

## **Table of Contents**

<b>Array .....</b>	<b>2</b>
<b>2d Array.....</b>	<b>6</b>
<b>Vector.....</b>	<b>11</b>
<b>List.....</b>	<b>14</b>
<b>Stacks.....</b>	<b>17</b>
<b>Queue.....</b>	<b>25</b>
<b>Dequeue.....</b>	<b>35</b>
<b>Trees.....</b>	<b>38</b>
<b>Binary Search trees.....</b>	<b>46</b>
<b>Singly link list.....</b>	<b>56</b>
<b>Doubly link list.....</b>	<b>59</b>
<b>Circular link list .....</b>	<b>64</b>

# **LAB MANUAL**

**NAME: M.HAMZA RIZWAN**

**REGISTRATION NUMBER : 2023-BS-AI-005**

## **ARRAY :**

An array is a linear data structure that stores a collection of elements of the same data type in contiguous memory locations.

### **SYNTAX :**

data\_type array\_name[size];

### **PROGRAM 1**

```
#include <iostream>
```

```
using namespace std;
```

```
int main() {
```

```
    int arr[5] = {10, 20, 30, 40, 50};
```

```
    int sum = 0;
```

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

```
        sum += arr[i];
```

```
    }
```

```
    cout << "Sum of array elements: " << sum << endl;
```

```
    return 0;
```

```
}
```

```
Sum of array elements: 150
```

### **PROGRAM 2**

```

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

```

```
Largest element in the array: 40
```

### **PROGRAM 3**

```

#include <iostream>
using namespace std;
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int start = 0;
    int end = 4;
    while (start < end) {
        // Swap elements
        int temp = arr[start];
        arr[start] = arr[end];

```

```

    arr[end] = temp;
    start++;
    end--;

}

cout << "Reversed array: ";
for (int i = 0; i < 5; i++) {
    cout << arr[i] << " ";
}

cout << endl;
return 0;
}

```

```
Reversed array: 5 4 3 2 1
```

## **PROGRAM 4**

```

#include <iostream>
using namespace std;
int main() {
    int arr[5] = {10, 21, 32, 43, 54};
    int evenCount = 0, oddCount = 0;
    for (int i = 0; i < 5; i++) {
        if (arr[i] % 2 == 0) {
            evenCount++;
        } else {
            oddCount++;
        }
    }
}

```

```
Even numbers count: 3
Odd numbers count: 2
```

```

    }
    cout << "Even numbers count: " << evenCount << endl;
    cout << "Odd numbers count: " << oddCount << endl;
    return 0;
}

```

## **PROGRAM 5**

```

#include <iostream>
using namespace std;
int main() {
    int arr1[5] = {1, 2, 3, 4, 5};
    int arr2[5];
    // Copying elements from arr1 to arr2
    for (int i = 0; i < 5; i++) {
        arr2[i] = arr1[i];
    }
    cout << "Elements of the second array: ";
    for (int i = 0; i < 5; i++) {
        cout << arr2[i] << " ";
    }
    cout << endl;
    return 0; }

```

```
Elements of the second array: 1 2 3 4 5
```

## 2D ARRAY :

### PROGRAM 1

```
#include <iostream>
using namespace std;
int main() {
    int arr[3][3] = {

        { 1, 2, 3 },
        { 4, 5, 6 },
        { 7, 8, 9 }
    };
    cout << "2D Array:" << endl;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            cout << arr[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

2D Array:

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

### PROGRAM 2

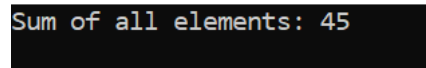
```
#include <iostream>
```

```

using namespace std;

int main() {
    int arr[3][3] = {
        { 1, 2, 3},
        { 4, 5, 6},
        { 7, 8, 9}
    };
    int sum = 0;
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            sum += arr[i][j];
        }
    }
    cout << "Sum of all elements: " << sum << endl;
    return 0;
}

```



Sum of all elements: 45

### **PROGRAM 3**

```

#include <iostream>

using namespace std;

int main() {
    int arr[3][3] = {
        { 10, 20, 30},
        { 40, 5, 60},
        { 70, 80, 90}
    };
}

```

```

int largest = arr[0][0];
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (arr[i][j] > largest) {
            largest = arr[i][j];
        }
    }
}

cout << "Largest element in the 2D array: " << largest << endl;
return 0;
}

```

```
Largest element in the 2D array: 90
```

## **PROGRAM 4**

```

#include <iostream>
using namespace std;
int main() {
    int arr[3][3] = {
        { 1, 2, 3},
        { 4, 5, 6},
        { 7, 8, 9}
    };
    int transposed[3][3];
    // Transposing the array
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            transposed[j][i] = arr[i][j];

```



```

    }
}
cout << "Transposed 2D Array:" << endl;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        cout << transposed[i][j] << " ";
    }
    cout << endl;
}
return 0;
}

```

```

Transposed 2D Array:
1 4 7

```

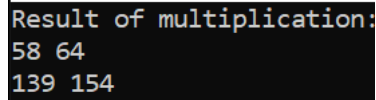
## **PROGRAM 5**

```

#include <iostream>
using namespace std;
int main() {
    int a[2][3] = {
        { 1, 2, 3},
        { 4, 5, 6}
    };
    int b[3][2] = {
        { 7, 8},
        { 9, 10},
        { 11, 12}
    };
    int result[2][2] = {0}; // Initialize result array to 0
    // Multiplying two matrices
}

```

```
for (int i = 0; i < 2; i++) {  
    for (int j = 0; j < 2; j++) {  
        for (int k = 0; k < 3; k++) {  
            result[i][j] += a[i][k] * b[k][j];    }  
        }  
    }  
    cout << "Result of multiplication:" << endl;  
    for (int i = 0; i < 2; i++) {  
        for (int j = 0; j < 2; j++) {  
            cout << result[i][j] << " ";  
        }  
        cout << endl;    }  
    return 0;  
}
```



Result of multiplication:  
58 64  
139 154

A terminal window with a black background and white text. It displays the output of a C++ program, showing the result of a matrix multiplication. The output is formatted with a title line followed by two rows of two numbers each, separated by spaces.

## **VECTOR :**

A vector is defined as a sequence container that encapsulates dynamic size arrays. It allows you to store elements of the same type and provides functionalities such as adding, removing, and accessing elements.

### **SYNTAX**

```
std::vector<data_type> vector_name;
```

OR

Using vector library

### **Program 1**

```
#include <iostream>
#include <vector>
using namespace std;
int main() {
    // Initialize a vector with values
    vector<int> numbers = {10, 20, 30, 40, 50};
    // Print the elements of the vector
    cout << "Elements in the vector: ";
    for (int num : numbers) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

```
Elements in the vector: 10 20 30 40 50
```

## **PROGRAM 2**

```
#include <iostream>

#include <vector>

using namespace std;

int main() {

    vector<int> numbers;

    // Adding elements to the vector
    for (int i = 1; i <= 5; i++) {
        numbers.push_back(i * 10); // Add multiples of 10
    }

    // Print the elements of the vector
    cout << "Elements in the vector after adding: ";
    for (int num : numbers) {
        cout << num << " ";
    }
    cout << endl;
    return 0;
}
```

```
Elements in the vector after adding: 10 20 30 40 50
```

## **PROGRAM 3**

```
#include <iostream>

#include <vector>

using namespace std;

int main() {

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

    // Remove the last element
```

```

numbers.pop_back();
// Print the elements of the vector after removal
cout << "Elements in the vector after pop_back: ";
for (int num : numbers) {
    cout << num << " ";
}
cout << endl;
return 0;
}

```

```
Elements in the vector after pop_back: 10 20 30 40
```

## **PROGRAM 4**

```

#include <iostream>
#include <vector>
using namespace std;
int main() {
    vector<int> numbers = {1, 2, 3, 4, 5};
    cout << "Size of the vector: " << numbers.size() << endl;
    numbers.push_back(6);
    numbers.push_back(7);
    cout << "New size of the vector after adding elements: " << numbers.size() << endl;
    return 0;
}

```

```
Size of the vector: 5
New size of the vector after adding elements: 7
```

## **PROGRAM 5**

```

#include <iostream>
#include <vector>

```

```
#include <algorithm>
```

```
using namespace std;
```

```
int main() {
```

```
    vector<int> numbers = {5, 3, 8, 1, 2};
```

```
    sort(numbers.begin(), numbers.end());
```

```
    cout << "Sorted elements in the vector: ";
```

```
    for (int num : numbers) {
```

```
        cout << num << " "; }
```

```
    cout << endl;
```

```
    return 0;
```

```
}
```

```
Sorted elements in the vector: 1 2 3 5 8
```

## LIST

### Definition:

A list is a collection of ordered elements of the same type, where each element is stored sequentially in memory or logically linked.

### Syntax:

```
list<int> numbers; // Declares a list of integers
```

### program 1:

```
#include<iostream>
```

```
#include<string>
```

```
using namespace std;
```

```
int main(){
```

```
    string fruits[5]={"Apple","Banana","Mango","Orange","Grapes"};
```

```

        for(int i=0;i<5;i++){
            cout<<fruits[i]<<endl;

        }
    }
}

```

```

Apple
Banana
Mango
Orange
Grapes

```

## **program 2:**

```

#include<iostream>

#include<string>

using namespace std;

int main(){

    string vegetables[5]={"Tomato","Carrot","Potato","Onion","Spinach"};

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

        cout<<vegetables[i]<<endl;    }

    }
}

```

```

Tomato
Carrot
Potato
Onion
Spinach

```

## **program 3:**

```

#include<iostream>

#include<string>

using namespace std;

int main(){

    string

```

```

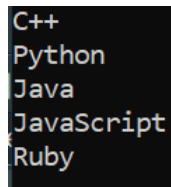
USA
Canada
UK
Australia
Germany

```

```
countries[5]={"USA","Canada","UK","Australia","Germany"};
    for(int i=0;i<5;i++){
        cout<<countries[i]<<endl;    }
}
```

### **program 4:**

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



```
int main(){
    string languages[5]={"C++","Python","Java","JavaScript","Ruby"};
    for(int i=0;i<5;i++){
        cout<<languages[i]<<endl;    }
}
```

### **program 5:**

```
#include<iostream>
using namespace std;
int main(){
    int primes[5] = {2, 3, 5, 7, 11};
    for(int i = 0; i < 5; i++){
        cout << primes[i]<<endl; }
}
```



```
2
3
5
7
11
```

## Stacks

### Definition:

a stack is a linear data structure that follows the Last In, First Out (LIFO) principle, meaning the last element added to the stack is the first one to be removed. It is often used to solve problems that require backtracking, such as evaluating expressions, parsing syntax, and managing function calls.

### Program 1:

```
#include <iostream>

#include <stack>

int main() {
    std::stack<int> stk;

    // Push elements onto the stack
    stk.push(10);
    stk.push(20);
    stk.push(30);

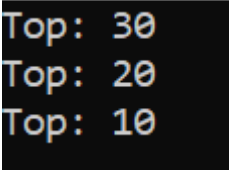
    // Pop and display all elements
```

```

while (!stk.empty()) {
    std::cout << "Top: " << stk.top() << std::endl;
    stk.pop();
}

return 0;
}

```



```

Top: 30
Top: 20
Top: 10

```

## **Program 2:**

```

#include <iostream>

#define MAX 100

class Stack {
    int arr[MAX];
    int top;
public:
    Stack() { top = -1; }
    void push(int x) {
        if (top == MAX - 1) {
            std::cout << "Stack Overflow\n";
            return;
        }
        arr[++top] = x;
    }
}

```

```

void pop() {
    if (top == -1) {
        std::cout << "Stack Underflow\n";
        return;
    }
    top--;
}

int peek() {
    if (top == -1) {
        std::cout << "Stack is Empty\n";
        return -1;
    }
    return arr[top];
}

bool isEmpty() {
    return top == -1;
}

};

int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);

```

```

Top: 30
Top: 20
Top: 10

```

```

while (!stack.isEmpty()) {
    std::cout << "Top: " << stack.peek() << std::endl;
    stack.pop();
}
return 0;
}

```

### **Program 3:**

```

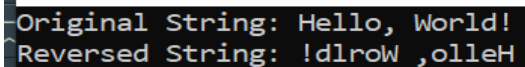
#include <iostream>
#include <stack>

std::string reverseString(const std::string &str) {
    std::stack<char> stk;
    for (char ch : str) {
        stk.push(ch);
    }

    std::string reversed;
    while (!stk.empty()) {
        reversed += stk.top();
        stk.pop();
    }
    return reversed;
}

int main() {
    std::string input = "Hello, World!";
    std::cout << "Original String: " << input << std::endl;

```



```

Original String: Hello, World!
Reversed String: !dlrow ,olleH

```

```
std::cout << "Reversed String: " << reverseString(input) << std::endl;
return 0;
}
```

### **Program 4:**

```
#include <iostream>
#include <stack>

bool isBalanced(const std::string &str) {
    std::stack<char> stk;
    for (char ch : str) {
        if (ch == '(' || ch == '{' || ch == '[') {
            stk.push(ch);
        } else if (ch == ')' || ch == '}' || ch == ']') {
            if (stk.empty() ||
                (ch == ')' && stk.top() != '(') ||
                (ch == '}' && stk.top() != '{') ||
                (ch == ']' && stk.top() != '[')) {
                return false;
            }
            stk.pop();
        }
    }
    return stk.empty();
}

int main() {
    std::string expression = "{[()]}";
```

```

if (isBalanced(expression)) {
    std::cout << "The parentheses are balanced.\n";
} else {
    std::cout << "The parentheses are not balanced.\n";
}

```

```
The parentheses are balanced.
```

```

return 0;
}

```

### **Program 5:**

```

#include <iostream>
#include <stack>
#include <cctype>

int precedence(char op) {
    if (op == '+' || op == '-') return 1;
    if (op == '*' || op == '/') return 2;
    return 0;
}

std::string infixToPostfix(const std::string &exp) {
    std::stack<char> stk;
    std::string result;
    for (char ch : exp) {
        if (std::isalnum(ch)) {
            result += ch; // Add operand to result
        } else if (ch == '(') {
            stk.push(ch);

```

```

    } else if (ch == ')') {
        while (!stk.empty() && stk.top() != '(') {
            result += stk.top();
            stk.pop();
        }
        stk.pop(); // Pop '('
    } else { // Operator
        while (!stk.empty() && precedence(stk.top()) >= precedence(ch)) {
            result += stk.top();
            stk.pop();
        }
        stk.push(ch);
    }
}

while (!stk.empty()) {
    result += stk.top();
    stk.pop();
}

return result;
}

int main() {
    std::string infix = "a+b*(c^d-e)^(f+g*h)-i";
    std::cout << "Infix: " << infix << std::endl;
    std::cout << "Postfix: " << infixToPostfix(infix) << std::endl;
}

```

```
return 0;  
}
```

```
< Infix: a+b*(c^d-e)^(f+g*h)-i  
Postfix:  
-----
```



# QUEUES

## Definition:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle, meaning the first element added to the queue is the first one to be removed. It is widely used for tasks that require order preservation, such as scheduling and buffering.

### Program 1:

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

```
int main() {
    queue<int> q;
```

```
    // Enqueue elements
```

```
    q.push(10);
```

```
    q.push(20);
```

```
    q.push(30);
```

```
    // Display and dequeue elements
```

```
Queue elements:
Front: 10
Front: 20
Front: 30
```

```
cout << "Queue elements:" << endl;
while (!q.empty()) {
    cout << "Front: " << q.front() << endl;
    q.pop();
}

return 0;
}
```

### **Program 2:**

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

class Queue {
    int front, rear, size;
    int* arr;

public:
    Queue(int capacity) {
        size = capacity;
        arr = new int[size];
        front = rear = -1;
    }

    ~Queue() { delete[] arr; }
```

```
void enqueue(int x) {  
    if (rear == size - 1) {  
        cout << "Queue Overflow\n";  
        return;  
    }  
    if (front == -1) front = 0;  
    arr[++rear] = x;  
}
```

```
void dequeue() {  
    if (front == -1 || front > rear) {  
        cout << "Queue Underflow\n";  
        return;  
    }  
    front++;  
}
```

```
int getFront() {  
    if (front == -1 || front > rear) {  
        cout << "Queue is Empty\n";  
        return -1;  
    }  
    return arr[front];  
}
```

```

bool isEmpty() {
    return front == -1 || front > rear;
}

};

int main() {
    Queue q(5);

    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);

    while (!q.isEmpty()) {
        cout << "Front: " << q.getFront() << endl;
        q.dequeue();
    }

    return 0;
}

```

```

Front: 10
Front: 20
Front: 30

```

### **Program 3:**

```

#include <iostream>
using namespace std;

class CircularQueue {

```

```
int *arr, front, rear, size;
```

```
public:
```

```
CircularQueue(int capacity) {
```

```
    size = capacity;
```

```
    arr = new int[size];
```

```
    front = rear = -1;
```

```
}
```

```
~CircularQueue() { delete[] arr; }
```

```
void enqueue(int x) {
```

```
    if ((rear + 1) % size == front) {
```

```
        cout << "Queue Overflow\n";
```

```
        return;
```

```
    }
```

```
    if (front == -1) front = 0;
```

```
    rear = (rear + 1) % size;
```

```
    arr[rear] = x;
```

```
}
```

```
void dequeue() {
```

```
    if (front == -1) {
```

```
        cout << "Queue Underflow\n";
```

```
        return;
```

```
    }  
    if (front == rear) {  
        front = rear = -1;  
    } else {  
        front = (front + 1) % size;  
    }  
}
```

```
int getFront() {  
    if (front == -1) {  
        cout << "Queue is Empty\n";  
        return -1;  
    }  
    return arr[front];  
}
```

```
bool isEmpty() {  
    return front == -1;  
}  
};
```

```
int main() {  
    CircularQueue q(5);  
  
    q.enqueue(10);
```

```
q.enqueue(20);  
q.enqueue(30);
```

```
Front: 10  
Front: 20  
Front: 30
```

```
while (!q.isEmpty()) {  
    cout << "Front: " << q.getFront() << endl;  
    q.dequeue();  
}
```

```
return 0;
```

```
}
```

#### **Program 4:**

```
#include <iostream>
```

```
#include <queue>
```

```
using namespace std;
```

```
struct Node {
```

```
    int data;
```

```
    Node* left;
```

```
    Node* right;
```

```
    Node(int val) : data(val), left(nullptr), right(nullptr) {}
```

```
};
```

```
void levelOrder(Node* root) {
```

```
    if (!root) return;
```

```

queue<Node*> q;
q.push(root);

while (!q.empty()) {
    Node* current = q.front();
    q.pop();
    cout << current->data << " ";

    if (current->left) q.push(current->left);
    if (current->right) q.push(current->right);
}
}

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

    Level Order Traversal: 1 2 3 4 5

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

```



```
    return 0;
}
```

### **Program 5:**

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

```
int main() {
    priority_queue<int> pq; // Max-Heap by default

    // Enqueue elements
    pq.push(10);
    pq.push(30);
    pq.push(20);

    cout << "Priority Queue (Max-Heap):" << endl;
    while (!pq.empty()) {
        cout << pq.top() << " "; // Top element
        pq.pop();
    }
    cout << endl;

    // Min-Heap using greater<int>
    priority_queue<int, vector<int>, greater<int>> minHeap;
    minHeap.push(10);
```

```
minHeap.push(30);
minHeap.push(20);
cout << "Priority Queue (Min-Heap):" << endl;
while (!minHeap.empty()) {
    cout << minHeap.top() << " ";
    minHeap.pop();
}

return 0;
}
```

```
Priority Queue (Max-Heap):
30 20 10
Priority Queue (Min-Heap):
10 20 30
```

# DEQUE

## Definition:

A **deque** (double-ended queue) is a dynamic data structure provided by the Standard Template Library (STL) that allows efficient insertion and deletion of elements at both the **front** and the back.

## Syntax:

```
deque<int> dq; // Declares a deque of integers
```

### program 1:

```
#include <iostream>
#include <deque>
using namespace std;
int main() {
    deque<int> dq;
    dq.push_back(1);
    dq.push_back(2);
    cout << "Front: " << dq.front() << endl;
    dq.pop_front();
    cout << "After pop, Front: " << dq.front() << endl;
    return 0;
}
```

```
Front: 1
After pop, Front: 2
```

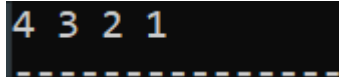
### program 2:

```

#include <iostream>
#include <deque>
#include <algorithm>
using namespace std;
int main() {
    deque<int> dq = {1, 2, 3, 4};
    reverse(dq.begin(), dq.end());
    for (int x : dq) cout << x << " ";

    return 0;
}

```



4 3 2 1

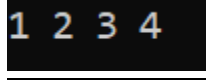
### **program 3:**

```

#include <iostream>
#include <deque>
using namespace std;
int main() {
    deque<int> dq = {1, 2, 3, 4};
    for (int x : dq) cout << x << " ";

    return 0;
}

```



1 2 3 4

### **program 4:**

```

#include <iostream>

```

```

#include <deque>
using namespace std;
int main() {
    deque<int> dq;
    dq.push_back(5);
    dq.push_back(10);

    dq.pop_back();
    cout << "Top of stack: " << dq.back() << endl;
    return 0;
}

```

```
Top of stack: 5
```

## **program 5:**

```

#include <iostream>
#include <deque>

using namespace std;
int main() {
    deque<int> dq = {1, 2, 3};
    cout << "Size: " << dq.size() << endl;
    cout << "Is Empty: " << (dq.empty() ? "Yes" : "No") << endl;
    return 0;
}

```

```
Size: 3
Is Empty: No
```

# Trees

## Definition:

A **tree** is a hierarchical data structure that consists of nodes connected by edges. It is a type of graph that is connected and acyclic, meaning there is exactly one path between any two nodes.

## Program 1:

```
#include <iostream>

using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

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

void inorder(Node* root) {
```

```

    if (root == nullptr) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

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

```

```

int main() {

```

```

Preorder Traversal: 1 2 4 5 3
Inorder Traversal: 4 2 5 1 3
Postorder Traversal: 4 5 2 3 1

```

```

    Node* root = new Node(1);
    root->left = new Node(2);
    root->right = new Node(3);
    root->left->left = new Node(4);
    root->left->right = new Node(5);
    cout << "Preorder Traversal: ";
    preorder(root);
    cout << "\nInorder Traversal: ";
    inorder(root);
    cout << "\nPostorder Traversal: ";
    postorder(root);
    return 0;

```

```
}
```

## **Program 2:**

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node {
```

```
    int data;
```

```
    Node* left;
```

```
    Node* right;
```

```
    Node(int value) {
```

```
        data = value;
```

```
        left = nullptr;
```

```
        right = nullptr;
```

```
    }
```

```
};
```

```
Node* insert(Node* root, int value) {
```

```
    if (root == nullptr) return new Node(value);
```

```
    if (value < root->data)
```

```
        root->left = insert(root->left, value);
```

```
    else
```

```
        root->right = insert(root->right, value);
```

```
    return root;
```

```
}
```

```
bool search(Node* root, int key) {
```

```
    if (root == nullptr) return false;
```

```
    if (root->data == key) return true;
```



```

    if (key < root->data) return search(root->left, key);
    return search(root->right, key);
}

```

```

Search 40: Found
Search 90: Not Found

```

```

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

    cout << "Search 40: " << (search(root, 40) ? "Found" : "Not Found") << endl;
    cout << "Search 90: " << (search(root, 90) ? "Found" : "Not Found") << endl;

    return 0;
}

```

### **Program 3:**

```

#include <iostream>

using namespace std;

struct Node {
    int data;
    Node* left;

```

```

Node* right;
Node(int value) {
    data = value;
    left = nullptr;
    right = nullptr;
}
};

int height(Node* root) {
    if (root == nullptr) return 0;
    int leftHeight = height(root->left);
    int rightHeight = height(root->right);
    return max(leftHeight, rightHeight) + 1;
}

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

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

```

```
Height of the tree: 3
```

#### **Program 4:**

```
#include <iostream>
```

```

#include <queue>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

void levelOrder(Node* root) {
    if (root == nullptr) return;
    queue<Node*> q;
    q.push(root);
    while (!q.empty()) {
        Node* current = q.front();
        q.pop();
        cout << current->data << " ";
        if (current->left) q.push(current->left);
        if (current->right) q.push(current->right);
    }
}

Level Order Traversal: 1 2 3 4 5

int main() {

```

```

Node* root = new Node(1);
root->left = new Node(2);
root->right = new Node(3);
root->left->left = new Node(4);
root->left->right = new Node(5);
cout << "Level Order Traversal: ";
levelOrder(root);
return 0;
}

```

### **Program 5:**

```

#include <iostream>
using namespace std;

struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

bool isBST(Node* root, Node* minNode, Node* maxNode) {
    if (root == nullptr) return true;

```

```

if ((minNode && root->data <= minNode->data) ||
    (maxNode && root->data >= maxNode->data)) {
    return false;
}
return isBST(root->left, minNode, root) && isBST(root->right, root, maxNode);
}

```

```
The tree is a valid Binary Search Tree.
```

```

int main() {
    Node* root = new Node(10);
    root->left = new Node(5);
    root->right = new Node(20);
    root->left->left = new Node(3);
    root->left->right = new Node(7);
    if (isBST(root, nullptr, nullptr))
        cout << "The tree is a valid Binary Search Tree.\n";
    else
        cout << "The tree is not a Binary Search Tree.\n";
    return 0;
}

```

# Binary search Trees

## Definition:

A **Binary Search Tree (BST)** is a specialized type of binary tree where each node has at most two children

### Program 1:

```
#include <iostream>
using namespace std;
struct Node {
    int data;

    Node* left;
    Node* right;
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};
```

```
// Function to insert a node into the BST
```

```
Node* insert(Node* root, int value) {
    if (root == nullptr) return new Node(value);
    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
```

```
Search for 40: Found
Search for 90: Not Found
```

```

    return root;
}

// Function to search for a key in the BST
bool search(Node* root, int key) {
    if (root == nullptr) 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 = nullptr;
    // Insert values into BST
    root = insert(root, 50);
    insert(root, 30);
    insert(root, 20);
    insert(root, 40);
    insert(root, 70);
    insert(root, 60);
    insert(root, 80);
    // Search for values
    cout << "Search for 40: " << (search(root, 40) ? "Found" : "Not Found") <<
endl;
    cout << "Search for 90: " << (search(root, 90) ? "Found" : "Not Found") <<
endl;
    return 0;
}

```

## **Program 2:**

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node {
```

```
    int data;
```

```
    Node* left;
```

```
    Node* right;
```

```
    Node(int value) {
```

```
        data = value;
```

```
        left = nullptr;
```

```
        right = nullptr;
```

```
    }
```

```
};
```

```
Node* insert(Node* root, int value) {
```

```
    if (root == nullptr) return new Node(value);
```

```
    if (value < root->data)
```

```
        root->left = insert(root->left, value);
```

```
    else if (value > root->data)
```

```
        root->right = insert(root->right, value);
```

```
    return root;
```

```
}
```

```
// In-order Traversal (Left -> Root -> Right)
```

```
void inorder(Node* root) {
```

```
    if (root == nullptr) return;
```

```
In-order Traversal: 20 30 40 50 60 70 80
Pre-order Traversal: 50 30 20 40 70 60 80
Post-order Traversal: 20 40 30 60 80 70 50
```



```

    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

// Pre-order Traversal (Root -> Left -> Right)
void preorder(Node* root) {
    if (root == nullptr) return;
    cout << root->data << " ";
    preorder(root->left);
    preorder(root->right);
}

// Post-order Traversal (Left -> Right -> Root)
void postorder(Node* root) {
    if (root == nullptr) return;
    postorder(root->left);
    postorder(root->right);
    cout << root->data << " ";
}

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

```

```

insert(root, 60);
insert(root, 80);
cout << "In-order Traversal: ";
inorder(root);
cout << "\nPre-order Traversal: ";
preorder(root);
cout << "\nPost-order Traversal: ";
postorder(root);
return 0;
}

```

### **Program 3:**

```

#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int value) {
        data = value;
        left = nullptr;
        right = nullptr;
    }
};

Node* insert(Node* root, int value) {
    if (root == nullptr) return new Node(value);

```

```

    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
    return root;
}

// Find the minimum value in the BST
int findMin(Node* root) {
    while (root->left != nullptr) {
        root = root->left;
    }
    return root->data;
}

// Find the maximum value in the BST
int findMax(Node* root) {
    while (root->right != nullptr) {
        root = root->right;
    }
    return root->data;
}

int main() {
    Node* root = nullptr;
    root = insert(root, 50);
    insert(root, 30);

```

```

Minimum value: 20
Maximum value: 80

```

```

insert(root, 20);
insert(root, 40);
insert(root, 70);
insert(root, 60);
insert(root, 80);

cout << "Minimum value: " << findMin(root) << endl;
cout << "Maximum value: " << findMax(root) << endl;

return 0;
}

```

#### **Program 4:**

```

#include <iostream>

using namespace std;

struct Node {

    int data;

    Node* left;

    Node* right;

    Node(int value) {

        data = value;

        left = nullptr;

        right = nullptr;

    }

};

Node* insert(Node* root, int value) {

    if (root == nullptr) return new Node(value);

```

```

    if (value < root->data)
        root->left = insert(root->left, value);
    else if (value > root->data)
        root->right = insert(root->right, value);
    return root;
}

// Find the minimum node in a BST
Node* findMinNode(Node* root) {
    while (root->left != nullptr) {
        root = root->left;
    }
    return root;
}

// Delete a node in the BST
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 {
        // Node with one or no child
        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 with two children
        Node* temp = findMinNode(root->right);
        root->data = temp->data;
        root->right = deleteNode(root->right, temp->data);
    }
}

```

```

        return temp;
    }
    if (root->right == nullptr) {
        Node* temp = root->left;
        delete root;
        return temp;
    }
    // Node with two children: Get the inorder successor
    Node* temp = findMinNode(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) return;
    inorder(root->left);
    cout << root->data << " ";
    inorder(root->right);
}

int main() {
    Node* root = nullptr;

    root = insert(root, 50);

```

```
insert(root, 30);
```

```
insert(root, 20);
```

```
insert(root, 40);
```

```
insert(root, 70);
```

```
insert(root, 60);
```

```
In-order before deletion: 20 30 40 50 60 70 80
```

```
In-order after deletion: 20 30 40 60 70 80
```

```
insert(root, 80);
```

```
cout << "In-order before deletion: ";
```

```
inorder(root);
```

```
root = deleteNode(root, 50);
```

```
cout << "\nIn-order after deletion: ";
```

```
inorder(root);
```

```
return 0;
```

```
}
```

### **Program 5:**

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node {
```

```
    int data;
```

```
    Node* left;
```

```
    Node* right;
```

```
    Node(int value) {
```

```
        data = value;
```

```
        left = nullptr;
```

```
        right = nullptr;
```

```

    }
};

bool isBST(Node* root, Node* minNode = nullptr, Node* maxNode = nullptr) {
    if (root == nullptr) return true;
    if (minNode && root->data <= minNode->data) return false;
    if (maxNode && root->data >= maxNode->data) return false;
    return isBST(root->left, minNode, root) && isBST(root->right, root, maxNode);
}

int main() {
    Node* root = new Node(10);
    root->left = new Node(5);
    root->right = new Node(20);
    root->left->left = new Node(3);
    root->left->right = new Node(7);
    if (isBST(root))
        cout << "The tree is a valid Binary Search Tree.\n";
    else
        cout << "The tree is not a Binary Search Tree.\n";
    return 0;
}

```

```
The tree is a valid Binary Search Tree.
```



# Singly Link List

- **Definition:** A Linked List is a linear data structure where elements, called nodes, are connected using pointers. Each node contains two parts.

## Example Programs

### program 1:

```
void insertEnd(Node*& head, int val) {  
    Node* newNode = new Node(val);  
    if (head == nullptr) {  
        head = newNode;  
        return;  
    }  
    Node* temp = head;  
    while (temp->next != nullptr) {  
        temp = temp->next;  
    }  
    temp->next = newNode;  
}
```

### Output

```
10 -> 20 -> 30 -> nullptr  
-----
```

### program 2:

```
void deleteNode(Node*& head, int val) {  
    if (head == nullptr) return;  
    if (head->data == val) {  
        Node* temp = head;  
        head = head->next;  
        delete temp;  
    }
```

### Output

```
Original List: 10 -> 20 -> 30 -> nullptr  
After Deletion: 10 -> 30 -> nullptr  
-----
```

```

        return;
    }

    Node* temp = head;
    while (temp->next != nullptr && temp->next->data != val) {
        temp = temp->next;
    }
    if (temp->next == nullptr) return;
    Node* toDelete = temp->next;
    temp->next = toDelete->next;
    delete toDelete;
}

```

### **program 3:**

```

void insertAtBegin(int data) {
    node *n = new node();
    n->data = data;
    n->link = head;
    head = n;
}

```

### **Output**

```

10 -> 20 -> 30 -> nullptr
-----

```

### **program 4:**

```

void display(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
}

```

### **Output**

```

10 -> 20 -> 30 -> nullptr
-----

```

```
}  
cout << "nullptr" << endl;
```

# Doubly Link List

- Definition

A **Doubly Linked List** is a type of linked data structure where each node contains three components:

- ❑ **Data:** The actual value stored in the node.
- ❑ **Next Pointer:** A pointer to the next node in the sequence.
- ❑ **Previous Pointer:** A pointer to the previous node in the sequence.

## Programs

### program 1:

```
struct DoublyNode {  
    int data;  
    DoublyNode* prev;  
    DoublyNode* next;
```

```
    DoublyNode(int val) : data(val), prev(nullptr), next(nullptr) { }  
};
```

### Output

```
10 <--> 20 <--> 30 <--> nullptr  
-----
```

### program 2:

```
void insertEnd(DoublyNode*& head, int val) {  
    DoublyNode* newNode = new DoublyNode(val);  
    if (head == nullptr) {  
        head = newNode;  
        return;  
    }
```

### Output

```
10 <--> 20 <--> 30 <--> nullptr  
-----
```

```

DoublyNode* temp = head;
while (temp->next != nullptr) {
    temp = temp->next;
}
temp->next = newNode;
newNode->prev = temp;}

```

### **program 3:**

```

void deleteNode(DoublyNode*& head, int val) {

```

```

    if (head == nullptr) return;

```

```

    if (head->data == val) {

```

```

        DoublyNode* temp = head;

```

```

        head = head->next;

```

```

        if (head != nullptr) head->prev = nullptr;

```

```

        delete temp;

```

```

        return;

```

```

    }

```

```

    DoublyNode* temp = head;

```

```

    while (temp != nullptr && temp->data != val) {

```

```

        temp = temp->next;

```

```

    }

```

```

    if (temp == nullptr) return;

```

```

    if (temp->next != nullptr) temp->next->prev = temp->prev;

```

```

    if (temp->prev != nullptr) temp->prev->next = temp->next;

```

```

    delete temp;

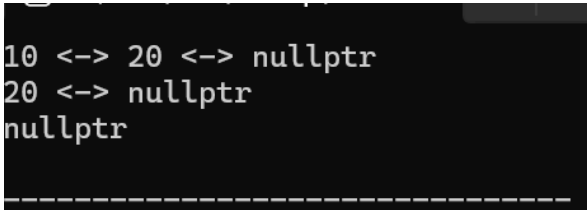
```

```

}

```

### **Output**



```

10 <-> 20 <-> nullptr
20 <-> nullptr
nullptr

```

#### **program 4:**

```
void display(DoublyNode* head) {
    DoublyNode* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
    cout << "nullptr" << endl;}
```

#### **Output**

```
10 <-> 20 <-> 30 <-> nullptr
-----
```

#### **program 5:**

```
void insertatposition(int position,int data){
    node *n = new node();
    n->data=data;
    if(position==1){
        n->next=head;
        if(head!=NULL){
            head->prev=n;}
        head = n;
        if(tail==NULL){
            tail=n;}}
    else{
        current = head;
        int count = 1;
        while(current!=nullptr && count<position-1){
            current = current->next;
            count++;}
```

#### **Output**

```
10 <-> 15 <-> 20 <-> nullptr
-----
Process exited after 0.06351 seconds with return value 0
Press any key to continue . . . |
```

```
if(current==NULL){
    cout<<"position out of bounds";
    delete n;
    return;}
n->next=current->next;
n->prev=current;
if (current->next != nullptr) {
current->next->prev = n;} else { // If inserting at the tail
tail = n;}
current->next = n;}}
```

# Lab 14: Circular Link List

## Characteristics

- The last node points to the first node.
- Can be singly or doubly linked.
- Enables circular traversal.

## • Example Programs

### program 1:

```
struct CNode {  
    int data;  
    CNode* next;
```

```
    CNode(int val) : data(val), next(nullptr) {}  
};
```

### program 2:

```
void insertEnd(CNode*& head, int val) {  
    CNode* newNode = new CNode(val);  
    if (head == nullptr) {  
        head = newNode;  
        newNode->next = head;  
        return;  
    }  
    CNode* temp = head;  
    while (temp->next != head) {  
        temp = temp->next;
```

## Output

```
10 20 30  
-----  
Process exited after 0.06958 seconds with return value 0  
Press any key to continue . . . |
```

## Output

```
10 20 30  
-----  
Process exited after 0.06958 seconds with return value 0  
Press any key to continue . . . |
```



```

    }

    temp->next = newNode;
    newNode->next = head;
}

```

### **program 3:**

```

void deleteNode(CNode*& head, int val) {
    if (head == nullptr) return;

    if (head->data == val && head->next == head) {
        delete head;
        head = nullptr;
        return;
    }

    CNode* temp = head;
    CNode* prev = nullptr;
    do {
        if (temp->data == val) break;
        prev = temp;
        temp = temp->next;
    } while (temp != head);
    if (temp == head && temp->data != val) return;
    if (temp == head) {
        prev = head;
        while (prev->next != head) prev = prev->next;
        head = head->next;
    }
}

```

### **Output**

```

10 20 30
10 30
-----
Process exited after 0.062 seconds with return value 0
Press any key to continue . . . |

```

```

        prev->next = head;
    } else {
        prev->next = temp->next;
    }
    delete temp;
} DoublyNode* temp = head;
while (temp != nullptr && temp->data != val) {
    temp = temp->next;
}
if (temp == nullptr) return;
if (temp->next != nullptr) temp->next->prev = temp->prev;
if (temp->prev != nullptr) temp->prev->next = temp->next;
delete temp;
}

```

#### **program 4:**

```

void display(CNode* head) {
    if (head == nullptr) return;
    CNode* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);
    cout << "(head)" << endl;
}

```

#### **program 5:**

#### **Output**

```

10 20 30
-----
Process exited after 0.06958 seconds with return value 0
Press any key to continue . . . |

```

#### **Output**

```

void insertAtPosition(int data, int position) {
    node *n = new node();
    n->data = data;
    if (position == 0) { // Insert at the beginning
        n->link = head;
        head = n;
        if (tail == nullptr) {
            tail = n;
        }
    } else {
        current = head;
        for (int i = 0; i < position - 1 && current != nullptr; ++i) {
            current = current->link;
        }
        if (current != nullptr) {
            n->link = current->link;
            current->link = n;
            if (n->link == nullptr) {
                tail = n;
            }
        } else {
            cout << "Position out of bounds" << endl;
        }
    }
}

```

```
10 20 30
```

```
-----
Process exited after 0.06958 seconds with return value 0
Press any key to continue . . . |
```