

**LAB MANUALS**

**Submitted by** : Qandeel asim

**Registration no:** 2023-bs-ai-037

**Submitted to:** Miss Irsha Qureshi

**Department:** BS Artificial Intelligence

**Introductory Lab**

**Data structure**:

A data structure is a way of organizing and storing data in a computer so that it can be used efficiently. Think of it like a container or box where you can keep things (like numbers, words, or other data) in a specific order or arrangement. The way you organize the data depends on how you want to access it or use it later.

**Types of data structure:**

**1. Linear Data Structures**

These structures organize data in a linear, sequential order**.**

* **Array**: A collection of elements, stored in contiguous memory locations, accessible by an index**.**
* **Linked List:** A series of nodes where each node contains data and a reference to the next node**.**
  + **Types:** Singly Linked List, Doubly Linked List, Circular Linked List.
* **Stack:** Follows the Last In, First Out (LIFO) principle, like stacking plates.
* **Queue:** Follows the First In, First Out (FIFO) principle, like a line of people.
  + **Types:** Simple Queue, Circular Queue, Priority Queue, Deque (Double-Ended Queue**).**

**2. Non-Linear Data Structures**

These structures organize data in a hierarchical or interconnected manner.

* **Tree:** A hierarchical structure where each node has a value and references to child nodes**.**
  + **Types:** Binary Tree, Binary Search Tree, AVL Tree, Heap, B-tree.
* **Graph:** Represents relationships or connections using nodes (vertices) and edges.
  + **Types**: Directed, Undirected, Weighted, Unweighted**.**

**LAB 01**

**Array**

Arrays are a collection of elements, all of the same type, stored in contiguous memory locations.

**Syntax:**

type arrayName[size];

**code 1**

#include<iostream>

using namespace std;

int main()

{ string groceryItems[5] = {"oil", "bread", "eggs", "vegetables", "fruits"};

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

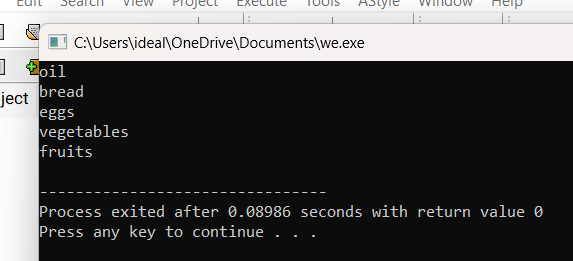
{ cout << groceryItems[index] << endl;

}

return 0;

}

**Output:**

:

**Code 2:**

#include<iostream>

using namespace std;

int main()

{

float arr[4]={56 , 54.5 , 85 , 60 };

for(int i=0 ; i<4; i++)

{

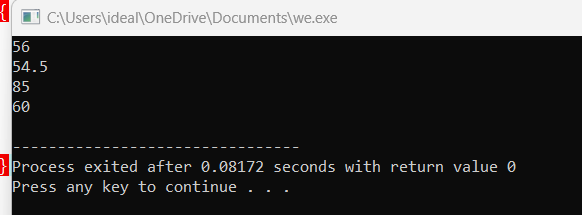
cout<<arr[i]<<endl;

}

return 0;

}

**Output:**



**Code 3:**

#include<iostream>

using namespace std;

int main()

{

float arr[5]={ 54.5 , 60 , 50.5 , 66 , 25.5 };

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

{

cout<<sizeof(arr[i])<<endl;

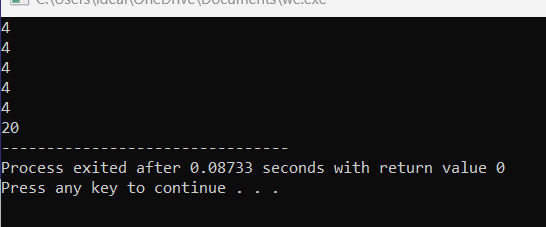
}

cout<<sizeof(arr);

return 0;

}

**Output:**

****

**Code 4:**

#include <iostream>

#include <string>

using namespace std;

int main() {

string fruits[3];

fruits[0] = "Apple";

fruits[1] = "Banana";

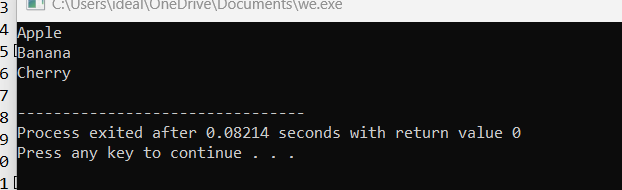
fruits[2] = "Cherry";

for(int i = 0; i < 3; i++) {

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

}

**Output:**



**Code 5:**

#include <iostream>

using namespace std;

int main()

{

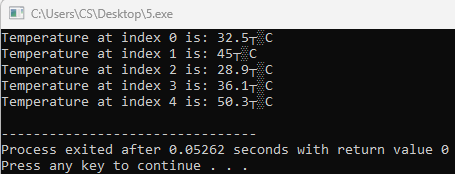
double temperatures[5] = {32.5, 45.0, 28.9, 36.1, 50.3};

for (int index = 0; index < 5; index++) {

cout << "Temperature at index " << index << " is: " << temperatures[index] << "°C" << endl;

}

return 0;}

**output: **

**LAB 02**

**Multi-Dimensional array**

A multi-dimensional array is an array that contains one or more arrays. It can be considered as an array of arrays, where each element itself is an array. The most common multi-dimensional arrays are 2D arrays (two-dimensional arrays), but they can have more than two dimensions (3D, 4D, etc.).

**Syntax:**

type array\_name[row\_size][column\_size];

**code 1**

#include <iostream>

using namespace std;

int main() {

int matrix[2][3] = {

{1, 2, 3},

{4, 5, 6}

};

cout << "2D Array Elements:\n";

for (int row = 0; row < 2; row++) {

for (int col = 0; col < 3; col++) {

cout << matrix[row][col] << " ";

}

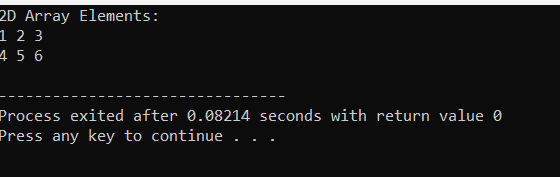
cout << endl;

}

return 0;

}

**Output :**

****

**Code 2:**

#include <iostream>

using namespace std;

int main() {

int matrix[2][2];

cout << "Enter 4 numbers for a 2x2 matrix:\n";

for (int row = 0; row < 2; row++) {

for (int col = 0; col < 2; col++) {

cout << "Element [" << row << "][" << col << "]: ";

cin >> matrix[row][col];

}

}

cout << "\nMatrix Elements:\n";

for (int row = 0; row < 2; row++) {

for (int col = 0; col < 2; col++) {

cout << matrix[row][col] << " ";

}

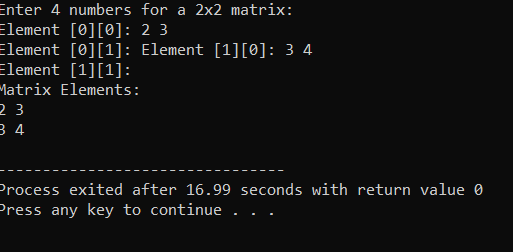
cout << endl;

}

return 0;

}

**Output :**

****

**Code 3:**

#include <iostream>

using namespace std;

int main() {

int matrix1[2][2] = {{1, 2}, {3, 4}};

int matrix2[2][2] = {{5, 6}, {7, 8}};

int sum[2][2];

for (int row = 0; row < 2; row++) {

for (int col = 0; col < 2; col++) {

sum[row][col] = matrix1[row][col] + matrix2[row][col];

}

}

cout << "Sum of the matrices:\n";

for (int row = 0; row < 2; row++) {

for (int col = 0; col < 2; col++) {

cout << sum[row][col] << " ";

}

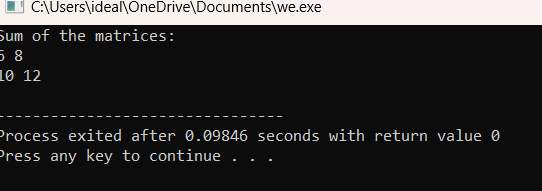
cout << endl;

}

return 0;

}

**Output:**

****

**Code 4:**

#include <iostream>

using namespace std;

int main() {

int matrix1[2][2] = {{1, 2}, {3, 4}};

int matrix2[2][2] = {{2, 0}, {1, 3}};

int product[2][2] = {{0, 0}, {0, 0}};

for (int row = 0; row < 2; row++) {

for (int col = 0; col < 2; col++) {

for (int k = 0; k < 2; k++) {

product[row][col] += matrix1[row][k] \* matrix2[k][col];

}

}

}

cout << "Product of the matrices:\n";

for (int row = 0; row < 2; row++) {

for (int col = 0; col < 2; col++) {

cout << product[row][col] << " ";

}

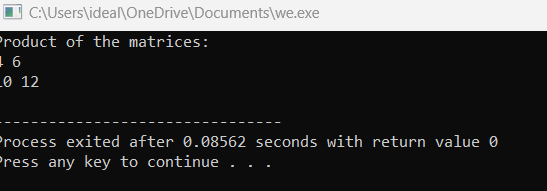
cout << endl;

}

return 0;

}

**Output;**

****

**Code 5:**

#include <iostream>

using namespace std;

int main() {

int cube[2][2][2] = {

{{1, 2}, {3, 4}},

{{5, 6}, {7, 8}}

};

cout << "3D Array Elements:\n";

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 2; j++) {

for (int k = 0; k < 2; k++) {

cout << "cube[" << i << "][" << j << "][" << k << "] = " << cube[i][j][k] << endl;

}

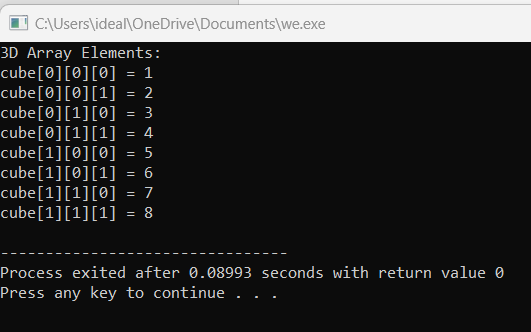
}

}

return 0;

}

**Output:**

****

**Lab 03**

**Vectors**

Vectors are part of the **Standard Template Library (STL)** and are a dynamic array-like structure that can grow and shrink in size as needed. Unlike regular arrays, vectors can dynamically resize, making them a flexible and efficient way to store collections of data.

**Syntax:**

#include <vector>

std::vector<type> vector\_name;

**code 1:**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> numbers = {10, 20, 30, 40, 50};

cout << "Vector elements are:\n";

for (int num : numbers) {

cout << num << " ";

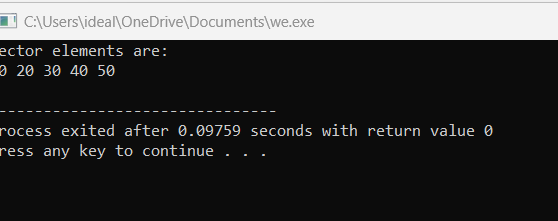
}

cout << endl;

return 0;

}

**Output:**

****

**Code 2:**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> numbers;

// Adding elements to the vector

numbers.push\_back(5);

numbers.push\_back(10);

numbers.push\_back(15);

cout << "Vector elements after adding values:\n";

for (int num : numbers) {

cout << num << " ";

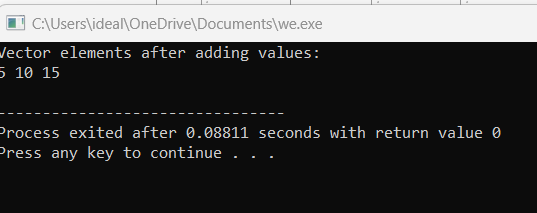
}

cout << endl;

return 0;

}

**Output :**



**Code 3:**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> numbers = {10, 20, 30, 40};

// Removing the last element

numbers.pop\_back();

cout << "Vector elements after removing the last value:\n";

for (int num : numbers) {

cout << num << " ";

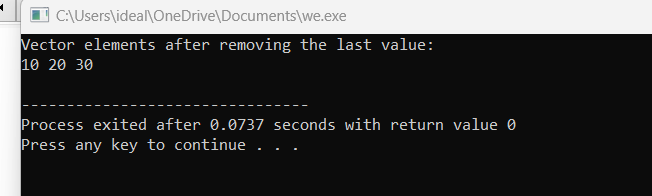
}

cout << endl;

return 0;

}

**Output:**



**Code 4:**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<string> fruits = {"Apple", "Banana", "Cherry"};

cout << "There are " << fruits.size() << " fruits in the vector:\n";

for (size\_t i = 0; i < fruits.size(); i++) {

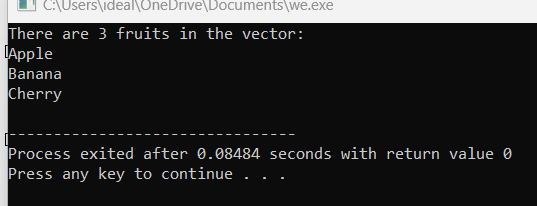
cout << fruits[i] << endl;

}

return 0;

}

**Output:**



**Code 5:**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> numbers = {1, 2, 4, 5};

// Insert 3 at the 3rd position (index 2)

numbers.insert(numbers.begin() + 2, 3);

// Erase the 2nd element (index 1)

numbers.erase(numbers.begin() + 1);

cout << "Vector after insertion and erasure:\n";

for (int num : numbers) {

cout << num << " ";

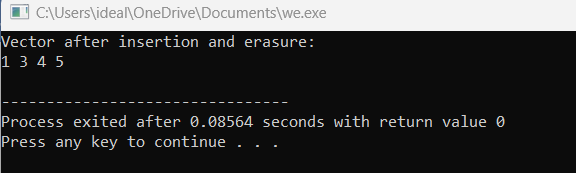
}

cout << endl;

return 0;

}

**Output:**

****

**LAB 04**

**List**

List is part of the **Standard Template Library (STL)**. It is a **doubly linked list** implementation that allows for efficient insertions and deletions from both ends of the list. Unlike vectors, lists do not allow random access to elements (i.e., you cannot directly access an element using an index), but they provide fast operations for adding and removing elements anywhere in the list.

**Syntax:**

#include <list>

std::list<type> list\_name;

**code 1:**

#include <iostream>

#include <list>

using namespace std;

int main() {

list<int> numbers = {10, 20, 30, 40, 50};

cout << "List elements are:\n";

for (int num : numbers) {

cout << num << " ";

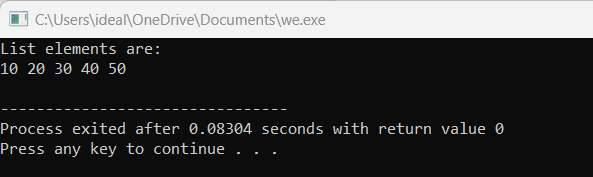
}

cout << endl;

return 0;

}

**Output:**

****

**Code 2:**

#include <iostream>

#include <list>

using namespace std;

int main() {

list<int> numbers;

// Adding elements to the list

numbers.push\_back(5); // Add at the end

numbers.push\_back(10);

numbers.push\_front(1); // Add at the front

cout << "List elements after adding values:\n";

for (int num : numbers) {

cout << num << " ";

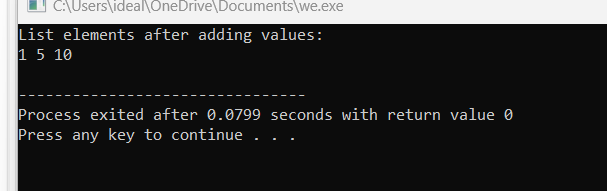
}

cout << endl;

return 0;

}

**Output:**



**Code 3:**

#include <iostream>

#include <list>

using namespace std;

int main() {

list<int> numbers = {10, 20, 30, 40};

// Removing the first and last elements

numbers.pop\_front();

numbers.pop\_back();

cout << "List elements after removing values:\n";

for (int num : numbers) {

cout << num << " ";

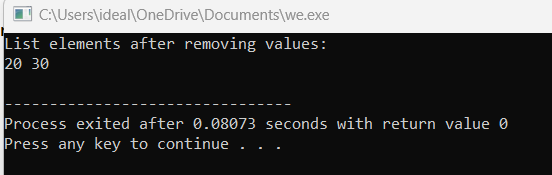
}

cout << endl;

return 0;

}

**Output:**



**Code 4:**

#include <iostream>

#include <list>

using namespace std;

int main() {

list<string> fruits = {"Apple", "Banana", "Cherry"};

cout << "List elements are:\n";

for (list<string>::iterator it = fruits.begin(); it != fruits.end(); ++it) {

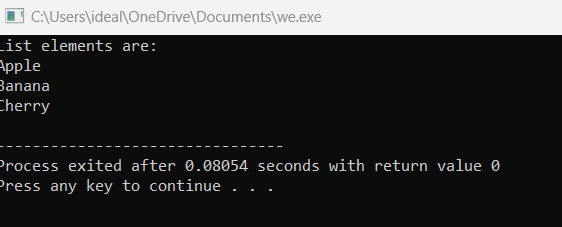
cout << \*it << endl;

}

return 0;

}

**Output:**



**Code 5:**

#include <iostream>

#include <list>

using namespace std;

int main() {

list<int> numbers = {1, 2, 4, 5};

// Insert 3 before the 3rd element

auto it = numbers.begin();

advance(it, 2); // Move iterator to the 3rd position

numbers.insert(it, 3);

// Erase the 2nd element

it = numbers.begin();

advance(it, 1); // Move iterator to the 2nd position

numbers.erase(it);

cout << "List after insertion and erasure:\n";

for (int num : numbers) {

cout << num << " ";

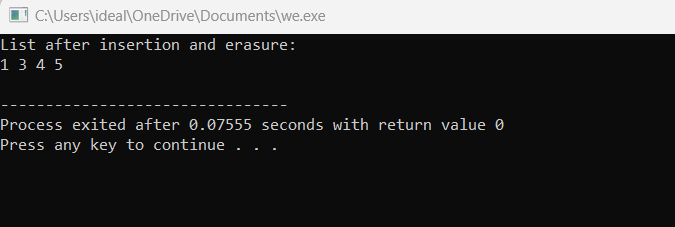
}

cout << endl;

return 0;

}

**Output**:



**Lab 05**

**Stack**

A stack in C++ is a linear data structure that follows the Last In, First Out (LIFO) principle. This means that the last element added to the stack is the first one to be removed. Stacks are useful for tasks such as expression evaluation, backtracking, and managing function calls.

**Syntax:**

#include <stack>

stack<type> st;

stack<int> integer\_stack;

stack<string> string\_stack;

**code 1:**

#include <iostream>

#include <stack>

using namespace std;

int main() {

stack<int> numbers;

// Pushing elements onto the stack

numbers.push(10);

numbers.push(20);

numbers.push(30);

cout << "Stack elements (top to bottom):\n";

while (!numbers.empty()) {

cout << numbers.top() << " "; // Accessing the top element

numbers.pop(); // Removing the top element

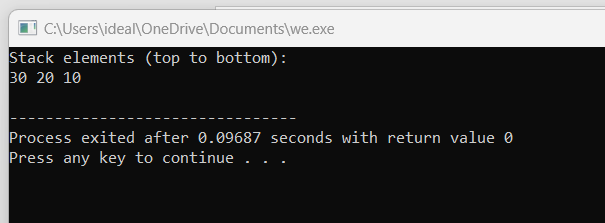
}

cout << endl;

return 0;

}

**Output :**

****

**Code 2:**

#include <iostream>

#include <stack>

using namespace std;

int main() {

stack<int> numbers;

// Checking if the stack is empty

if (numbers.empty()) {

cout << "The stack is empty.\n";

} else {

cout << "The stack is not empty.\n";

}

// Push an element and check again

numbers.push(100);

if (numbers.empty()) {

cout << "The stack is empty.\n";

} else {

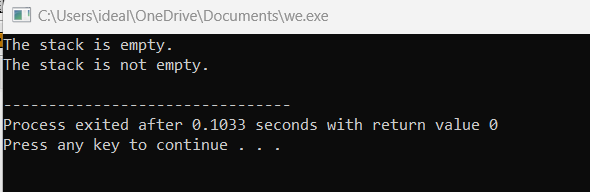
cout << "The stack is not empty.\n";

}

return 0;

}

**Output:**

****

**Code 3:**

#include <iostream>

#include <stack>

using namespace std;

int main() {

stack<int> numbers;

numbers.push(5);

numbers.push(15);

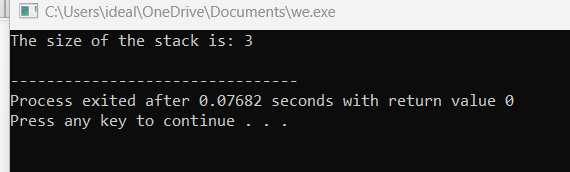
numbers.push(25);

cout << "The size of the stack is: " << numbers.size() << endl;

return 0;

}

**Output:**

****

**Code 4:**

#include <iostream>

#include <stack>

using namespace std;

int main() {

string input = "hello";

stack<char> charStack;

// Push characters onto the stack

for (char c : input) {

charStack.push(c);

}

cout << "Reversed string: ";

// Pop characters from the stack to reverse the string

while (!charStack.empty()) {

cout << charStack.top();

charStack.pop();

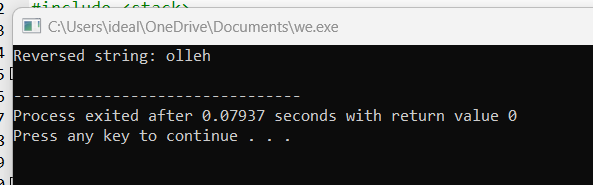
}

cout << endl;

return 0;

}

**Output:**

****

**Code 5:**

#include <iostream>

#include <stack>

using namespace std;

bool isBalanced(const string& expression) {

stack<char> brackets;

for (char c : expression) {

if (c == '(') {

brackets.push(c);

} else if (c == ')') {

if (brackets.empty()) {

return false;

}

brackets.pop();

}

}

return brackets.empty();

}

int main() {

string expression = "(a + b) \* (c - d)";

if (isBalanced(expression)) {

cout << "The expression has balanced parentheses.\n";

} else {

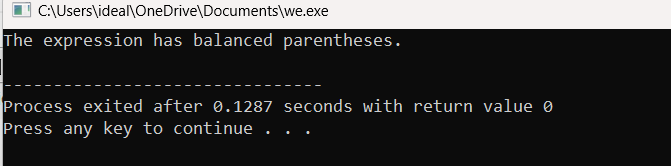
cout << "The expression does not have balanced parentheses.\n";

}

return 0;

}

**Output :**

****

**Lab 06**

**Queue**

A queue stores multiple elements in a specific order, called **FIFO**. **FIFO** stands for **First in, First Out**. To visualize FIFO, think of a queue as people standing in line in a supermarket. The first person to stand in line is also the first who can pay and leave the supermarket. This way of organizing elements is called FIFO in computer science and programming.

**Syntax:**

#include <queue>

queue<type> q;

queue<int> integer\_queue;

queue<string> string\_queue;

**code 1:**

**Code 1:**

**Pop front, back:**

#include <iostream>

#include <queue>

using namespace std;

int main() {

queue<int> q;

q.push(5);

q.push(10);

q.push(15);

cout << "Front: " << q.front() << ", Back: " << q.back() << endl;

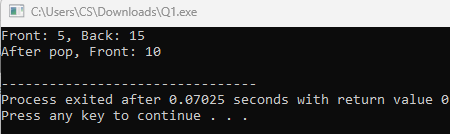
q.pop();

cout << "After pop, Front: " << q.front() << endl;

return 0;

}

**Output:**



**Code 2:**

**Queue size:**

#include <iostream>

#include <queue>

using namespace std;

int main() {

queue<int> q;

q.push(1); q.push(2); q.push(3);

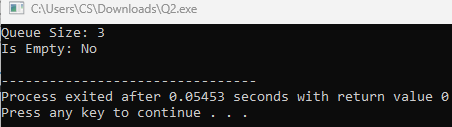
cout << "Queue Size: " << q.size() << endl;

cout << "Is Empty: " << (q.empty() ? "Yes" : "No") << endl;

return 0;

}

**Output:**



**Code 3:**

**Push elements:**

#include <iostream>

#include <queue>

using namespace std;

int main() {

queue<string> q;

q.push("qandeel");

q.push("abeer");

cout << "Front: " << q.front() << ", Back: " << q.back() << endl;

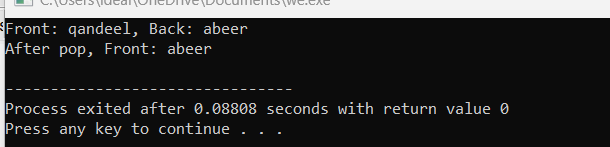
q.pop();

cout << "After pop, Front: " << q.front() << endl;

return 0;

}

**Output:**

****

**Code 4:**

**Empty queue:**

#include <iostream>

#include <queue>

using namespace std;

int main() {

queue<int> q;

for (int i = 1; i <= 5; ++i) q.push(i);

while (!q.empty()) {

cout << q.front() << " ";

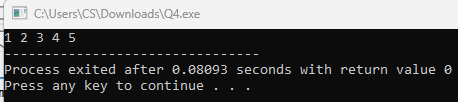
q.pop();

}

return 0;

}

**Output:**



**Code 5:**

#include <iostream>

#include <queue>

using namespace std;

int main() {

queue<int> q1, q2;

q1.push(1); q1.push(2);

q2.push(3); q2.push(4);

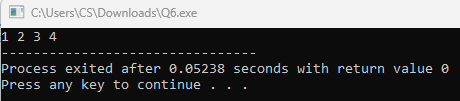
while (!q1.empty()) { cout << q1.front() << " "; q1.pop(); }

while (!q2.empty()) { cout << q2.front() << " "; q2.pop(); }

return 0;

}

**Output:**



**Lab 07**

**Single Link list**

The singly linked list is the simplest form of linked list in which the node contain two members data and a next pointer that stores the address of the next node. Each node is a singly linked list is connected through the next pointer and the next pointer of the last node points to NULL denoting the end of the linked list.

**Syntax:**

struct node

{

int data;

struct node \*next;

};

**Code**

**Linear search:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node(int data) {

val = data;

next = NULL;

}

};

void insert(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == NULL) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void search(Node\* head, int v) {

Node\* temp = head;

bool found = false;

while (temp != NULL) {

if (temp->val == v) {

cout << "Value " << v << " found at node: " << temp << endl;

found = true;

}

temp = temp->next;

}

if (!found) {

cout << "Value " << v << " not found in the list." << endl;

}

}

void displayList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val << " -> ";

temp = temp->next;

}

cout << "NULL\n";

}

int main() {

Node\* head = NULL;

insert(head, 5);

insert(head, 10);

insert(head, 15);

insert(head, 20);

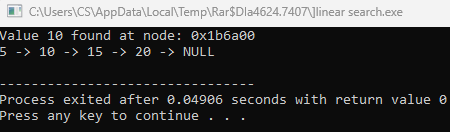
search(head, 10);

displayList(head);

return 0;

}

**Output:**



**Deletion**

**Code 1:**

**Delete tail:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node(int data) {

val = data;

next = NULL;

}

};

void insert(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == NULL) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void delTail(Node\* head){

Node\* secondlast=head;

while (secondlast->next->next != NULL){

secondlast=secondlast->next;

}

Node\* temp=secondlast->next;

secondlast->next = NULL;

delete temp;

}

void displayList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val << " -> ";

temp = temp->next;

}

cout << "NULL\n";

}

int main() {

Node\* head = NULL;

insert(head, 5);

insert(head, 10);

insert(head, 15);

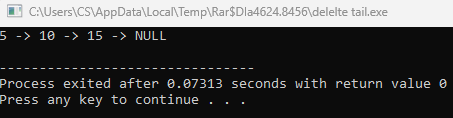
insert(head, 20);

delTail(head);

displayList(head);

}

**Output:**

****

**Code 2:**

**Delete at position:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node(int data) {

val = data;

next = NULL;

}

};

void insert(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == NULL) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void delatP(Node\* &head, int pos){

Node\* prev=head;

int currentpos=0;

while (currentpos != pos-1){

prev = prev->next;

currentpos++;

}

Node\* temp = prev->next;

prev->next= prev->next->next;

delete temp;

}

void displayList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val << " -> ";

temp = temp->next;

}

cout << "NULL\n";

}

int main() {

Node\* head = NULL;

insert(head, 5);

insert(head, 10);

insert(head, 15);

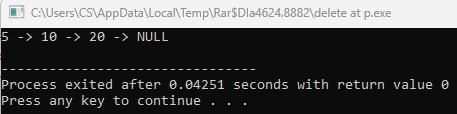
insert(head, 20);

delatP(head,2);

displayList(head);

}

**Output:**

****

**Code 3:**

**Delete at start:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node(int data) {

val = data;

next = NULL;

}

};

void insert(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == NULL) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void delatstart(Node\*& head) {

if (head == NULL) return;

Node\* temp = head;

head = head->next;

delete temp;

}

void displayList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val << " -> ";

temp = temp->next;

}

cout << "NULL\n";

}

int main() {

Node\* head = NULL;

insert(head, 5);

insert(head, 10);

insert(head, 15);

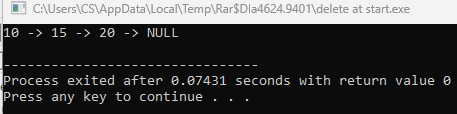
insert(head, 20);

delatstart(head);

displayList(head);

}

**Output:**



**Code 4:**

**Insert at position:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node(int data) {

val = data;

next = NULL;

}

};

void insert(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == NULL) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void insertatstart(Node\* &head, int data){

Node\* newnode = new Node(data);

newnode->next = head;

head = newnode;

}

void insertatP(Node\* &head, int val, int pos){

if(pos==0){

insertatstart(head,val);

return;

}

Node\* newnode = new Node(val);

Node\* temp = head;

int currentpos=0;

while (currentpos!=pos-1){

temp=temp->next;

currentpos++;

}

newnode->next=temp->next;

temp->next = newnode;

}

void displayList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val << " -> ";

temp = temp->next;

}

cout << "NULL\n";

}

int main() {

Node\* head = NULL;

insert(head, 5);

insert(head, 10);

insert(head, 15);

insert(head, 20);

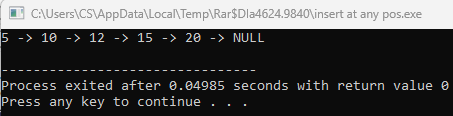
insertatP(head,12,2);

displayList(head);

return 0;

}

**Output:**



**Insertion**

**Code 1:**

**Insert at start:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node(int data) {

val = data;

next = NULL;

}

};

void insertatstart(Node\* &head, int data){

Node\* newnode = new Node(data);

newnode->next = head;

head = newnode;

}

void display(Node\* head){

Node\* temp = head;

while (temp!=NULL){

cout << temp->val << "->";

temp=temp->next;

}

cout << "NULL";

}

int main (){

Node\* head = NULL;

insertatstart(head,4);

display(head);

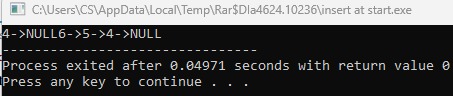
insertatstart(head,5);

insertatstart(head,6);

display(head);

}

**Output:**



**Code 2:**

**Insert at tail:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node(int data) {

val = data;

next = NULL;

}

};

void insert(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == NULL) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void insertAtTail(Node\* &head,int val){

Node\* newnode=new Node(val);

Node\* temp=head;

while(temp->next!=NULL){

temp = temp->next;

}

temp->next=newnode;

}

void displayList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val << " -> ";

temp = temp->next;

}

cout << "NULL\n";

}

int main() {

Node\* head = NULL;

insert(head, 5);

insert(head, 10);

insert(head, 15);

insert(head, 20);

displayList(head);

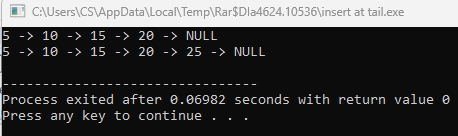
insertAtTail(head,25);

displayList(head);

return 0;

}

**Output:**



**Code 3:**

**Only insert:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node(int data) {

val = data;

next = NULL;

}

};

void insert(Node\*& head, int data) {

Node\* newNode = new Node(data);

if (head == NULL) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != NULL) {

temp = temp->next;

}

temp->next = newNode;

}

void displayList(Node\* head) {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val << " -> ";

temp = temp->next;

}

cout << "NULL\n";

}

int main() {

Node\* head = NULL;

insert(head, 5);

insert(head, 10);

insert(head, 15);

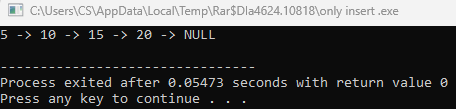
insert(head, 20);

displayList(head);

return 0;

}

**Output:**



**Lab 8**

**Circular link list**

A **Circular Linked List** is a variation of a linked list in which all nodes are connected to form a circle.

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

**Insertion**

**Code 1:**

**Inert at beginning:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

void insertBegin(Node\*\* head, int data) {

Node\* newNode = new Node();

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

newNode->next = newNode;

\*head = newNode;

} else {

Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

\*head = newNode;

}

}

void display(Node\* head) {

if (head == NULL) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

int main() {

Node\* head = NULL;

insertBegin(&head, 45);

insertBegin(&head, 80);

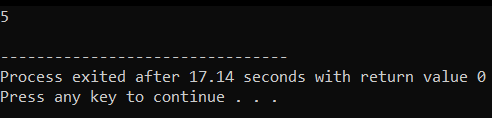
insertBegin(&head, 15);

display(head);

return 0;

}

**Output:**



**Code 2:**

**Insert at beginning:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

void insertMid(Node\*\* head, int data) {

Node\* newNode = new Node();

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

newNode->next = newNode;

\*head = newNode;

} else {

Node\* slow = \*head;

Node\* fast = \*head;

while (fast->next != \*head && fast->next->next != \*head) {

slow = slow->next;

fast = fast->next->next;

}

newNode->next = slow->next;

slow->next = newNode;

}

}

void display(Node\* head) {

if (head == NULL) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;}

int main() {

Node\* head = NULL;

insertMid(&head, 10);

insertMid(&head, 20);

insertMid(&head, 30);

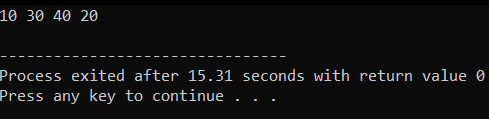
insertMid(&head, 40);

display(head);

return 0;

}

**Output:**



**Code 3:**

**Insert at end:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

void insertEnd(Node\*\* head, int data) {

Node\* newNode = new Node();

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

newNode->next = newNode;

\*head = newNode;

} else {

Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

}

void display(Node\* head) {

if (head == NULL) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

int main() {

Node\* head = NULL;

insertEnd(&head, 45);

insertEnd(&head, 80);

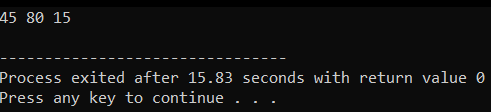
insertEnd(&head, 15);

display(head);

return 0;

}

**Output:**



**Deletion**

**Code 1:**

**Delete at start:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

void deleteStart(Node\*\* head) {

if (\*head == NULL) {

cout << "List is empty." << endl;

return;

}

Node\* temp = \*head;

if ((\*head)->next == \*head) {

delete \*head;

\*head = NULL;

return;

}

Node\* last = \*head;

while (last->next != \*head) {

last = last->next;

}

Node\* newHead = (\*head)->next;

last->next = newHead;

delete \*head;

\*head = newHead;

}

void insertEnd(Node\*\* head, int data) {

Node\* newNode = new Node();

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

newNode->next = newNode;

\*head = newNode;

} else {

Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

}

void display(Node\* head) {

if (head == NULL) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

int main() {

Node\* head = NULL;

insertEnd(&head, 45);

insertEnd(&head, 80);

insertEnd(&head, 15);

cout << "Original List: ";

display(head);

deleteStart(&head);

cout << "After Deletion at Start: ";

display(head);

deleteStart(&head);

cout << "After Deleting Again: ";

display(head);

deleteStart(&head);

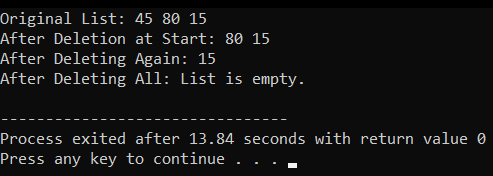
cout << "After Deleting All: ";

display(head);

return 0;

}

**Output:**



**Code 2:**

**Delete at mid:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

void deleteMid(Node\*\* head) {

if (\*head == NULL) {

cout << "List is empty." << endl;

return;

}

if ((\*head)->next == \*head) {

delete \*head;

\*head = NULL;

return;

}

Node\* slow = \*head;

Node\* fast = \*head;

Node\* prev = NULL;

while (fast != \*head && fast->next != \*head) {

prev = slow;

slow = slow->next;

fast = fast->next->next;

}

if (prev != NULL) {

prev->next = slow->next;

if (slow == \*head) {

\*head = slow->next;

}

delete slow;

}

}

void insertEnd(Node\*\* head, int data) {

Node\* newNode = new Node();

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

newNode->next = newNode;

\*head = newNode;

} else {

Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

}

void display(Node\* head) {

if (head == NULL) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

int main() {

Node\* head = NULL;

insertEnd(&head, 45);

insertEnd(&head, 80);

insertEnd(&head, 15);

insertEnd(&head, 60);

insertEnd(&head, 25);

cout << "Original List: ";

display(head);

deleteMid(&head);

cout << "After Deletion at Mid: ";

display(head);

deleteMid(&head);

cout << "After Deleting Again: ";

display(head);

deleteMid(&head);

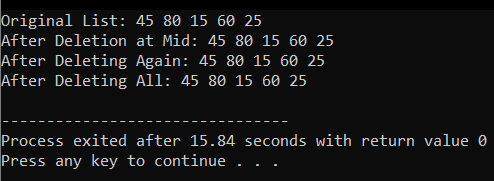
cout << "After Deleting All: ";

display(head);

return 0;

}

**Output:**



**Code 3:**

**Delete at end:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

void deleteEnd(Node\*\* head) {

if (\*head == NULL) {

cout << "List is empty." << endl;

return;

}

if ((\*head)->next == \*head) {

delete \*head;

\*head = NULL;

return;

}

Node\* temp = \*head;

Node\* prev = NULL;

while (temp->next != \*head) {

prev = temp;

temp = temp->next;

}

prev->next = \*head;

delete temp;

}

void insertEnd(Node\*\* head, int data) {

Node\* newNode = new Node();

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

newNode->next = newNode;

\*head = newNode;

} else {

Node\* temp = \*head;

while (temp->next != \*head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

}

void display(Node\* head) {

if (head == NULL) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

int main() {

Node\* head = NULL;

insertEnd(&head, 45);

insertEnd(&head, 80);

insertEnd(&head, 15);

insertEnd(&head, 60);

insertEnd(&head, 25);

cout << "Original List: ";

display(head);

deleteEnd(&head);

cout << "After Deletion at End: ";

display(head);

deleteEnd(&head);

cout << "After Deleting Again: ";

display(head);

deleteEnd(&head);

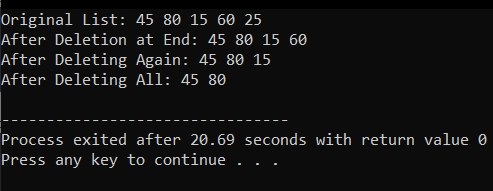
cout << "After Deleting All: ";

display(head);

return 0;

}

**Output:**



**Lab 9**

**Doubly link list**

A doubly linked list is a type of linked list where each node contains three parts:

* Data – Stores the value.
* Next Pointer – Points to the next node.
* Previous Pointer – Points to the previous node.

**Insertion**

**Code 1:**

**Insert at start:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node\* prev;

Node(int data)

{

val=data;

next=NULL;

prev=NULL;

}

};

class DOUBLELINKLIST{

public:

Node\* head;

Node\* tail;

DOUBLELINKLIST(){

head = NULL;

tail = NULL;

}

void insertHead(int val){

Node\* new\_node = new Node(val);

if (head==NULL){

head=new\_node;

tail= new\_node;

return;

}

new\_node-> next = head;

head->prev=new\_node;

head=new\_node;

return;

}

void display(){

Node\* temp=head;

while (temp!=NULL){

cout << temp->val << "<->"<<endl;

temp=temp->next;

}

cout << endl;

}

};

int main ()

{

DOUBLELINKLIST dll;

dll.insertHead(3);

dll.display();

dll.insertHead(2);

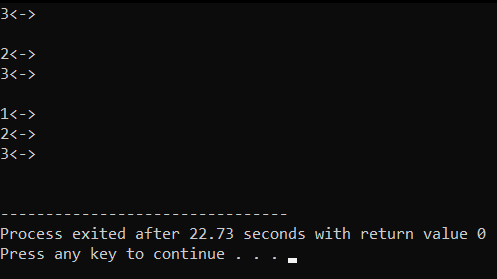
dll.display();

dll.insertHead(1);

dll.display();

}

**Output:**



**Code 2:**

**Insert at any position:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node\* prev;

Node(int data){

val=data;

next=NULL;

prev=NULL;

}

};

class DOUBLELINKLIST{

public:

Node\* head;

Node\* tail;

DOUBLELINKLIST(){

head = NULL;

tail = NULL;

}

void insertend(int val){

Node\* new\_node = new Node(val);

if (tail==NULL){

head=new\_node;

tail= new\_node;

return;

}

new\_node-> prev = tail;

tail->next=new\_node;

tail=new\_node;

return;

}

void display(){

Node\* temp=head;

while (temp!=NULL){

cout << temp->val << "<->"<<endl;

temp=temp->next;

}

cout << endl;

}

void insertatP(int val, int k){

int count=0;

Node\* temp = head;

while (count <(k-1)){

temp = temp->next;

count++;

}

Node\* new\_node = new Node(val);

new\_node->next = temp->next;

temp->next=new\_node;

new\_node->prev=temp;

new\_node->next->prev=new\_node;

return;

}

};

int main (){

DOUBLELINKLIST dll;

dll.insertend(4);

dll.display();

dll.insertend(5);

dll.display();

dll.insertend(6);

dll.display();

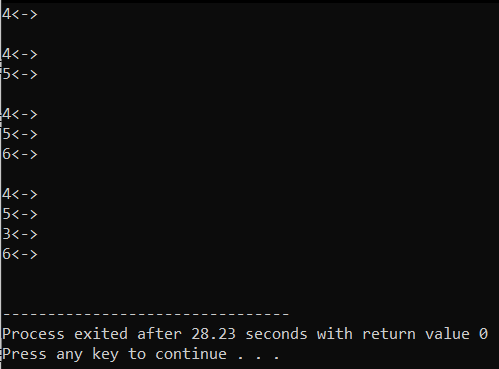
dll.insertatP(3,2);

dll.display();

return 0;

}

**Output:**



**Deletion**

**Code 1:**

**Delete at start:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node\* prev;

Node(int data) {

val = data;

next = NULL;

prev = NULL;

}

};

class DOUBLELINKLIST {

public:

Node\* head;

Node\* tail;

DOUBLELINKLIST() {

head = NULL;

tail = NULL;

}

void insert(int val) {

Node\* new\_node = new Node(val);

if (head == NULL) {

head = new\_node;

tail = new\_node;

} else {

tail->next = new\_node;

new\_node->prev = tail;

tail = new\_node;

}

}

void deleteAThead() {

if (head == NULL) {

return;

}

Node\* temp = head;

head = head->next;

if (head == NULL) {

tail = NULL;

} else {

head->prev = NULL;

}

delete temp;

}

void display() {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val;

if (temp->next != NULL) {

cout << " <-> ";

}

temp = temp->next;

}

cout << endl;

}

};

int main() {

DOUBLELINKLIST dll;

dll.insert(3);

dll.insert(2);

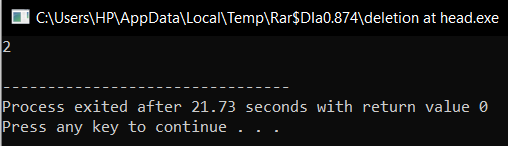
dll.deleteAThead();

dll.display();

return 0;

}

**Output:**



**Code:**

**Delete at any position:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node\* prev;

Node(int data) {

val = data;

next = NULL;

prev = NULL;

}

};

class DOUBLELINKLIST {

public:

Node\* head;

Node\* tail;

DOUBLELINKLIST() {

head = NULL;

tail = NULL;

}

void insert(int val) {

Node\* new\_node = new Node(val);

if (head == NULL) {

head = new\_node;

tail = new\_node;

} else {

tail->next = new\_node;

new\_node->prev = tail;

tail = new\_node;

}

}

void del(int p) {

if (head == NULL) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

int count = 1;

while (temp != NULL && count < p) {

temp = temp->next;

count++;

}

if (temp == NULL) {

cout << "Position out of bounds." << endl;

return;

}

if (temp->prev != NULL)

temp->prev->next = temp->next;

if (temp->next != NULL)

temp->next->prev = temp->prev;

if (temp == head)

head = temp->next;

if (temp == tail)

tail = temp->prev;

delete temp;

}

void display() {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val;

if (temp->next != NULL) {

cout << " <-> ";

}

temp = temp->next;

}

cout << endl;

}

};

int main() {

DOUBLELINKLIST dll;

dll.insert(3);

dll.insert(2);

dll.insert(1);

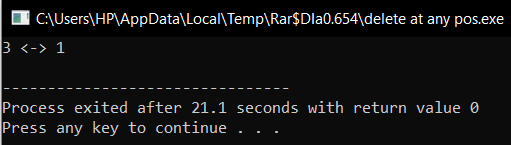
dll.del(2);

dll.display();

return 0;

}

**Output:**



**Code:**

**Delete at end:**

#include <iostream>

using namespace std;

class Node {

public:

int val;

Node\* next;

Node\* prev;

Node(int data) {

val = data;

next = NULL;

prev = NULL;

}

};

class DOUBLELINKLIST {

public:

Node\* head;

Node\* tail;

DOUBLELINKLIST() {

head = NULL;

tail = NULL;

}

void insert(int val) {

Node\* new\_node = new Node(val);

if (head == NULL) {

head = new\_node;

tail = new\_node;

} else {

tail->next = new\_node;

new\_node->prev = tail;

tail = new\_node;

}

}

void del(){

if (head==NULL){

return;

}

Node\* temp = tail;

tail = tail->prev;

if(head==NULL){

tail=NULL;

else{

tail->next = NULL;

}

delete temp;

}

void display() {

Node\* temp = head;

while (temp != NULL) {

cout << temp->val;

if (temp->next != NULL) {

cout << " <-> ";

}

temp = temp->next;

}

cout << endl;

}

};

int main() {

DOUBLELINKLIST dll;

dll.insert(3);

dll.insert(2);

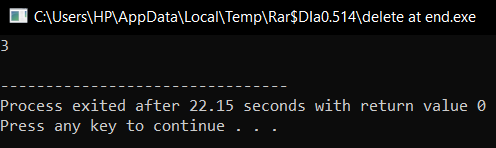
dll.del();

dll.display();

return 0;

}

**Output:**



**Binary search tree:**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = nullptr;

right = nullptr;

}

};

// Insert function

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

if (root == nullptr) {

return new Node(key);

}

if (key < root->data) {

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

} else if (key > root->data) {

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

}

return root;

}

// Search function

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

if (root == nullptr || root->data == key) {

return root;

}

if (key < root->data) {

return search(root->left, key);

} else {

return search(root->right, key);

}

}

// Find the minimum value node

Node\* findMin(Node\* root) {

while (root && root->left != nullptr) {

root = root->left;

}

return root;

}

// Delete function

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

if (root == nullptr) {

return root;

}

if (key < root->data) {

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

} else if (key > root->data) {

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

} else {

if (root->left == nullptr) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == nullptr) {

Node\* temp = root->left;

delete root;

return temp;

}

Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

// In-order traversal

void inOrder(Node\* root) {

if (root != nullptr) {

inOrder(root->left);

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

inOrder(root->right);

}

}

// Pre-order traversal

void preOrder(Node\* root) {

if (root != nullptr) {

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

preOrder(root->left);

preOrder(root->right);

}

}

// Post-order traversal

void postOrder(Node\* root) {

if (root != nullptr) {

postOrder(root->left);

postOrder(root->right);

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

}

}

// Main function to test the BST

int main() {

Node\* root = nullptr;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

cout << "In-order traversal: ";

inOrder(root);

cout << endl;

cout << "Pre-order traversal: ";

preOrder(root);

cout << endl;

cout << "Post-order traversal: ";

postOrder(root);

cout << endl;

cout << "\nDeleting 20\n";

root = deleteNode(root, 20);

cout << "In-order traversal: ";

inOrder(root);

cout << endl;

cout << "\nDeleting 30\n";

root = deleteNode(root, 30);

cout << "In-order traversal: ";

inOrder(root);

cout << endl;

cout << "\nDeleting 50\n";

root = deleteNode(root, 50);

cout << "In-order traversal: ";

inOrder(root);

cout << endl;

Node\* found = search(root, 60);

if (found != nullptr) {

cout << "\nFound: " << found->data << endl;

} else {

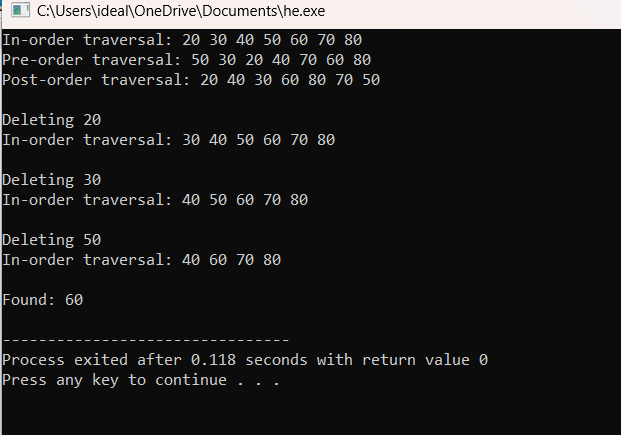
cout << "\nNot Found" << endl;

}

return 0;

}

**Output:**

****