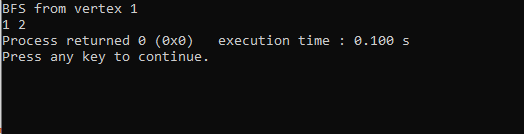
g.add\_edge(3, 3);

cout << "BFS from vertex 1 \n"; g.BFS(1);

return 0;

}

# OUTPUT:



1. **Write a program to solve the Knapsack algorithm problem CODE:**

#include <iostream> #include <ctime> using namespace std; int max(int x, int y) {

return (x > y) ? x : y;

}

int knapSack(int B, int w[], int v[], int n) { int i, wt;

int K[n + 1][B + 1];

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

for (wt = 0; wt <= B; wt++) { if (i == 0 || wt == 0) K[i][wt] = 0;

else if (w[i - 1] <= wt)

K[i][wt] = max(v[i - 1] + K[i - 1][wt - w[i - 1]], K[i - 1][wt]); else

K[i][wt] = K[i - 1][wt];

}

}

return K[n][B];

//CALCULATE AND RETURNS THE VALUE

}

int main() {

cout << "Enter the number of items :"; int n, B;

cin >> n;

int v[n], w[n];

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

cout << "Enter value and weight for item " << i << ":"; cin >> v[i];

cin >> w[i];

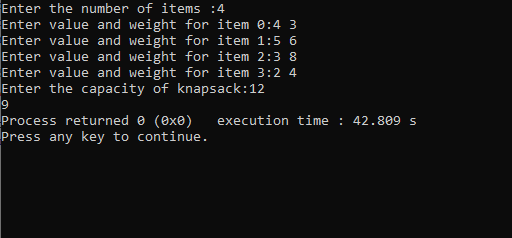
}

cout << "Enter the capacity of knapsack:"; cin >> B;

cout << knapSack(B, w, v, n); return 0;

}

# OUTPUT:



1. **Write a program to implement Insertion Sort CODE:**

#include<iostream> #include <bits/stdc++.h> #include<ctime> #include <stdlib.h> using namespace std;

void print(int a[], int n)

{

int i;

for (i = 0; i < n; i++) cout << a[i] << " ";

cout << endl;

}

void insertionsort(int a[], int n)

{

int i, key, j;

for (i = 1; i < n; i++)

{

key = a[i]; j = i - 1;

while (j >= 0 && a[j] > key)

{

a[j + 1] = a[j]; j = j - 1;

}

a[j + 1] = key;

}

print(a,n);

}

int main()

{

int a[6],n;

cout<<"Enter the no of elements: "; cin>>n;

cout<<endl;

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

{

cout<<"Element "<<i+1<<": "; cin>>a[i];

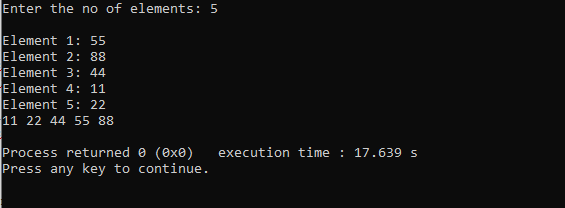
}

insertionsort(a,n);

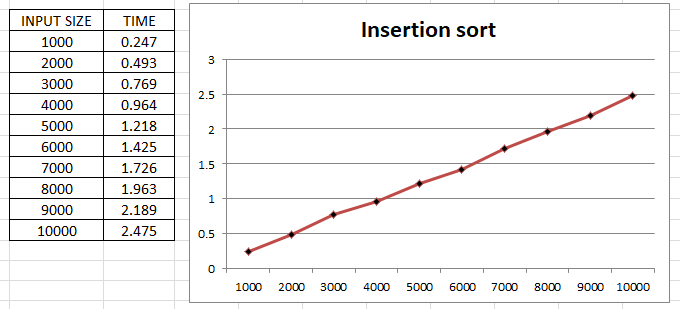
//cout<<"sorted array is "<<print(a,n);

}

# OUTPUT:



**EFFICIENCY:**



# Write a program to find the Diameter of a Tree CODE:

# include <iostream> # include <cstdlib> using namespace std;

struct node

{

int info;

struct node \*left; struct node \*right;

}\*root ;

void insrt(node \*tree, node \*newnode)

{

if (root == NULL)

{

root = new node;

root->info = newnode->info; root->left = NULL;

root->right = NULL;

cout<<"Root Node is Added"<<endl; return;

}

if (tree->info == newnode->info)

{

cout<<"Element already in the tree"<<endl; return;

}

if (tree->info > newnode->info)

{

if (tree->left != NULL)

{

insrt(tree->left, newnode);

}

else

{

tree->left = newnode; (tree->left)->left = NULL;

(tree->left)->right = NULL; cout<<"Node Added To Left"<<endl; return;

}

}

else

{

if (tree->right != NULL)

{

insrt(tree->right, newnode);

}

else

{

tree->right = newnode; (tree->right)->left = NULL;

(tree->right)->right = NULL; cout<<"Node Added To Right"<<endl; return;

}

}

}

int height(struct node\* node)

{

if (node == NULL) return 0;

return 1 + max(height(node->left), height(node->right));

}

int diameter(struct node\* tree)

{

if (tree == NULL) return 0;

int lheight = height(tree->left); int rheight = height(tree->right);

int ldiameter = diameter(tree->left); int rdiameter = diameter(tree->right);

return max(lheight + rheight + 1, max(ldiameter, rdiameter));

}

int main()

{

int n,i=0;

struct node \*temp;

cout<<"Enter number of nodes: "; cin>>n;

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

{

cout<<"Enter node "<<i+1<<": "; temp = new node;

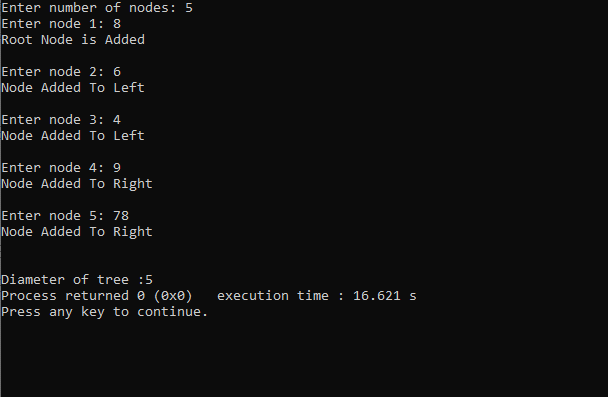
cin>>temp->info; insrt(root, temp); cout<<"\n";

}

cout<<"\nDiameter of tree :"; cout<<diameter(root);

}

# OUTPUT:



1. **Write a program to find the Height of a Tree CODE:**

# include <iostream> # include <cstdlib> using namespace std;

struct node

{

int info;

struct node \*left; struct node \*right;

}\*root ;

void insrt(node \*tree, node \*newnode)

{

if (root == NULL)

{

root = new node;

root->info = newnode->info; root->left = NULL;

root->right = NULL;

cout<<"Root Node is Added"<<endl; return;

}

if (tree->info == newnode->info)

{

cout<<"Element already in the tree"<<endl; return;

}

if (tree->info > newnode->info)

{

if (tree->left != NULL)

{

insrt(tree->left, newnode);

}

else

{

tree->left = newnode; (tree->left)->left = NULL;

(tree->left)->right = NULL; cout<<"Node Added To Left"<<endl; return;

}

}

else

{

if (tree->right != NULL)

{

insrt(tree->right, newnode);

}

else

{

tree->right = newnode; (tree->right)->left = NULL;

(tree->right)->right = NULL; cout<<"Node Added To Right"<<endl; return;

}

}

}

int height(struct node\* node)

{

if (node == NULL) return 0;

return 1 + max(height(node->left), height(node->right));

}

int main()

{

int n,i=0;

struct node \*temp;

cout<<"Enter number of nodes: "; cin>>n;

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

{

cout<<"Enter node "<<i+1<<": "; temp = new node;

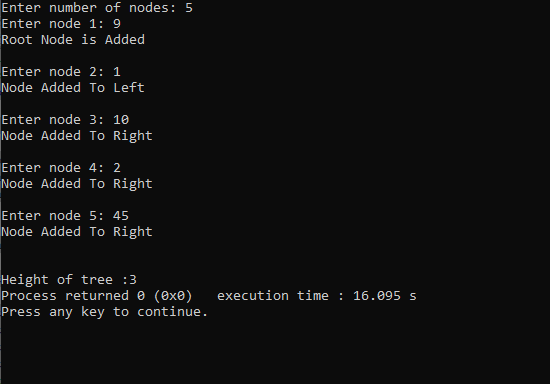
cin>>temp->info; insrt(root, temp); cout<<"\n";

}

cout<<"\nHeight of tree :"; cout<<height(root);

}

# OUTPUT:



1. **Write a program to implement Quick Sort CODE:**

#include <bits/stdc++.h> using namespace std;

void swap(int\* a, int\* b)

{

int t = \*a;

\*a = \*b;

\*b = t;

}

//last pivot

int partition (int arr[], int low, int high)

{

int pivot = arr[high]; int i = (low - 1);

for (int j = low; j <= high - 1; j++)

{

if (arr[j] < pivot)

{

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]); return (i + 1);

}

void quickSort(int arr[], int low, int high)

{

if (low < high)

{

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1); quickSort(arr, pi + 1, high);

}

}

void printArray(int arr[], int size)

{

int i;

for (i = 0; i < size; i++) cout << arr[i] << " ";

cout << endl;

}

int main()

{

int m;

cout<<"Enter number of elements: "; cin>>m;

int arr[m];

for(int i=0;i<m;i++)

{

cout<<"Enter element "<<i+1<<": "; cin>>arr[i];

}

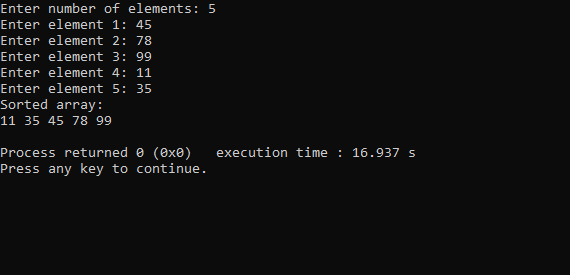
int n = sizeof(arr) / sizeof(arr[0]); quickSort(arr, 0, n - 1);

cout << "Sorted array: \n"; printArray(arr, n);

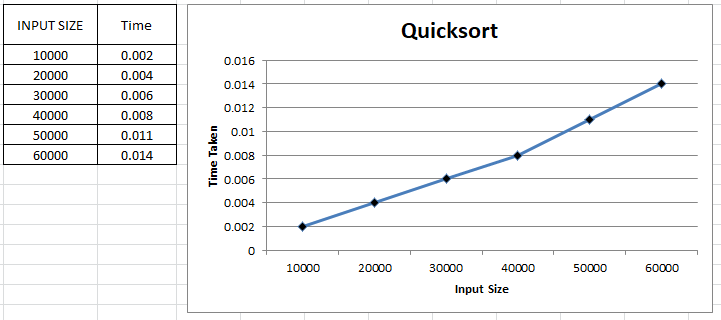
return 0;

}

# OUTPUT:



**EFFICIENCY:**



# Write a program to determine the efficiency of Binary Search Tree Traversals CODE:

# include <iostream> # include <cstdlib>

# include<ctime> using namespace std;

struct node

{

int info;

struct node \*left; struct node \*right;

}\*root ;

void insrt(node \*tree, node \*newnode)

{

if (root == NULL)

{

root = new node;

root->info = newnode->info; root->left = NULL;

root->right = NULL;

//cout<<"Root Node is Added"<<endl; return;

}

if (tree->info == newnode->info)

{

//cout<<"Element already in the tree"<<endl; return;

}

if (tree->info > newnode->info)

{

if (tree->left != NULL)

{

insrt(tree->left, newnode);

}

else

{

tree->left = newnode; (tree->left)->left = NULL;

(tree->left)->right = NULL;

//cout<<"Node Added To Left"<<endl; return;

}

}

else

{

if (tree->right != NULL)

{

insrt(tree->right, newnode);

}

else

{

tree->right = newnode; (tree->right)->left = NULL;

(tree->right)->right = NULL;

//cout<<"Node Added To Right"<<endl; return;

}

}

}

void preorder(node \*ptr)

{

if (root == NULL)

{

//cout<<"Tree is empty"<<endl; return ;

}

if (ptr != NULL)

{

//cout<<ptr->info<<" "; preorder(ptr->left); preorder(ptr->right);

}

}

void inorder(node \*ptr)

{

if (root == NULL)

{

//cout<<"Tree is empty"<<endl; return;

}

if (ptr != NULL)

{

inorder(ptr->left);

//cout<<ptr->info<<" "; inorder(ptr->right);

}

}

void postorder(node \*ptr)

{

if (root == NULL)

{

//cout<<"Tree is empty"<<endl;

return;

}

if (ptr != NULL)

{

postorder(ptr->left); postorder(ptr->right);

//cout<<ptr->info<<" ";

}

}

int height(node\* node)

{

if (node == NULL) return 0;

else

{

int lheight = height(node->left); int rheight = height(node->right);

if (lheight > rheight) return(lheight + 1);

else return(rheight + 1);

}

}

void printGivenLevel(node\* root, int level)

{

if (root == NULL) return;

if (level == 1) root->info ;

else if (level > 1)

{

printGivenLevel(root->left, level-1); printGivenLevel(root->right, level-1);

}

}

void levelorder(node \*ptr)

{

int h = height(root); int i;

for (i = 1; i <= h; i++) printGivenLevel(root, i);

}

void display(node \*ptr, int level)

{

int i;

if (ptr != NULL)

{

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

if (ptr == root) cout<<"Root: ";

else

{

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

}

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

}

}

int main()

{

const int n =40000; int i=0,x;

clock\_t time\_req; struct node \*temp;

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

{

x = rand();

//cout<<"Enter node "<<i+1<<": "; temp = new node;

temp->info = x; insrt(root, temp);

//cout<<"\n";

}

//display(root,1); time\_req = clock();

cout << "Timer starts"<<endl; cout<<"\nInorder Traversal: "; inorder(root);

cout << "\nTimer ends"<<endl; time\_req = clock() - time\_req;

double time\_taken = ((double)time\_req)/CLOCKS\_PER\_SEC; cout<<"Time taken: "<<time\_taken<<endl;

time\_req = clock();

cout << "Timer starts"<<endl; cout<<"\nPreorder Traversal: "; preorder(root);

cout << "\nTimer ends"<<endl; time\_req = clock() - time\_req;

time\_taken = ((double)time\_req)/CLOCKS\_PER\_SEC; cout<<"Time taken: "<<time\_taken<<endl;

time\_req = clock();

cout << "Timer starts"<<endl; cout<<"\nPostorder Traversal: "; postorder(root);

cout << "\nTimer ends"<<endl; time\_req = clock() - time\_req;

time\_taken = ((double)time\_req)/CLOCKS\_PER\_SEC; cout<<"Time taken: "<<time\_taken<<endl;

time\_req = clock();

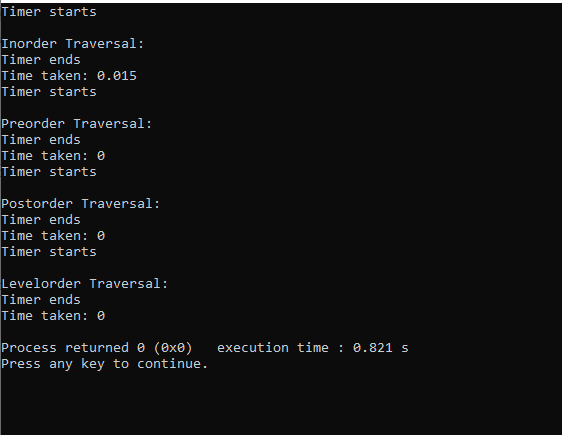
cout << "Timer starts"<<endl; cout<<"\nLevelorder Traversal: "; levelorder(root);

cout << "\nTimer ends"<<endl; time\_req = clock() - time\_req;

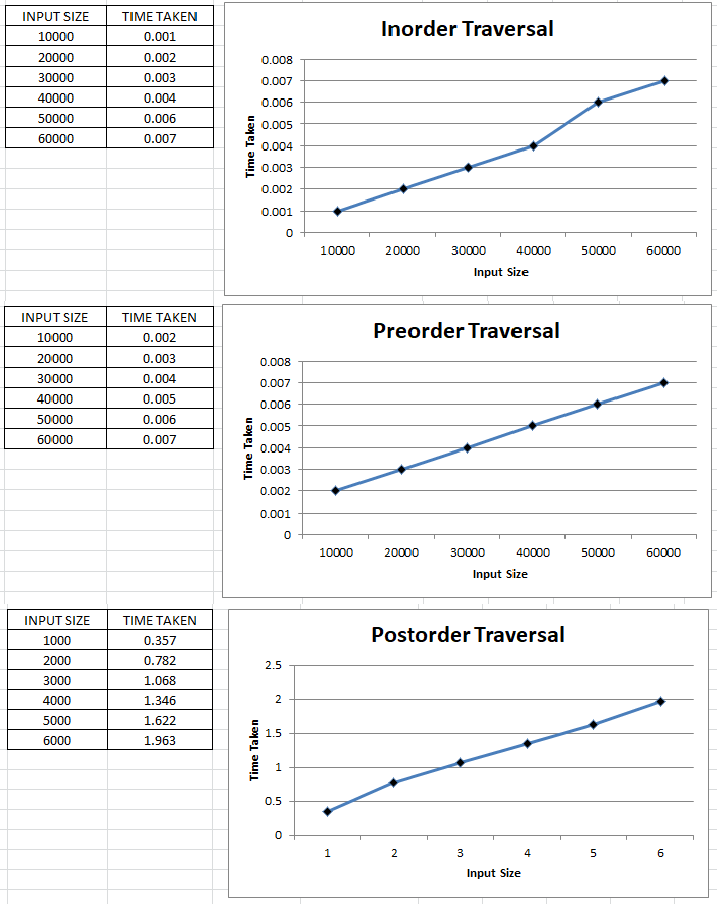
time\_taken = ((double)time\_req)/CLOCKS\_PER\_SEC; cout<<"Time taken: "<<time\_taken<<endl;

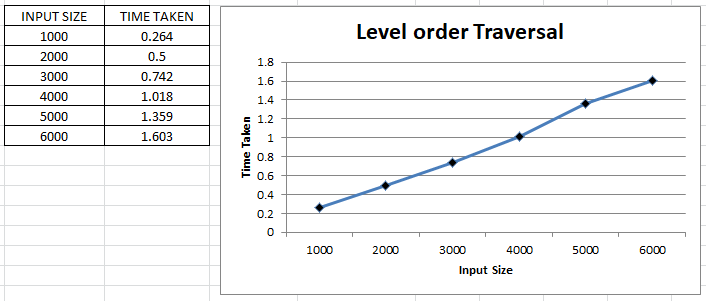
}

# OUTPUT:



**EFFICIENCY**





# Write a program to construct the AVL Tree

**CODE:**

#include <iostream> using namespace std;

class Node{ public:

int key; Node \*left; Node \*right; int height;

};

int max(int a, int b);

int height(Node \*N){ if (N == NULL)

return 0; return N->height;

}

int max(int a, int b){ return (a > b)? a : b;

}

Node\* newNode(int key){ Node\* node = new Node(); node->key = key;

node->left = NULL; node->right = NULL; node->height = 1;

return(node);

}

Node \*rightRotate(Node \*y){ Node \*x = y->left;

Node \*T2 = x->right;

x->right = y; y->left = T2;

y->height = max(height(y->left),

height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1; return x;

}

Node \*leftRotate(Node \*x){ Node \*y = x->right; Node \*T2 = y->left;

y->left = x;

x->right = T2;

x->height = max(height(x->left),

height(x->right)) + 1; y->height = max(height(y->left),

height(y->right)) + 1;

return y;

}

int getBalance(Node \*N){ if (N == NULL)

return 0;

return height(N->left) - height(N->right);

}

Node\* insert(Node\* node, int key){ if (node == NULL)

return(newNode(key));

if (key < node->key)

node->left = insert(node->left, key); else if (key > node->key)

node->right = insert(node->right, key); else

return node;

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalance(node);

if (balance > 1 && key < node->left->key) return rightRotate(node);

if (balance < -1 && key > node->right->key) return leftRotate(node);

if (balance > 1 && key > node->left->key)

{

node->left = leftRotate(node->left); return rightRotate(node);

}

if (balance < -1 && key < node->right->key)

{

node->right = rightRotate(node->right); return leftRotate(node);

}

return node;

}

void Inorder(Node\* root){ if(root==NULL) return; Inorder(root->left); cout<< root->key<< " "; Inorder(root->right);

}

int main(){

Node \*root = NULL; int n,x;

cout<<"Enter number of elements: "; cin>>n;

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

{

cout<<"Enter element "<<i+1<<": "; cin>>x;

root = insert(root, x);

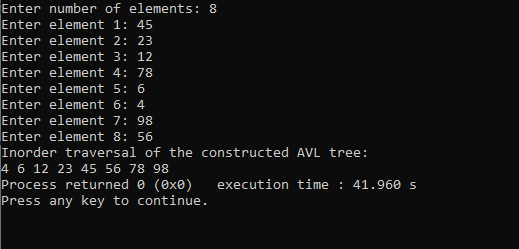
}

cout << "Inorder traversal of the constructed AVL tree: \n"; Inorder(root);

return 0;

}

# OUTPUT:



1. **Write a program to construct a heap using the bottom-up approach CODE:**

#include <iostream>

using namespace std;

void heapify(int arr[], int n, int i)

{

int largest = i; int l = 2 \* i + 1; int r = 2 \* i + 2;

if (l < n && arr[l] > arr[largest]) largest = l;

if (r < n && arr[r] > arr[largest]) largest = r;

if (largest != i) {

swap(arr[i], arr[largest]); heapify(arr, n, largest);

}

}

void buildHeap(int arr[], int n)

{

int startIdx = (n / 2) - 1;

for (int i = startIdx; i >= 0; i--) { heapify(arr, n, i);

}

}

void printHeap(int arr[], int n)

{

cout << "Array representation of Heap is:\n";

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

cout << arr[i] << " "; cout << "\n";

}

int main()

{ int n;

cout<<"Enter number of elements: "; cin>>n;

int arr[n];

cout<<"Enter elements: "; for(int i=0; i<n; i++)

{

cin>>arr[i];

}

int m = sizeof(arr) / sizeof(arr[0]); buildHeap(arr, m);

printHeap(arr, m); return 0;

}

# OUTPUT:

