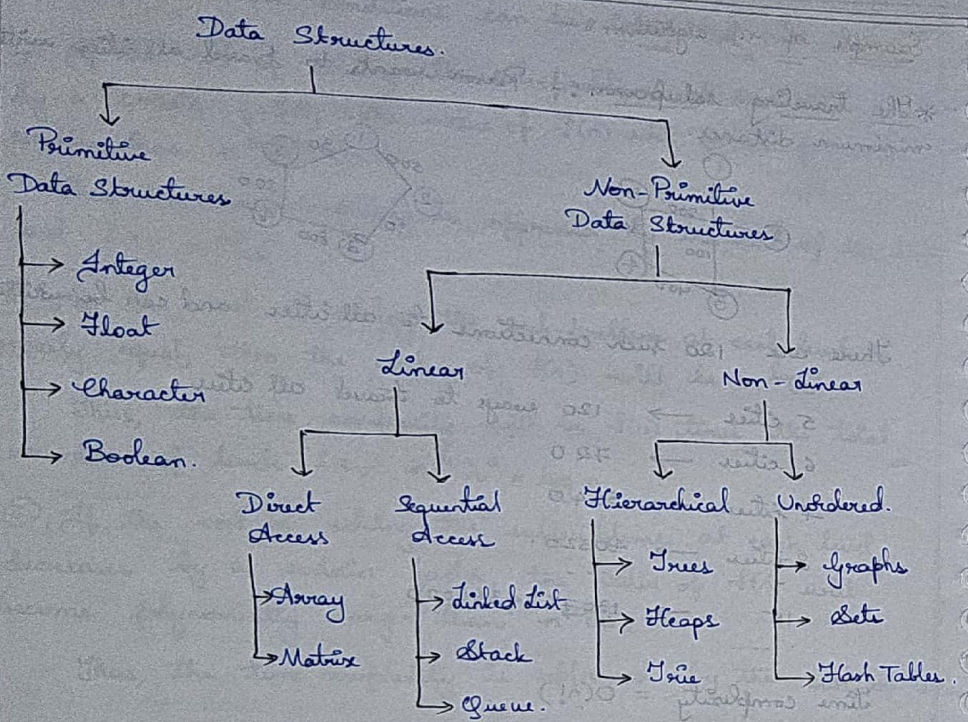
TYPES OF DATA STRUCTURES



Fig, Types of Data Structures.

* Arrays and Linked Lists :-

If you want to insert in a sequential manner, use arrays or linked lists.

Pros & Cons :-

	Arrays	Lists
Reading	O(1)	O(n)
Insertion	O(n)	O(1)

If you write down everything you spent on everyday and if you want to review the expenses at end of each month, which DS you prefer to save this data?

"LINKED LISTS."

* Inserting into the middle.

While using list, it is as easy as pointing the address of previous element to the node being inserted.

But for the arrays, you have to shift all the rest of the elements down. And if there's no space, you might have to copy everything to a new location!

* Deletion:-

Again lists would be better. Changing the next address of prev node to the next node's address. [O(1)].

But for arrays, deletion requires elements to be moved up [O(n)]

	Arrays	Lists
READING	O(1)	O(n)
INSERTION	O(n)	O(1)
DELETION	O(n)	O(1)

* Access :-

Random Access – Arrays
Sequential Access – Linked Lists.

Examples:-

① Restaurants - Orderqueue:-

Servers keep adding orders from the end.
Chefs keep removing/collecting orders from the front.
So LINKED LISTS are better to implement Queue.

② Facebook usernames:-

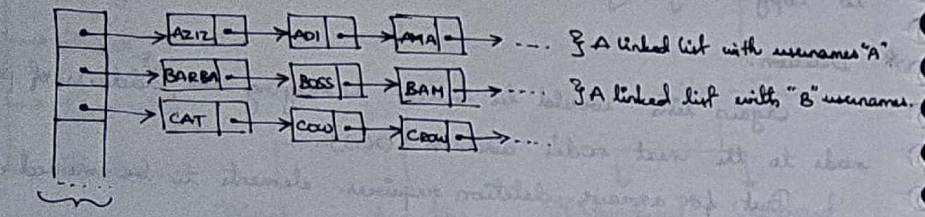
Facebook searches username in database using Binary Search. This needs random access. So ARRAYS are better.

③. Peoply signup for Facebook prilly often too. What happens when you add new users to an array?

In reality, Facebook uses neither an array nor a linked list to store user information.

Hybrid data structure: array of linked list.

Array of 26 slots each pointing to a linked list.



AN ARRAY

To add Aditi B as she signed up for Facebook, go to slot 1 and add the username to the list.

If Zakhir signs, search for the username in slot 26.

DS 1. ARRAYS

Static vs. Dynamic Sized Arrays:-

① Static-sized Arrays:- have a fixed size, determined at compile time. Once declared, the size of a static array cannot be changed.

* Characteristics :-

1) Fixed Size
2) Memory Allocation - memory allocated on stack
3) Performance - fast as they are contiguous and indexed.

* `int[] staticArray = new int[5];`
`staticArray[0] = 1;`

② Dynamic-sized Arrays:- can change size during runtime. They can grow or shrink as needed, offering more flexibility.

* Characteristics :-

1) Resizable
2) Memory Allocation - memory allocated on the heap, which allows them to have a flexible size but also means that memory management (allocation and deallocation) is more complex and slightly slower.
3) Efficiency Considerations - resizing operations and copying existing elements to the new location, which can be costly in terms of performance.

* `ArrayList<Integer> dynamicArray = new ArrayList<>();`
`dynamicArray.add(1);`

③ Usage:-

Choice depends on specific requirements:
① size of collection known ahead of time ② variability in the size of elements to be stored.

Basic Concepts and Operations :-

Operations	Big-O
Access	O(1)
Insert (At the end)	O(1)
Insert (At specific index)	O(n)
Delete (From the end)	O(1)
Delete (At specific index)	O(n)
Search	O(n)
Search (Sorted array)	O(log n)
Updating	O(1)

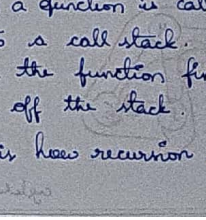
DS ⇧ STACK

- push()
- pop()
- peek()
- isEmpty()

```
MyStack<T> {
    Stack<T> stack = new Stack<>();
    public boolean isEmpty() {
        return stack.empty();
    }
    public void push(T data) {
        stack.push(data);
    }
    public T pop() {
        if (isEmpty()) throw new EmptyStackException();
        return stack.pop();
    }
    public T peek() {
        if (isEmpty()) throw new EmptyStackException();
        return stack.peek();
    }
}
```

* Implementing stack using Arrays.
* Implementing stack using Linked Lists.
* Applications of Stack :-

① Memory management. → when a function is called, the system 'pushes' it onto a call stack.
when the function finishes running, it's 'popped' off the stack.
This is how recursion works in programming.



② Expression Evaluation and Syntax Parsing -

2 + 3 * 4

BODMAS rule - Bracket, Order, Division and Multiplication, Addition and Subtraction.

Storing operators in a stack can help manage their execution order. Similar logic applies when compilers parse code syntax. They use stacks to check if all opening brackets have matching closing ones, which helps validate the syntax.

③ Undo Mechanism in Software Applications -

Recently pushed change will be popped and that's how it's undone

④ Backtracking Algorithms -

Backtracking algorithms solve problems by trying out solutions and undoing them if they don't lead to a solution.

Eg:- Sudoku.

8 Queens problem. imp **

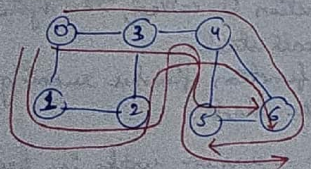
Knight's Tour problem.

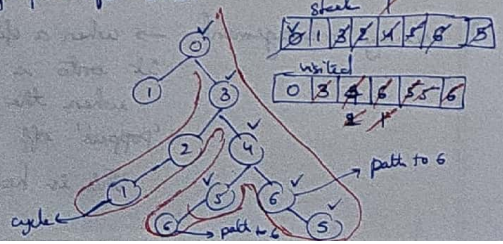
In such scenarios, stacks are used to store the intermediate stages of the problem. When an attempted solution doesn't work out, the algorithm can 'pop' back to a previous state and try a different path.

It's like having a 'save game' feature when you're tackling a challenging puzzle!

⑤ Depth - First Search (DFS) -

Stacks are used in graphs for DFS.





DFS explores a graph by visiting a node and recursively investigating all its unvisited neighbours. The algorithm uses a stack to remember which nodes to visit next when it finishes exploring a path.

By 'pushing' unvisited nodes onto the stack and 'popping' them once visited, DFS systematically explores every node in the graph.

* Network routing
* AI behavior in games.
* detecting cycles in a graph.) imp **

⑥ Web Page History in Browsers -

click a link → page is 'pushed' → a new page appears.
hit 'back' → topmost page is 'popped' → takes back to previous

PATTERNS 1. TWO POINTERS

Introduction to Two Pointer Pattern :-

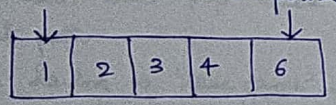
In problems where we deal with sorted arrays (or linked lists), and need to find a set of elements that fulfill certain constraints, the Two Pointers approach becomes quite useful. The set of elements could be a pair, a triplet or even a subarray.

Eg:- Given an array of sorted numbers and target sum, find a pair in the array whose sum is equal to the given target.

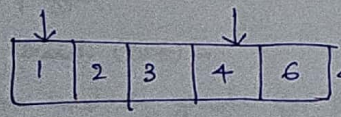
Given an input array is sorted, an efficient approach would be to start with one pointer at the beginning and another pointer at the end. At every step, we will check if the numbers indicated by the two pointers add up to the target sum. If they do not, we have two options:

① If the sum of the two numbers indicated by the two pointers is greater than the target sum, this means that we need a pair with a smaller sum. To explore more pairs, we can decrement the end-pointer.

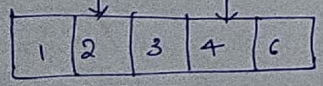
② If the sum of the two numbers indicated by the two pointers is smaller than the target sum, this means that we need a pair with a larger sum. To explore more pairs, we can increment the start-pointer.



>6 target sum = 6



<6



= 6

time complexity = O(N)

DS III. QUEUE

1. INTRODUCTION TO QUEUES

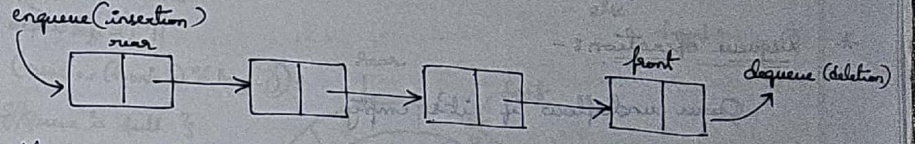
① **Introduction to Queues :-**

① What is a Queue?

FIFO - "First In, First Out"

② **Basic Terminology :-**

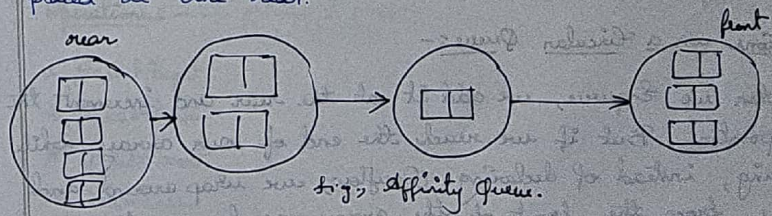
* Enqueue :- add an element at the end.
* Dequeue :- removing element from the front.



* Front :- start of the Queue.
* Rear :- end of the Queue.

③ **Types of Queues :-**

1) Simple Queue FIFO (Basic Queue)
2) Deque (Double Ended Queue) Elements can be added & removed from both ends of the Queue. Eg: Coffee Shop line where orders can be taken from both ends!
3) Circular Queue Last element points to the first, making a loop!
4) Priority Queue In this type, some elements are more important and get to jump the line.
5) Affinity Queue. In this type, every element has an affinity & is placed with an element having the same affinity; otherwise placed at the rear.



Eg:- Affinity Queue.

2. WORKING WITH SIMPLE QUEUES

* **Operations on Simple Queues:-**

Enque (add)
Deque (remove)
peek()
isEmpty()

* **Enqueue Operation:-**

Queue overflow if it's full.

* **Dequeue operation:-**

Queue underflow if it's empty.

* **Peek and IsEmpty:-**

peek shows the front element.

Operations	Big-O
Enqueue/Offer	O(1)
Dequeue/Poll	O(1)
Peek/Front	O(1)
IsEmpty	O(1)

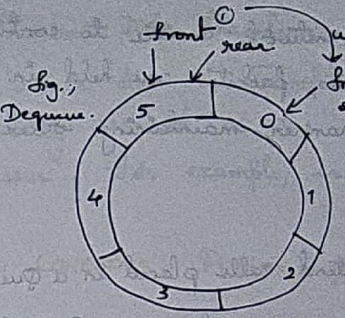
3. DIVING DEEPER – CIRCULAR QUEUES AND DEQUES

* **Understanding Circular Queues:-**

* **Operations on a Circular Queue:-**

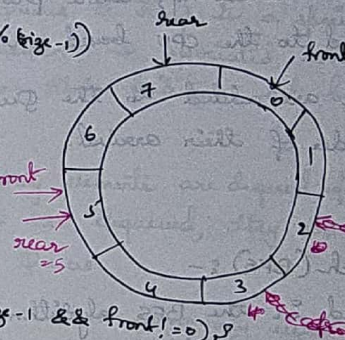
→ When we Enqueue, we add it at the rear and increment the rear pointer. But if we reach the end of our array while enqueueing, instead of declaring Overflow, we wrap around and continue from the front of the array, as long as there is space.

→ When we Deque, we remove it from the front and increment the front pointer. But if reach the



ENQUEUE? -

```
if((front==0 &&
rear==size-1) ||
(rear == (front-1) % (size-1))
{// Queue is full }
else if (front == -1) {
    front = 0
    rear = 0
    queue[rear] = e;
}
else if (rear == size-1 && front != 0) {
    rear = 0
    queue[rear] = e;
}
else {
    rear = rear+1;
    queue[rear] = e;
}
```



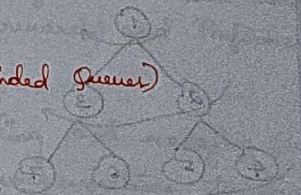
(a)

{= (front-1)%(size-1)}
= 10 % 7
= 3

DEQUEUE? -

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

* **Introduction to Dequeues (Double-Ended Queues)**



Operations: -

* Insert Front
* Insert Rear
* Delete Front
* Delete Rear

4. APPLICATIONS AND ADVANCED CONCEPTS

* **DAILY LIVES :-**

(1) **Data Traffic Management :-**

Queues are used in network devices to control the amount of data traffic. The data packets are held in a Queue and processed in a FIFO manner, maintaining order and ensuring fair access.

(2) **Call Centres :-** Agents attend calls placed in a Queue.

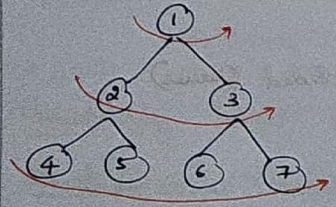
(3) **Operating Systems :-** To manage processes in Operating Systems. They are used in task scheduling, where processes are kept in a Queue and allocated to the CPU based on scheduling algorithms.

(4) **Printers :-** Print jobs are added to the Queue and are executed in the order of their arrival.

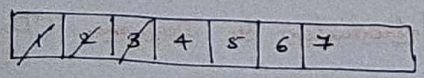
* **PROGRAMMING :-**

(1) **Breadth - First Search (BFS) :-**

Queues are used in the BFS algorithm to visit nodes of a tree or a graph in a breadth-wise manner. It starts at a given node (often the root of a tree or any node of a graph), then explores all of the neighbour nodes at the current depth before moving on to nodes at the next depth level.



Queue



popping & adding children.

(2) **Caching :-** When cache is full, oldest item in cache is removed (this is using queues).

(3) **Asynchronous Data Transfer :-** Queues are used in scenarios where data is transferred asynchronously (data is not necessarily received at the same rate it's sent) between two processes. For example, in I/O Buffers.

* **ADVANCED CONCEPTS - PRIORITY QUEUES AND QUEUEING THEORY :-**

(1) **Priority Queues - Because Order Matters :-**

A Priority Queue & Heap is a special type of Queue in which each element is associated with a priority. Elements are served based on their priority - not based on their position in the Queue.

Highest Priority elements are dequed first. If elements with equal priorities are dequeued, they are served according to their order in the Queue.

Priority Queues are commonly used in operating systems for process scheduling

(2) **Queueing Theory - Behind the Scenes of Queues.**

Queueing Theory is a mathematical study of waiting lines, or Queues. It's used to predict queue lengths and wait times. It's a key component of Operations research and Performance Engineering.

Queueing Theory helps in understanding and managing Queues better in various fields, from telecommunications to traffic engineering. It forms the backbone of efficient resource allocation.