STACK – Data Structure

A **Stack** is a **linear data structure** that follows the **LIFO** principle:
LIFO → Last In, First Out
The element inserted last is removed first.

Real-Life Examples of Stack

- A pile of plates
- Books stacked on top of each other
- Undo/Redo operations in software
- Browser back button history

Major Operations of Stack

Operation	Meaning
push(x)	Insert an element on top
pop()	Remove and return the top element
peek() / top()	View (not remove) the top element
isEmpty()	Check if the stack is empty
isFull() *(in static stack)*	Check if stack is full
size()	Number of elements in stack

Why Use Stack? (Advantages)

- Easy to implement
- Efficient: **O(1)** insertion and deletion
- Used for **function calls**, **recursion**, **expression evaluation**, **backtracking**, etc.

Limitations of Stack

- Limited size (in static array implementation)
- Can only access the **top** element
- Not suitable for random access

Stack Implementation Approaches

(A) Using Array (Static Stack)

- Fixed size
- Fast operations
- Used in simple programs

(B) Using Linked List (Dynamic Stack)

- No fixed size
- Grows dynamically

- Used when size is unknown

Time Complexity

Operation	Time
push()	O(1)
pop()	O(1)
peek()	O(1)
isEmpty()	O(1)

Applications of Stack

Stack is used in many core areas:

1. Expression Evaluation

- Infix → Prefix → Postfix conversion
- Postfix evaluation

2. Function Calls

- Call stack
- Recursion stack

3. Backtracking

- Maze solving
- Undo operations

4. Symbol Balancing

- Parentheses matching
  Example: ((a+b) * (c+d))

5. Browser Navigation

- Back/Forward history

Array-Based Stack Code (Most Used in Interviews)

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

class Stack {
private:
    int top;
    int capacity;
    int *arr;
```

```
public:
    Stack(int size) {
        capacity = size;
        arr = new int[capacity];
        top = -1;
    }

    void push(int x) {
        if (top == capacity - 1) {
            cout << "Stack Overflow!" << endl;
            return;
        }
        arr[++top] = x;
    }

    int pop() {
        if (top == -1) {
            cout << "Stack Underflow!" << endl;
            return -1;
        }
        return arr[top--];
    }

    int peek() {
        if (top == -1) {
            cout << "Stack is Empty!" << endl;
            return -1;
        }
        return arr[top];
    }

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

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

int main() {
    Stack s(5);

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

    cout << "Top element: " << s.peek() << endl;

    cout << "Popped: " << s.pop() << endl;

    cout << "Current size: " << s.size() << endl;

    return 0;
}
```

Important Interview Questions on Stack

1. Implement a stack using array
2. Implement a stack using linked list

3. Check balanced parentheses
4. Implement Min Stack
5. Evaluate postfix expression
6. Convert infix to postfix
7. Reverse a string using stack
8. Design a stack with push, pop, top in O(1)

What is Recursion?

Recursion is a programming technique where a function **calls itself** to solve a smaller part of the problem.

A recursive function has:

1. **Base Case** → The condition where recursion stops.
2. **Recursive Case** → Function calls itself with smaller input.

Why Use Recursion?

- Reduces complex problems into smaller subproblems
- Clean and simple logic
- Useful for tree and graph problems
- Ideal for mathematical computations

How Recursion Works?

Recursion uses **function call stack**.

Each function call is stored temporarily, and when the base case is reached, the stack unwinds.

Example:
`fact(5)` → calls `fact(4)` → `fact(3)` → `fact(2)` → `fact(1)` → return.

Example: Factorial Using Recursion

```
int factorial(int n) {
    if (n == 0) return 1;  // Base case
    return n * factorial(n - 1); // Recursive call
}
```

Example: Fibonacci Using Recursion

```
int fib(int n) {
    if (n <= 1) return n;  // Base case
    return fib(n - 1) + fib(n - 2); // Recursive calls
}
```

Tail Recursion

When the **recursive call is the last step** in the function.

```
int tailFact(int n, int result = 1) {
    if (n == 0) return result;
    return tailFact(n - 1, result * n);
}
```

Applications of Recursion

Mathematical Computations

Factorial
Fibonacci
GCD / LCM
Power calculation

Example — Power:

```
int power(int a, int b) {
    if (b == 0) return 1;
    return a * power(a, b - 1);
}
```

Recursion in Arrays

a. Print array elements

```
void printArray(int arr[], int i, int n) {
    if (i == n) return;
    cout << arr[i] << " ";
    printArray(arr, i + 1, n);
}
```

b. Search element (Linear Search)

```
bool search(int arr[], int n, int target, int i) {
    if (i == n) return false;
    if (arr[i] == target) return true;
    return search(arr, n, target, i + 1);
}
```

Recursion in Strings

Reverse a string:

```
string reverseStr(string s) {
    if (s.length() <= 1) return s;
    return reverseStr(s.substr(1)) + s[0];
}
```

Advantages of Recursion

Clean & easy to understand
Reduces complex problem size
Better for tree/graph
Ideal for divide & conquer

Disadvantages of Recursion

More memory usage (stack calls)
Slower than loops for simple tasks
Risk of stack overflow without base case

When to Use Recursion?

Use recursion when:

- Problem can be divided into smaller problems
- Tree/graph traversal needed
- Backtracking required
- Divide & Conquer algorithm used

Avoid recursion when:

- Simple loops can solve quickly
- Memory is limited

QUEUE – Data Structure

A **Queue** is a linear data structure that works on the **FIFO principle**:

FIFO → First In, First Out

Example:

- People standing in a line
- Print queue
- Task scheduling

Major Queue Operations

Operation	Meaning
enqueue(x)	Insert item at rear
dequeue()	Remove item from front
peek() / front()	Get first element
isEmpty()	Check if queue is empty
isFull()	(For array-based queue)

Operation	Meaning
display()	Show queue

Applications of Queue

- CPU scheduling
- BFS (Breadth-First Search in graphs)
- Printers / I/O buffering
- Task scheduling systems
- Call center systems

QUEUE IMPLEMENTATION USING ARRAY

Issues with simple array

- After some dequeues, empty space at front cannot be reused.
- This leads to **wastage of space**.

Therefore, **Circular Queue** is used in real scenarios — but for basic queue we use simple implementation.

Code: Queue Using Array

(Includes enqueue, dequeue, peek, isEmpty, isFull, display)

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

class Queue {
private:
    int *arr;
    int front;
    int rear;
    int capacity;

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

    void enqueue(int value) {
        if (rear == capacity - 1) {
            cout << "Queue Overflow!" << endl;
            return;
        }

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

        arr[++rear] = value;
    }
```

```
    int dequeue() {
        if (front == -1 || front > rear) {
            cout << "Queue Underflow!" << endl;
            return -1;
        }
        return arr[front++];
    }

    int peek() {
        if (front == -1 || front > rear) {
            cout << "Queue is Empty!" << endl;
            return -1;
        }
        return arr[front];
    }

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

    bool isFull() {
        return rear == capacity - 1;
    }

    void display() {
        if (isEmpty()) {
            cout << "Queue is Empty!" << endl;
            return;
        }

        cout << "Queue: ";
        for (int i = front; i <= rear; i++)
            cout << arr[i] << " ";
        cout << endl;
    }
};
```

QUEUE IMPLEMENTATION USING LINKED LIST

Why use Linked List?

- No size limit
- Dynamic memory allocation
- No overflow (unless memory full)

Code: Queue Using Linked List

(Includes enqueue, dequeue, peek, isEmpty, display)

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

class Node {
public:
    int data;
    Node* next;

    Node(int value) {
```

```
        data = value;
        next = nullptr;
    }
};

class LinkedListQueue {
private:
    Node* front;
    Node* rear;

public:
    LinkedListQueue() {
        front = rear = nullptr;
    }

    // Insert at rear
    void enqueue(int value) {
        Node* newNode = new Node(value);

        if (rear == nullptr) {
            front = rear = newNode;
            return;
        }

        rear->next = newNode;
        rear = newNode;
    }

    // Remove from front
    int dequeue() {
        if (front == nullptr) {
            cout << "Queue Underflow!" << endl;
            return -1;
        }

        Node* temp = front;
        int value = temp->data;

        front = front->next;

        if (front == nullptr)
            rear = nullptr;

        delete temp;
        return value;
    }

    int peek() {
        if (front == nullptr) {
            cout << "Queue is Empty!" << endl;
            return -1;
        }
        return front->data;
    }

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

    void display() {
        if (front == nullptr) {
```

```
            cout << "Queue is Empty!" << endl;
            return;
        }

        Node* temp = front;

        cout << "Queue: ";
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }

        cout << endl;
    }
};
```

Time Complexity

Operation	Array	Linked List
enqueue	O(1)	O(1)
dequeue	O(1)	O(1)
peek	O(1)	O(1)
isEmpty	O(1)	O(1)

DEQUE (Double-Ended Queue)

A **Deque (Double-Ended Queue)** is a linear data structure where **insertion and deletion can be performed from both ends**:

- **Front**
- **Rear**

Deque = “Double Ended Queue”

Types of Deque

1. **Input-Restricted Deque**
   - Insertions only at rear
   - Deletion at both ends
2. **Output-Restricted Deque**
   - Deletions only at front
   - Insertion at both ends

Applications of Deque

- Sliding window maximum/minimum
- Browser history
- Implementing stacks and queues
- Palindrome checking
- Job scheduling

- Undo/Redo operations

Operations of Deque

Operation	Meaning
insertFront(x)	Insert at front
insertRear(x)	Insert at rear
deleteFront()	Delete from front
deleteRear()	Delete from rear
getFront()	Show front element
getRear()	Show rear element
isEmpty()	Check queue empty
isFull()	(In array version)
display()	Print deque

DEQUE Using Array (Circular Array Implementation)

Why circular array?

To avoid space wastage after deletions.

C++ Code: Deque Using Circular Array

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

class Deque {
private:
    int *arr;
    int front, rear, size;

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

    // Check full
    bool isFull() {
        return (front == 0 && rear == size - 1) || (front == rear + 1);
    }

    // Check empty
    bool isEmpty() {
        return front == -1;
    }

    // Insert at front
    void insertFront(int x) {
        if (isFull()) {
```

```
        cout << "Deque Overflow!" << endl;
        return;
    }

    if (front == -1) { // empty
        front = rear = 0;
    }
    else if (front == 0) {
        front = size - 1;
    }
    else {
        front--;
    }

    arr[front] = x;
}

// Insert at rear
void insertRear(int x) {
    if (isFull()) {
        cout << "Deque Overflow!" << endl;
        return;
    }

    if (front == -1) {
        front = rear = 0;
    }
    else if (rear == size - 1) {
        rear = 0;
    }
    else {
        rear++;
    }

    arr[rear] = x;
}

// Delete at front
void deleteFront() {
    if (isEmpty()) {
        cout << "Deque Underflow!" << endl;
        return;
    }

    if (front == rear) {
        front = rear = -1;
    }
    else if (front == size - 1) {
        front = 0;
    }
    else {
        front++;
    }
}

// Delete at rear
void deleteRear() {
    if (isEmpty()) {
        cout << "Deque Underflow!" << endl;
        return;
    }
```

```
        if (front == rear) {
            front = rear = -1;
        }
        else if (rear == 0) {
            rear = size - 1;
        }
        else {
            rear--;
        }
    }

    int getFront() {
        if (isEmpty()) {
            cout << "Deque is Empty!" << endl;
            return -1;
        }
        return arr[front];
    }

    int getRear() {
        if (isEmpty()) {
            cout << "Deque is Empty!" << endl;
            return -1;
        }
        return arr[rear];
    }

    // Display deque
    void display() {
        if (isEmpty()) {
            cout << "Deque is Empty!" << endl;
            return;
        }

        cout << "Deque: ";
        int i = front;

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

DEQUE Using Doubly Linked List

- No size limit
- Simpler insert/delete operations
- No overflow

C++ Code: Deque Using Doubly Linked List

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

class Node {
```

```cpp
public:
    int data;
    Node* next;
    Node* prev;

    Node(int value) {
        data = value;
        next = prev = nullptr;
    }
};

class LinkedListDeque {
private:
    Node* front;
    Node* rear;

public:
    LinkedListDeque() {
        front = rear = nullptr;
    }

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

    // Insert at front
    void insertFront(int value) {
        Node* newNode = new Node(value);

        if (isEmpty()) {
            front = rear = newNode;
        }
        else {
            newNode->next = front;
            front->prev = newNode;
            front = newNode;
        }
    }

    // Insert at rear
    void insertRear(int value) {
        Node* newNode = new Node(value);

        if (isEmpty()) {
            front = rear = newNode;
        }
        else {
            rear->next = newNode;
            newNode->prev = rear;
            rear = newNode;
        }
    }

    // Delete front
    void deleteFront() {
        if (isEmpty()) {
            cout << "Deque Underflow!" << endl;
            return;
        }

        Node* temp = front;
```

```
        if (front == rear) {
            front = rear = nullptr;
        }
        else {
            front = front->next;
            front->prev = nullptr;
        }

        delete temp;
    }

    // Delete rear
    void deleteRear() {
        if (isEmpty()) {
            cout << "Deque Underflow!" << endl;
            return;
        }

        Node* temp = rear;

        if (front == rear) {
            front = rear = nullptr;
        }
        else {
            rear = rear->prev;
            rear->next = nullptr;
        }

        delete temp;
    }

    int getFront() {
        if (isEmpty()) return -1;
        return front->data;
    }

    int getRear() {
        if (isEmpty()) return -1;
        return rear->data;
    }

    // Display
    void display() {
        if (isEmpty()) {
            cout << "Deque is Empty!" << endl;
            return;
        }

        cout << "Deque: ";
        Node* temp = front;
        while (temp != nullptr) {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
```

Time Complexity

Operation	Array	Linked List
insertFront	O(1)	O(1)
insertRear	O(1)	O(1)
deleteFront	O(1)	O(1)
deleteRear	O(1)	O(1)
getFront	O(1)	O(1)
getRear	O(1)	O(1)