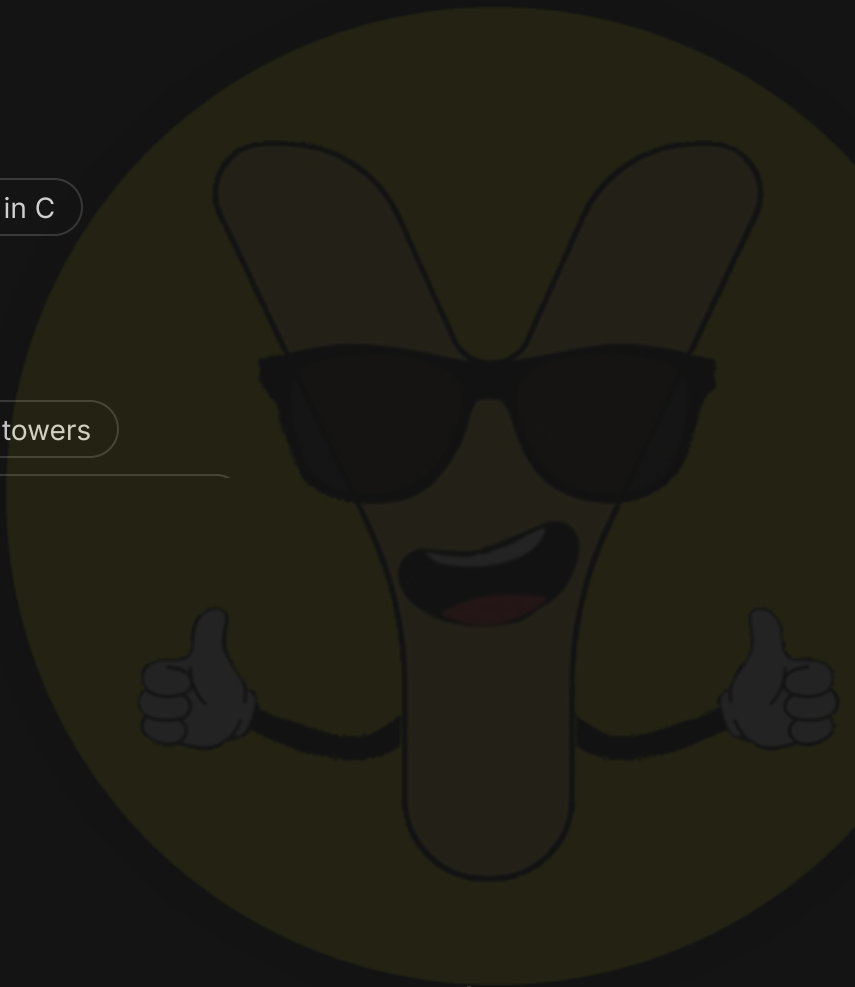
AKTU · 2ND YEAR · SEMESTER 3 · BCS301

Data Structure

Unit 2 — Stacks, Recursion, and Queues

- Stacks: Abstract Data Type, Primitive Stack operations: Push & Pop
- Array and Linked Implementation of Stack in C
- Application of stack: Prefix and Postfix Expressions, Evaluation of postfix expression
- Iteration and Recursion: Principles of recursion, Tail recursion, Removal of recursion
- Problem solving using iteration and recursion with examples such as binary search, Fibonacci numbers, and Hanoi towers



DECODED BY

Chirag Jain

Founder, Ycotes · Gold Medalist in Artificial Intelligence, IIT Madras · taught 10,000+ students

Unit 2 Syllabus Overview

This unit covers linear data structures focused on restricted access and self-referential function calls.

- Stacks: ADT, Push/Pop operations, Array and Linked implementations in C.
- Stack Applications: Prefix, Postfix, and evaluation of postfix expressions.
- Recursion: Principles, Tail recursion, Removal of recursion, and tradeoffs.
- Problem Solving: Binary search, Fibonacci numbers, and Tower of Hanoi.
- Queues: Operations, Circular queues, Array/Linked implementations, Deque, Priority Queue.

Stack Abstract Data Type

STACK

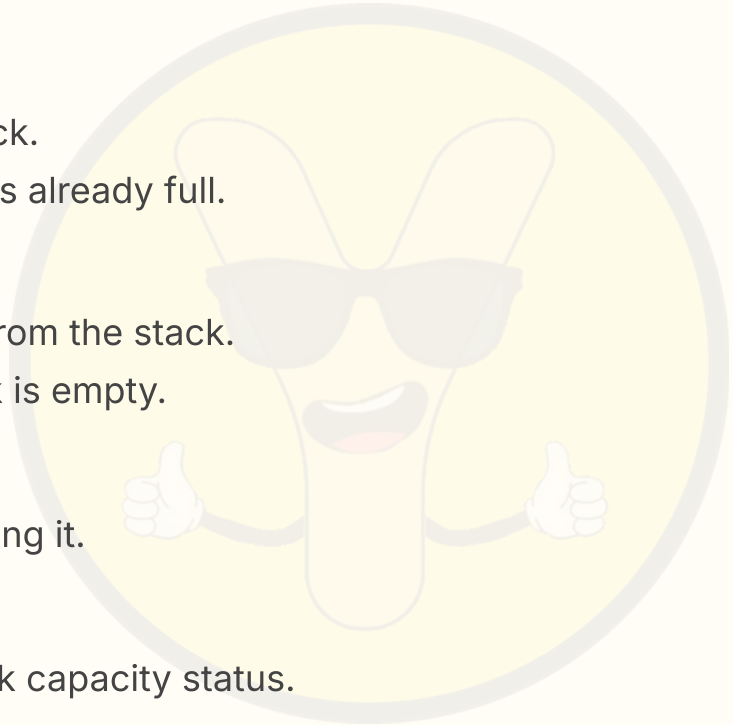
A linear data structure that follows the Last In, First Out (LIFO) principle, where insertion and deletion occur only at one end called the top.

- LIFO order means the last element inserted is the first one to be removed.
- Restricted access structure restricting operations to the top pointer.
- Logical view shows elements stacked vertically.

Primitive Stack Operations

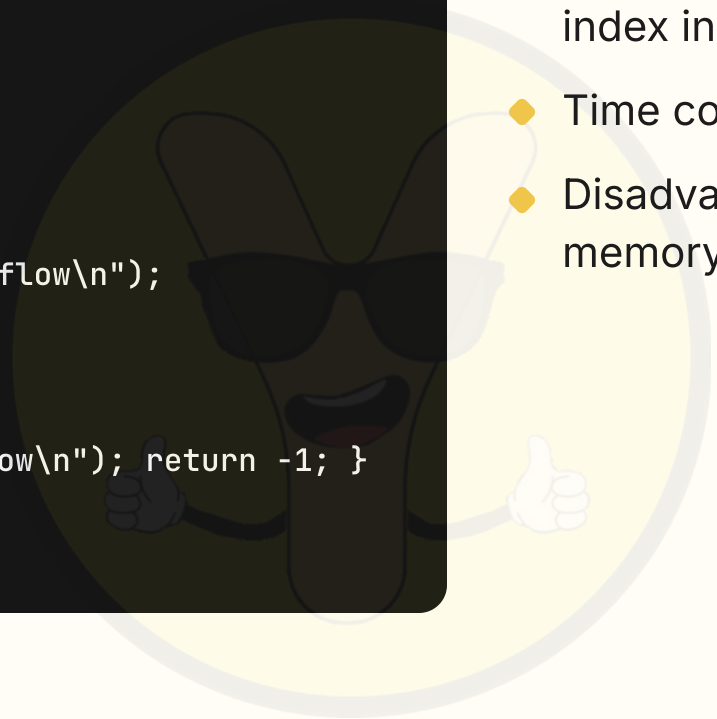
Basic operations supported by every stack implementation.

- Push(item)
  - Inserts an element at the top of the stack.
  - Overflow condition occurs if the stack is already full.
- Pop()
  - Removes and returns the top element from the stack.
  - Underflow condition occurs if the stack is empty.
- Peek() / Top()
  - Returns the top element without removing it.
- IsEmpty() & IsFull()
  - Boolean helper functions to check stack capacity status.



Array Implementation of Stack in C

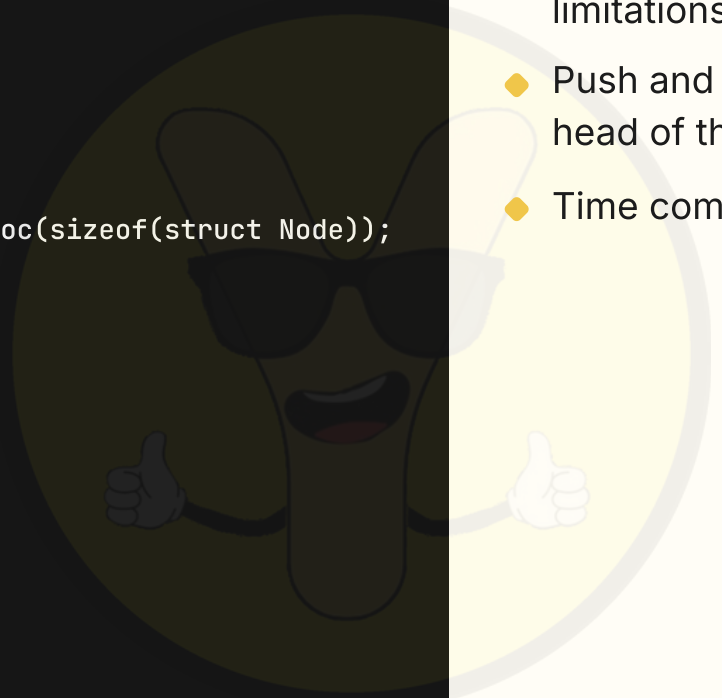
```
#define MAX 100
struct Stack {
    int arr[MAX];
    int top;
};
void push(struct Stack *s, int val) {
    if (s->top == MAX - 1) printf("Overflow\n");
    else s->arr[++(s->top)] = val;
}
int pop(struct Stack *s) {
    if (s->top == -1) { printf("Underflow\n"); return -1; }
    else return s->arr[(s->top)--];
}
```



- Uses a fixed-size array and an integer top index initialized to -1.
- Time complexity of Push and Pop is $O(1)$.
- Disadvantage: Fixed size leading to potential memory wastage or overflow.

Linked List Implementation of Stack in C

```
struct Node {
    int data;
    struct Node* next;
};
struct Node* top = NULL;
void push(int val) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode→data = val;
    newNode→next = top;
    top = newNode;
}
int pop() {
    if (top == NULL) return -1;
    struct Node* temp = top;
    int popped = temp→data;
    top = top→next;
    free(temp);
    return popped;
}
```

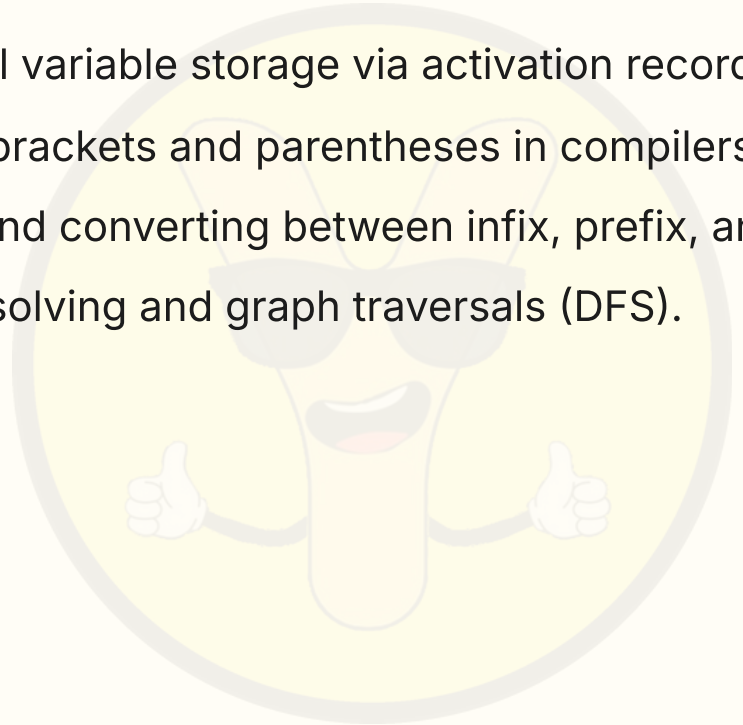


- Dynamic memory allocation removes fixed-size limitations.
- Push and Pop operations insert and delete at the head of the linked list.
- Time complexity for both operations is $O(1)$.

Applications of Stacks

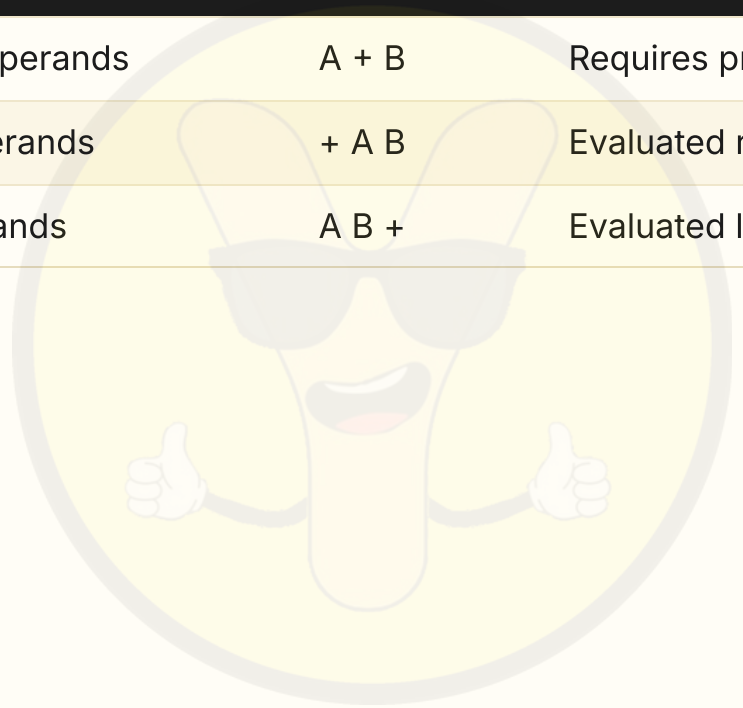
Stacks are extensively used in system programming and expression evaluation.

- Function call management and local variable storage via activation records (call stack).
- Syntax parsing, such as balancing brackets and parentheses in compilers.
- Evaluating arithmetic expressions and converting between infix, prefix, and postfix notations.
- Backtracking algorithms like maze solving and graph traversals (DFS).



Comparison of Expression Notations

Notation	Position of Operator	Example	Evaluation Method
Infix	Between operands	A + B	Requires precedence rules and parentheses
Prefix (Polish)	Before operands	+ A B	Evaluated right to left using a stack
Postfix (Reverse Polish)	After operands	A B +	Evaluated left to right using a stack



Evaluation of Postfix Expression

Evaluate the postfix expression: 5 3 + 6 2 / *

1. Read 5: Push 5. Stack: [5]
2. Read 3: Push 3. Stack: [5, 3]
3. Read +: Pop 3 and 5. Compute 5 + 3 = 8. Push 8. Stack: [8]
4. Read 6: Push 6. Stack: [8, 6]
5. Read 2: Push 2. Stack: [8, 6, 2]
6. Read /: Pop 2 and 6. Compute 6 / 2 = 3. Push 3. Stack: [8, 3]
7. Read *: Pop 3 and 8. Compute 8 * 3 = 24. Push 24. Stack: [24]

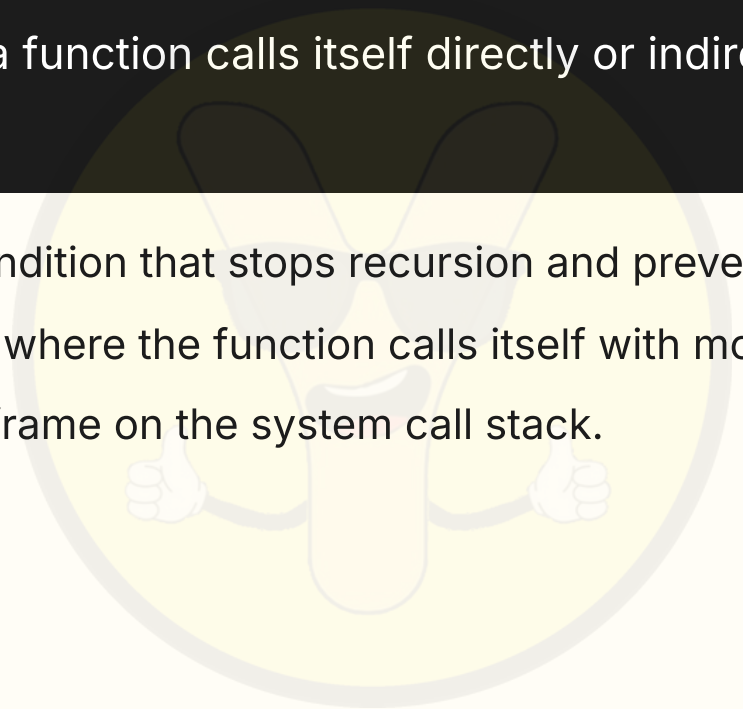
Final Result = 24

Principles of Recursion

RECURSION

A programming technique where a function calls itself directly or indirectly to solve a smaller instance of the same problem.

- Base Condition: The terminating condition that stops recursion and prevents infinite loops.
- Recursive Step: The reduction step where the function calls itself with modified parameters.
- Every recursive call creates a new frame on the system call stack.



Tail Recursion & Removal of Recursion

Understanding optimization and conversion techniques for recursive functions.

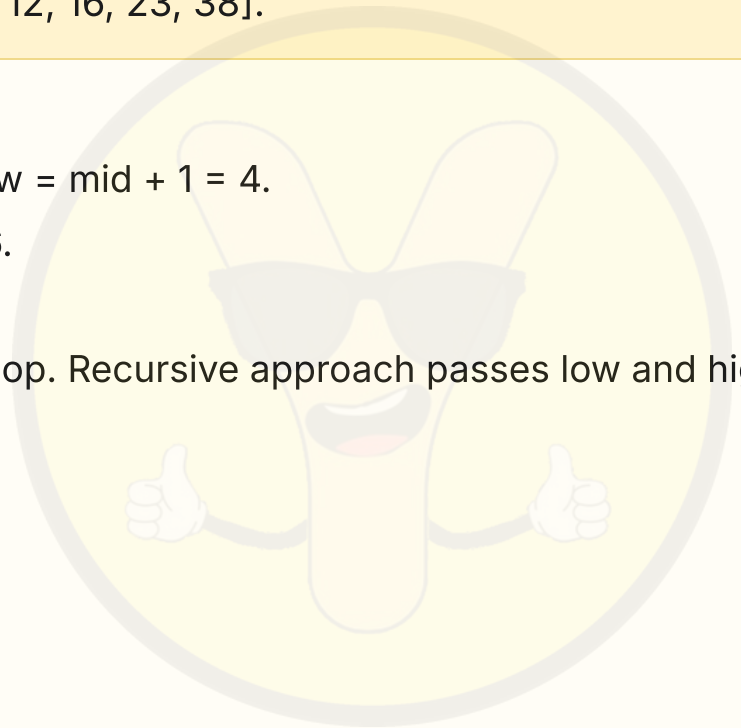
- Tail Recursion
  - A recursive function is tail-recursive if the recursive call is the absolute last operation performed by the function.
  - Compilers can optimize tail recursion (Tail Call Optimization) to reuse stack frames.
- Removal of Recursion
  - Any recursive algorithm can be converted into an iterative algorithm.
  - Requires explicit user-defined stack data structures to simulate system call stacks.
  - Eliminates function call overhead and protects against stack overflow errors.

Binary Search: Iteration vs Recursion

Search target 23 in sorted array [2, 5, 8, 12, 16, 23, 38].

1. Initialize low = 0, high = 6.
2. Mid = (0+6)/2 = 3 (val 12). 23 > 12, so low = mid + 1 = 4.
3. New range [16, 23, 38], low = 4, high = 6.
4. Mid = (4+6)/2 = 5 (val 23). Match found!
5. Iterative approach uses a simple while loop. Recursive approach passes low and high as parameters.

Element found at index 5.



Fibonacci Numbers

Compute the 4th Fibonacci number $F(4)$ using recursion.

1. Base cases: $F(0) = 0$, $F(1) = 1$.
2. Formula: $F(n) = F(n-1) + F(n-2)$.
3. $F(4) = F(3) + F(2)$
4. $F(3) = F(2) + F(1) = (F(1) + F(0)) + 1 = (1 + 0) + 1 = 2$
5. $F(2) = F(1) + F(0) = 1 + 0 = 1$
6. $F(4) = 2 + 1 = 3$

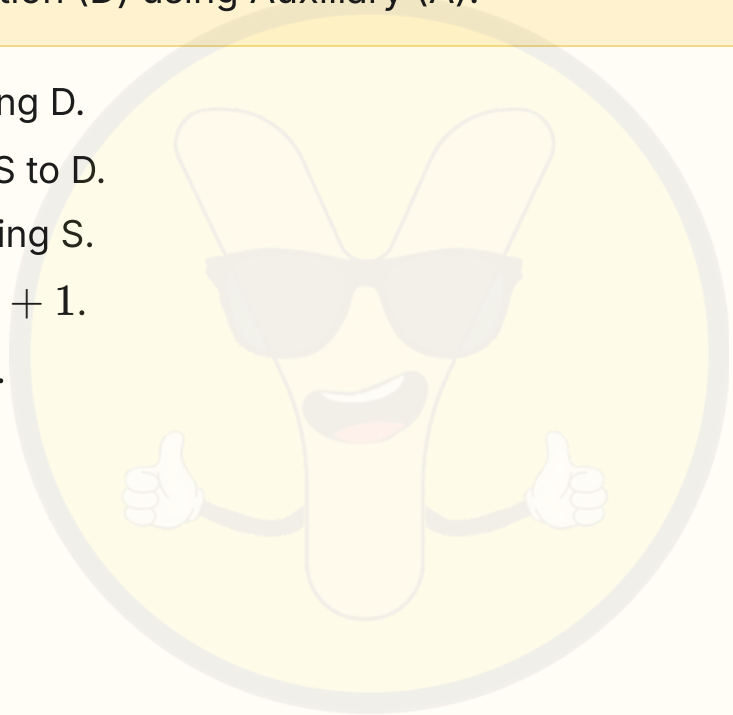
F(4) = 3 (Note: Pure recursion is $O(2^n)$ time due to overlapping subproblems).

Tower of Hanoi Problem

Move n disks from Source (S) to Destination (D) using Auxiliary (A).

1. Step 1: Move $n-1$ disks from S to A using D.
2. Step 2: Move the nth (largest) disk from S to D.
3. Step 3: Move $n-1$ disks from A to D using S.
4. Recurrence relation: $T(n) = 2T(n-1) + 1$.
5. Total moves required for n disks: $2^n - 1$.

Solved in $O(2^n)$ time complexity.



Tradeoffs: Iteration vs Recursion

Parameter	Iteration	Recursion
Speed	Generally faster (no overhead)	Slower due to function call overhead
Memory	Uses constant auxiliary memory $O(1)$	Uses stack memory proportional to depth $O(n)$
Code Length	Can be longer or complex for trees/graphs	Short, clean, and intuitive for hierarchical problems
Risk	No risk of stack overflow	Risk of Stack Overflow if base case is missing

Queue Abstract Data Type

QUEUE

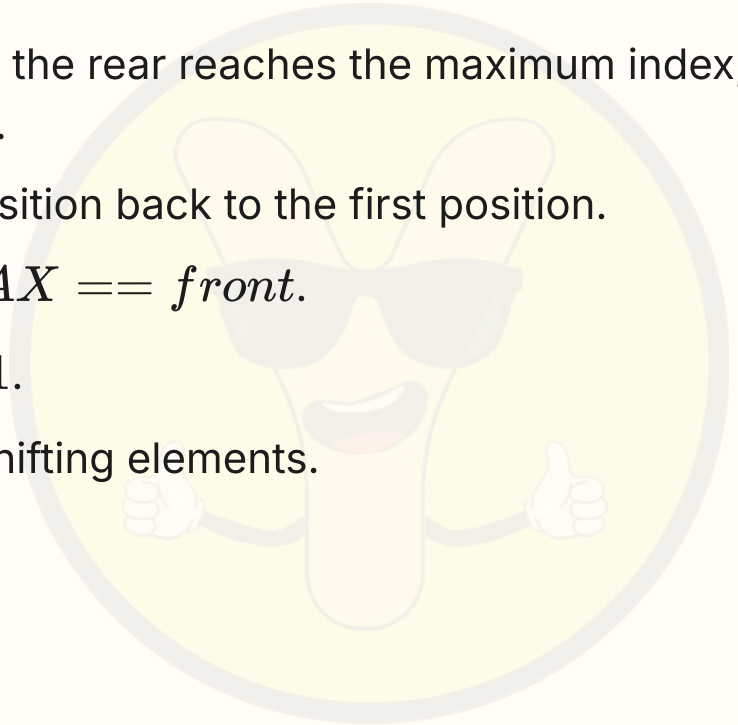
A linear data structure that follows the First In, First Out (FIFO) principle, where insertions happen at one end (rear) and deletions at the other end (front).

- FIFO order means the first element added is the first one removed.
- Analogy: A real-world queue or line of people waiting for service.
- Primary operations: Enqueue (Add), Dequeue (Delete), Create, IsEmpty, IsFull.

Circular Queue

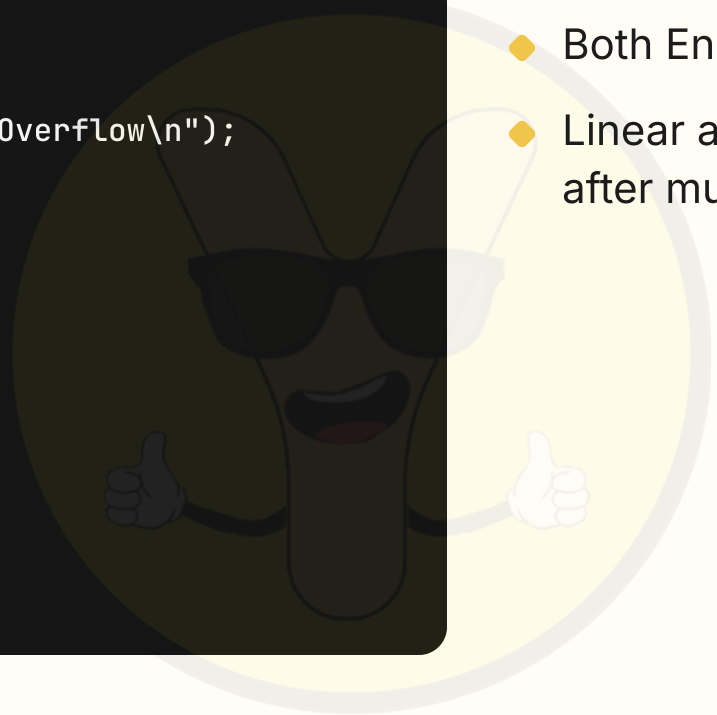
Overcoming the linear queue space limitation.

- In a linear queue using arrays, once the rear reaches the maximum index, no further elements can be added even if front elements were deleted.
- Circular queue connects the last position back to the first position.
- Condition for Full: $(rear + 1)\%MAX == front$.
- Condition for Empty: $front == -1$.
- Optimizes memory usage without shifting elements.



Array Implementation of Queue in C

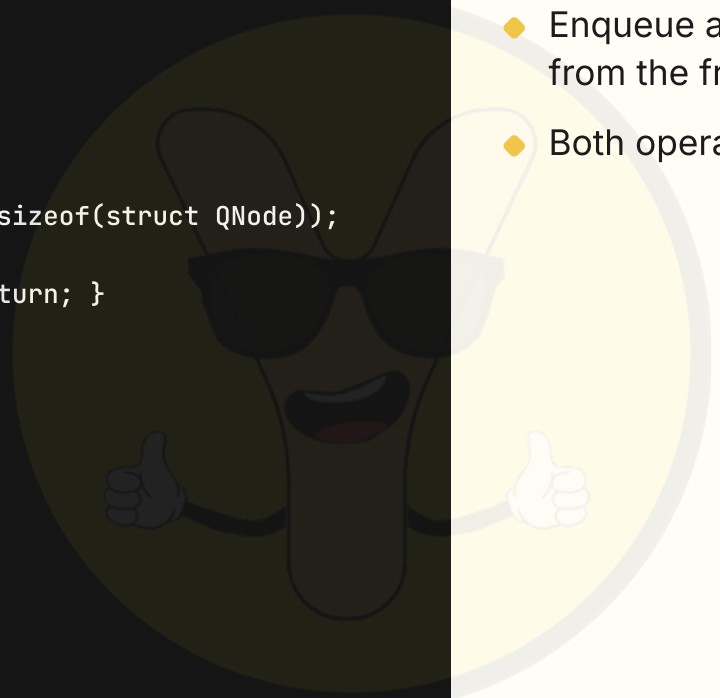
```
#define MAX 100
int queue[MAX], front = -1, rear = -1;
void enqueue(int val) {
    if (rear == MAX - 1) printf("Queue Overflow\n");
    else {
        if (front == -1) front = 0;
        queue[++rear] = val;
    }
}
int dequeue() {
    if (front == -1 || front > rear) {
printf("Underflow\n"); return -1; }
    else return queue[front++];
}
```



- Maintains two integer pointers: front and rear.
- Both Enqueue and Dequeue run in $O(1)$ time.
- Linear array implementation wastes space after multiple dequeues.

Linked List Implementation of Queue in C

```c
struct QNode {
    int data;
    struct QNode* next;
};
struct QNode *front = NULL, *rear = NULL;
void enqueue(int val) {
    struct QNode* temp = (struct QNode*)malloc(sizeof(struct QNode));
    temp→data = val; temp→next = NULL;
    if (rear == NULL) { front = rear = temp; return; }
    rear→next = temp;
    rear = temp;
}
int dequeue() {
    if (front == NULL) return -1;
    struct QNode* temp = front;
    int val = temp→data;
    front = front→next;
    if (front == NULL) rear = NULL;
    free(temp);
    return val;
}
```



- Dynamic implementation avoids fixed-size constraints.
- Enqueue adds at the rear pointer; Dequeue removes from the front pointer.
- Both operations execute in $O(1)$ time.

Deque (Double-Ended Queue)

DEQUE

A generalized queue data structure that supports insertion and deletion at both ends: the front and the rear.

- Input-restricted deque: Insertion is allowed at only one end, but deletion is allowed at both ends.
- Output-restricted deque: Deletion is allowed at only one end, but insertion is allowed at both ends.
- Implemented using circular arrays or doubly linked lists in $O(1)$ time.

Priority Queue

PRIORITY QUEUE

A specialized queue where each element is associated with a priority, and elements are served based on their priority rather than arrival order.

- If two elements have the same priority, they are served according to their FIFO order.
- Min-Priority Queue: Element with the lowest value has highest priority.
- Max-Priority Queue: Element with the highest value has highest priority.
- Efficiently implemented using Binary Heaps.

Exam Corner

- **Q1** Write an algorithm to convert an infix expression to a postfix expression using a stack.
- **Q2** Explain array implementation of a circular queue and write C functions for Enqueue and Dequeue.
- **Q3** Compare recursion and iteration with respect to time, space, and memory overhead.
- **Q4** Solve the Tower of Hanoi problem for 3 disks and write its recursive C function.
- **Q5** Explain Priority Queues and their applications in operating systems.
- **Q6** Write a C program to evaluate a given postfix expression using a stack.
- **Q7** What is stack overflow and underflow? How are they handled in linked list stack implementations?

Unit 2 done. You are exam-ready.

Revise the Exam corner once more, then open Unit 3 on theycotes.com. Every AKTU subject, unit by unit, free. Notes by Chirag Jain, Ycotes.

