

The
University of
Faisalabad

Data Structure **Lab Manual**

<u>Name:</u>	Wishaq Akbar
<u>Reg#:</u>	2023-BS-AI-041
<u>Section:</u>	A
<u>Program:</u>	Artificial Intelligence
<u>Submitted to:</u>	Mrs. Irsha Qureshi
<u>Course Code:</u>	CS-126
<u>Date Submitted:</u>	13-01-2025

Table of Contents

Lab 01:	Introduction to DSA	3
Lab 02:	Array	4
Lab 03:	2D Array	7
Lab 04:	Vector	10
Lab 05:	List	13
Lab 06:	Stack	16
Lab 07:	Queue	20
Lab 08:	Dequeue	24
Lab 09:	Trees	25
Lab 10:	Binary Search Tree	31
Lab 11:	Singly Link List	33
Lab 12:	Doubly Link List	36
Lab 13:	Circular Link List	39

What is a Data Structure?

A data structure is a specialized format for organizing, managing, and storing data in a way that enables efficient access and modification. It is a foundational concept in computer science, enabling algorithms to work efficiently.

Importance of Data Structures

- Efficient data storage and retrieval
- Better algorithm design
- Effective use of memory
- Enhanced performance in software applications

Types of Data Structures

- **Primitive Data Structures:**
Examples: Integer, Float, Character, Boolean
- **Non-Primitive Data Structures:**
 - **Linear Data Structures:** Arrays, Linked Lists, Stacks, Queues
 - **Non-Linear Data Structures:** Trees, Graphs

Applications of Data Structures

- **Arrays:** Used for static data storage like matrices, tables.
- **Linked Lists:** Dynamic memory usage, manipulation of data sequences.
- **Stacks and Queues:** Used in parsing expressions, task scheduling.
- **Trees:** Hierarchical data storage such as file systems.
- **Graphs:** Representing networks like social connections or transport systems.

An array is a **collection of elements**, each identified by at least one array **index or key**. Arrays are used to store multiple values of the **same type** in a contiguous block of memory.

Program 1: Array Modifying

```
#include <iostream>
using namespace std;
int main() {
    int arr[4] = {5, 10, 15, 20};
    arr[2] = 100;
    cout << "Modified array: ";
    for (int i = 0; i < 4; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output

```
Modified array: 5 10 100 20
-----
Process exited after 0.1653 seconds with return value 0
Press any key to continue . . . |
```

Program 2: Max Element

```
#include <iostream>
using namespace std;
int main() {
    int arr[6] = {10, 20, 30, 40, 5, 25};
    int max = arr[0];
    for (int i = 1; i < 6; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    cout << "Maximum element in the array: " << max << endl;
    return 0;
}
```

Output

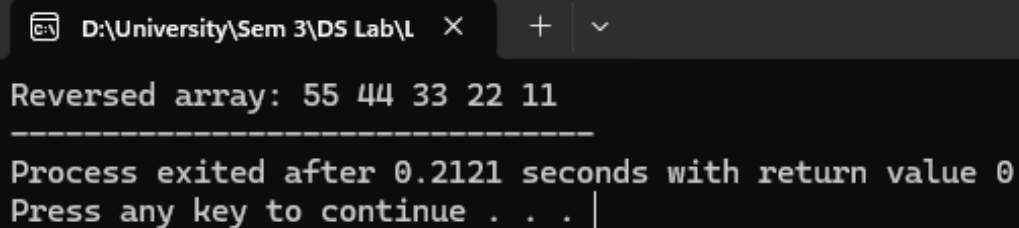
```
Maximum element in the array: 40
-----
Process exited after 0.1785 seconds with return value 0
Press any key to continue . . . |
```

Program 3: Reversed Array

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

int main() {
    int arr[5] = {11, 22, 33, 44, 55};
    cout << "Reversed array: ";
    for (int i = 4; i >= 0; i--) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output



```
Reversed array: 55 44 33 22 11
-----
Process exited after 0.2121 seconds with return value 0
Press any key to continue . . . |
```

Program 4: Copying an Array

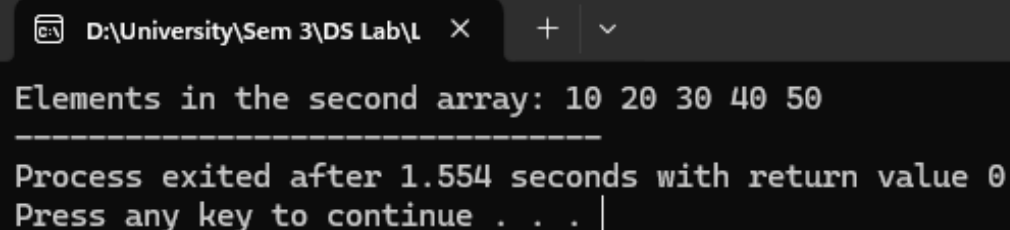
```
#include <iostream>
using namespace std;

int main() {
    int arr1[5] = {10, 20, 30, 40, 50};
    int arr2[5];

    for (int i = 0; i < 5; i++) {
        arr2[i] = arr1[i];
    }

    cout << "Elements in the second array: ";
    for (int i = 0; i < 5; i++) {
        cout << arr2[i] << " ";
    }
    return 0;
}
```

Output



```
Elements in the second array: 10 20 30 40 50
-----
Process exited after 1.554 seconds with return value 0
Press any key to continue . . . |
```

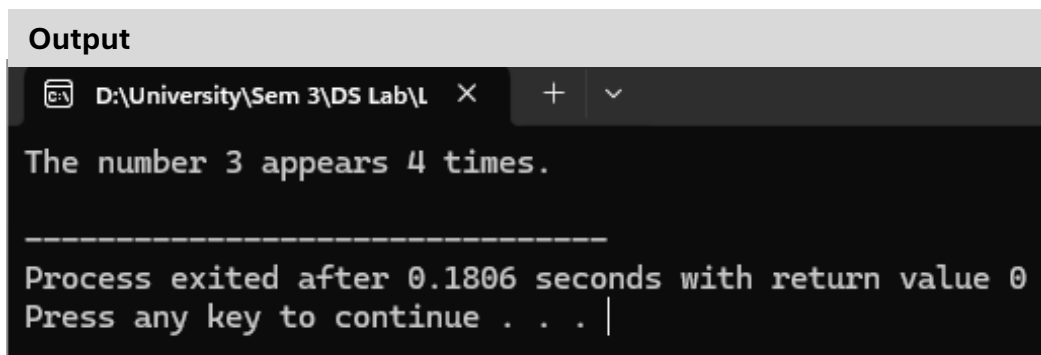
Program 5: Number Occurrence in an Array

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

int main() {
    int arr[8] = {5, 3, 7, 3, 9, 3, 6, 3};
    int target = 3;
    int count = 0;

    for (int i = 0; i < 8; i++) {
        if (arr[i] == target) {
            count++;
        }
    }

    cout << "The number " << target << " appears " << count << "
times." << endl;
    return 0;
}
```

The screenshot shows a dark-themed IDE window titled "Output". The output text is: "The number 3 appears 4 times." followed by a horizontal line and "Process exited after 0.1806 seconds with return value 0". At the bottom, it says "Press any key to continue . . . |".

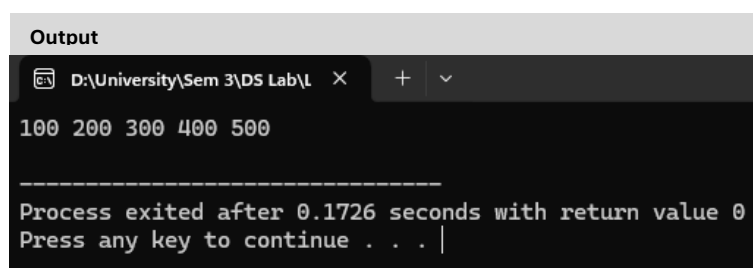
```
Output
D:\University\Sem 3\DS Lab\L  X  +  v
The number 3 appears 4 times.
-----
Process exited after 0.1806 seconds with return value 0
Press any key to continue . . . |
```

Program 6: Print Array Function

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

void printArray(int arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arr[5] = {100, 200, 300, 400, 500};
    printArray(arr, 5);
    return 0;
}
```

The screenshot shows a dark-themed IDE window titled "Output". The output text is: "100 200 300 400 500" followed by a horizontal line and "Process exited after 0.1726 seconds with return value 0". At the bottom, it says "Press any key to continue . . . |".

```
Output
D:\University\Sem 3\DS Lab\L  X  +  v
100 200 300 400 500
-----
Process exited after 0.1726 seconds with return value 0
Press any key to continue . . . |
```

A 2D array is an array of arrays where data is stored in a **matrix format**, i.e., rows and columns.

Program 1: 2D array Matrix

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

int main() {
    int matrix[3][3] = {
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    };

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}
```

Output

```
1 2 3
4 5 6
7 8 9
```

```
-----
Process exited after 0.07298 seconds with return value 0
Press any key to continue . . . |
```

Program 2: Transpose a 2D matrix

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

int main() {
    int matrix[2][2] = {
        {1, 2},
        {3, 4}
    };

    cout << "Transpose of the matrix: \n";
    for (int i = 0; i < 2; i++) {
```

```

        for (int j = 0; j < 2; j++) {
            cout << matrix[j][i] << " ";
        }
        cout << endl;
    }
    return 0;
}

```

Output

Transpose of the matrix:

1 3

2 4

 Process exited after 0.05844 seconds with return value 0
 Press any key to continue . . . |

Program 3: Add two matrices

```

#include <iostream>
using namespace std;

int main() {
    int matrix1[2][2] = {
        {1, 2},
        {3, 4}
    };
    int matrix2[2][2] = {
        {5, 6},
        {7, 8}
    };
    int result[2][2];

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            result[i][j] = matrix1[i][j] + matrix2[i][j];
        }
    }

    cout << "Sum of the matrices: \n";
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            cout << result[i][j] << " ";
        }
        cout << endl;
    }

    return 0;
}

```


Output

```
Sum of the matrices:
6 8
10 12

-----
Process exited after 0.0703 seconds with return value 0
Press any key to continue . . . |
```

Program 4: Find the Largest Element in a 2D Array

```
#include <iostream>
using namespace std;
int main() {
    int arr[3][3], maxElement;
    cout << "Enter 3x3 matrix elements: \n";
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cin >> arr[i][j];
        }
    }
    maxElement = arr[0][0];
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (arr[i][j] > maxElement) {
                maxElement = arr[i][j];
            }
        }
    }
    cout << "Largest element in the matrix: " << maxElement << endl;
    return 0;
}
```

Output

```
Enter 3x3 matrix elements:
2 4 6
5 1 2
5 8 3
Largest element in the matrix: 8

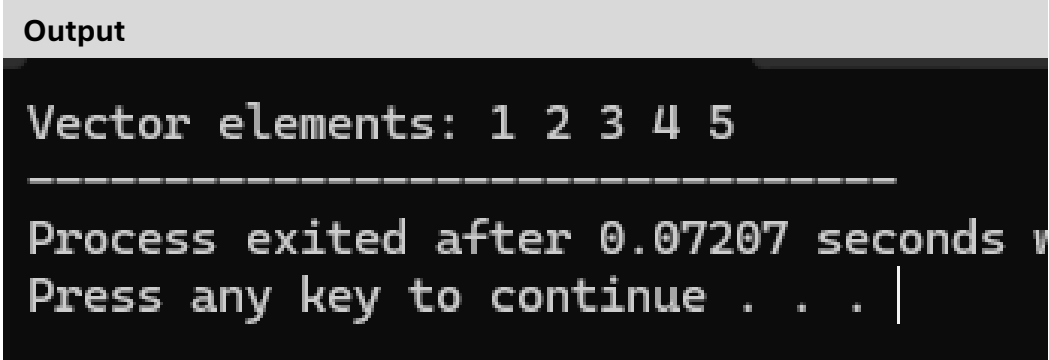
-----
Process exited after 13.52 seconds with return value 0
Press any key to continue . . . |
```

A **vector** in C++ is a dynamic array provided by the Standard Template Library (STL). Unlike static arrays, vectors can resize automatically when elements are added or removed. They offer functionalities like insertion, deletion, and traversal.

Program 1: Initialize and Print a Vector

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
    cout << "Vector elements: ";
    for (int i = 0; i < vec.size(); i++) {
        cout << vec[i] << " ";
    }
    return 0;
}
```

Output



```
Vector elements: 1 2 3 4 5
-----
Process exited after 0.07207 seconds
Press any key to continue . . . |
```

Program 2: Add Elements to a Vector

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec;
    vec.push_back(10);
    vec.push_back(20);
    vec.push_back(30);
    cout << "Vector after adding elements: ";
    for (int val : vec) {
        cout << val << " ";
    }
    return 0;
}
```

Output

```
Vector after adding elements: 10 20 30
-----
Process exited after 0.07885 seconds with return value 0
Press any key to continue . . . |
```

Program 3: Remove an Element from a Vector

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec = {10, 20, 30, 40};
    vec.pop_back();
    cout << "Vector after removing the last element: ";
    for (int val : vec) {
        cout << val << " ";
    }
    return 0;
}
```

Output

```
C:\Users\Wishaq Akbar\Down X + v
Vector after removing the last element: 10 20 30
-----
Process exited after 0.07906 seconds with return value 0
Press any key to continue . . . |
```

Program 4: Find the Size of a Vector

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec = {1, 2, 3, 4, 5};
    cout << "The size of the vector is: " << vec.size() << endl;
    return 0;
}
```

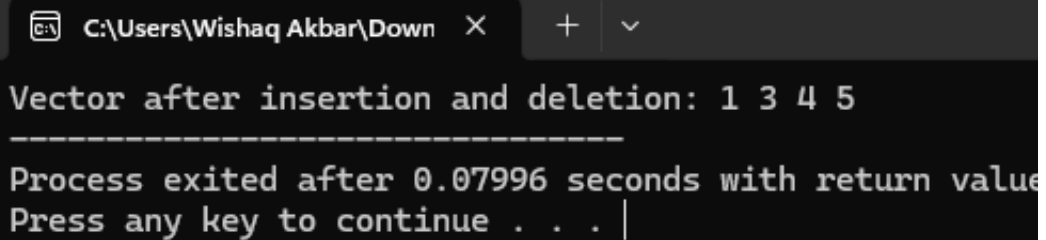
Output

```
C:\Users\Wishaq Akbar\Down X + v
Vector after removing the last element: 10 20 30
-----
Process exited after 0.07906 seconds with return value 0
Press any key to continue . . . |
```

Program 5: Insert and Erase Elements

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> vec = {1, 2, 3, 5};
    vec.insert(vec.begin() + 3, 4);
    vec.erase(vec.begin() + 1);
    cout << "Vector after insertion and deletion: ";
    for (int val : vec) {
        cout << val << " ";
    }
    return 0;
}
```

Output



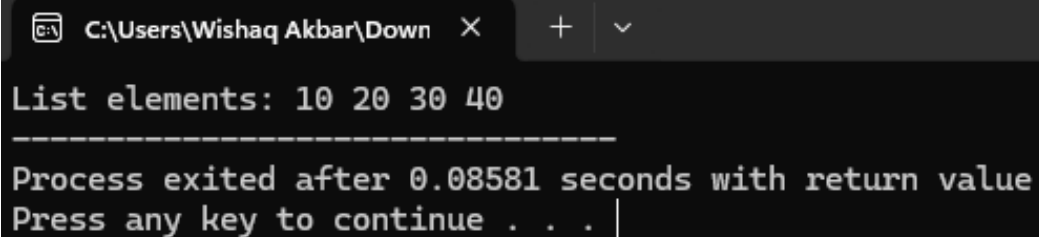
```
C:\Users\Wishaq Akbar\Down X + v
Vector after insertion and deletion: 1 3 4 5
-----
Process exited after 0.07996 seconds with return value
Press any key to continue . . . |
```

A **list** in C++ is provided by the Standard Template Library (STL). Unlike arrays or vectors, lists allow efficient insertion and deletion of elements at both ends and in the middle.

Program 1: Initialize List

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> myList = {10, 20, 30, 40};
    cout << "List elements: ";
    for (int val : myList) {
        cout << val << " ";
    }
    return 0;
}
```

Output



```
List elements: 10 20 30 40
-----
Process exited after 0.08581 seconds with return value
Press any key to continue . . . |
```

Program 2: Merge Two Lists

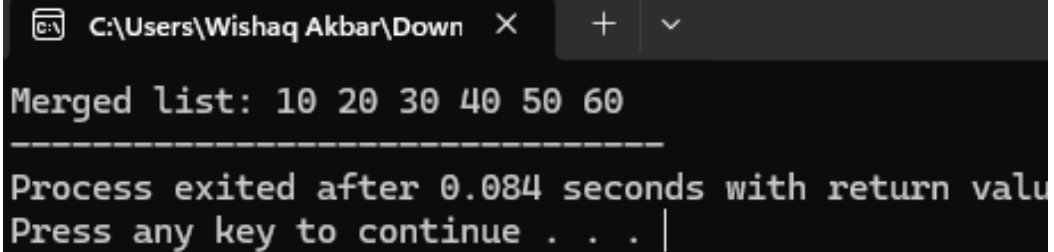
```
#include <iostream>
#include <list>
using namespace std;

int main() {
    list<int> list1 = {10, 20, 30};
    list<int> list2 = {40, 50, 60};

    list1.merge(list2);

    cout << "Merged list: ";
    for (int val : list1) {
        cout << val << " ";
    }
    return 0;
}
```

Output

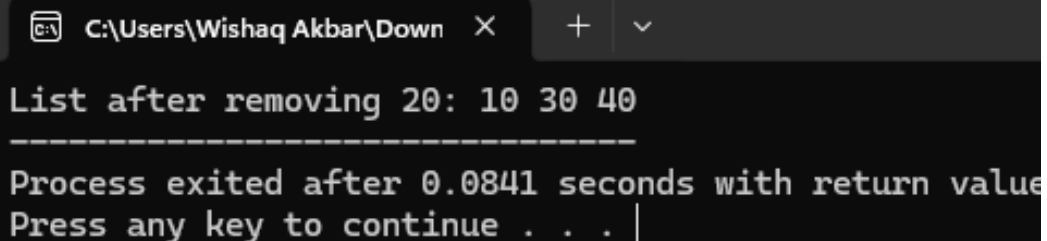


```
C:\Users\Wishaq Akbar\Down X + v
Merged list: 10 20 30 40 50 60
-----
Process exited after 0.084 seconds with return value
Press any key to continue . . . |
```

Program 3: Remove Specific Elements

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> myList = {10, 20, 30, 20, 40, 20};
    myList.remove(20);
    cout << "List after removing 20: ";
    for (int val : myList) {
        cout << val << " ";
    }
    return 0;
}
```

Output

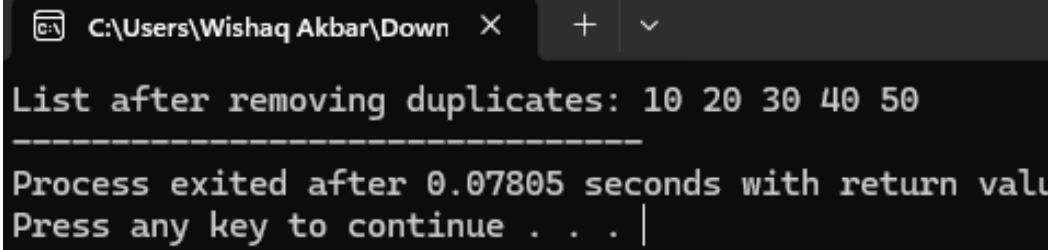


```
C:\Users\Wishaq Akbar\Down X + v
List after removing 20: 10 30 40
-----
Process exited after 0.0841 seconds with return value
Press any key to continue . . . |
```

Program 4: Remove Duplicates

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> myList = {10, 20, 20, 30, 40, 40, 50};
    myList.unique(); // Remove consecutive duplicates
    cout << "List after removing duplicates: ";
    for (int val : myList) {
        cout << val << " ";
    }
    return 0;
}
```

Output



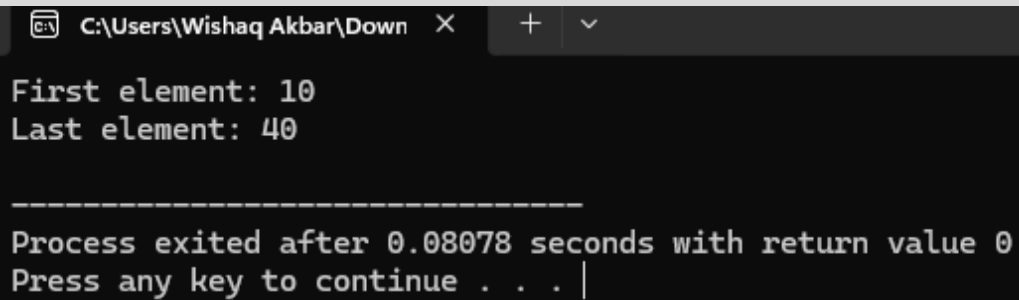
```
C:\Users\Wishaq Akbar\Down X + v

List after removing duplicates: 10 20 30 40 50
-----
Process exited after 0.07805 seconds with return value
Press any key to continue . . . |
```

Program 5: Access Front and Back Elements

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    list<int> myList = {10, 20, 30, 40};
    cout << "First element: " << myList.front() << endl;
    cout << "Last element: " << myList.back() << endl;
    return 0;
}
```

Output



```
C:\Users\Wishaq Akbar\Down X + v

First element: 10
Last element: 40
-----
Process exited after 0.08078 seconds with return value 0
Press any key to continue . . . |
```

A **stack** is a linear data structure that follows the **LIFO (Last In, First Out)** principle, meaning the last element added to the stack is the first one to be removed. Common operations in a stack are:

- **Push:** Add an element to the top of the stack.
- **Pop:** Remove an element from the top of the stack.
- **Top/Peak:** View the top element without removing it.

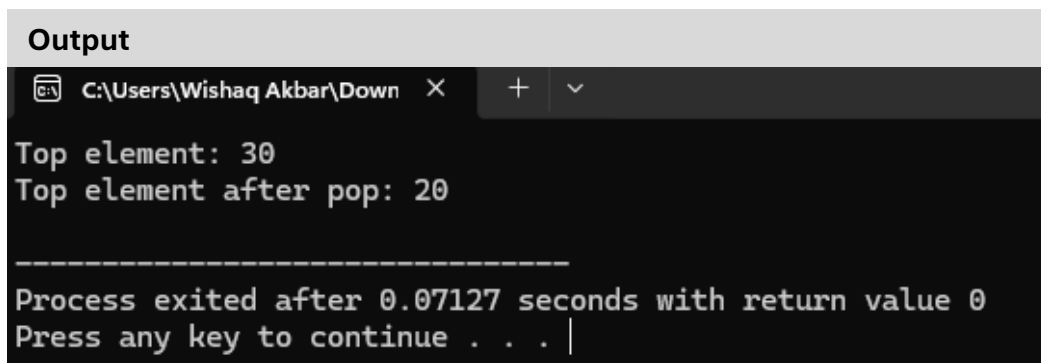
Program 1: Basic Stack Operations

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> myStack;

    // Push elements onto the stack
    myStack.push(10);
    myStack.push(20);
    myStack.push(30);
    cout << "Top element: " << myStack.top() << endl;
    myStack.pop();
    cout << "Top element after pop: " << myStack.top() << endl;

    return 0;
}
```



```
Output
C:\Users\Wishaq Akbar\Down X + v

Top element: 30
Top element after pop: 20

-----
Process exited after 0.07127 seconds with return value 0
Press any key to continue . . . |
```

Program 2: Check if Stack is Empty

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> myStack;

    if (myStack.empty()) {
        cout << "Stack is empty." << endl;
    }
}
```



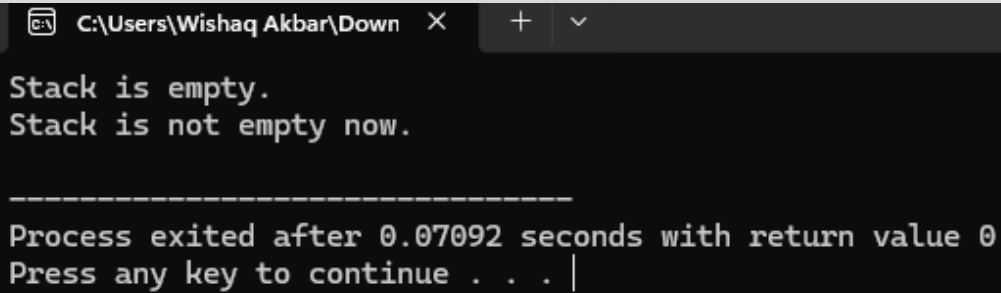
```

    } else {
        cout << "Stack is not empty." << endl;
    }
    myStack.push(50);
    if (!myStack.empty()) {
        cout << "Stack is not empty now." << endl;
    }

    return 0;
}

```

Output



```

Stack is empty.
Stack is not empty now.

-----

Process exited after 0.07092 seconds with return value 0
Press any key to continue . . . |

```

Program 3: Stack Size

```

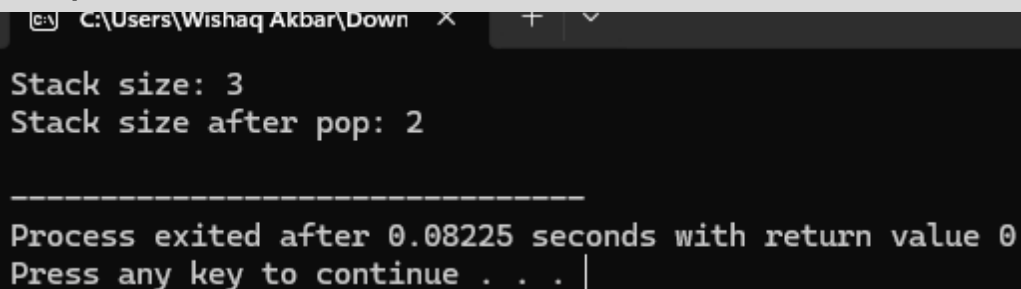
#include <iostream>
#include <stack>
using namespace std;

int main() {
    stack<int> myStack;
    myStack.push(1);
    myStack.push(2);
    myStack.push(3);
    cout << "Stack size: " << myStack.size() << endl;
    myStack.pop();
    cout << "Stack size after pop: " << myStack.size() << endl;

    return 0;
}

```

Output



```

Stack size: 3
Stack size after pop: 2

-----

Process exited after 0.08225 seconds with return value 0
Press any key to continue . . . |

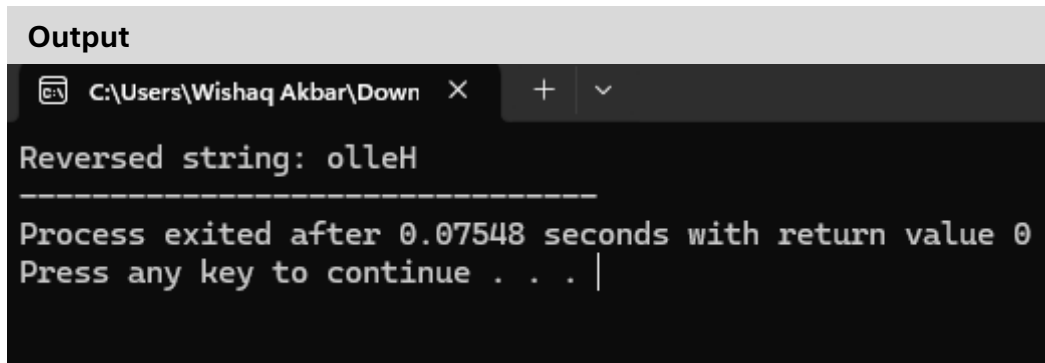
```

Program 4: Reverse a String Using a Stack

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    string str = "Hello";
    stack<char> charStack;
    for (char ch : str) {
        charStack.push(ch);
    }
    cout << "Reversed string: ";
    while (!charStack.empty()) {
        cout << charStack.top();
        charStack.pop();
    }

    return 0;
}
```



```
Output
C:\Users\Wishaq Akbar\Down X + v
Reversed string: olleH
-----
Process exited after 0.07548 seconds with return value 0
Press any key to continue . . . |
```

Program 5: Decimal to Binary Using a Stack

```
#include <iostream>
#include <stack>
using namespace std;

int main() {
    int num = 25;
    stack<int> binaryStack;
    while (num > 0) {
        binaryStack.push(num % 2);
        num /= 2;
    }
    cout << "Binary representation: ";
    while (!binaryStack.empty()) {
        cout << binaryStack.top();
        binaryStack.pop();
    }

    return 0;
}
```

Output



C:\Users\Wishaq Akbar\Down



Binary representation: 11001

Process exited after 0.07858 seconds with return value 0

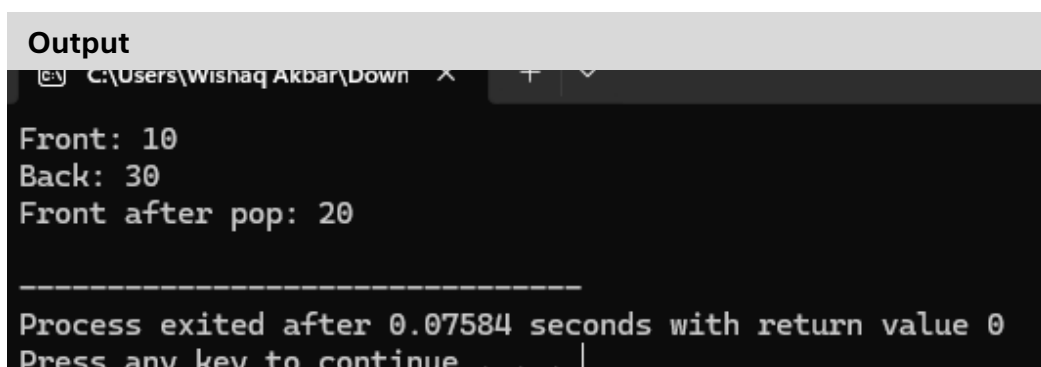
Press any key to continue . . . |

A **queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle, meaning the first element added is the first one removed.

Program 1: Basic Queue Operations

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    queue<int> myQueue;
    myQueue.push(10);
    myQueue.push(20);
    myQueue.push(30);
    cout << "Front: " << myQueue.front() << endl;
    cout << "Back: " << myQueue.back() << endl;
    myQueue.pop();
    cout << "Front after pop: " << myQueue.front() << endl;
    return 0;
}
```



```
Output
C:\Users\Wishaq Akbar\Down x + v
Front: 10
Back: 30
Front after pop: 20

-----
Process exited after 0.07584 seconds with return value 0
Press any key to continue . . .
```

Program 2: Reverse a Queue Using a Stack

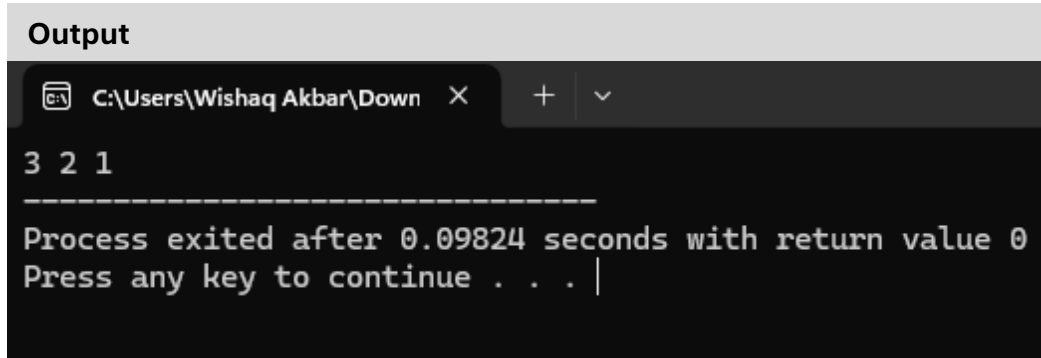
```
#include <iostream>
#include <queue>
#include <stack>
using namespace std;

int main() {
    queue<int> myQueue;
    stack<int> myStack;
    myQueue.push(1);
    myQueue.push(2);
    myQueue.push(3);
    while (!myQueue.empty()) {
        myStack.push(myQueue.front());
        myQueue.pop();
    }
```

```

    while (!myStack.empty()) {
        myQueue.push(myStack.top());
        myStack.pop();
    }
    while (!myQueue.empty()) {
        cout << myQueue.front() << " ";
        myQueue.pop();
    }
    return 0;
}

```



```

Output
C:\Users\Wishaq Akbar\Down
3 2 1
-----
Process exited after 0.09824 seconds with return value 0
Press any key to continue . . .

```

Program 3: Reverse a Queue Using a Stack

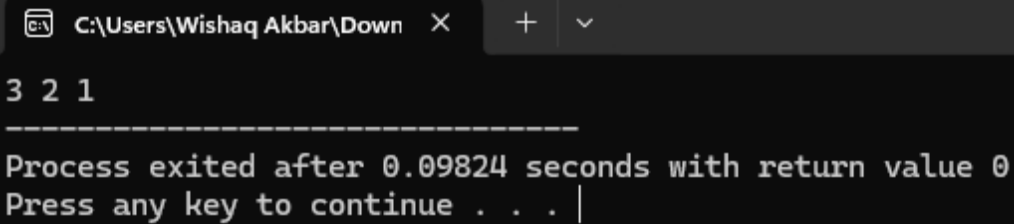
```

#include <iostream>
#include <queue>
#include <stack>
using namespace std;

int main() {
    queue<int> myQueue;
    stack<int> myStack;
    myQueue.push(1);
    myQueue.push(2);
    myQueue.push(3);
    while (!myQueue.empty()) {
        myStack.push(myQueue.front());
        myQueue.pop();
    }
    while (!myStack.empty()) {
        myQueue.push(myStack.top());
        myStack.pop();
    }
    while (!myQueue.empty()) {
        cout << myQueue.front() << " ";
        myQueue.pop();
    }
    return 0;
}

```

Output



```
C:\Users\Wishaq Akbar\Down X + v
3 2 1
-----
Process exited after 0.09824 seconds with return value 0
Press any key to continue . . . |
```

Program 4: Implement a Circular Queue

```
#include <iostream>
using namespace std;
class CircularQueue {
    int *arr, front, rear, size, capacity;
public:
    CircularQueue(int cap) {
        capacity = cap;
        arr = new int[capacity];
        size = 0;
        front = rear = -1;
    }
    bool isFull() { return size == capacity; }
    bool isEmpty() { return size == 0; }
    void enqueue(int val) {
        if (isFull()) return;
        if (rear == -1) front = rear = 0;
        else rear = (rear + 1) % capacity;
        arr[rear] = val;
        size++;
    }
    void dequeue() {
        if (isEmpty()) return;
        front = (front + 1) % capacity;
        size--;
        if (size == 0) front = rear = -1;
    }
    int getFront() { return isEmpty() ? -1 : arr[front]; }
};

int main() {
    CircularQueue cq(3);
    cq.enqueue(10);
    cq.enqueue(20);
    cq.enqueue(30);
    cout << "Front: " << cq.getFront() << endl;
    cq.dequeue();
    cout << "Front after dequeue: " << cq.getFront() << endl;
    return 0;
}
```

Output

```
C:\Users\Wishag Akbar\Down X + v
Front: 10
Front after dequeue: 20

-----
Process exited after 0.06208 seconds with return value 0
Press any key to continue . . . |
```

Program 5: Priority Queue Implementation

```
#include <iostream>
#include <queue>
using namespace std;

int main() {
    priority_queue<int> pq;
    pq.push(15);
    pq.push(5);
    pq.push(20);
    pq.push(10);
    cout << "Priority Queue elements in descending order: ";
    while (!pq.empty()) {
        cout << pq.top() << " ";
        pq.pop();
    }
    return 0;
}
```

Output

```
C:\Users\Wishag Akbar\Down X + v
Priority Queue elements in descending order: 20 15 10 5
-----
Process exited after 0.06126 seconds with return value 0
Press any key to continue . . . |
```

A **Dequeue** (Double-Ended Queue) is a data structure where elements can be added or removed from both ends. It can be implemented using arrays or linked lists and supports operations like insertion, deletion, and traversal from both ends.

Program 1: Dequeue Implementation

```
#include <iostream>
#include <deque>
using namespace std;

int main() {
    deque<int> dq;

    dq.push_back(10);
    dq.push_front(20);
    dq.push_back(30);
    dq.push_front(40);

    cout << "Deque elements: ";
    for (int x : dq) cout << x << " ";
    cout << endl;

    dq.pop_front();
    dq.pop_back();

    cout << "After removing front and back: ";
    for (int x : dq) cout << x << " ";
    cout << endl;

    cout << "Front element: " << dq.front() << endl;
    cout << "Back element: " << dq.back() << endl;

    return 0;
}
```

Output

```
Deque elements: 40 20 10 30
After removing front and back: 20 10
Front element: 20
Back element: 10

-----
Process exited after 0.2354 seconds with return value 0
```


A **Tree** is a hierarchical data structure consisting of nodes, where each node has a value and references to its child nodes. The topmost node is called the **root**, and nodes without children are called **leaves**. Common tree types include Binary Trees, Binary Search Trees, and AVL Trees.

Program 1: Binary Tree Traversal

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

struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

void preorder(Node* root) {
    if (root == NULL) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

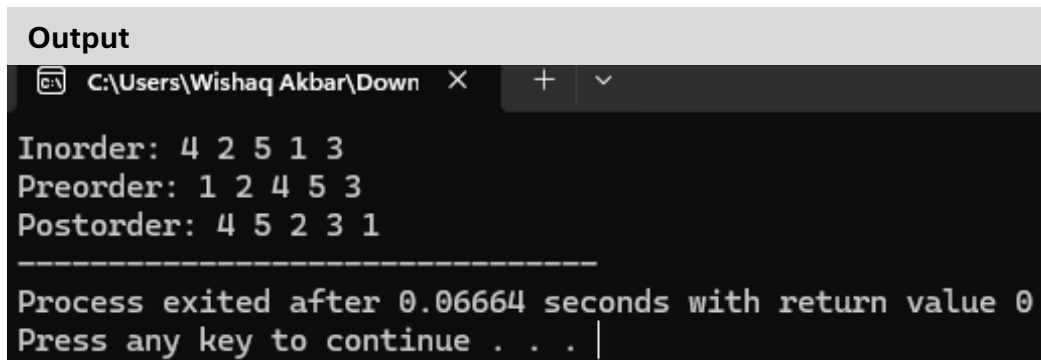
int main() {
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
```

```

    root->left->right = createNode(5);

    cout << "Inorder: ";
    inorder(root);
    cout << "\nPreorder: ";
    preorder(root);
    cout << "\nPostorder: ";
    postorder(root);
    return 0;
}

```



```

Output
C:\Users\Wishaq Akbar\Down X + v
Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Postorder: 4 5 2 3 1
-----
Process exited after 0.06664 seconds with return value 0
Press any key to continue . . . |

```

Program 2: Insertion and Search

```

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

Node* insert(Node* root, int data) {
    if (root == NULL) return createNode(data);
    if (data < root->data) root->left = insert(root->left, data);
    else if (data > root->data) root->right = insert(root->right,
data);
    return root;
}

bool search(Node* root, int key) {
    if (root == NULL) return false;
    if (root->data == key) return true;
    if (key < root->data) return search(root->left, key);
}

```

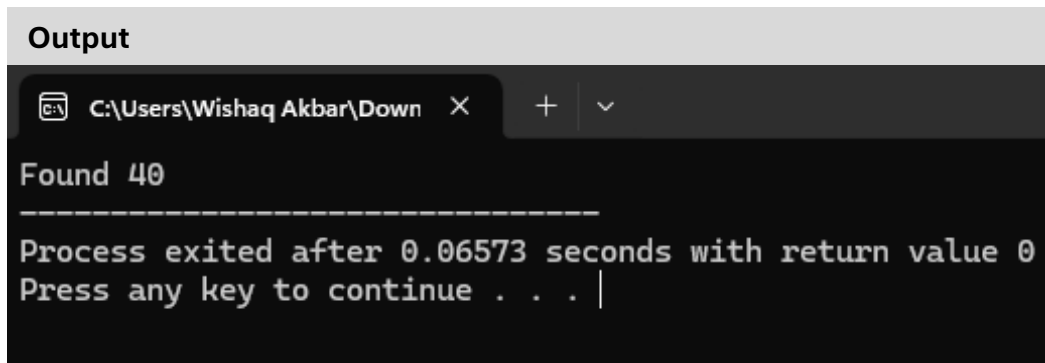
```

        return search(root->right, key);
    }

int main() {
    Node* root = NULL;
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);

    int key = 40;
    if (search(root, key)) cout << "Found " << key;
    else cout << key << " not found";
    return 0;
}

```



```

Output
C:\Users\Wishaq Akbar\Down  X  +  v
Found 40
-----
Process exited after 0.06573 seconds with return value 0
Press any key to continue . . . |

```

Program 3: Level Order Traversal

```

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

struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

void levelOrder(Node* root) {
    if (root == NULL) return;
    queue<Node*> q;
    q.push(root);
}

```

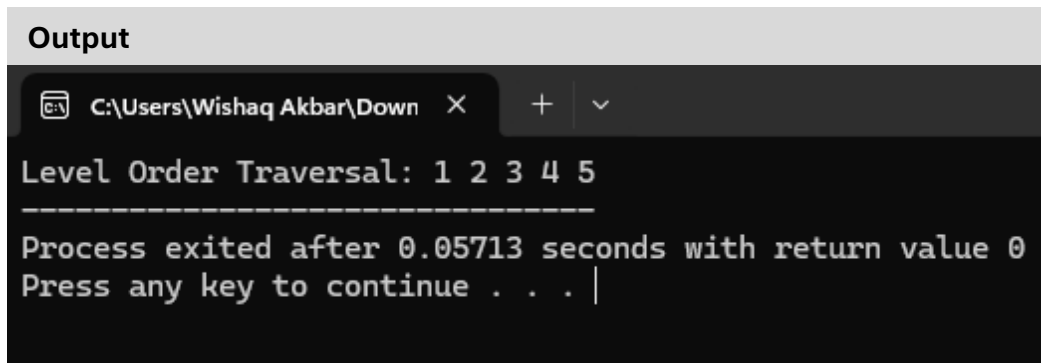
```

while (!q.empty()) {
    Node* current = q.front();
    q.pop();
    cout << current->data << " ";
    if (current->left != NULL) q.push(current->left);
    if (current->right != NULL) q.push(current->right);
}
}

int main() {
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    cout << "Level Order Traversal: ";
    levelOrder(root);
    return 0;
}

```



```

Output
C:\Users\Wishaq Akbar\Down >
Level Order Traversal: 1 2 3 4 5
-----
Process exited after 0.05713 seconds with return value 0
Press any key to continue . . . |

```

Program 4: Height of a Tree

```

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

int findHeight(Node* root) {
    if (root == NULL) return 0;

```

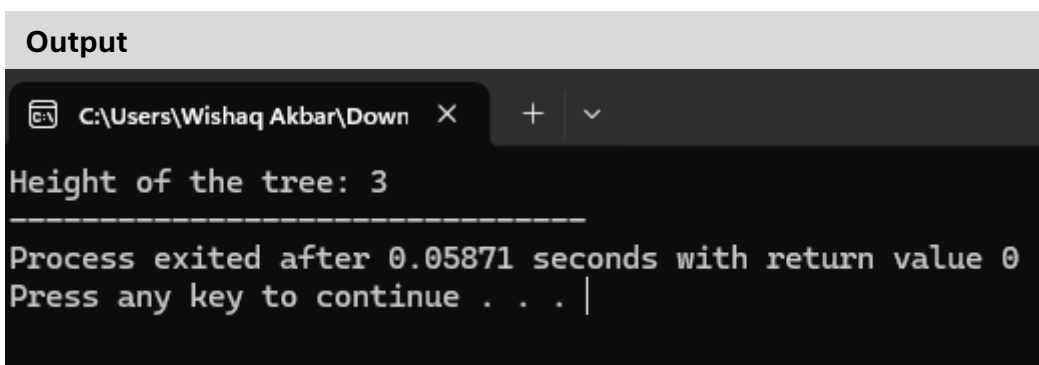
```

    int leftHeight = findHeight(root->left);
    int rightHeight = findHeight(root->right);
    return max(leftHeight, rightHeight) + 1;
}

int main() {
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
    root->left->right = createNode(5);

    cout << "Height of the tree: " << findHeight(root);
    return 0;
}

```



```

Output
C:\Users\Wishaq Akbar\Down X + v
Height of the tree: 3
-----
Process exited after 0.05871 seconds with return value 0
Press any key to continue . . . |

```

Program 5: Count Nodes in a Tree

```

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

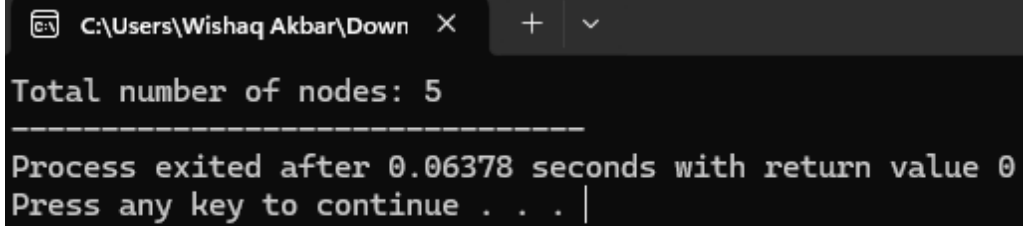
int countNodes(Node* root) {
    if (root == NULL) return 0;
    return 1 + countNodes(root->left) + countNodes(root->right);
}

int main() {
    Node* root = createNode(1);

```

```
    root->left = createNode(2);  
    root->right = createNode(3);  
    root->left->left = createNode(4);  
    root->left->right = createNode(5);  
  
    cout << "Total number of nodes: " << countNodes(root);  
    return 0;  
}
```

Output



```
C:\Users\Wishaq Akbar\Down X + v  
Total number of nodes: 5  
-----  
Process exited after 0.06378 seconds with return value 0  
Press any key to continue . . . |
```

A **Tree** is a hierarchical data structure consisting of nodes, where each node has a value and references to its child nodes. The topmost node is called the **root**, and nodes without children are called **leaves**. Common tree types include Binary Trees, Binary Search Trees, and AVL Trees.

Program 1: Binary Tree Traversal

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

struct Node {
    int data;
    Node* left;
    Node* right;
};

Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

void inorder(Node* root) {
    if (root == NULL) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

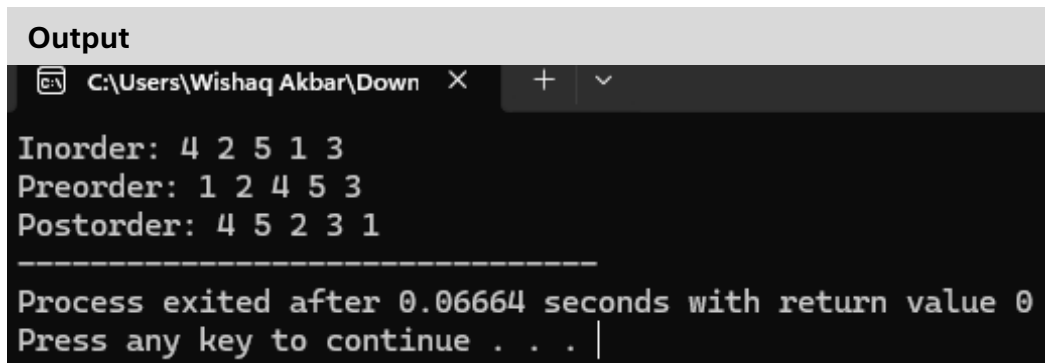
void preorder(Node* root) {
    if (root == NULL) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

void postorder(Node* root) {
    if (root == NULL) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

int main() {
    Node* root = createNode(1);
    root->left = createNode(2);
    root->right = createNode(3);
    root->left->left = createNode(4);
```

```
    root->left->right = createNode(5);

    cout << "Inorder: ";
    inorder(root);
    cout << "\nPreorder: ";
    preorder(root);
    cout << "\nPostorder: ";
    postorder(root);
    return 0;
}
```



```
Output
C:\Users\Wishaq Akbar\Down X + v

Inorder: 4 2 5 1 3
Preorder: 1 2 4 5 3
Postorder: 4 5 2 3 1
-----
Process exited after 0.06664 seconds with return value 0
Press any key to continue . . . |
```


A Singly Linked List is a linear data structure consisting of nodes. Each node contains two parts:

1. Data: The value stored in the node.
2. Pointer (next): A reference to the next node in the sequence.

The first node is called the **head**, and the last node's pointer is **NULL**, indicating the end of the list. Singly linked lists allow efficient insertion and deletion at any point in the sequence, but traversal is one-way, from the head to the end.

Program 1: Link List Operations

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

// Definition of a Node
struct Node {
    int data;
    Node* next;
};

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

// Function to insert a node at the end of the linked list
void insertAtEnd(Node*& head, int data) {
    Node* newNode = createNode(data);
    if (head == NULL) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Function to display the linked list
void displayList(Node* head) {
    Node* temp = head;
    while (temp != NULL) {
        cout << temp->data << " -> ";
    }
}
```

```

        temp = temp->next;
    }
    cout << "NULL" << endl;
}

// Function to search for a value in the linked list
bool search(Node* head, int value) {
    Node* temp = head;
    while (temp != NULL) {
        if (temp->data == value) {
            return true;
        }
        temp = temp->next;
    }
    return false;
}

// Function to delete a node by value
void deleteNode(Node*& head, int value) {
    if (head == NULL) return;

    // If the head node is the one to be deleted
    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    // Traverse to find the node to delete
    Node* temp = head;
    while (temp->next != NULL && temp->next->data != value) {
        temp = temp->next;
    }

    // If the node to delete was found
    if (temp->next != NULL) {
        Node* nodeToDelete = temp->next;
        temp->next = temp->next->next;
        delete nodeToDelete;
    }
}

int main() {
    Node* head = NULL;

    // Insert elements into the linked list
    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);
    insertAtEnd(head, 40);
}

```

```

    cout << "Linked List: ";
    displayList(head);

    // Search for a value in the list
    int searchValue = 20;
    if (search(head, searchValue)) {
        cout << "Value " << searchValue << " found in the list." <<
endl;
    } else {
        cout << "Value " << searchValue << " not found in the list."
<< endl;
    }

    // Delete a value from the list
    int deleteValue = 30;
    cout << "Deleting value " << deleteValue << " from the list." <<
endl;
    deleteNode(head, deleteValue);

    // Display the list after deletion
    cout << "Linked List after deletion: ";
    displayList(head);

    return 0;
}

```

Output

```

Linked List: 10 -> 20 -> 30 -> 40 -> NULL
Value 20 found in the list.
Deleting value 30 from the list.
Linked List after deletion: 10 -> 20 -> 40 -> NULL

```

```

-----
Process exited after 0.5078 seconds with return value 0

```

A Doubly Linked List is a linear data structure where each node contains three parts:

1. Data: The value stored in the node.
2. Pointer to the Next Node (next): A reference to the next node in the sequence.
3. Pointer to the Previous Node (prev): A reference to the previous node in the sequence.

This allows traversal in **both forward and backward directions**, making it more versatile than a singly linked list. However, it requires additional memory for the prev pointer and slightly more complex insertion/deletion operations.

Program 1: Binary Tree Traversal

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

// Definition of a Node
struct Node {
    int data;        // Stores the data value
    Node* next;      // Pointer to the next node
    Node* prev;      // Pointer to the previous node
};

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node(); // Allocate memory for a new node
    newNode->data = data;        // Assign the value to the node
    newNode->next = NULL;        // Initialize the next pointer to
    NULL
    newNode->prev = NULL;        // Initialize the prev pointer to
    NULL
    return newNode;
}

// Function to insert a node at the end of the doubly linked list
void insertAtEnd(Node*& head, int data) {
    Node* newNode = createNode(data); // Create a new node
    if (head == NULL) {                // If the list is empty
        head = newNode;                // Set the new node as the
        return;
    }
    Node* temp = head;                 // Start from the head
    while (temp->next != NULL) {        // Traverse to the last node
        temp = temp->next;
    }
    temp->next = newNode;               // Link the last node to the
    new node
```

```

        newNode->prev = temp;                // Link the new node back to
the last node
    }

// Function to display the list in forward direction
void displayForward(Node* head) {
    Node* temp = head;
    cout << "Doubly Linked List (Forward): ";
    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}

// Function to display the list in reverse direction
void displayReverse(Node* head) {
    if (head == NULL) return;

    // Traverse to the last node
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    // Traverse backward from the last node
    cout << "Doubly Linked List (Reverse): ";
    while (temp != NULL) {
        cout << temp->data << " -> ";
        temp = temp->prev;
    }
    cout << "NULL" << endl;
}

// Function to delete a node by value
void deleteNode(Node*& head, int value) {
    if (head == NULL) return; // If the list is empty, return

    Node* temp = head;

    // Traverse the list to find the node with the given value
    while (temp != NULL && temp->data != value) {
        temp = temp->next;
    }

    if (temp == NULL) return; // If the value is not found, return

    // If the node to be deleted is the head
    if (temp == head) {
        head = head->next;
        if (head != NULL) {

```

```

        head->prev = NULL;
    }
    delete temp;
    return;
}

// If the node to be deleted is in the middle or end
if (temp->next != NULL) {
    temp->next->prev = temp->prev;
}
if (temp->prev != NULL) {
    temp->prev->next = temp->next;
}

delete temp;
}

int main() {
    Node* head = NULL; // Initialize the head of the doubly linked
list

    // Insert elements into the doubly linked list
    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);
    insertAtEnd(head, 40);

    // Display the list in forward and reverse order
    displayForward(head);
    displayReverse(head);

    // Delete a value from the list
    int deleteValue = 20;
    cout << "Deleting value " << deleteValue << " from the list." <<
endl;
    deleteNode(head, deleteValue);

    // Display the list after deletion
    displayForward(head);
    displayReverse(head);

    return 0;
}

```

Output

```

Doubly Linked List (Forward): 10 -> 20 -> 30 -> 40 -> NULL
Doubly Linked List (Reverse): 40 -> 30 -> 20 -> 10 -> NULL
Deleting value 20 from the list.
Doubly Linked List (Forward): 10 -> 30 -> 40 -> NULL
Doubly Linked List (Reverse): 40 -> 30 -> 10 -> NULL

-----
Process exited after 0.4333 seconds with return value 0

```

A **Circular Linked List** is a linear data structure where the last node of the list points back to the first node, forming a circle. Circular linked lists can be:

1. **Singly Circular Linked List:** Each node has a next pointer pointing to the next node, and the last node points to the head.
2. **Doubly Circular Linked List:** Each node has next and prev pointers, with the next of the last node pointing to the head, and the prev of the head pointing to the last node.

This structure is useful for scenarios like implementing a circular queue or managing tasks in a round-robin scheduling system.

Program 1: Circular Singly Linked List

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

// Definition of a Node
struct Node {
    int data;          // Stores the data value
    Node* next;        // Pointer to the next node
};

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node(); // Allocate memory for the new node
    newNode->data = data;        // Assign the value
    newNode->next = NULL;        // Initialize the next pointer
    return newNode;
}

// Function to insert a node at the end of the circular linked list
void insertAtEnd(Node*& head, int data) {
    Node* newNode = createNode(data); // Create a new node
    if (head == NULL) {                // If the list is empty
        head = newNode;
        newNode->next = head;          // Point the new node to
        itself
        return;
    }
    Node* temp = head;                 // Start from the head
    while (temp->next != head) {        // Traverse to the last node
        temp = temp->next;
    }
    temp->next = newNode;               // Link the last node to the
    new node
    newNode->next = head;               // Point the new node to the
    head
```

```

}

// Function to display the circular linked list
void display(Node* head) {
    if (head == NULL) {                // If the list is empty
        cout << "Circular Linked List is empty." << endl;
        return;
    }
    Node* temp = head;
    cout << "Circular Linked List: ";
    do {                                // Traverse and print each
node        cout << temp->data << " -> ";
            temp = temp->next;
    } while (temp != head);            // Stop when we return to the
head        cout << "(head)" << endl;
}

// Function to delete a node by value
void deleteNode(Node*& head, int value) {
    if (head == NULL) return;          // If the list is empty,
return

    Node* temp = head;
    Node* prev = NULL;

    // Find the node with the given value
    do {
        if (temp->data == value) {
            if (temp == head) {        // If the node to be deleted
is the head        Node* last = head;
                    while (last->next != head) { // Find the last node
                        last = last->next;
                    }
                    if (head == head->next) { // If there's only one
node                delete head;
                    head = NULL;
                } else {
                    last->next = head->next; // Update last node's
next                head = head->next;      // Move head to the
next node        delete temp;
                }
            } else {                    // If the node to be deleted
is in the middle or end        prev->next = temp->next;
                                delete temp;
}

```



```

        }
        return;
    }
    prev = temp;
    temp = temp->next;
} while (temp != head);           // Stop when we return to the
head
}

int main() {
    Node* head = NULL; // Initialize the head of the circular linked
list

    // Insert elements into the circular linked list
    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);
    insertAtEnd(head, 40);

    // Display the circular linked list
    display(head);

    // Delete a node by value
    int deleteValue = 20;
    cout << "Deleting value " << deleteValue << " from the list." <<
endl;
    deleteNode(head, deleteValue);

    // Display the list after deletion
    display(head);

    return 0;
}

```

Output

```

C:\Users\wisha\OneDrive\Doc  X
Circular Linked List: 10 -> 20 -> 30 -> 40 -> (head)
Deleting value 20 from the list.
Circular Linked List: 10 -> 30 -> 40 -> (head)

-----
Process exited after 0.4471 seconds with return value 0

```

Program 1: Circular Doubly Linked List

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

// Definition of a Node
struct Node {
    int data;          // Data stored in the node
    Node* next;        // Pointer to the next node
    Node* prev;        // Pointer to the previous node
};

// Function to create a new node
Node* createNode(int data) {
    Node* newNode = new Node();
    newNode->data = data; // Assign data
    newNode->next = NULL; // Initialize next pointer
    newNode->prev = NULL; // Initialize prev pointer
    return newNode;
}

// Function to insert a node at the end of the doubly circular
// linked list
void insertAtEnd(Node*& head, int data) {
    Node* newNode = createNode(data); // Create a new node
    if (head == NULL) {                // If the list is empty
        head = newNode;
        newNode->next = newNode;        // Point to itself
        newNode->prev = newNode;        // Point to itself
        return;
    }
    Node* tail = head->prev;            // Get the last node (tail)
    tail->next = newNode;               // Link last node's next to
new node                               // Link new node's prev to
    newNode->prev = tail;               // Link new node's next to
    newNode->next = head;               // Link head's prev to new
head                                  node
    head->prev = newNode;
}

// Function to display the doubly circular linked list
void display(Node* head) {
    if (head == NULL) {                // If the list is empty
        cout << "Doubly Circular Linked List is empty." << endl;
        return;
    }
    Node* temp = head;
    cout << "Doubly Circular Linked List: ";
    do {
        cout << temp->data << " <-> "; // Traverse the list
    } while (temp->next != head);
}
```

```

        temp = temp->next;
    } while (temp != head);          // Stop when we reach the head
again
    cout << "(head)" << endl;
}

// Function to delete a node by value
void deleteNode(Node*& head, int value) {
    if (head == NULL) return;        // If the list is empty

    Node* temp = head;

    // Find the node with the given value
    do {
        if (temp->data == value) {
            if (temp->next == temp && temp->prev == temp) { //
Single node case
                delete temp;
                head = NULL;
                return;
            }
            Node* prevNode = temp->prev;    // Node before the
current node
            Node* nextNode = temp->next;    // Node after the
current node
            prevNode->next = nextNode;      // Update the links
            nextNode->prev = prevNode;
            if (temp == head) {             // If the head is
deleted
                head = nextNode;
            }
            delete temp;                    // Delete the node
            return;
        }
        temp = temp->next;
    } while (temp != head);              // Stop when we reach
the head again
}

// Main function
int main() {
    Node* head = NULL; // Initialize the head of the doubly circular
linked list

    // Insert elements into the doubly circular linked list
    insertAtEnd(head, 10);
    insertAtEnd(head, 20);
    insertAtEnd(head, 30);
    insertAtEnd(head, 40);

    // Display the list

```

```

    display(head);

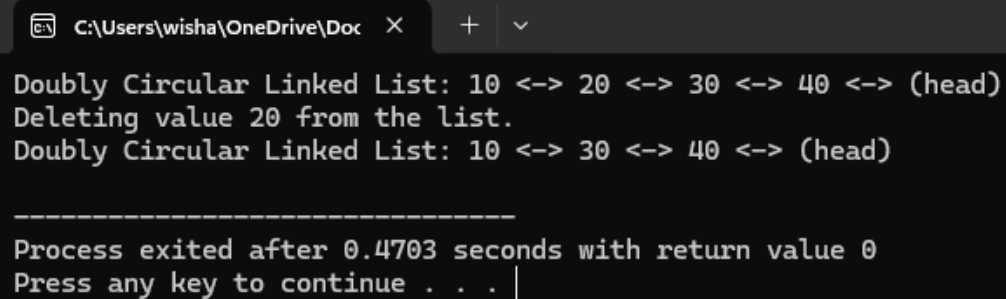
    // Delete a node by value
    int deleteValue = 20;
    cout << "Deleting value " << deleteValue << " from the list." <<
endl;
    deleteNode(head, deleteValue);

    // Display the list after deletion
    display(head);

    return 0;
}

```

Output



```

C:\Users\wisha\OneDrive\Doc  X  +  v
Doubly Circular Linked List: 10 <--> 20 <--> 30 <--> 40 <--> (head)
Deleting value 20 from the list.
Doubly Circular Linked List: 10 <--> 30 <--> 40 <--> (head)

-----
Process exited after 0.4703 seconds with return value 0
Press any key to continue . . .

```