**Name: Saba Zulfiqar**

**Roll NO :SU92-BSSEM-S24-107**

**Section : 3A**

**Lab 10** -

**Stack with Linkedlist and Array**

**Tasks:**

**1. With Array; Push, Pop, Display**

**2. With Linkedlist; Push, Pop, Display.**

#include<iostream>

using namespace std ;

const int size = 100;

int stack [size];

int top = -1;

void push(int value){

if (top == size - 1){

cout<<"stack overflow";

}

else{

top++;

stack[top] = value;

}

}

void pop(){

if (top == -1){

cout<<"stack overflow";

}

else{

cout<<"popped : "<<stack[top]<<endl;

top--;

}

}

void display(){

if(top == -1){

cout<<"stack is empty";

}

else{

cout<<"stack elements:" ;

for (int i = top ; i>=0 ; i--){

cout << stack[i] << " ";

}

}

cout << endl;

}

int main() {

push(10);

push(20);

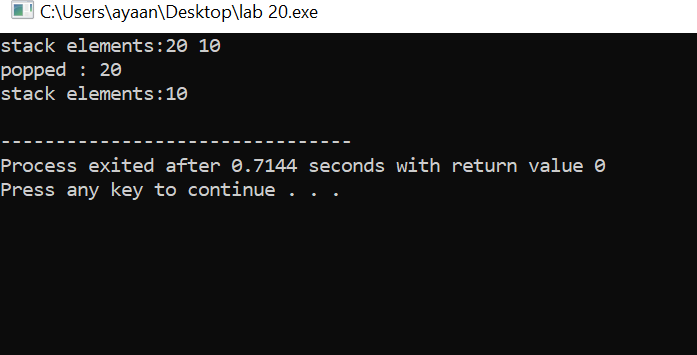
display();

pop();

display();

return 0;

}



**//Stack using Linked List**

struct Node{

int data;

Node\* next;

};

Node\*top = nullptr;

void push ( int value) {

Node \*newNode = new Node;

newNode->data = value;

newNode->next = top;

top = newNode;

}

void pop(){

if (top == nullptr) {

cout<<"stack underflow";

}

else{

cout<< "Popped : "<<top->data<<endl;

Node\* temp = top;

top = top ->next;

delete temp;

}

}

void display (){

if (top == nullptr){

cout<<"stack is empty" ;

}

else{

cout<<"stack elements"<<endl;

Node\* current = top;

while (current != nullptr) {

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

}

int main() {

push(100);

push(200);

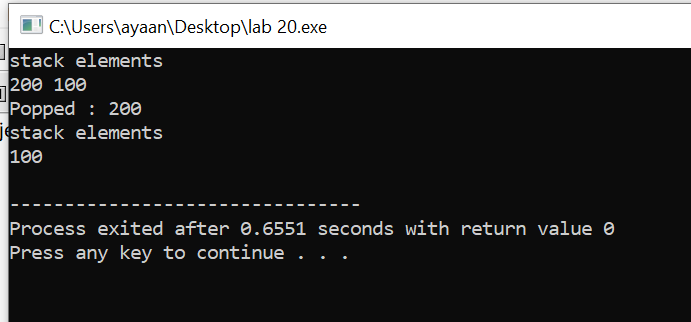
display();

pop();

display();

return 0;

}



**Explanation:**

This code shows how a **stack** works in two ways — using an **array** and a **linked list**.  
In both, you can **push** (add) items, **pop** (remove) the top item, and **display** the current stack.  
The **array version** has a fixed size, so it can become **"full"** (overflow).  
The **linked list version** can grow dynamically, so it's more flexible and doesn’t overflow unless memory is full.  
The **top** always points to the latest added item — which is the first one to be removed (LIFO).  
**Array-based stack**: Uses a fixed-size array to store elements.

**Linked list-based stack**: **Linked list-based stack**: Uses dynamically created nodes where each Node holds a value and a pointer to the next.

**Lab 11 - Queue with Linkedlist and Array**

**Tasks:**

1. **With Array; Enqueue, Dequeue, Display**

#include <iostream>

using namespace std;

const int size = 100;

int queueArray[size];

int f = -1, rear = -1;

void enqueue(int value) {

if (rear == size - 1) {

cout << "Queue Overflow";

} else {

if (f == -1) f = 0;

queueArray[++rear] = value;

}

}

void dequeue() {

if (f == -1 || f > rear) {

cout << "Queue Underflow";

} else {

cout << "Dequeued: " << queueArray[f++] << endl;

}

}

void display() {

if (f == -1 || f > rear) {

cout << "Queue is empty";

} else {

cout << "Queue elements: ";

for (int i = f; i <= rear; i++) {

cout << queueArray[i] << " ";

}

cout << endl;

}

}

int main() {

enqueue(10);

enqueue(20);

enqueue(30);

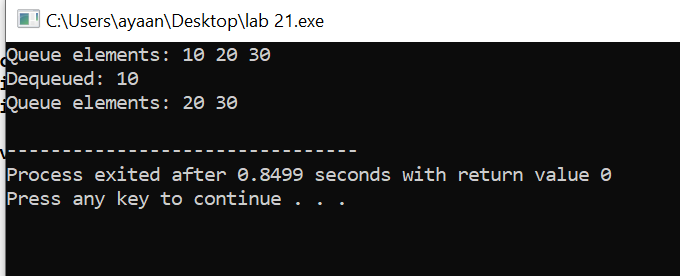
display();

dequeue();

display();

return 0;

}



1. **With Linkedlist; Enqueue, Dequeue, Display**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

Node\* f = nullptr;

Node\* r = nullptr;

//

void enqueue(int v){

Node\* newNode = new Node;

newNode->data = v;

newNode->next = nullptr;

if (r == nullptr) {

f = r= newNode;

} else {

r->next = newNode;

r = newNode;

}

}

//

void dequeue() {

if (f == nullptr) {

cout << "Queue Underflow";

} else {

cout << "Dequeued: " << f->data << endl;

Node\* temp = f;

f = f->next;

if (f == nullptr) r = nullptr;

delete temp;

}

}

void display() {

if (f == nullptr) {

cout << "Queue is empty";

} else {

cout << "Queue elements: ";

Node\* current = f;

while (current != nullptr) {

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

}

//

int main() {

enqueue(100);

enqueue(200);

enqueue(300);

display();

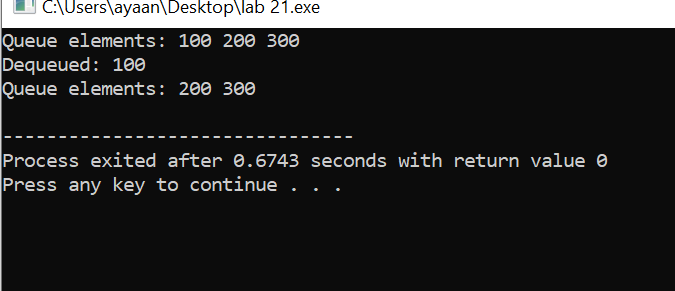
dequeue();

display();

return 0;

}

//



**Explanation:**

This code shows how a **queue** works in two ways using an **array** and a **linked list**.  
In a queue, elements are added from the **rear** (enqueue) and removed from the **front** (dequeue), following the **FIFO** rule . First In, First Out.  
The **array version** uses a fixed-size array where we track front and rear positions to insert or remove elements.  
If the array gets full or empty, it shows helpful messages like **“Queue Overflow”** or **“Queue Underflow.”**  
The **linked list version** creates new nodes as needed, so it grows dynamically and doesn’t have a fixed size.  
Both versions have a **display()** function to show current elements from front to rear.

**Lab 12 - BST and AVL**

1. **Insert and Traverse for BST**

#include<iostream>

#include <algorithm>

using namespace std;

struct Node{

int key;

Node\* left;

Node\* right;

Node (int val){

key = val;

left = nullptr;

right = nullptr;

}

};

Node\* insert(Node\* root , int key ){

if(!root){

return new Node(key);

}

if(key<root->key)

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

else

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

return root;

}

void inorder(Node\* root){

if(root){

inorder(root->left) ;

cout<<root->key <<" ";

inorder(root->right) ;

}

}

int main() {

Node\* Root = nullptr;

int arr[] = {30, 20, 40, 10, 25};

for (int val : arr)

Root = insert(Root, val);

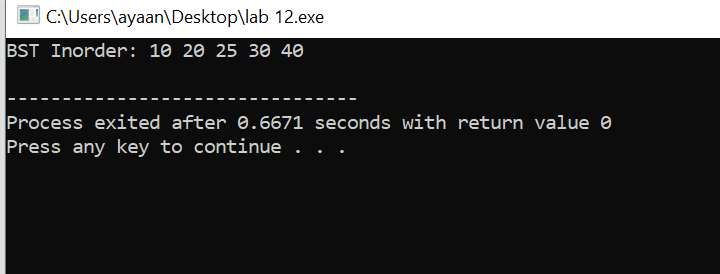
cout << "BST Inorder: ";

inorder(Root);

cout << endl;

return 0;

}



**2. Insert and Traverse for AVL**

#include<iostream>

#include<algorithm>

using namespace std;

struct AVLNode {

int key, height;

AVLNode \*left, \*right;

AVLNode(int val) {

key = val;

height = 1;

left = nullptr;

right = nullptr;

}

};

int getHeight(AVLNode\* node) {

return node ? node->height : 0;

}

int getbalance(AVLNode\* node) {

return node ? getHeight(node->left) - getHeight(node->right) : 0;

}

AVLNode\* rightRotate(AVLNode\* y) {

AVLNode\* x = y->left;

AVLNode\* T2 = x->right;

x->right = y;

y->left = T2;

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

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

return x;

}

AVLNode\* leftRotate(AVLNode\* x) {

AVLNode\* y = x->right;

AVLNode\* T2 = y->left;

y->left = x;

x->right = T2;

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

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

return y;

}

AVLNode\* insertAVL(AVLNode\* root, int key) {

if (!root) return new AVLNode(key);

if (key < root->key)

root->left = insertAVL(root->left, key);

else if (key > root->key)

root->right = insertAVL(root->right, key);

else

return root;

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

int balance = getbalance(root);

if (balance > 1 && key < root->left->key)

return rightRotate(root);

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

return leftRotate(root);

if (balance > 1 && key > root->left->key) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

if (balance < -1 && key < root->right->key) {

root->right = rightRotate(root->right);

return leftRotate(root);

}

return root;

}

void inorderAVL(AVLNode\* root) {

if (root) {

inorderAVL(root->left);

cout << root->key << " ";

inorderAVL(root->right);

}

}

int main() {

AVLNode\* avlRoot = nullptr;

int arr[] = {40, 20, 30, 50, 25};

for (int val : arr)

avlRoot = insertAVL(avlRoot, val);

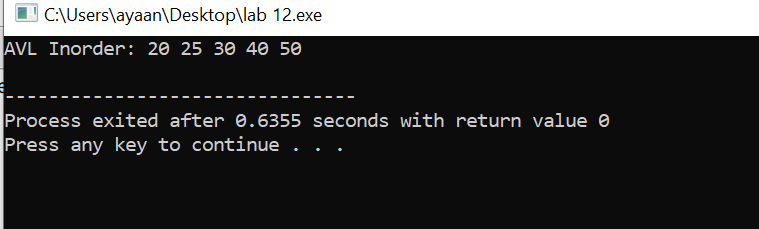
cout << "AVL Inorder: ";

inorderAVL(avlRoot);

cout << endl;

return 0;

}



**Explanation:**

This code demonstrates how to **insert and traverse** in two types of binary trees: **BST (Binary Search Tree)** and **AVL Tree (self-balancing BST)**.  
In both, elements are inserted based on comparisons smaller to the left, larger to the right and then printed in **inorder** (which gives sorted output).  
The **BST version** is basic and doesn’t handle imbalance, so it can become skewed like a linked list if inserted in order.  
The **AVL version** fixes that using **rotations** (left/right) to stay balanced after every insertion.  
It uses a **balance factor** to check if the tree is leaning too much to one side and then rotates accordingly.  
The AVL tree keeps search time efficient by maintaining height balance — great for real-time systems or databases.

**Lab 13 - DFS and BFS**

1. **Insert and Traverse for DFS in tree**

#include<iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node (int val){

data = val;

left = right = nullptr;

}

};

void inorder(Node\* root){

if(root){

inorder(root->left) ;

cout<< root->data<<" ";

inorder(root->right);

}

}

void preorder(Node\* root){

if(root){

cout<< root->data<<" ";

preorder(root->left) ;

preorder(root->right);

}

}

void postorder(Node\* root){

if(root){

postorder(root->left);

postorder(root->right);

cout<< root->data<<" ";

}

}

int main() {

Node\* root = new Node(10);

root->left = new Node(5);

root->right= new Node(15);

root->left->left = new Node(2);

root->left->right = new Node(7);

root->right->right = new Node(20);

cout << "Inorder Traversal: ";

inorder(root);

cout << "\nPreorder Traversal: ";

preorder(root);

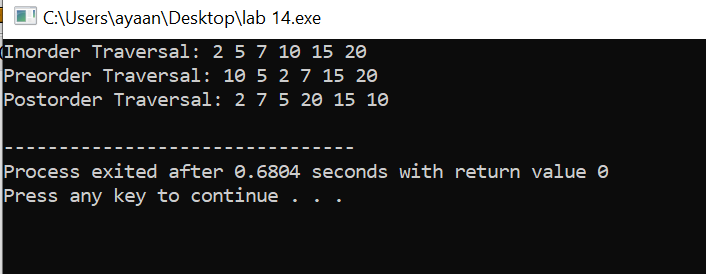
cout << "\nPostorder Traversal: ";

postorder(root);

cout << endl;

return 0;

}



**Explanation:**

**Tree Traversals (Inorder, Preorder, Postorder)**:  
A binary tree is created with nodes having values. Three functions print the tree using different traversal order.

**Inorder (LNR)**: left → node → right

**Preorder (NLR)**: node → left → right

**Postorder (LRN)**: left → right → node

Each gives a different view of how the tree is explored.

1. **Insert and Traverse for DFS in graph**

#include <iostream>

#include <vector>

using namespace std;

void DFS\_Graph(int node, vector<vector<int>> &adj, vector<bool> &visited){

visited[node] = true;

cout << node << " ";

for(int neighbor : adj[node]){

if(!visited[neighbor])

DFS\_Graph(neighbor, adj, visited);

}

}

int main(){

int V = 5;

vector<vector<int>> adj(V);

adj[0] = {1,2};

adj[1] = {0,3};

adj[2] = {0,4};

adj[3] = {1};

adj[4] = {2};

vector<bool> visited(V, false);

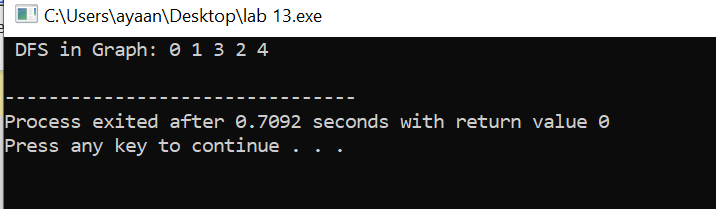
cout << " DFS in Graph: " ;

DFS\_Graph(0, adj, visited);

cout << endl;

return 0;

}



**Explanation:**

This code performs **Depth First Search** on a graph using recursion. Starting from node 0, it explores as far as possible along each branch before backtracking.  
The graph is represented using an adjacency list. A visited array keeps track of which nodes have already been visited to avoid infinite loops.

**3. Insert and Traverse for BFS in tree**

#include<iostream>

#include<queue>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node (int val){

data = val;

left = right = nullptr;

}

};

void bfs (Node\* root){

if (!root) return ;

queue<Node\*> q;

q.push(root);

while (!q.empty()){

Node\* cur = q.front ();

q.pop();

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

if(cur->left)q.push(cur->left);

if(cur->right)q.push(cur->right);

}

}

int main() {

Node\* root = new Node(1);

root->left = new Node(2);

root->right= new Node(3);

root->left->left = new Node(4);

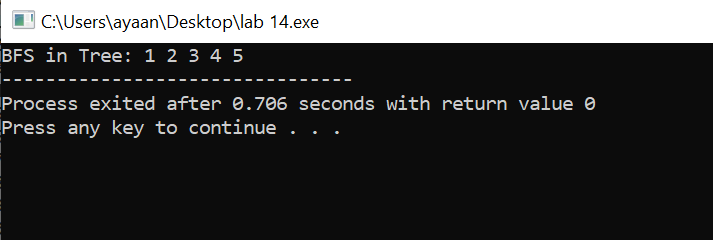
root->left->right = new Node(5);

cout << "BFS in Tree: ";

bfs(root);

return 0;

}



**Explanation:**

**BFS on Binary Tree**:  
This code performs **Breadth First Search** (level-order traversal) on a binary tree.It uses a queue to visit each level of the tree from left to right, one level at a time.  
Nodes are printed in the order they appear in each level.

**4.**

#include <iostream>

#include <vector>

#include <queue>

using namespace std;

void BFS\_Graph(int start, vector<vector<int>> &adj, vector<bool> &visited){

queue<int> q;

q.push(start);

visited[start] = true;

while(!q.empty()){

int node = q.front();

q.pop();

cout << node << " ";

for (int neighbor : adj[node]) {

if(!visited[neighbor]){

visited[neighbor] = true;

q.push(neighbor);

}

}

}

}

int main (){

int V = 5;

vector<vector<int>> adj(V);

adj[0] = {1, 2};

adj[1] = {1, 3};

adj[2] = {0, 4};

adj[3] = {1};

adj[4] = {2};

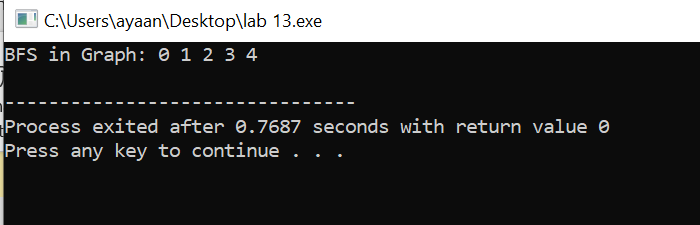
vector<bool> visited(V, false);

cout << "BFS in Graph: ";

BFS\_Graph(0, adj, visited);

cout << endl;

}



**Explanation:**

**BFS on Graph**:  
This performs **Breadth First Search** on a graph using a queue. Starting from a node, it visits all neighbors first before going deeper.  
The visited array ensures nodes are only visited once.