



The
University of
Faisalabad

LAB MANUAL

SUBMITTED BY : **LAIBA FATIMA**

CLASS : **BS AI 3rd SEM**

SUBMITTED TO : **MISS IRSHA QURESHI**

REGISTRATION # : **2023-BS-AI-047**

SECTION : **A**

Table of Contents

Contents

LAB 1.....	4
INTRODUCTION TO DSA :	4
1. Array	4
2. Stack	4
3. Linked List.....	5
4. Queue	5
5. Binary Tree	5
LAB 2.....	6
ARRAYS.....	6
Introduction:	6
DIAGRAMS:	11
LAB 3.....	12
STACK:.....	12
Introduction:	12
DIAGRAM / PICTORIAL REPRESENTATION:	21
LAB 4.....	22
Queue(Linear , Double , Circular)	22
Introduction:	22
DIAGRAM / PICTORIAL REPRESENTATION	33
LAB 5.....	34
LINKLIST (SINGLE,DOUBLE,CIRCULAR).....	34
Introduction:	34
DIAGRAMS.....	45
LAB 6.....	46
TREES.....	46
Introduction:	46
Key Concepts in Trees:.....	46
DIAGRAM:.....	54

LAB 7.....	55
VECTORS.....	55
Introduction:	55
DIAGRAM :	58
LAB 8.....	59
Binary Search Tree (BST).....	59
Introduction:	59
DIAGRAM :	67

LAB 1

INTRODUCTION TO DSA :

DSA (Data Structures and Algorithms) is a combination of two foundational concepts in computer science.

1. **Data Structures:** Efficient ways to organize, store, and manage data for easy access and modification. Examples include arrays, stacks, linked lists, and trees.
2. **Algorithms:** Step-by-step procedures or methods for solving problems or performing tasks efficiently, such as sorting, searching, and pathfinding.

Together, DSA forms the backbone of software development and is essential for writing optimized and scalable code.

1. Array

Definition: An array is a collection of elements of the same type, stored in contiguous memory locations and accessed using an index.

Syntax (C++):

```
int arr[5] = {1, 2, 3, 4, 5};
```

2. Stack

Definition: A stack is a linear data structure that follows the Last In First Out (LIFO) principle, where elements are added and removed from the same end.

Syntax (C++ using STL):

```
#include <stack>

stack<int> s;
```

3. Linked List

Definition: A linked list is a linear data structure where elements, called nodes, are connected using pointers, with each node containing data and a reference to the next node.

Syntax (C++):

```
struct Node {  
    int data;  
    Node* next;  
};
```

4. Queue

Definition: A queue is a linear data structure that follows the First In First Out (FIFO) principle, where elements are added from the rear and removed from the front.

Syntax (C++ using STL):

```
#include <queue>  
  
queue<int> q;
```

5. Binary Tree

Definition: A binary tree is a hierarchical data structure in which each node has at most two children, referred to as the left and right child.

Syntax (C++):

```
struct TreeNode {  
    int data;  
    TreeNode* left;  
    TreeNode* right;  
};
```

LAB 2

ARRAYS:

Introduction:

Arrays are one of the most fundamental data structures in programming, providing a way to store a fixed-size sequential collection of elements of the same type. They are used to efficiently

Operations Covered in This Lab:

- Summation of array elements.
- Finding the maximum element.
- Reversing an array.
- Counting occurrences of specific elements.
- Merging two arrays.

Syntax (C++):

```
int arr[5] = {1, 2, 3, 4, 5};
```

Program 1

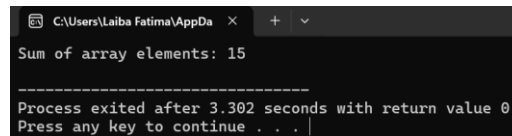
Statement:

The program calculates and displays the sum of all the elements in a given array.

```
#include <iostream>
using namespace std;
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int sum = 0;
    int size = sizeof(arr) / sizeof(arr[0]);
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    cout << "Sum of array elements: " << sum << endl;
```

```
    return 0;
}
```

OUTPUT



```
C:\Users\Laiba Fatima\AppData...
Sum of array elements: 15
-----
Process exited after 3.302 seconds with return value 0
Press any key to continue . . . |
```

Program 2

Statement:

The program finds and displays the maximum element in a given array of integers.

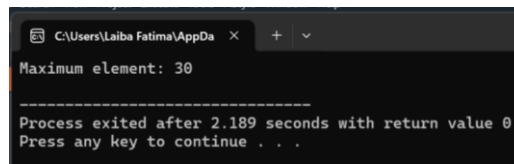
```
#include <iostream>
using namespace std;

int main() {
    int arr[] = {10, 20, 5, 30, 25};
    int maxElement = arr[0];
    int size = sizeof(arr) / sizeof(arr[0]);

    for (int i = 1; i < size; i++) {
        if (arr[i] > maxElement) {
            maxElement = arr[i];
        }
    }

    cout << "Maximum element: " << maxElement << endl;
    return 0;
}
```

OUTPUT



```
C:\Users\Laiba Fatima\AppData
Maximum element: 30
-----
Process exited after 2.189 seconds with return value 0
Press any key to continue . . .
```

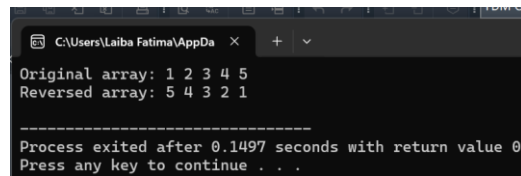
Program 3

Statement:

The program displays the elements of a given array in their original order and then in reverse order.

```
#include <iostream>
using namespace std;
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int size = sizeof(arr) / sizeof(arr[0]);
    cout << "Original array: ";
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
    cout << "Reversed array: ";
    for (int i = size - 1; i >= 0; i--) {
        cout << arr[i] << " ";
    }
    cout << endl;
    return 0;
}
```


OUTPUT



```
C:\Users\Laiba Fatima\AppData...  
Original array: 1 2 3 4 5  
Reversed array: 5 4 3 2 1  
-----  
Process exited after 0.1497 seconds with return value 0  
Press any key to continue . . .
```

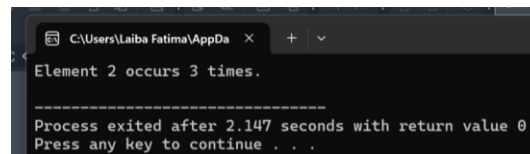
Program 4

STATEMENT:

Count the number of occurrences of a specific "element" within an integer array.

```
#include <iostream>  
using namespace std;  
int main() {  
    int arr[] = {1, 2, 3, 4, 2, 2, 5};  
    int element = 2;  
    int count = 0;  
    int size = sizeof(arr) / sizeof(arr[0]);  
    for (int i = 0; i < size; i++) {  
        if (arr[i] == element) {  
            count++;  
        }  
    }  
    cout << "Element " << element << " occurs " << count << " times." << endl;  
    return 0;  
}
```

OUTPUT



```
C:\Users\Laiba Fatima\AppData...
Element 2 occurs 3 times.
-----
Process exited after 2.147 seconds with return value 0
Press any key to continue . . .
```

Program 5

Statement:

This C++ code demonstrates the process of merging two integer arrays, arr1 and arr2, into a single array, merged

```
#include <iostream>

using namespace std;

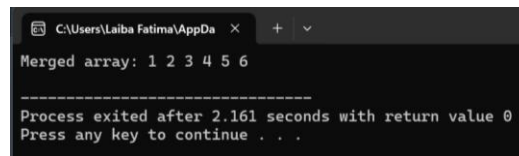
int main() {
    int arr1[] = {1, 2, 3};
    int arr2[] = {4, 5, 6};
    int size1 = sizeof(arr1) / sizeof(arr1[0]);
    int size2 = sizeof(arr2) / sizeof(arr2[0]);
    int mergedSize = size1 + size2;
    int merged[mergedSize];

    // Copy elements of arr1 to merged array
    for (int i = 0; i < size1; i++) {
        merged[i] = arr1[i];
    }

    // Copy elements of arr2 to merged array
    for (int i = 0; i < size2; i++) {
        merged[size1 + i] = arr2[i];
    }
}
```

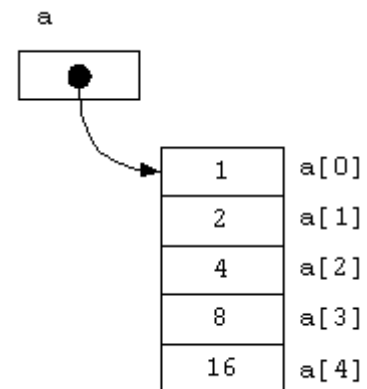
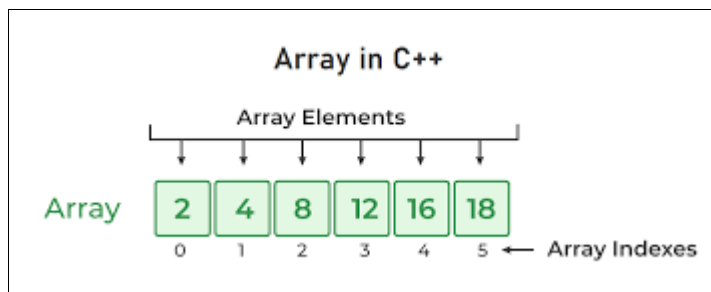
```
cout << "Merged array: ";  
for (int i = 0; i < mergedSize; i++) {  
    cout << merged[i] << " ";  
}  
cout << endl;  
return 0;  
}
```

OUTPUT



```
C:\Users\Laiba Fatima\AppData...  
Merged array: 1 2 3 4 5 6  
-----  
Process exited after 2.161 seconds with return value 0  
Press any key to continue . . .
```

DIAGRAMS:



LAB 3

STACK:

Introduction:

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The element added last is removed first.

Key Operations in Stacks:

1. **Push** : Adds an element to the top of the stack.
2. **Pop** : Removes the top element of the stack.
3. **Peek** : Retrieves the top element without removing it.
4. **IsEmpty and IsFull** : Checks stack state.

Applications of Stacks:

1. Function call management in recursion.
2. Reversing strings.
3. Parsing expressions (e.g., in compilers).

Syntax (C++ using STL):

```
#include <stack>
```

```
stack<int> s;
```

Program 1

Statement:

This C++ code implements a basic Stack data structure using an array.

```
#include <iostream>
```

```
using namespace std;
```

```
class Stack {
```

```
private:
```

```
    int top;
```

```
int arr[5];
```

```
public:
```

```
Stack() {  
    top = -1;  
    for (int i = 0; i < 5; i++) {  
        arr[i] = 0;  
    }  
}
```

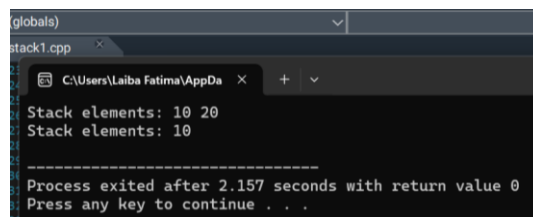
```
void push(int value) {  
    if (top == 4) {  
        cout << "Stack Overflow!" << endl;  
    } else {  
        top++;  
        arr[top] = value;  
    }  
}
```

```
void pop() {  
    if (top == -1) {  
        cout << "Stack Underflow!" << endl;  
    } else {  
        arr[top] = 0;  
        top--;  
    }  
}
```

```
void display() {  
    cout << "Stack elements: ";  
    for (int i = 0; i <= top; i++) {  
        cout << arr[i] << " ";  
    }  
    cout << endl;  
}  
};
```

```
int main() {  
    Stack stack;  
    stack.push(10);  
    stack.push(20);  
    stack.display();  
    stack.pop();  
    stack.display();  
    return 0;  
}
```

OUTPUT

A screenshot of a terminal window showing the output of a C++ program. The window has a title bar with 'globals' and a dropdown arrow. Below the title bar, there's a tab labeled 'stack1.cpp'. The main content area shows the output: 'Stack elements: 10 20' on the first line, 'Stack elements: 10' on the second line, followed by a separator line of dashes, and then 'Process exited after 2.157 seconds with return value 0' and 'Press any key to continue . . .'.

```
globals  
stack1.cpp  
C:\Users\Laiba Fatima\AppData  
Stack elements: 10 20  
Stack elements: 10  
-----  
Process exited after 2.157 seconds with return value 0  
Press any key to continue . . .
```

Program 2

Statement :

This code provides a basic example of how to implement a stack data structure in C++ with the key operations of pushing elements onto the stack and retrieving the top element

```
#include <iostream>

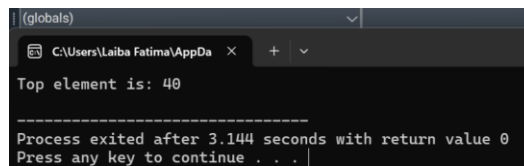
using namespace std;

class Stack {
private:
    int top;
    int arr[5];
public:
    Stack() {
        top = -1;
    }
    void push(int value) {
        if (top == 4) {
            cout << "Stack Overflow!" << endl;
        } else {
            top++;
            arr[top] = value;
        }
    }
    int peek() {
        if (top == -1) {
            cout << "Stack is empty!" << endl;
            return -1;
        } else {
```

```
        return arr[top];
    }
}
};

int main() {
    Stack stack;
    stack.push(30);
    stack.push(40);
    cout << "Top element is: " << stack.peek() << endl;
    return 0;
}
```

OUTPUT



```
(globals)
C:\Users\Laiba Fatima\AppData...
Top element is: 40
-----
Process exited after 3.144 seconds with return value 0
Press any key to continue . . . |
```

Program 3

Statement :

This C++ code demonstrates a basic implementation of a Stack data structure with functionalities for checking if the stack is empty, pushing elements onto the stack, and popping elements from the stack.

```
#include <iostream>

using namespace std;

class Stack {
private:
    int top;
```



```
int arr[5];

public:

Stack() {
    top = -1;
}

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

void push(int value) {
    if (top == 4) {
        cout << "Stack Overflow!" << endl;
    } else {
        top++;
        arr[top] = value;
    }
}

void pop() {
    if (isEmpty()) {
        cout << "Stack is already empty!" << endl;
    } else {
        top--;
    }
}

};

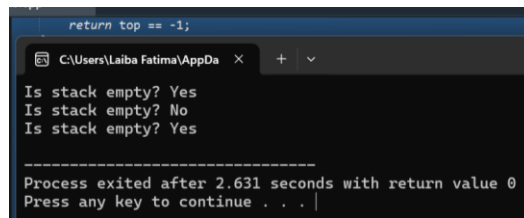
int main() {
    Stack stack;

    cout << "Is stack empty? " << (stack.isEmpty() ? "Yes" : "No") << endl;

    stack.push(50);
```

```
cout << "Is stack empty? " << (stack.isEmpty() ? "Yes" : "No") << endl;
stack.pop();
cout << "Is stack empty? " << (stack.isEmpty() ? "Yes" : "No") << endl;
return 0;
}
```

OUTPUT



```
return top == -1;
C:\Users\Laiba Fatima\AppData...
Is stack empty? Yes
Is stack empty? No
Is stack empty? Yes
-----
Process exited after 2.631 seconds with return value 0
Press any key to continue . . . |
```

Program 4

Statement :

This code highlights the importance of checking for stack overflow when pushing elements onto a stack, preventing potential errors and ensuring proper stack behavior.

```
#include <iostream>
using namespace std;
class Stack {
private:
    int top;
    int arr[5];

public:
    Stack() {
        top = -1;
    }
}
```

```
bool isFull() {
    return top == 4;
}

void push(int value) {
    if (isFull()) {
        cout << "Stack is full!" << endl;
    } else {
        top++;
        arr[top] = value;
    }
}

};

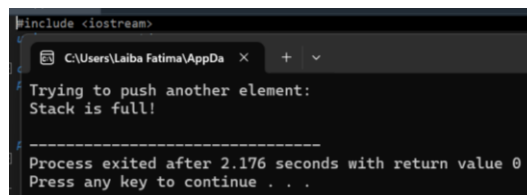
int main() {
    Stack stack;

    stack.push(10);
    stack.push(20);
    stack.push(30);
    stack.push(40);
    stack.push(50);

    cout << "Trying to push another element:" << endl;
    stack.push(60); // Should display "Stack is full!"

    return 0;
}
```

OUTPUT



```
#include <iostream>

C:\Users\Laiba Fatima\AppData...
Trying to push another element:
Stack is full!

-----
Process exited after 2.176 seconds with return value 0
Press any key to continue . . .
```

Program 5

Statement :

This code provides a simple example of how to determine the current size of a stack, which can be useful in various applications where tracking the number of elements in the stack is necessary.

```
#include <iostream>

using namespace std;

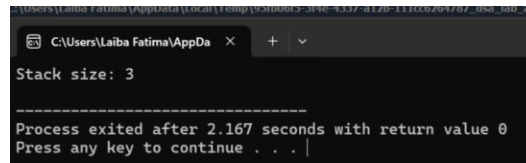
class Stack {
private:
    int top;
    int arr[5];
public:
    Stack() {
        top = -1;
    }

    void push(int value) {
        if (top == 4) {
            cout << "Stack Overflow!" << endl;
        } else {
            top++;
            arr[top] = value;
        }
    }

    int size() {
        return top + 1;
    }
}
```

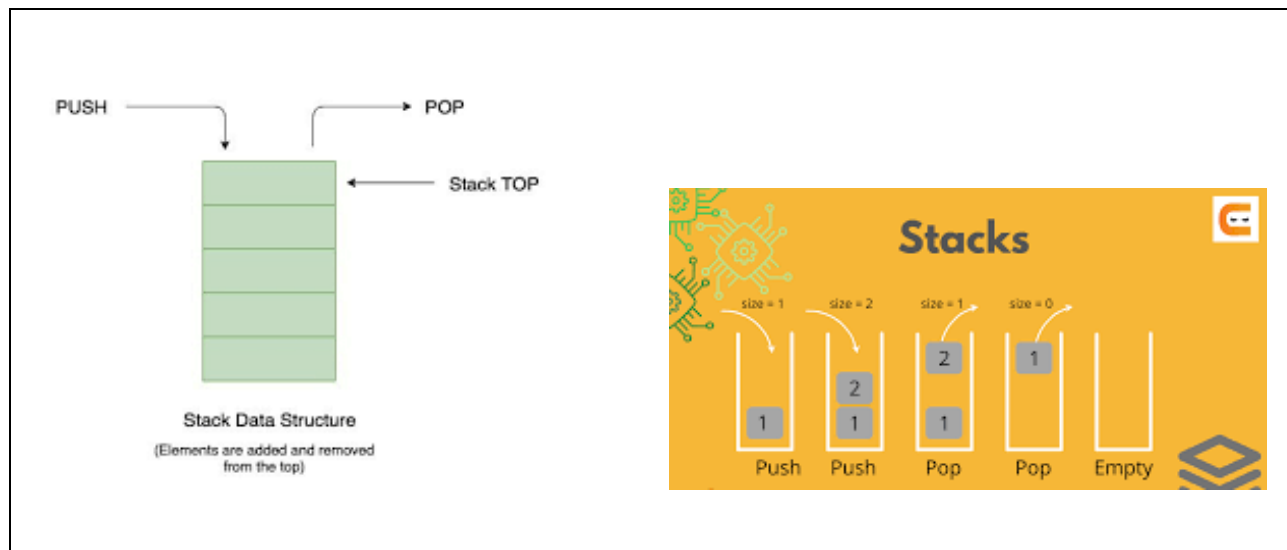
```
};  
  
int main() {  
    Stack stack;  
    stack.push(10);  
    stack.push(20);  
    stack.push(30);  
    cout << "Stack size: " << stack.size() << endl;  
    return 0;  
}
```

OUTPUT



```
C:\Users\Laiba Fatima\AppData... x + v  
Stack size: 3  
-----  
Process exited after 2.167 seconds with return value 0  
Press any key to continue . . . |
```

DIAGRAM / PICTORIAL REPRESENTATION:



LAB 4

Queue(Linear , Double , Circular)

Introduction:

A queue is a linear data structure that follows the First In, First Out (FIFO) principle. It ensures that the first element added is the first to be removed. A queue allows insertion at one end (rear) and deletion from the other end (front).

Types of Queues:

- **Linear Queue:** A straightforward queue where elements are processed sequentially.
- **Circular Queue:** A queue where the last position connects to the first, optimizing memory usage.
- **Priority Queue:** Elements are dequeued based on priority rather than order.

Syntax (C++ using STL):

```
#include <queue>

queue<int> q;
```

Program 1

Statement:

Demonstrates enqueue and dequeue operations using arrays

```
#include <iostream>

using namespace std;

#define SIZE 5

class Queue {
    int arr[SIZE], front, rear;
public:
    Queue() : front(-1), rear(-1) {}
```

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

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

void display() {
    if (front == -1 || front > rear) {
        cout << "Queue is empty\n";
        return;
    }
    cout << "Queue elements: ";
    for (int i = front; i <= rear; i++)
        cout << arr[i] << " ";
    cout << endl;
}
};
```

```
int main() {  
    Queue q;  
    q.enqueue(10);  
    q.enqueue(20);  
    q.enqueue(30);  
    q.display();  
    q.dequeue();  
    q.display();  
    return 0;  
}
```

OUTPUT

```
10  enqueued  
20  enqueued  
30  enqueued  
Queue elements: 10 20 30  
10  dequeued  
Queue elements: 20 30
```

Program 2

Statement:

Checks if the queue is empty before performing operations.

```
#include <iostream>  
using namespace std;  
#define SIZE 5
```



```
class Queue {
    int arr[SIZE], front, rear;
public:
    Queue() : front(-1), rear(-1) {}
    bool isEmpty() {
        return (front == -1 || front > rear);
    }
    void enqueue(int val) {
        if (rear == SIZE - 1) {
            cout << "Queue Overflow\n";
            return;
        }
        if (front == -1) front = 0;
        arr[++rear] = val;
    }

    void display() {
        if (isEmpty()) {
            cout << "Queue is empty\n";
            return;
        }
        cout << "Queue elements: ";
        for (int i = front; i <= rear; i++)
            cout << arr[i] << " ";
        cout << endl;
    }
};
```

```
int main() {  
    Queue q;  
    cout << "Is queue empty? " << (q.isEmpty() ? "Yes" : "No") << endl;  
    q.enqueue(5);  
    cout << "Is queue empty? " << (q.isEmpty() ? "Yes" : "No") << endl;  
    q.display();  
    return 0;  
}
```

OUTPUT

```
Is queue empty? Yes  
Is queue empty? No  
Queue elements: 5
```

Program 3

Statement:

Implements a circular queue using an array.

```
#include <iostream>  
using namespace std;  
#define SIZE 5  
class CircularQueue {  
    int arr[SIZE], front, rear;  
  
public:
```

```
CircularQueue() : front(-1), rear(-1) {}
```

```
void enqueue(int val) {
```

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

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

```
        return;
```

```
    }
```

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

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

```
    arr[rear] = val;
```

```
    cout << val << " enqueued\n";
```

```
}
```

```
void dequeue() {
```

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

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

```
        return;
```

```
    }
```

```
    cout << arr[front] << " dequeued\n";
```

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

```
    else front = (front + 1) % SIZE;
```

```
}
```

```
void display() {
```

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

```
        cout << "Queue is empty\n";
```

```
        return;
```

```
    }
```

```
    cout << "Queue elements: ";
```

```
        int i = front;
        while (true) {
            cout << arr[i] << " ";
            if (i == rear) break;
            i = (i + 1) % SIZE;
        }
        cout << endl;
    }
};

int main() {
    CircularQueue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();
    q.dequeue();
    q.display();
    return 0;
}
```

OUTPUT

```
10  enqueued
20  enqueued
30  enqueued
Queue elements: 10 20 30
10  dequeued
Queue elements: 20 30
```

Program 4

Statement:

Reverses the queue using an auxiliary array.

```
#include <iostream>

using namespace std;

#define SIZE 5

class Queue {
    int arr[SIZE], front, rear;
public:
    Queue() : front(-1), rear(-1) {}
    void enqueue(int val) {
        if (rear == SIZE - 1) {
            cout << "Queue Overflow\n";
            return;
        }
        if (front == -1) front = 0;
        arr[++rear] = val;
    }
    void reverseQueue() {
        if (front == -1 || front > rear) {
            cout << "Queue is empty\n";
            return;
        }
        int temp[SIZE];
        int j = 0;
        for (int i = rear; i >= front; i--)
```

```
        temp[j++] = arr[i];
    for (int i = 0; i < j; i++)
        arr[front + i] = temp[i];
    cout << "Queue reversed\n";
}

void display() {
    if (front == -1 || front > rear) {
        cout << "Queue is empty\n";
        return;
    }
    cout << "Queue elements: ";
    for (int i = front; i <= rear; i++)
        cout << arr[i] << " ";
    cout << endl;
}

};

int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    q.display();
    q.reverseQueue();
    q.display();
    return 0;
}
```

OUTPUT

```
Queue elements: 10 20 30
Queue reversed
Queue elements: 30 20 10
```

Program 5

Statement:

Implements a priority queue using manual sorting.

```
#include <iostream>
using namespace std;
#define SIZE 5
class PriorityQueue {
    int arr[SIZE], rear;

public:
    PriorityQueue() : rear(-1) {}
    void enqueue(int val) {
        if (rear == SIZE - 1) {
            cout << "Queue Overflow\n";
            return;
        }
        int i = ++rear;
        while (i > 0 && arr[i - 1] > val) {
            arr[i] = arr[i - 1];
            i--;
        }
    }
};
```

```
    }
    arr[i] = val;
    cout << val << " enqueued\n";
}

void dequeue() {
    if (rear == -1) {
        cout << "Queue Underflow\n";
        return;
    }
    cout << arr[0] << " dequeued\n";
    for (int i = 0; i < rear; i++)
        arr[i] = arr[i + 1];
    rear--;
}

void display() {
    if (rear == -1) {
        cout << "Queue is empty\n";
        return;
    }
    cout << "Queue elements: ";
    for (int i = 0; i <= rear; i++)
        cout << arr[i] << " ";
    cout << endl;
}

};

int main() {
    PriorityQueue pq;
    pq.enqueue(30);
```

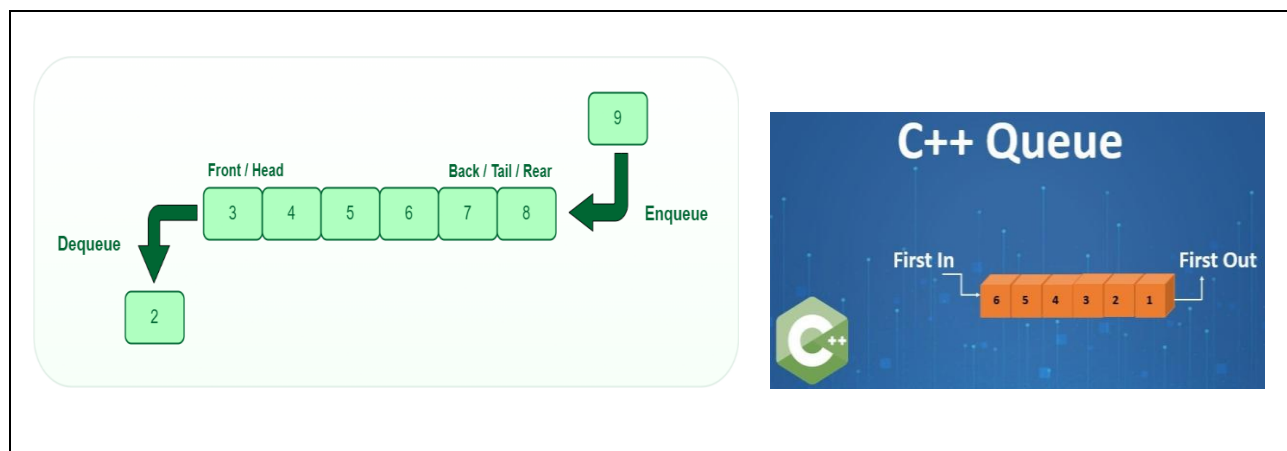


```
pq.enqueue(10);  
pq.enqueue(20);  
pq.display();  
pq.dequeue();  
pq.display();  
return 0;  
}
```

OUTPUT

```
30  enqueued  
10  enqueued  
20  enqueued  
Queue elements: 10 20 30  
10  dequeued  
Queue elements: 20 30
```

DIAGRAM / PICTORIAL REPRESENTATION



LAB 5

LINKLIST (SINGLE,DOUBLE,CIRCULAR)

Introduction:

A linked list is a dynamic data structure consisting of nodes, where each node contains data and a pointer to the next (or previous) node.

Types of Linked Lists:

- **Single Linked List:** Each node points to the next node in sequence.
- **Double Linked List:** Each node contains pointers to both its previous and next nodes.
- **Circular Linked List:** The last node connects back to the first, creating a loop.

Syntax (C++):

```
struct Node {  
    int data;  
    Node* next;  
};
```

Program 1

Statement:

Demonstrates insertion, deletion, and display operations in a single linked list.

```
#include <iostream>  
using namespace std;  
struct Node {  
    int data;  
    Node* next;  
};  
class SingleLinkedList {
```

```
Node* head;
```

```
public:
```

```
SingleLinkedList() : head(nullptr) {}
```

```
void insertAtEnd(int val) {
```

```
    Node* newNode = new Node{val, nullptr};
```

```
    if (!head) {
```

```
        head = newNode;
```

```
        return;
```

```
    }
```

```
    Node* temp = head;
```

```
    while (temp->next) temp = temp->next;
```

```
    temp->next = newNode;
```

```
}
```

```
void deleteFromFront() {
```

```
    if (!head) {
```

```
        cout << "List is empty\n";
```

```
        return;
```

```
    }
```

```
    Node* temp = head;
```

```
    head = head->next;
```

```
    delete temp;
```

```
}
```

```
void display() {
```

```
    if (!head) {
```

```
        cout << "List is empty\n";
        return;
    }
    Node* temp = head;
    while (temp) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
};
```

```
int main() {
    SingleLinkedList list;
    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.display();
    list.deleteFromFront();
    list.display();
    return 0;
}
```

OUTPUT

```
10 -> 20 -> 30 -> NULL
20 -> 30 -> NULL
```

Program 2

Statement:

Demonstrates insertion at the front and display operations in a double linked list.

```
#include <iostream>
using namespace std;
struct Node {
    int data;
    Node* prev;
    Node* next;
};
class DoubleLinkedList {
    Node* head;
public:
    DoubleLinkedList() : head(nullptr) {}

    void insertAtFront(int val) {
        Node* newNode = new Node{val, nullptr, head};
        if (head) head->prev = newNode;
        head = newNode;
    }

    void display() {
        if (!head) {
            cout << "List is empty\n";
            return;
        }
    }
```

```
Node* temp = head;
while (temp) {
    cout << temp->data << " <-> ";
    temp = temp->next;
}
cout << "NULL\n";
}
};
```

```
int main() {
    DoubleLinkedList list;
    list.insertAtFront(10);
    list.insertAtFront(20);
    list.insertAtFront(30);
    list.display();
    return 0;
}
```

OUTPUT

```
30 <-> 20 <-> 10 <-> NULL
```

Program 3

Statement :

Demonstrates insertion at the end and display operations in a circular single linked list.

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node {  
    int data;  
    Node* next;  
};
```

```
class CircularSingleLinkedList {  
    Node* head;
```

```
public:
```

```
    CircularSingleLinkedList() : head(nullptr) {}
```

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

```
    void display() {  
        if (!head) {
```

```
        cout << "List is empty\n";
        return;
    }
    Node* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);
    cout << "(head)\n";
}
};

int main() {
    CircularSingleLinkedList list;
    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.display();
    return 0;
}
```

OUTPUT

```
10 -> 20 -> 30 -> (head)
```

Program 4

Statement:

Demonstrates insertion and display operations in a circular double linked list.


```
#include <iostream>

using namespace std;

struct Node {
    int data;
    Node* prev;
    Node* next;
};

class CircularDoubleLinkedList {
    Node* head;

public:
    CircularDoubleLinkedList() : head(nullptr) {}

    void insertAtEnd(int val) {
        Node* newNode = new Node{val, nullptr, nullptr};
        if (!head) {
            head = newNode;
            head->next = head;
            head->prev = head;
            return;
        }
        Node* tail = head->prev;
        tail->next = newNode;
        newNode->prev = tail;
        newNode->next = head;
    }
};
```

```
        head->prev = newNode;
    }

    void display() {
        if (!head) {
            cout << "List is empty\n";
            return;
        }
        Node* temp = head;
        do {
            cout << temp->data << " <-> ";
            temp = temp->next;
        } while (temp != head);
        cout << "(head)\n";
    }
};

int main() {
    CircularDoubleLinkedList list;
    list.insertAtEnd(10);
    list.insertAtEnd(20);
    list.insertAtEnd(30);
    list.display();
    return 0;
}
```

OUTPUT

```
10 <-> 20 <-> 30 <-> (head)
```

Program 5

Statement:

Demonstrates reversing a single linked list using iteration.

```
#include <iostream>
```

```
using namespace std;
```

```
struct Node {
```

```
    int data;
```

```
    Node* next;
```

```
};
```

```
class SingleLinkedList {
```

```
    Node* head;
```

```
public:
```

```
    SingleLinkedList() : head(nullptr) {}
```

```
    void insertAtEnd(int val) {
```

```
        Node* newNode = new Node{val, nullptr};
```

```
        if (!head) {
```

```
            head = newNode;
```

```
            return;
```

```
        }
```

```
        Node* temp = head;
```

```
        while (temp->next) temp = temp->next;
```

```
        temp->next = newNode;
```

```
    }
```

```
void reverse() {
    Node* prev = nullptr;
    Node* current = head;
    Node* next = nullptr;

    while (current) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    head = prev;
}

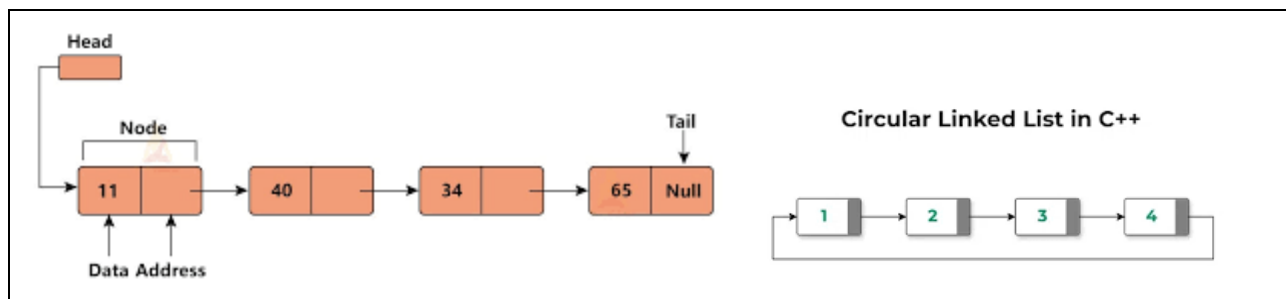
void display() {
    if (!head) {
        cout << "List is empty\n";
        return;
    }
    Node* temp = head;
    while (temp) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL\n";
}
};
```

```
int main() {  
    SingleLinkedList list;  
    list.insertAtEnd(10);  
    list.insertAtEnd(20);  
    list.insertAtEnd(30);  
    list.display();  
    list.reverse();  
    list.display();  
    return 0;  
}
```

OUTPUT

```
10 -> 20 -> 30 -> NULL  
30 -> 20 -> 10 -> NULL
```

DIAGRAMS



LAB 6

TREES

Introduction:

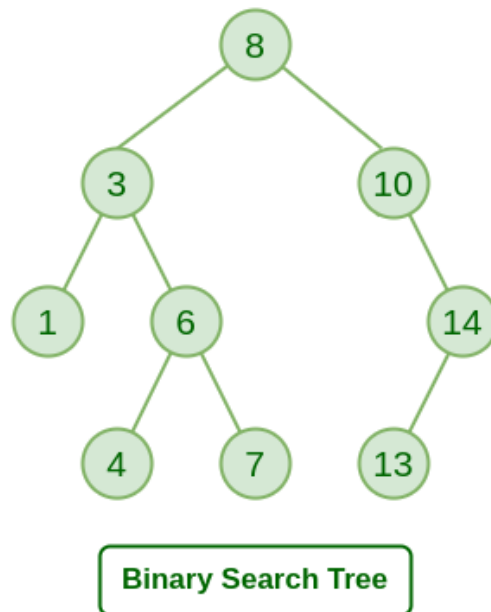
Trees are hierarchical data structures where elements are organized in a tree-like structure with a root node and sub-nodes. A tree is a collection of nodes connected by edges. It starts with a root node and branches into child nodes.

Key Concepts in Trees:

Binary Tree: Each node has at most two children (left and right).

Binary Search Tree (BST): A binary tree where the left subtree contains nodes smaller than the root, and the right subtree contains larger nodes.

Tree Traversals: Methods to visit all nodes, such as preorder, inorder, and postorder.



Program 1

Statement:

Demonstrates insertion into a binary tree and a recursive preorder traversal.

```
#include <iostream>

using namespace std;

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

    Node(int value) : data(value), left(nullptr), right(nullptr) {}
};

class BinaryTree {
    Node* root;

public:
    BinaryTree() : root(nullptr) {}
    Node* getRoot() { return root; }
    void insert(int value) {
        if (!root) {
            root = new Node(value);
            return;
        }
        insertHelper(root, value);
    }
};
```

```
void insertHelper(Node* node, int value) {
    if (value < node->data) {
        if (!node->left)
            node->left = new Node(value);
        else
            insertHelper(node->left, value);
    } else {
        if (!node->right)
            node->right = new Node(value);
        else
            insertHelper(node->right, value);
    }
}

void preorderTraversal(Node* node) {
    if (!node) return;

    cout << node->data << " ";
    preorderTraversal(node->left);
    preorderTraversal(node->right);
}

};

int main() {
    BinaryTree tree;
    tree.insert(10);
    tree.insert(5);
    tree.insert(15);
    tree.insert(3);
    cout << "Preorder Traversal: ";
```



```
tree.preorderTraversal(tree.getRoot());  
cout << endl;  
return 0;  
}
```

OUTPUT

Preorder Traversal: 10 5 3 15

Program 2

Statement:

Demonstrates searching for a value in a binary search tree (BST).

```
#include <iostream>  
using namespace std;  
  
struct Node {  
    int data;  
    Node* left;  
    Node* right;  
  
    Node(int value) : data(value), left(nullptr), right(nullptr) {}  
};  
  
class BST {  
    Node* root;
```

public:

```
BST() : root(nullptr) {}
```

```
void insert(int value) {  
    root = insertHelper(root, value);  
}
```

```
Node* insertHelper(Node* node, int value) {  
    if (!node) return new Node(value);  
  
    if (value < node->data)  
        node->left = insertHelper(node->left, value);  
    else  
        node->right = insertHelper(node->right, value);  
  
    return node;  
}
```

```
bool search(int value) {  
    return searchHelper(root, value);  
}  
  
bool searchHelper(Node* node, int value) {  
    if (!node) return false;  
    if (node->data == value)  
        return true;  
    if (value < node->data)  
        return searchHelper(node->left, value);  
    else
```

```
        return searchHelper(node->right, value);
    }
};

int main() {
    BST tree;
    tree.insert(20);
    tree.insert(10);
    tree.insert(30);
    tree.insert(5);

    cout << "Search 10: " << (tree.search(10) ? "Found" : "Not Found") << endl;
    cout << "Search 25: " << (tree.search(25) ? "Found" : "Not Found") << endl;

    return 0;
}
```

OUTPUT

```
Search 10: Found
Search 25: Not Found
```

Program 3

Statement :

Demonstrates level-order traversal using a queue.

```
#include <iostream>
```

```
#include <queue>
```

```
using namespace std;
```

```
struct Node {  
    int data;  
    Node* left;  
    Node* right;  
  
    Node(int value) : data(value), left(nullptr), right(nullptr) {}  
};
```

```
class BinaryTree {  
    Node* root;  
  
public:  
    BinaryTree() : root(nullptr) {}  
    Node* getRoot() { return root; }  
    void insert(int value) {  
        if (!root) {  
            root = new Node(value);  
            return;  
        }  
        insertHelper(root, value);  
    }
```

```
    void insertHelper(Node* node, int value) {  
        if (value < node->data) {  
            if (!node->left)  
                node->left = new Node(value);  
            else
```

```
        insertHelper(node->left, value);
    } else {
        if (!node->right)
            node->right = new Node(value);
        else
            insertHelper(node->right, value);
    }
}

void levelOrderTraversal() {
    if (!root) {
        cout << "Tree is empty\n";
        return;
    }

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

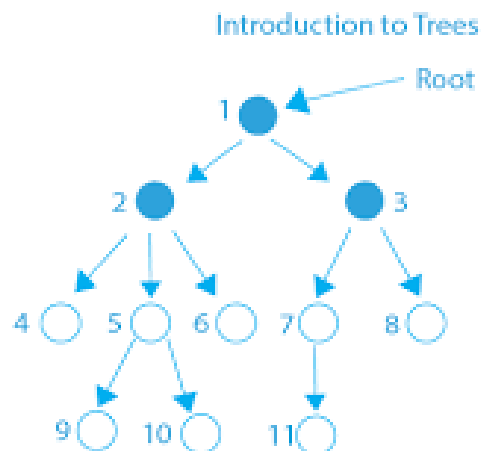
    while (!q.empty()) {
        Node* current = q.front();
        q.pop();
        cout << current->data << " ";
        if (current->left) q.push(current->left);
        if (current->right) q.push(current->right);
    }
    cout << endl;
}
};
```

```
int main() {  
    BinaryTree tree;  
    tree.insert(10);  
    tree.insert(5);  
    tree.insert(15);  
    tree.insert(3);  
    cout << "Level Order Traversal: ";  
    tree.levelOrderTraversal();  
    return 0;  
}
```

OUTPUT

Level Order Traversal: 10 5 15 3

DIAGRAM:



LAB 7

VECTORS

Introduction:

Vectors are dynamic arrays provided by the C++ Standard Template Library (STL). Unlike arrays, they can resize automatically.

Key Features of Vectors:

- Dynamic resizing.
- Random access through indexing.
- Rich set of functions for manipulation (e.g., sort, search, insert).

Applications of Vectors:

- Managing dynamic datasets where size varies.
- Simplifying operations on arrays with STL functions.

Program 1

Statement :

Demonstrates initialization, insertion, access, and iteration through a vector.

```
#include <iostream>

#include <vector>

using namespace std;

int main() {

    // Initializing a vector

    vector<int> vec;

    // Adding elements to the vector

    vec.push_back(10);

    vec.push_back(20);
```

```
vec.push_back(30);

// Accessing elements
cout << "Vector elements: ";
for (int i = 0; i < vec.size(); i++) {
    cout << vec[i] << " ";
}
cout << endl;

// Removing the last element
vec.pop_back();

// Iterating using range-based for loop
cout << "After pop_back: ";
for (int val : vec) {
    cout << val << " ";
}
cout << endl;

return 0;
}
```

OUTPUT

```
Vector elements: 10 20 30
After pop_back: 10 20
```


Program 2

Statement :

Demonstrates sorting a vector and searching for an element.

```
#include <iostream>

#include <vector>

#include <algorithm> // For sort and binary_search
using namespace std;

int main() {
    // Initializing a vector
    vector<int> vec = {40, 10, 20, 30};

    // Sorting the vector
    sort(vec.begin(), vec.end());

    cout << "Sorted vector: ";
    for (int val : vec) {
        cout << val << " ";
    }
    cout << endl;

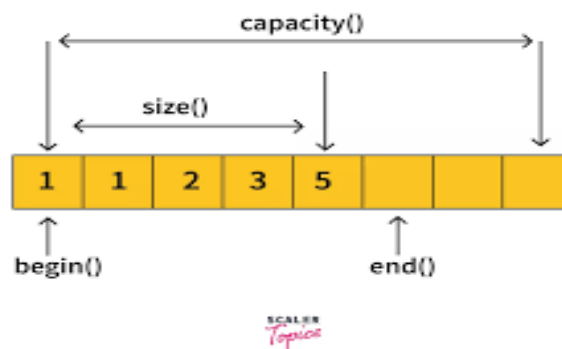
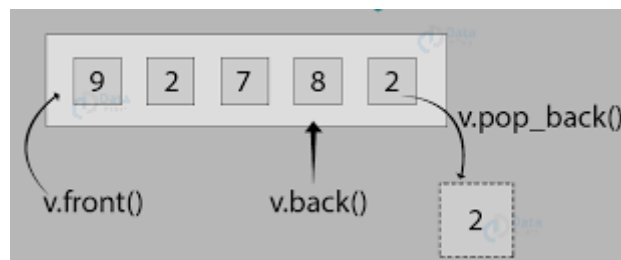
    // Searching for an element
    int key = 20;
    if (binary_search(vec.begin(), vec.end(), key)) {
        cout << "Element " << key << " found in the vector.\n";
    } else {
        cout << "Element " << key << " not found in the vector.\n";
    }
}
```

```
}  
  
return 0;  
}
```

OUTPUT

```
Sorted vector: 10 20 30 40  
Element 20 found in the vector.
```

DIAGRAM :



LAB 8

Binary Search Tree (BST)

Introduction:

A Binary Search Tree (BST) is a hierarchical data structure where each node has at most two children, and:

The left child contains values less than the parent node.

The right child contains values greater than the parent node.

We will adapt the stack-like operations (push, pop, etc.) to work with BST.

Program 1

Statement :

Basic BST Implementation

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

class BSTNode {
public:
    int value;
    BSTNode* left;
    BSTNode* right;

    BSTNode(int val) : value(val), left(nullptr), right(nullptr) {}
};
```

```
class BST {
private:
    BSTNode* root;

    BSTNode* insert(BSTNode* node, int value) {
        if (node == nullptr) {
            return new BSTNode(value);
        }
        if (value < node->value) {
            node->left = insert(node->left, value);
        } else if (value > node->value) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    void inorderTraversal(BSTNode* node) {
        if (node != nullptr) {
            inorderTraversal(node->left);
            cout << node->value << " ";
            inorderTraversal(node->right);
        }
    }

public:
    BST() : root(nullptr) {}

    void push(int value) {
```

```
        root = insert(root, value);
    }

    void display() {
        cout << "BST elements (In-order Traversal): ";
        inorderTraversal(root);
        cout << endl;
    }
};

int main() {
    BST bst;
    bst.push(30);
    bst.push(20);
    bst.push(40);
    bst.display();
    return 0;
}
```

Output:

BST elements (In-order Traversal): 20 30 40

Program 2

Statement :

Peek Operation in BST

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

```
class BSTNode {
```

```
public:
    int value;
    BSTNode* left;
    BSTNode* right;

    BSTNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

class BST {
private:
    BSTNode* root;

    BSTNode* insert(BSTNode* node, int value) {
        if (node == nullptr) {
            return new BSTNode(value);
        }
        if (value < node->value) {
            node->left = insert(node->left, value);
        } else if (value > node->value) {
            node->right = insert(node->right, value);
        }
        return node;
    }

    BSTNode* findMax(BSTNode* node) {
        while (node && node->right != nullptr) {
            node = node->right;
        }
    }
};
```

```
        return node;
    }

public:
    BST() : root(nullptr) {}

    void push(int value) {
        root = insert(root, value);
    }

    int peek() {
        BSTNode* maxNode = findMax(root);
        return maxNode ? maxNode->value : -1;
    }
};

int main() {
    BST bst;
    bst.push(10);
    bst.push(20);
    bst.push(30);
    cout << "Peek (Max element): " << bst.peek() << endl;
    return 0;
}
```

Output:

Peek (Max element): 30

Program 3**Statemen****Checking if BST is Empty**

```
#include <iostream>

using namespace std;

class BSTNode {
public:
    int value;
    BSTNode* left;
    BSTNode* right;

    BSTNode(int val) : value(val), left(nullptr), right(nullptr) {}
};

class BST {
private:
    BSTNode* root;

public:
    BST() : root(nullptr) {}

    bool isEmpty() {
        return root == nullptr;
    }

    void push(int value) {
        root = new BSTNode(value);
    }
}
```



```
    }  
};  
  
int main() {  
    BST bst;  
    cout << "Is BST empty? " << (bst.isEmpty() ? "Yes" : "No") << endl;  
    bst.push(50);  
    cout << "Is BST empty? " << (bst.isEmpty() ? "Yes" : "No") << endl;  
    return 0;  
}
```

Output:

Is BST empty? Yes

Is BST empty? No

Program 4: Check BST Fullness

cpp

Copy code

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

```
class BSTNode {  
public:  
    int value;  
    BSTNode* left;  
    BSTNode* right;  
  
    BSTNode(int val) : value(val), left(nullptr), right(nullptr) {}  
};
```

```
class BST {  
private:  
    BSTNode* root;  
  
public:  
    BST() : root(nullptr) {}  
  
    bool isFull() {  
        return false; // BSTs are dynamic; they don't get "full" like arrays.  
    }  
  
    void push(int value) {  
        if (isFull()) {  
            cout << "BST is full!" << endl;  
        } else {  
            root = new BSTNode(value);  
        }  
    }  
};  
  
int main() {  
    BST bst;  
    bst.push(10);  
    cout << "BST doesn't have a full state." << endl;  
    return 0;  
}
```

DIAGRAM :

Binary Search Tree

