UNIT-III

1. Stacks

1.1 Definition of Stack

A **stack** is a linear data structure that follows the **Last In, First Out (LIFO)** principle. In this structure, the last element added is the first one to be removed. A stack is often visualized as a vertical pile of elements where insertion and deletion occur at the top of the stack.

Features of Stack:

- **LIFO Order:** Last element added is the first one removed.
- **Operations at One End:** All operations are performed at the top of the stack.
- **Dynamic Nature:** Can dynamically grow or shrink in linked list implementations.

1.2 Representation of Stacks

1.2.1 Array Representation

A stack can be implemented using an array where:

- A fixed size array holds the elements.
- A variable (`top`) keeps track of the last inserted element.

Example Code (C-like):

```c
#define MAX 100
int stack[MAX];
int top = -1;

void push(int value) {
    if (top == MAX - 1) {
        printf("Stack Overflow\n");
    } else {
        stack[++top] = value;
    }
}

int pop() {
    if (top == -1) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        return stack[top--];
    }
}
```

1.2.2 Linked List Representation

In a linked list implementation:

- Each node contains the data and a pointer to the next node.
- The top of the stack points to the last inserted node.

Example Code (C-like):

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

Node* top = NULL;

void push(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = top;
    top = newNode;
}

int pop() {
    if (top == NULL) {
        printf("Stack Underflow\n");
        return -1;
    } else {
        int value = top->data;
        Node* temp = top;
        top = top->next;
        free(temp);
        return value;
    }
}
```

1.3 Stack Operations

1.3.1 Insertion (Push)

Add an element to the top of the stack.

Steps:

1. Check for overflow (in array-based implementation).
2. Increment the `top` pointer.
3. Insert the element at the `top`.

1.3.2 Deletion (Pop)

Remove an element from the top of the stack.

Steps:

1. Check for underflow.
2. Retrieve the element at the `top`.
3. Decrement the `top` pointer.

1.3.3 Traversal

Access and display all elements in the stack.

Steps:

1. Start from the `top` pointer.
2. Iterate backward until the stack is empty.

1.4 Applications of Stack

1.4.1 Recursion

Stacks are used to implement recursion, as they help manage function call states.

1.4.2 Polish Notation

- **Prefix Notation (Polish):** Operators appear before operands (e.g., `+AB`).
- **Postfix Notation (Reverse Polish):** Operators appear after operands (e.g., `AB+`).

Stacks are used to evaluate or convert expressions in these notations.

---

2. Queues

2.1 Definition of Queue

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle. The first element added is the first one removed. Queues are commonly used in scheduling and buffering.

Features of Queue:

- **FIFO Order:** First element inserted is the first to be removed.
- **Two Pointers:** Front and rear pointers track the start and end of the queue.

2.2 Representation of Queues

2.2.1 Array Representation

In an array implementation:

- A fixed size array holds the elements.
- Two pointers (`front` and `rear`) track the start and end of the queue.

Example Code (C-like):

```c
#define MAX 100
int queue[MAX];
int front = -1, rear = -1;

void enqueue(int value) {
    if (rear == MAX - 1) {
        printf("Queue Overflow\n");
    } else {
        if (front == -1) front = 0;
        queue[++rear] = value;
    }
}

int dequeue() {
    if (front == -1 || front > rear) {
        printf("Queue Underflow\n");
        return -1;
    } else {
        return queue[front++];
    }
}
```

2.2.2 Linked List Representation

In a linked list implementation:

- Each node contains the data and a pointer to the next node.
- The `front` points to the first node, and the `rear` points to the last node.

Example Code (C-like):

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

Node* front = NULL;
Node* rear = NULL;

void enqueue(int value) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = value;
    newNode->next = NULL;
    if (rear == NULL) {
        front = rear = newNode;
    } else {
        rear->next = newNode;
        rear = newNode;
    }
}

int dequeue() {
    if (front == NULL) {
        printf("Queue Underflow\n");
        return -1;
    } else {
        int value = front->data;
        Node* temp = front;
        front = front->next;
        free(temp);
        if (front == NULL) rear = NULL;
        return value;
    }
}
```

2.3 Types of Queues

2.3.1 Simple Queue

A basic queue that follows the FIFO principle.

2.3.2 Circular Queue

In a circular queue, the last position is connected to the first, forming a circle.
Advantages: Optimizes memory usage.

2.3.3 Priority Queue

Elements are dequeued based on priority rather than insertion order.

2.3.4 Double-Ended Queue (Deque)

Elements can be added or removed from both ends.

2.4 Operations on Queues

2.4.1 Insertion (Enqueue)

Add an element to the rear of the queue.

2.4.2 Deletion (Dequeue)

Remove an element from the front of the queue.

2.4.3 Traversal

Access and display all elements in the queue.

2.5 Applications of Queues

- **Task Scheduling:** In operating systems for process management.
- **Data Buffering:** Used in input/output buffers.
- **Breadth-First Search (BFS):** Used in graph traversal.

2.6 Operations on Circular Queue

Circular queues are an advanced form of queues where the rear connects back to the front, creating a circle. This allows efficient utilization of space by reusing freed positions.

2.6.1 Insertion in Circular Queue

To insert an element:

1. Check if the queue is full (`(rear + 1) % size == front`).
2. If the queue is not full:
    - Update the `rear` pointer using `(rear + 1) % size` .
    - Add the element at the `rear` .

Example Code (C-like):

```c
#define MAX 5
int queue[MAX];
int front = -1, rear = -1;

void enqueue(int value) {
    if ((rear + 1) % MAX == front) {
        printf("Queue Overflow\n");
    } else {
        if (front == -1) front = 0;
        rear = (rear + 1) % MAX;
        queue[rear] = value;
    }
}
```

2.6.2 Deletion in Circular Queue

To remove an element:

1. Check if the queue is empty (`front == -1`).
2. Retrieve the element at `front` .
3. Update the `front` pointer:
    - If `front == rear` , reset both pointers to `-1` (queue becomes empty).
    - Otherwise, move `front` to `(front + 1) % size` .

Example Code (C-like):

```c
int dequeue() {
    if (front == -1) {
        printf("Queue Underflow\n");
        return -1;
    } else {
        int value = queue[front];
        if (front == rear) {
            front = rear = -1;
        } else {
            front = (front + 1) % MAX;
        }
        return value;
    }
}
```

2.6.3 Traversal in Circular Queue

To traverse a circular queue:

1. Start at `front`.
2. Use `(index + 1) % size` to move through the queue.
3. Stop when the `rear` is reached.

Example Code (C-like):

```
void traverse() {
    if (front == -1) {
        printf("Queue is Empty\n");
    } else {
        int i = front;
        printf("Queue Elements: ");
        while (1) {
            printf("%d ", queue[i]);
            if (i == rear) break;
            i = (i + 1) % MAX;
        }
        printf("\n");
    }
}
```

2.7 Applications of Circular Queues

- **Resource Management:** Used in operating systems for scheduling processes.
- **Buffering:** Efficiently manage circular buffers in data streaming.
- **Paging:** Memory paging in OS uses circular queue structures.

2.8 Comparison of Stack and Queue

Feature	Stack	Queue
Order	LIFO (Last In, First Out)	FIFO (First In, First Out)
Operations Allowed	Push, Pop	Enqueue, Dequeue
Access	Top only	Front and Rear
Use Cases	Function calls, Backtracking	Scheduling, BFS, Resource Mgmt