ARRAY EXAMPLE

Using 85, 90, 72, 64, 95 in C, Java and Python

C LANGUAGE

Code

```c
#include <stdio.h>
int main()
{
    int marks[5] = {85, 90, 72, 64, 95};
    int i;
    printf("Array elements:\n");
    for(i = 0; i < 5; i++)
        printf("%d ", marks[i]);
    printf("\n");
    return 0;
}
```

Explanation

- `int marks[5]` declares an array of 5 integers.
- The array is initialized with values 85, 90, 72, 64, 95.
- A `for` loop is used to print each element.

Memory Representation

Index	0	1	2	3	4
marks	85	90	72	64	95

Output

```
Array elements:
85 90 72 64 95
```

JAVA LANGUAGE

Code

```java
public class ArrayExample {
    public static void main(String[] args) {
        int[] marks = {85, 90, 72, 64, 95};
        System.out.println("Array elements:");
        for (int i = 0; i < marks.length; i++) {
            System.out.print(marks[i] + " ");
        }
        System.out.println();
    }
}
```

Explanation

- `int[] marks` declares an array of integers.
- The array is initialized with 85, 90, 72, 64, 95.
- `marks.length` gives the size of the array.
- A `for` loop is used to print each element.

Memory Representation

Index	0	1	2	3	4
marks	85	90	72	64	95

Output

```
Array elements:
85 90 72 64 95
```

PYTHON LANGUAGE

Code

```python
marks = [85, 90, 72, 64, 95]
print("Array elements:")
for num in marks:
    print(num, end = " ")
print()
```

Explanation

- `marks` is a list that stores the values.
- Lists in Python are dynamic arrays.
- A `for` loop is used to print each element.

Memory Representation

Index	0	1	2	3	4
marks	85	90	72	64	95

Output

```
Array elements:
85 90 72 64 95
```

KEY COMPARISON

C
- Uses fixed-size arrays. Size must be known in advance.
- Efficient and close to hardware.

Java
- Uses arrays with fixed size.
- Safer with bounds checking and rich libraries.

Python
- Uses lists which are dynamic in size.
- Easier syntax and built-in functions.

Index:	0	1	2	3	4
Value:	85	90	72	64	95

Same data stored in different languages

DYNAMIC ARRAY EXAMPLE

Storing 85, 90, 72, 64, 95 and then adding 88 (Size grows automatically)

C LANGUAGE

Code

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *arr = NULL;
    int size = 5, i;
    arr = (int*)malloc(size * sizeof(int));

    int values[] = {85, 90, 72, 64, 95};
    for(i = 0; i < size; i++)
        arr[i] = values[i];

    // Add 88 (resize array)
    size = size + 1;
    arr = (int *)realloc(arr, size * sizeof(int));
    arr[size-1]  = 88;

    printf("Array elements:\n");
    for(i = 0; i < size; i++)
        printf("%d ", arr[i]);
    free(arr);
    return 0;
}
```

Explanation

- Memory is allocated using `malloc()`.
- When size increases, `realloc()` is used to get a new block.
- In C, we must manage memory manually.

Memory Illustration

Initial (size = 5)

Index	0	1	2	3	4
Value	85	90	72	64	95

Add 88 (size = 6)

In 0	1	2	3	4	5
90	72	64	95	88	

Output

```
Array elements:
85  90  72  64  95  88
```

JAVA LANGUAGE

Code

```java
import java.util.ArrayList;

public class DynamicArrayExample {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(85);
        list.add(90);
        list.add(72);
        list.add(64);
        list.add(95);
        list.add(88); // size grows automatically

        System.out.println("Array elements: " + list);
    }
}
```

Explanation

- ArrayList is a dynamic array.
- When we add more elements, it automatically increases capacity.
- No need to manage memory.

Memory Illustration

Initial Capacity (example = 5)

Index	0	1	2	3	4
Value	85	90	72	64	95

After adding 88 → Capacity increases (example = 10)

Index	0	1	2	3	4	5	6	7	8	9
Value	85	90	72	64	95	88	_	_	_	_

Output

```
Array elements: [85, 90, 72, 64, 95, 88]
```

PYTHON LANGUAGE

Code

```python
arr = [85, 90, 72, 64, 95]

arr.append(88)   # size grows automatically

print("Array elements:", arr)
```

Explanation

- Python lists are dynamic arrays.
- append() automatically increases the size.
- Memory management is handled by Python.

Memory Illustration

Initial

Index	0	1	2	3	4
Value	85	90	72	64	95

After append(88)

Index	0	1	2	3	4	5
Value	85	90	72	64	95	88

Output

```
Array elements: [85, 90, 72, 64, 95, 88]
```

KEY COMPARISON

Feature	C	Java (ArrayList)	Python (list)
Creation	malloc()	new ArrayList<<()	[]
Size Growth	Manual (realloc)	Automatic	Automatic
Memory Management	Manual (malloc/free)	Automatic (Garbage Collector)	Automatic (Garbage Collector)
Ease of Use	★★☆☆☆	★★★★☆	★★★★★

REAL-WORLD EXAMPLES

- **Contact List** — New contacts added regularly
- **Shopping Cart** — Items added or removed
- **Task List** — Tasks keep growing
- **Stock Prices** — New data comes every second
- **Chat Messages** — Messages increase dynamically

QUICK TAKEAWAY

- Use C when you need maximum control over memory and performance.
- Use Java when you want safe, automatic memory management and rich libraries.
- Use Python when you want simplest syntax and fastest development.

LINKED LIST EXAMPLE – MUSIC PLAYLIST

A playlist where songs are stored in a linked list. Each song points to the next song.

PLAYLIST (INITIAL)

1. Shape of You
2. Believer
3. Thunder
4. Perfect
5. Someone You Loved

C LANGUAGE

1. Node Structure

```c
struct Node {
    char song[50];
    struct Node* next;
};
```

Each node stores a song and the address of next song.

2. Create Playlist

```c
struct Node* head = NULL;    // Empty playlist
```

3. Insert Song at End

```c
void insertEnd(struct Node** head, char* song) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    strcpy(newNode->song, song);
    newNode->next = NULL;
    if(*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while(temp->next != NULL) temp = temp->next;
    temp->next = newNode;
}
```

4. Traverse & Display Playlist

```c
void display(struct Node* head) {
    struct Node* temp = head;
    int i = 1;
    while(temp != NULL) {
        printf("%d. %s\n", i++, temp->song);
        temp = temp->next;
    }
}
```

📌 Memory is created using malloc() and must be freed using free().

JAVA LANGUAGE

1. Node Class

```java
class Node {
    String song;
    Node next;

    Node(String song) { this.song = song; this.next = null; }
}
```

2. Create Playlist

```java
Node head = null;     // Empty playlist
```

3. Insert Song at End

```java
void insertEnd(String song) {
    Node newNode = new Node(song);
    if (head == null) {
        head = newNode;
        return;
    }
    Node temp = head;
    while (temp.next != null) temp = temp.next;
    temp.next = newNode;
}
```

4. Traverse & Display Playlist

```java
void display() {
    Node temp = head;
    int i = 1;
    while (temp != null) {
        System.out.println(i++ + ". " + temp.song);
        temp = temp-next;
    }
}
```

📌 Java handles memory automatically (Garbage Collector).

PYTHON LANGUAGE

1. Node Class

```python
class Node:
    def __init__(self, song):
        self.song = song
        self.next = None
```

2. Create Playlist

```python
head = None     # Empty playlist
```

3. Insert Song at End

```python
def insert_end(song):
    globad head
    new_node = Node(song)
    if head is None:
        head = new_node
        return
    temp = head
    while temp.next:
        temp = temp.next
    temp.next = new_node
}
```

4. Traverse & Display Playlist

```python
def display();
    temp = head
    i = 1
    while temp:
        print(f"{i}. {temp.song}")
        temp = temp.next
    )  i += 1
}
```

📌 Python memory is managed automatically.

LINKED LIST REPRESENTATION

OPERATIONS ON PLAYLIST

Insert at Beginning (Add 'Blinding Lights')

Delete a Song (Delete 'Thunder')

Insert at End (Add 'Havana')

Insert After a Song (After 'Believer' add 'Levitating')

COMPARISON SUMMARY

Feature	C	Java	Python
Memory Management	Manual (malloc/free)	Automatic (GC)	Automatic (GC)
Syntax	Verbose	Moderate	Simple
Node Creation	malloc()	new Node()	Node()
Pointer / Reference	Pointers	References	References
Ease of Use	★★☆☆☆	★★★☆☆	★★★★☆
Performance	Very Fast	Fast	Moderate

COMPLETE PLAYLIST CODE (DEMO SONGS)

Songs: ["Shape of You", "Believer", "Thunder", "Perfect", "Someone You Loved"]

After Operations:

- Insert at Beginning -> "Blinding Lights"
- Insert After "Believer" -> "Levitating"
- Delete "Thunder"
- Insert at End -> "Havana"

Final Playlist Order:

Blinding Lights -> Shape of You -> Believer -> Levitating -> Perfect -> Someone You Loved -> Havana

REAL-WORLD APPLICATIONS

- Music / Video Playlist
- Browser History (Back / Forward)
- Undo / Redo in Editors
- Photo Viewer (Next / Previous)
- Task List Management
- Chat Applications
- File Systems (Linked Allocation)
- Operating System Process List

STACK EXAMPLE – DINING PLATES

A stack follows **LIFO (Last In First Out)**. The last plate placed on top is the first one removed.

C LANGUAGE

```c
// STACK USING ARRAY
#include <stdio.h>
#define MAX 5

int stack[MAX];
int top = -1;  // stack is empty

// Push: place a plate on top
void push(int x) {
    if (top == MAX-1) {
        printf("Stack Overflow!\n");
        return;
    }
    stack[++top] = x;
    printf("Pushed %d (Plate placed)\n", x);
}

// Pop: remove top plate
int pop() {
    if (top == -1) {
        printf("Stack Underflow!\n");
        return -1;
    }
    int x = stack[top--];
    printf("Popped %d (Plate removed)\n", x);
    return x;
}

// Display stack
void display() {
    printf("Plates in stack (Top -> Bottom): ");
    if (top == -1) { printf("(Empty)\n"); return; }
    for (int i = top; i >= 0; i--)
        printf("%d ", stack[i]);
        printf("\n");
}  }

// DEMO
int main() {
    push(1);      // Plate 1
    push(2);      // Plate 2
    push(3);      // Plate 3
    push(4);      // Plate 4
    display();
    printf("--- Now removing plates ---\n");
    pop();        // Removes Plate 4
    pop();        // Removes Plate 3
    display();
    return 0;
}
```

C OUTPUT (Sample)

```
Pushed 1 (Plate placed)
Pushed 2 (Plate placed)
Pushed 3 (Plate placed)
Pushed 4 (Plate placed)
Plates in stack (Top -> Bottom): 4 3 2 1
--- Now removing plates ---
Popped 4 (Plate removed)
Popped 3 (Plate removed)
Plates in stack (Top -> Bottom): 2 1
```

JAVA LANGUAGE

```java
import java.util.Stack;

public class StackPlates {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>();

        // Push (place plate)
        stack.push(1);    // Plate 1
        stack.push(2);    // Plate 2
        stack.push(3);    // Plate 3
        stack.push(4);    // Plate 4

        System.out.println("Plates in stack (Top -> Bottom):
        " + stack);

        // Pop (remove plate)
        System.out.println("--- Now removing plates ---");
        System.out.println("Popped " + stack.pop() + " (Plate
        removed)");
        System.out.println("Popped " + stack.pop() + " (Plate
        removed)");

        System.out.println("Plates in stack (Top -> Bottom): " +
    }
}
```

JAVA OUTPUT (Sample)

```
Plates in stack (Top -> Bottom): [1, 2, 3, 4]
--- Now removing plates ---
Popped 4 (Plate removed)
Popped 3 (Plate removed)
Plates in stack (Top -> Bottom): [1, 2]
```

PYTHON LANGUAGE

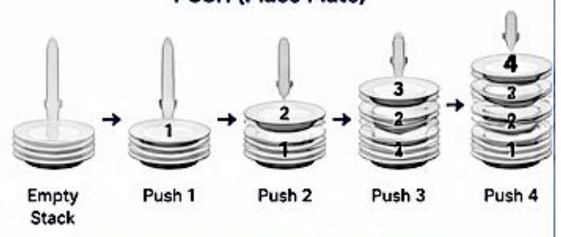
```python
# Stack using list
stack = []

# Push (place plate)
stack.append(1)   # Plate 1
stack.append(2)   # Plate 2
stack.append(3)   # Plate 3
stack.append(4)   # Plate 4

print("Plates in stack (Top -> Bottom):", stack)

# Pop (remove plate)
print("--- Now removing plates ---")
print("Popped", stack.pop(), "(Plate removed)")
print("Popped", stack.pop(), "(Plate removed)")

print("Plates in stack (Top -> Bottom):", stack)
```

PYTHON OUTPUT (Sample)

```
Plates in stack (Top -> Bottom): [1, 2, 3, 4]
--- Now removing plates ---
Popped 4 (Plate removed)
Popped 3 (Plate removed)
Plates in stack (Top -> Bottom): [1, 2]
```

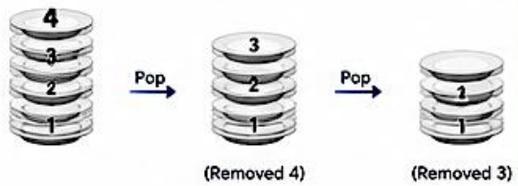
DINING PLATES ANALOGY

PUSH (Place Plate)



We keep placing new plates on the top.

POP (Remove Plate)



We always remove the top plate first.

KEY POINTS

- Stack follows LIFO (Last In First Out).
- Push = place a plate on top.
- Pop = remove the top plate.
- Operations allowed only at one end (Top).

REAL-LIFE USES

- Undo in Editors
- Browser Back Button
- Function Calls
- Expression Evaluation
- Backtracking Problems
- Syntax Parsing

COMPARISON TABLE

Feature	C (Array)	Java (Stack Class)	Python (List)
Implementation	Array + top index	Built-in Stack Class	List (built-in)
Push Operation	O(1)	O(1)	O(1)
Pop Operation	O(1)	O(1)	O(1)
Overflow Handling	Manual	Handled (StackException)	Handled (MemoryError if huge)
Underflow Handling	Manual	Handled (EmptyStackException)	Handled (IndexError)
Ease of Use	★★★☆☆	★★★★☆	★★★★★

STACK OPERATIONS COMPLEXITY

Operation	Time Complexity
Push	O(1)
Pop	O(1)
Peek (Top)	O(1)
IsEmpty	O(1)

SUMMARY

Just like dining plates are stacked one over another and the top plate is removed first, stack data structure works in the same way.

LIFO Principle: Last In First Out

Last In → 1 → 2 → 3 → 4 → First Out

(4 is last in, so 4 comes out first)

LEGEND

- Push / Place Plate
- Pop / Remove Plate

QUEUE EXAMPLE – TICKET COUNTER

A Queue follows **FIFO** (First In First Out). The first person who comes, gets the ticket first.

TICKET COUNTER ANALOGY

1. People stand in a line.
2. Person at FRONT gets the ticket first.
3. New people join at REAR.

C LANGUAGE

1. Queue using Array

```c
#include <stdio.h>
#define MAX 5
int queue[MAX], front = -1, rear = -1;

// Enqueue: Add person at rear
void enqueue(int x) {
   if (rear == MAX - 1) {
     printf("Queue is Full\n");
     return;
   }
   if (front == -1) front = 0;
   queue[++rear] = x;
   printf("Person %d joined the queue.\n", x);
}

// Dequeue: Remove person from front
int dequeue() {
  if (front == -1 || front > rear) {
     printf("Queue is Empty\n");
     return -1;
  }
   int x = queue[front++];
   if (front > rear) front = rear = -1; // reset
   printf("Person %d got the ticket.\n", x);
}  return x;

// Display Queue
void display() {
  if (front == -1) {
     printf("Queue is Empty\n");
     return;
  }
   printf("Queue (Front -> Rear): ");
   for (int i = front; i <= rear; i++)
        printf("%d ", queue[i]);
  printf("\n");
}

// Demo
int main() {
  enqueue(101);
  enqueue(102);
  enqueue(103);
  display();
  dequeue();
  display();
  enqueue(104);
  display();
  return 0;
}
```

SAMPLE OUTPUT

```
Person 101 joined the queue.
Person 102 joined the queue.
Person 103 joined the queue.
Queue (Front -> Rear):  101 102 103
Person 101 got the ticket.
Queue (Front -> Rear):  102 103
Person 104 joined the queue.
Queue (Front -> Rear):  102 103 104
```

JAVA LANGUAGE

1. Queue using LinkedList (Built-in)

```java
import java.util.*;

public class TicketCounter {
  public static void main(String[] args) {
    Queue<Integer> q = new LinkedList<>();

    // Enqueue:  Add person
    q.offer(101);   // Person 101
    q.offer(102);   // Person 102
    q.offer(103);   // Person 103

    System.out.println("Queue: " + q);

    // Dequeue: Remove person
    System.out.println("Ticket issued to: " + q.poll());
    System.out.println("Queue: " + q);

    // Enqueue more
    q.offer(104);   // Person 104
    System.out.println("Queue: " + q);
  }
}
```

SAMPLE OUTPUT

```
Queue: [101, 102, 103]
Ticket issued to: 101
Queue: [102, 103]
Queue: [102, 103, 104]
```

PYTHON LANGUAGE

1. Queue using collections.deque

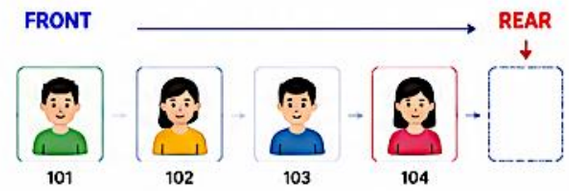
```python
from collections import deque

q = deque()

# Enqueue: Add person
q.append(101)   # Person 101
q.append(102)   # Person 102
q.append(103)   # Person 103

print("Queue:", list(q))

# Dequeue: Remove person
print("Ticket issued to:", q.popleft())
print("Queue:", list(q))

# Enqueue more
q.append(104)   #  Person 104
print("Queue:", list(q))
```

SAMPLE OUTPUT

```
Queue: [101, 102, 103]
Ticket issued to: 101
Queue: [102, 103]
Queue: [102, 103, 104]
```

TICKET COUNTER – VISUAL REPRESENTATION

1. People Join the Queue (Enqueue)



Queue: [101, 102, 103, 104]

2. Ticket Issued (Dequeue)

Queue: [102, 103, 104]

3. New Person Joins

Queue: [102, 103, 104, 105]

KEY POINTS

- Queue follows FIFO (First In First Out).
- Enqueue → Add at Rear (end of line).
- Dequeue → Remove from Front (start of line).
- Used when order of arrival matters.

C vs JAVA vs PYTHON – QUEUE COMPARISON

Feature	C (Array)	Java (Queue)	Python (deque)
Implementation	Manual (Array)	Built-in (LinkedList)	Built-in (deque)
Enqueue	O(1)	O(1)	O(1)
Dequeue	O(1)	O(1)	O(1)
Overflow Handling	Manual	Automatic	Automatic
Underflow Handling	Manual	Automatic	Automatic
Ease of Use	★★☆☆☆	★★★★☆	★★★★★

COMMON REAL-WORLD USES OF QUEUE

Ticket Booking Counters · Printer Spooling · Call Center Calls · CPU Scheduling · Network Packet Handling

WHEN TO USE QUEUE?

- When order of processing is important.
- When multiple requests are pending.
- When items are processed in the same order they arrive.

TIME COMPLEXITY

Enqueue (Insert)	: O(1)
Dequeue (Delete)	: O(1)
Peek (Front Element) :	: O(1)
IsEmpty	: O(1)
IsFull	: [For Array]

EMERGENCY TRIAGE LEVELS (Priority)			
Level	Priority	Description	Color
1 (Critical)	4	Life Threatening	
2 (High)	3	Serious – Needs immediate care	
3 (Medium)	2	Stable – Can wait	
4 (Low)	1	Minor – Can wait longer	

PRIORITY QUEUE EXAMPLE – EMERGENCY HOSPITAL

A Priority Queue is a data structure where each element has a priority.
The element with highest priority is served first.

GOAL

Treat patients based on severity (priority).
Highest priority (Critical) patients are treated first.

WHY PRIORITY QUEUE?

- Treat critical patients immediately
- Efficient management of emergency cases
- Fair and optimized resource allocation
- Ideal for real world scheduling problems

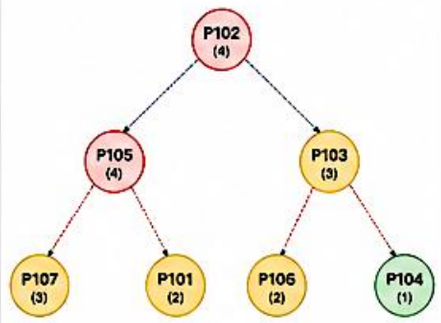
COMMON OPERATIONS	
Operation	Description
insert(patient)	Add patient to queue (based on priority)
peek()	View highest priority patient
extractMax()	Remove & return highest priority patient
isEmpty()	Check if queue is empty
isFull()	Check if queue is full

1. PATIENT ARRIVAL (Inserted into Priority Queue)

Arrival Order	Patient ID	Name	Emergency Level	Priority	Symptoms
1	P101	Ravi	3 (Medium)	2	Fever, Cold
2	P102	Sita	1 (Critical)	4	Heart Attack
3	P103	Arjun	2 (High)	3	Severe Bleeding
4	P104	Meena	4 (Low)	1	Sprain
5	P105	Kiran	1 (Critical)	4	Road Accident
6	P106	Divya	3 (Medium)	2	Breathing Problem
7	P107	Karthik	2 (High)	3	High Fever

Note: Higher number = Higher Priority

2. PRIORITY QUEUE (Max Heap Representation)



Heap Order: Parent priority ≥ Child priority

3. TREATMENT ORDER (Extracted by Priority)

Treatment Order	Patient ID	Name	Emergency Level	Priority
1	P102	Sita	1 (Critical)	4
2	P105	Kiran	1 (Critical)	4
3	P103	Arjun	2 (High)	3
4	P107	Karthik	2 (High)	3
5	P101	Ravi	3 (Medium)	2
6	P106	Divya	3 (Medium)	2
7	P104	Meena	4 (Low)	1

Critical Patients Treated First – Saves Lives

4. IMPLEMENTATION – PRIORITY QUEUE (MAX HEAP)

C (Using Array)

```c
#include <stdio.h>
#define MAX 100
int heap[MAX], size = 0;

void swap(int *a, int *b){
  int t=*a; *a=*b; *b=t;
}

void insert(int val){
  int i = size;
  heap[i] = val;
  while(i>0 && heap[i/2] < heap[i]){
    swap(&heap[i], &heap[i/2]);
    i = i/2;
  }
}

int extractMax(){
  if(size==0) return -1;
  int max = heap[1];
  heap[1] = heap[size--];
  int i=1;
  while(i*2<size){
    int largest = i;
    int l=i*2, r=i*2+1;
    if(l<=size && heap[l]>heap[largest])
      largest = l;
    if(r<=size && heap[r]>heap[largest])
      largest = r;
    if(largest==i) break;
    swap(&heap[i], &heap[largest]);
    i = largest;
  }
  return max;
}
```

Java (PriorityQueue)

```java
import java.util.*;
class Patient implements
Comparable<Patient>{
  String id, name;
  int level, priority;
Patient(String id, String name,
  int level, int priority){
  this.id=id; this.name=name;
  this.level=level; this.priority=priority;
}

public int compareTo(Patient p){
  return p.priority - this.priority;
} // Max Priority Queue

public String toString(){
  return id+"("+level+","+priority+")";
}
}
class Main{
  public static void main(String[] args){
    PriorityQueue<Patient> pq =
        new PriorityQueue<>();
    pq.add(new Patient("P102","Sita",1,4));
    pq.add(new Patient("P105","Kiran",1,4));
    pq.add(new Patient("P103","Arjun",2,3));
    pq.add(new Patient("P107","Karthik",2,3));
    pq.add(new Patient("P101","Ravi",3,2));
    pq.add(new Patient("P106","Divya",3,2));
    pq.add(new Patient("P104","Meena",4,1));
    }
    while(!pq.isEmpty())
      System.out.println(pq.poll());
} }
```

Python (heapq)

```python
import heapq

class Patient:
  def __init__(self, pid, name, level, priority):
    self.id = pid
    self.name = name
    self.level = level
    self.priority = priority

  def __lt__(self, other):
    # For max heap, use negative priority
    return self.priority > other.priority

  def __str__(self):
    return f"{self.id}({self.level},{self.priority})"

pq = []

patients = [
  Patient("P102","Sita",1,4),
  Patient("P105","Kiran",1,4),
  Patient("P103","Arjun",2,3),
  Patient("P107","Karthik",2,3),
  Patient("P101","Ravi",3,2),
  Patient("P106","Divya",3,2),
  Patient("P104","Meena",4,1)
]

for p in patients:
  heapq.heappush(pq, p)

while pq:
  print(heapq.heappop(pq))
```

5. STEP BY STEP EXECUTION (Nospital Flow)

Step	Operation	Patient	Queue After Operation (Front → Rear)	Explanation
1	Insert	P101 (2)	P101(2)	First patient arrives
2	Insert	P102 (4)	P102(4), P101(2)	P102 has higher priority → goes front
3	Insert	P103 (3)	P102(4), P103(3), P101(2)	P103 placed according to priority
4	Insert	P104 (1)	P102(4), P103(3), P101(2), P104(1)	Lowest priority at rear
5	Insert	P105 (4)	P102(4), P105(4), P101(2), P104(1), P103(3)	P105 (4) comes front
6	ExtractMax	P102 (4)	P105(4), P103(3), P101(2), P104(1), P106(2), P107(3)	P102 treated first (Critical)
7	ExtractMax	P105 (4)	P103(3), P107(3), P106(2), P101(2), P104(1)	Next critical treated
	...	...		Continue till empty

6. TIME COMPLEXITY

Operation	Time Complexity
Insertion (Enqueue)	O(log n)
Deletion (ExtractMax)	O(log n)
Peek (Top Priority)	O(1)
isEmpty()	O(1)
isFull()	O(1)

n = number of patients in queue

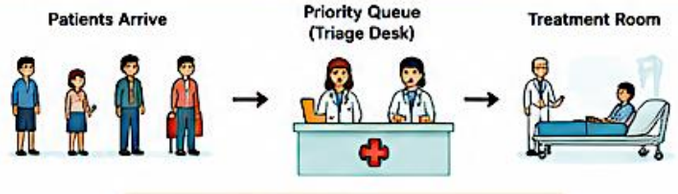
7. REAL LIFE USE CASES

- **Emergency Room Management** – Treat critical patients first
- **Air Traffic Control** – Handle urgent flights first
- **Fire Department Dispatch** – Send nearest/critical first
- **CPU Scheduling** – Execute high priority processes
- **Manufacturing Systems** – Handle urgent tasks first

8. PRIORITY QUEUE VS NORMAL QUEUE

Feature	Normal Queue (FIFO)	Priority Queue
Order	First Come First Serve	Highest Priority First
Insertion	O(1)	O(log n)
Deletion	O(1)	O(log n)
Use Case	Printing, Buffers	Scheduling, ER, CPU
Flexibility	Low	High
Complexity	Simple	More Efficient

9. HOSPITAL ANALOGY



Critical Patients are treated first
Improves survival rate & efficiency

10. KEY TAKEAWAYS

- Priority Queue serves highest priority element first.
- Ideal for scenarios where urgency matters.
- Max Heap is commonly used to implement PQ.
- Efficient and widely used in real world systems.
- Helps save time, resources and most importantly lives!

HASH MAP EXAMPLE – PHONE CONTACTS

Store phone number as key and contact name as value for fast lookup.

C LANGUAGE

Using Separate Chaining

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define SIZE 10

typedef struct Node {
   char name[50];
   char phone[15];
   struct Node* next;
} Node;

Node* table[SIZE];

int hash(char* phone) {
   long sum = 0;
   for (int i = 0; phone[i] != '\0'; i++)
      sum = (sum + phone[i]) % SIZE;
   return sum;
}

void put(char* phone, char* name) {
   int index = hash(phone);
   Node* newNode = (Node*)malloc(sizeof(Node));
   strcpy(newNode->phone, phone);
   strcpy(newNode->name, name);
   newNode->next = table[index];
   table[index] = newNode;
}

char* get(char* phone) {
   int index = hash(phone);
   Node* temp = table[index];
   while (temp != NULL) {
      if (strcmp(temp->phone, phone) == 0)
         return temp->name;
      temp = temp->next;
   }
   return NULL;
}

int main() {
   for(int i = 0; i < SIZE; i++) table[i] = NULL;

   put("9876543210", "Siva");
   put("9123456780", "Ramesh");
   put("9988776655", "Anitha");

   printf("Name: %s\n", get("9123456780"));
   return 0;
}
```

OUTPUT

```
Name: Ramesh
```

JAVA LANGUAGE

Using HashMap (Built-in)

```java
import java.util.HashMap;

public class PhoneContacts {
  public static void main(String[] args) {
    HashMap<String, String> contacts = new HashMap<>();

    // Add contacts
    contacts.put("9876543210", "Siva");
    contacts.put("9123456780", "Ramesh");
    contacts.put("9988776655", "Anitha");

    // Retrieve contact
    System.out.println("Name: " + contacts.get("9123456780"));

    // Check if contact exists
    if (contacts.containsKey("9988776655")) {
      System.out.println("Anitha's number exists.");
    }

    // Display all contacts
    System.out.println("\nAll Contacts:");
    for (String phone : contacts.keySet()) {
      System.out.println(phone + " -> " + contacts.get(phone));
    }
  }
}
```

SAMPLE OUTPUT

```
Name: Ramesh
Anitha's number exists.

All Contacts:
9876543210  -> Siva
9988776655  -> Anitha
9123456780  -> Ramesh
```

PYTHON LANGUAGE

Using dict (Built-in)

```python
# Create phone contacts dictionary
contacts = {
    "9876543210": "Siva",
    "9123456780": "Ramesh",
    "9988776655": "Anitha"
}

# Retrieve contact
print("Name:", contacts["9123456780"])

# Check if contact exists
if "9988776655" in contacts:
   print("Anitha's number exists.")

# Add new contact
contacts["9000001111"] = "Karthik"

# Display all contacts
print("\nAll Contacts:")
for phone, name in contacts.items():
   print(phone, "->", name)
```

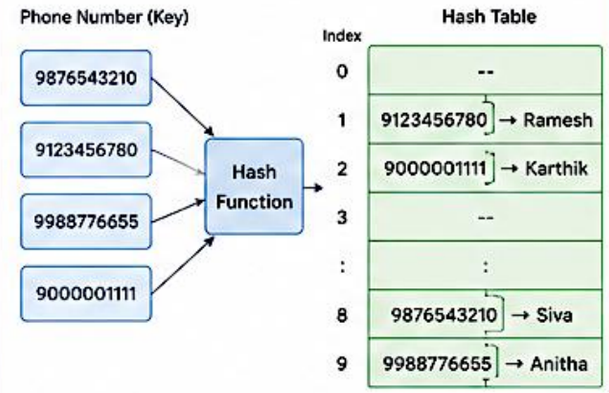
SAMPLE OUTPUT

```
Name: Ramesh
Anitha's number exists.

All Contacts:
9876543210  -> Siva
9123456780  -> Ramesh
9988776655  -> Anitha
9000001111  -> Karthik
```

PHONE CONTACTS (KEY → VALUE)

Phone Number (Key)	Contact Name (Value)
9876543210	Siva
9123456780	Ramesh
9988776655	Anitha
9000001111	Karthik

HOW HASH MAP WORKS



Each key is hashed to an index.
At that index, the key-value (contact) is stored.

OPERATIONS IN HASH MAP

Operation	Description	C (Custom)	Java (HashMap)	Python (dict)
Insert	Add a new contact	put()	put(key, value)	contacts[key] = value
Search	Find contact by number	get()	get(key)	contacts.get(key)
Delete	Remove a contact	remove()	remove(key)	del contacts[key]
Update	Update contact name	put()	put(key, newValue)	contacts[key] = newValue
Size	Number of contacts	count	size()	len(contacts)
Check Exists	Check if number exists	get()!=NULL	containsKey(key)	key in contacts

COMPARISON SUMMARY

Feature	C (Custom HashMap)	Java (HashMap)	Python (dict)
Implementation	Manual (Arrays + Linked List)	Built-in Class	Built-in (Optimized)
Ease of Use	★★☆☆☆	★★★★☆	★★★★★
Performance (Avg.)	O(1)	O(1)	O(1)
Memory Management	Manual	Automatic (GC)	Automatic (GC)
Best For	Learning Concepts	Enterprise Apps	Rapid Development

REAL-WORLD USES

- Phone Contacts
- Login Systems (User → Details)
- Bank Accounts (Acc No → Info)
- Caching in Web / Apps
- Student Records (Roll No → Info)
- E-commerce (Product ID → Info)

KEY TAKEAWAYS

- Hash Map stores data as key-value pairs.
- Each key is unique.
- Provides very fast search, insert and delete.
- Ideal for real-time applications like contacts, login, caching, etc.
- Java and Python have built-in, optimized implementations.

SET EXAMPLE – STUDENT ROLL NUMBERS

A Set stores only UNIQUE values. Duplicate roll numbers are not allowed.

Student Roll Numbers

101, 102, 103, 104, 105

C LANGUAGE

Using Array + Function (Simple Set Implementation)

```c
#include <stdio.h>
#define MAX 100

int set[MAX];
int size = 0;

// Check if roll number exists in set
int contains(int roll) {
    for (int i = 0; i < size; i++)
       if (set[i] == roll) return 1;
    return 0;
}

// Add roll number (only if not present)
void add(int roll) {
    if (!contains(roll)) {
       set[size++] = roll;
       printf("Added %d\n", roll);
    } else {
        printf("%d already exists. Duplicate ignored.\n", roll);
    }
}

// Display all roll numbers
void display() {
    printf("\nRoll Numbers in Set: { ");
    for (int i = 0; i < size; i++)
        printf("%d ", set[i]);
    printf("}\n");
}

int main() {
  int rolls[] = {101, 102, 103, 104, 102, 105, 101, 106};
  int n = sizeof(rolls)/sizeof(rolls[0]);
  for (int i = 0; i < n; i++) add(rolls[i]);
  display();
  return 0;
}
```

OUTPUT

```
Roll Numbers in Set: { 101 102 103 104 105 106 }
```

JAVA LANGUAGE

Using HashSet (Built-in)

```java
import java.util.HashSet;
import java.util.Set;

public class StudentSetExample {
    public static void main(String[] args) {
        Set<Integer> rollSet = new HashSet<>();

        // Roll numbers (with duplicates)
        int[] rolls = {101, 102, 103, 104, 102, 105, 101, 106};

        // Add all roll numbers to set
        for (int roll : rolls) {
            if (rollSet.add(roll))
                System.out.println(roll + " added to set");
            else
                System.out.println(roll + " already exists.");
        }

        System.out.println("\nRoll Numbers in Set: " + rollSet);
    }
}
```

OUTPUT (Sample)

```
101 added to set
102 added to set
103 added to set
104 added to set
102 already exists.
105 added to set
101 already exists.
106 added to set

Roll Numbers in Set: [101, 102, 103, 104, 105, 106]
```

PYTHON LANGUAGE

Using set (Built-in)

```python
# Roll numbers (with duplicates)
rolls = [101, 102, 103, 104, 102, 105, 101, 106]

# Add to set (duplicates automatically removed)
roll_set = set()

for roll in rolls:
    if roll in roll_set:
        print(f"{roll} already exists.")
    else:
        roll_set.add(roll)
        print(f"{roll} added to set")

print("\nRoll Numbers in Set: ",  roll_set)
```

OUTPUT (Sample)

```
101 added to set
102 added to set
103 added to set
104 added to set
102 already exists.
105 added to set
101 already exists.
106 added to set

Roll Numbers in Set: {101, 102, 103, 104, 105, 106}
```

STUDENT ROLL NUMBERS – VISUAL EXAMPLE

Input Roll Numbers (With Duplicates)

101	102	103	104	102	105	101	106

↓

Set (Duplicates Removed Automatically)

101	102	103	104	105	106

Order may vary (in Java/Python sets).

REAL-LIFE ANALOGY

Think of a class attendance list recorded in a register.
If a student's roll number is already written,
we don't write it again. Only unique roll numbers remain.

OPERATIONS IN SET

Operation	Description	C (Custom)	Java (HashSet)	Python (set)
Add	Add a roll number	add(roll)	add(roll)	add(roll)
Remove	Remove a roll number	remove(roll)	remove(roll)	remove(roll)
Contains	Check if roll exists	contains(roll)	contains(roll)	in operater
Size	Number of elements	size	size()	len(set)
Display	Show all elements	display()	print(set)	print(roll_set)
Duplicates	Allowed?	No	No	No
Order	Maintains insertion order?	Yes (if array)	No (unordered)	No (unordered)

COMPARISON SUMMARY

Feature	C (Custom Set)	Java (HashSet)	Python (set)
Implementation	Manual (Array)	Built-in Class	Built-in Type
Uniqueness	✓	✓	✓
Performance (Add/Search)	O(n)	O(1) Average	O(1) Average
Order	Depends on array	Unordered	Unordered
Ease of Use	★★☆☆☆	★★★★☆	★★★★☆

COMMON REAL-WORLD USES

- Unique student roll numbers
- Unique email IDs in a system
- Unique usernames
- Removing duplicates from data
- Set of books in a library

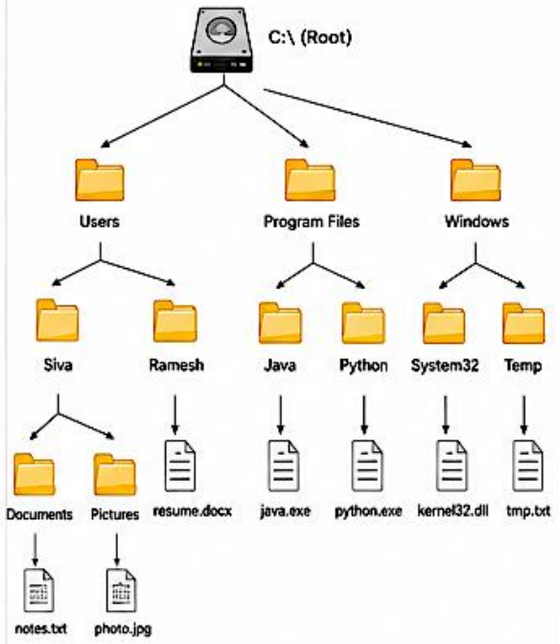
KEY TAKEAWAYS

- A Set stores only unique values.
- Duplicate roll numbers are ignored.
- Best data structure when uniqueness matters.
- HashSet (Java) and set (Python) provide very fast operations.

TREE EXAMPLE – FILES STRUCTURE

A tree is a hierarchical (parent-child) data structure.
Example: Files and Folders in a Computer

FILES STRUCTURE (ANALOGY)



Key Points:

- C:\ is the root of the tree.
- Folders are internal nodes.
- Files are leaf nodes.
- Each item (except root) has exactly one parent.
- A folder can have zero or more sub-folders/files (children).

C LANGUAGE (Using Structure)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAX_CHILDREN 5
#define MAX_NAME 50

typedef struct TreeNode {
    char name[MAX_NAME];
    struct TreeNode* children[MAX_CHILDREN];
    int childCount;
} TreeNode;

// Create a new node
TreeNode* createNode(char* name) {
    TreeNode* node = (TreeNode*)malloc(sizeof(TreeNode));
    strcpy(node->name, name);
    node->childCount = 0;
    for (int i = 0; i < MAX_CHILDREN; i++)
        node->children[i] = NULL;
    return node;
}

// Add child to parent
void addChild(TreeNode* parent, TreeNode* child) {
    if (parent->childCount < MAX_CHILDREN)
        parent->children[parent->childCount++] = child;
}

// Display tree
void printTree(TreeNode* root, int level) {
    if (root == NULL) return;
    for (int i = 0; i < level, i++) printf("    ");
        printf("- %s\n", root->name);
        for (int i = 0; i < root->childCount; i++)
            printTree(root->children[i], level + 1);
}
```

C OUTPUT (Tree View)

```
- C:\ (Root)
  - Users
    - Siva
      - Documents
        - notes.txt
      - Pictures
        - photo.jpg
    - Ramesh
      - resume.docx
  - Program Files
    - Java
      - java.exe
    - Python
      - python.exe
  - Windows
    - System32
      - kernel32.dll
    - Temp
      - tmp.txt
```

JAVA (Using Class)

```java
import java.util.*;

class TreeNode {
  String name;
  List<TreeNode> children;

  TreeNode(String name) {
    this.name = name;
    this.children = new ArrayList<>();
  }
}

public class FileStructureTree {
 static void addChild(TreeNode parent, TreeNode child) {
   parent.children.add(child);
 }

 static void printTree(TreeNode root, int level) {
   if (root == null) return;
   for (int i = 0; i < level; i++)
       System.out.print("    ");
     System.out.println("- " + root.name);
   for (TreeNode child : root.children)
       printTree(child, level + 1);
 }

public static void main(String[] args) {
  TreeNode root = new TreeNode("C:\\ (Root)");

  TreeNode users = new TreeNode("Users");
  TreeNode prog = new TreeNode("Program Files");
  TreeNode win = new TreeNode("Windows");
```

JAVA OUTPUT (Tree View)

```
- C:\ (Root)
  - Users
    - Siva
      - Documents
        - notes.txt
      - Pictures
        - photo.jpg
    - Ramesh
      - resume.docx
  - Program Files
    - Java
      - java.exe
    - Python
      - python.exe
  - Windows
    - System32
      - kernel32.dll
    - Temp
      - tmp.txt
```

PYTHON (Using Class)

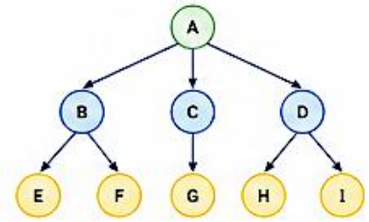
```python
class TreeNode:
    def __init__(self, name):
        self.name = name
        self.children = []

def add_child(parent, child):
    parent.children.append(child)

def print_tree(root, level=0):
    if root is None:
        return
    print("    " * level + "- " + root.name)
    for child in root.children:
        print_tree(child, level + 1)

# Build the file structure tree
root = TreeNode('C:\\ (Root)')
users = TreeNode('Users')
prog = TreeNode('Program Files')
win = TreeNode('Windows')
add_child(root, users)
add_child(root, prog)
add_child(root, win)
# ... (add remaining nodes similarly)

print_tree(root)
```

PYTHON OUTPUT (Tree View)

```
- C:\ (Root)
  - Users
    - Siva
      - Documents
        - notes.txt
      - Pictures
        - photo.jpg
    - Ramesh
      - resume.docx
  - Program Files
    - Java
      - java.exe
    - Python
      - python.exe
  - Windows
    - System32
      - kernel32.dll
    - Temp
      - tmp.txt
```

UNDERSTANDING THE TREE

Generic Tree Representation



Terminologies

- Root : Topmost node (C:\)
- Parent : A node that has children
- Child : A node directly below a parent
- Leaf Node : Node with no children (files)
- Internal Node : Node with at least one child (folders)
- Level : Depth of a node (Root level = 0)
- Height of Tree : Longest path from root to leaf

ADVANTAGES OF TREE

- Represents hierarchical data naturally.
- Easy to navigate parent-child relationships.
- Efficient for searching in hierarchical data.
- Used in file systems, databases, XML, organization charts, etc.

REAL-WORLD USES

- File Systems (Windows, Linux)
- Organization Hierarchy
- HTML DOM Structure
- Database Indexing
- XML/JSON Parsing

TREE OPERATIONS (COMMON)

Operation	Description
Create Node	Create a new node
Add Child	Add a child to a parent
Traverse	Visit all nodes (Preorder, Postorder, etc.)
Search	Search a node
Delete	Delete a node (and its subtree)

COMPARISON SUMMARY

Feature	C (Using Structure)	Java (Using Class)	Python (Using Class)
Implementation	Manual using structs & pointers	Built-in collections (ArrayList)	Dynamic lists
Syntax Simplicity	Complex	Simple	Very Simple
Memory Management	Manual	Automatic (Garbage Collector)	Automatic (Garbage Collector)
Flexibility	Less Flexible	More Flexible	Most Flexible
Best For	System Programming	Enterprise Applications	Scripting, AI, Data Science

KEY TAKEAWAYS

- Tree is perfect for hierarchical data like file systems.
- Folders can have many sub-folders, but each has only one parent.
- Files are the leaf nodes (no children).
- We can represent trees in C (structures), Java (classes), and Python (classes) easily.
- Trees help keep data organized and easy to access.

BINARY SEARCH – EXAMPLE

Binary Search works on **SORTED** data.

It repeatedly divides the search range in half. If the middle element is the target, return it.
If not, search in the left half or right half depending on the comparison.

TIME COMPLEXITY

Best / Avg / Worst

O(log n)

Very Efficient for large data!

EXAMPLE DATA (Sorted Array)

```
arr = [10, 23, 35, 47, 59, 62, 73, 88, 94]
```

We will search for **Target = 62**

STEP-BY-STEP SEARCH PROCESS

1. Start: low = 0, high = 8
   mid = (0 + 8) // 2 = 4 → arr[4] = 59
   59 < 62 → Search in right half (5 to 8)

10	23	35	47	59	62	73	88	94
0	1	2	3	4	5	6	7	8
				low				high

2. Now: low = 5, high = 8
   mid = (5 + 8) // 2 = 6 → arr[6] = 73
   73 > 62 → Search in left half (5 to 5)

10	23	35	47	59	62	73	88	94
0	1	2	3	4	5	6	7	8
					low	mid		high

3. Now: low = 5, high = 5
   mid = (5 + 5) // 2 = 5 → arr[5] = 62
   62 == 62 → Found at index 5

10	23	35	47	59	62	62	88	94
0	1	2	3	4	5	6	7	8
					low	mid	high	

RESULT: Target 62 found at index 5.

C LANGUAGE

```c
#include <stdio.h>

int binarySearch(int arr[], int n, int target) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;  // Not found
}

int main() {
    int arr[] = {10, 23, 35, 47, 59, 62, 73, 88, 94};
    int n = sizeof(arr) / sizeof(arr[0]);
    int target = 62;
    int result = binarySearch(arr, n, target);
    if (result != -1)
        printf("%d found at index %d\n", target, result);
    else
        printf("%d not found in the array\n", target);
    return 0;
}
```

C OUTPUT (Sample)

```
62 found at index 5
```

JAVA LANGUAGE

```java
public class BinarySearchExample {
    public static int binarySearch(int[] arr, int target){
        int low = 0, high = arr.length - 1;
        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (arr[mid] == target) return mid;
            else if (arr[mid] < target)
                low = mid + 1;
            else
                high = mid - 1;
        }
        return -1;  // Not found
    }

    public static void main(String[] args) {
        int[] arr = {10, 23, 35, 47, 59, 62, 73, 88, 94};
        int target = 62;
        int result = binarySearch(arr, target);
        if (result != -1)
            System.out.println(target + " found at index " + result);
        else
            System.out.println(target + " not found in the array");
    }
}
```

JAVA OUTPUT (Sample)

```
62 found at index 5
```

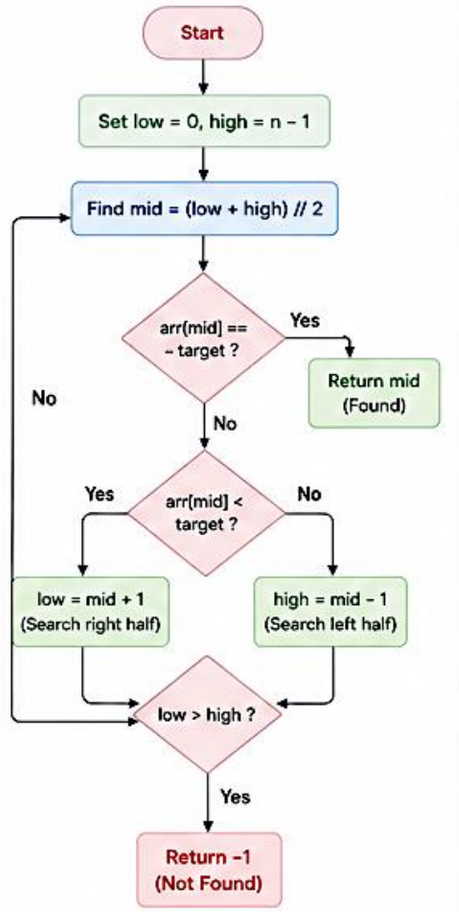
PYTHON LANGUAGE

```python
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1  # Not found


arr = [10, 23, 35, 47, 59, 62, 73, 88, 94]
target = 62
result = binary_search(arr, target)

if result != -1:
    print(f"{target} found at index {result}")
else:
    print(f"{target} not found in the array")
```

PYTHON OUTPUT (Sample)

```
62 found at index 5
```

HOW BINARY SEARCH WORKS



KEY POINTS

- Binary Search works only on sorted data.
- Compare the target with the middle element.
- Eliminate half of the data in each step.
- Repeat until the element is found or range is empty.
- Much faster than Linear Search for large datasets.

COMPARISON: LINEAR SEARCH vs BINARY SEARCH

Feature	Linear Search	Binary Search
Data Requirement	Works on any data	Requires sorted data
Approach	Check one by one	Divide & conquer (half each time)
Time Complexity	O(n)	O(log n)
Best For	Small / Unsorted data	Large / Sorted data
Comparisons (for n items)	Up to n	Up to $\log_2 n$

WHEN TO USE BINARY SEARCH?

- When data is sorted.
- When you need fast searches in large datasets.
- In systems where data doesn't change often (sorted once, searched many times).

EXAMPLE PRACTICE

Use the array:

```
arr = [10, 23, 35, 47, 59, 62, 73, 88, 94]
```

Try searching for:

10 **47** **73** **94** **20**

(20 is not in the array)

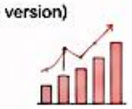
COMPLEXITY SNAPSHOT

Best Case : O(1) (If middle is the target)

Average Case : O(log n)

Worst Case : O(log n)

Space Complexity : O(1) (Iterative version)



EMPLOYEES (Unsorted)		
ID	Name	Salary (₹)
101	CEO	250000
102	CTO	200000
103	Manager	150000
104	Team Lead	120000
105	Developer	90000
106	Analyst	80000
107	Intern	40000

HEAP EXAMPLE – SALARY HIERARCHY

A Heap is a complete binary tree that follows the **Heap Property**.
We use a **Max Heap** to organize employees by salary (highest salary at the top).

GOAL **GOAL**

Organize employees to a hierarchy based on salary.
This helps in quickly finding the highest paid employee.

HEAP PROPERTY (Max Heap)

For every node:

Salary of parent ≥ Salary of children

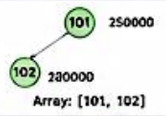
Example: 250000 ≥ 200000 and 150000

1. BUILD MAX HEAP (BY SALARY)

Insert employees one by one and maintain Max Heap property.

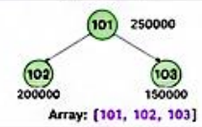
Insert 101 (250000)

Insert 102 (200000)



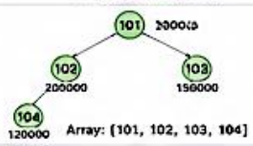
Array: [101, 102]

Insert 103 (150000)



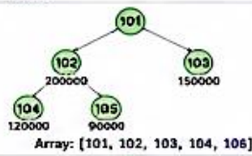
Array: [101, 102, 103]

Insert 104 (120000)



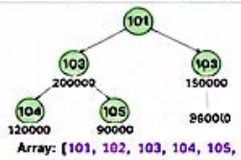
Array: [101, 102, 103, 104]

Insert 105 (90000)



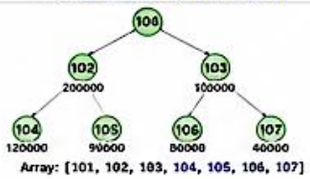
Array: [101, 102, 103, 104, 105]

Insert 106 (80000)



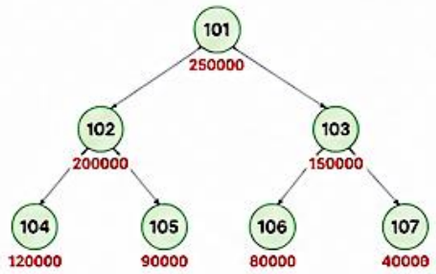
Array: [101, 102, 103, 104, 105, 106]

Insert 107 (40000)



Array: [101, 102, 103, 104, 105, 106, 107]

2. FINAL MAX HEAP (Salary Hierarchy)



Heap Array Representation (Level Order)

101	102	103	104	105	106	107
250000	200000	150000	120000	90000	80000	40000

3. OPERATIONS EXAMPLE

A. Peek (Get Highest Salary)

- Return the root.
- Output: 101 (CEO) with ₹250000

B. Extract-Max (Remove Highest Salary)

1. Remove root (101).
2. Move last element (107) to root.
3. Heapify down to maintain Max Heap.

New Heap Array: [102, 104, 103, 107, 105, 106]

C. Insert (Add New Employee)

1. Add at the end.
2. Heapify up.

Example: Insert 108 (160000)
New Heap Array: [102, 104, 108, 107, 105, 106, 103]
(108 moves up above 103 and below 104)

4. IMPLEMENTATION – ARRAY BASED HEAP

C (Structure + Functions)

```c
#include <stdio.h>
#define MAX 100

typedef struct {
   int id;
   int salary;
} Employee;

Employee heap[MAX];
int size = 0;

void swap(int i, int j){
   Employee temp = heap[i];
   heap[i] = heap[j];
   heap[j] = temp;
}

int parent(int i){ return (i-1)/2; }
int left(int i){ return 2*i+1; }
int right(int i){ return 2*i+2; }

void heapifyUp(int i){
   while(i>0 && heap[parent(i)].salary < heap[i].salary){
      swap(i, parent(i));
      i = parent(i);
   }
}

void heapifyDown(int i){
   int l = left(i), r = right(i), largest = i;
   if(l < size && heap[l].salary > heap[largest].salary)
      largest = l;
   if(r < size && heap[r].salary > heap[largest].salary)
      largest = r;
   if(largest != i){ swap(i, largest); heapifyDown(largest);}
}

void insert(Employee e){
   heap[size] = e;
   heapifyUp(size);
   size++;
}

Employee extractMax(){
   Employee root = heap[0];
   heap[0] = heap[size-1];
   size--;
   heapifyDown(0);
   return root;
}
```

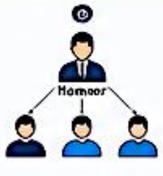
Java

```java
class Employee {
   int id;
   int salary;
   Employee(int id, int salary){
      this.id=id; this.salary=salary;
   }
}

class MaxHeap {
   Employee[] heap = new Employee[100];
   int size = 0;

   int parent(int i){ return (i-1)/2; }
   int left(int i){ return 2*i+1; }
   int right(int i){ return 2*i+2; }

   void swap(int i, int j){
      Employee t = heap[i];
      heap[i] = heap[j];
      heap[j] = t;
   }

   void insert(Employee e){
      heap[size] = e;
      heapifyUp(size);
      size++;
   }

   void heapifyUp(int i){
      while(i>0 && heap[parent(i)].salary < heap[i].salary){
         swap(i, parent(i));
         i = parent(i);
      }
   }

   void heapifyDown(int i){
      int l=left(i), r=right(i), largest=i;
      if(l<size && heap[l].salary>heap[largest].salary) largest=l;
      if(r<size && heap[r].salary>heap[largest].salary) largest=r;
      if(largest!=i){ swap(i,largest); heapifyDown(largest); }
   }

   Employee extractMax(){
      Employee root = heap[0];
      heap[0] = heap[size-1];
      size--;
      heapifyDown(0);
      return root;
   }
```

Python

```python
class MaxHeap:
   def __init__(self):
      self.heap = []

   def parent(self, i):
      return (i-1)//2

   def left(self, i):
      return 2*i+1

   def right(self, i):
      return 2*i+2

   def swap(self, i, j):
      self.heap[i], self.heap[j] = self.heap[j], self.heap[i]

   def insert(self, emp):
      self.heap.append(emp)
      self.heapify_up(len(self.heap)-1)

   def heapify_up(self, i):
      while i>0 and self.heap[self.parent(i)]['salary'] <
            self.heap[i]['salary']:
         self.swap(i, self.parent(i))
         i = self.parent(i)

   def heapify_down(self, i):
      l, r, largest = self.left(i), self.right(i), i
      if l < len(self.heap) and self.heap[l]['salary'] >
         self.heap[largest]['salary']:
            largest = l
      if r < len(self.heap) and self.heap[r]['salary'] >
         self.heap[largest]['salary']:
            largest = r
      if largest != i:
         self.swap(i, largest)
         self.heapify_down(largest)

   def extract_max(self):
      if not self.heap:
         return None
      root = self.heap[0]
      self.heap[0] = self.heap.pop()
      self.heapify_down(0)
      return root
```

5. REAL-LIFE ANALOGY



CEO has highest salary (top).
Below are managers, then team leads, developers, analysts, interns.
The organization is a hierarchy just like a Max Heap!

6. COMPLEXITY ANALYSIS

Operation	Time Complexity (n = number of employees)
Insert	O(log n)
Extract-Max	O(log n)
Peek (Get Max)	O(1)
Build Heap (n inserts)	O(n log n)
Build Heap (Bottom-Up)	O(n)

7. USE CASES

- Finding highest paid employee quickly
- Top K salary reports
- Job scheduling (highest priority first)
- Resource allocation
- Task scheduling in systems

8. KEY TAKEAWAYS

- Heap is a complete binary tree.
- Max Heap keeps highest salary at root.
- Efficient for retrieving highest/lowest priority.
- Array representation is memory efficient.
- Insert and Extract-Max take O(log n) time.

9. SAMPLE HEAP (Salary Order)

Level	Employee ID	Name	Salary (₹)
0	101	CEO	250000
1	102	CTO	200000
1	103	Manager	150000
2	104	Team Lead	120000
2	105	Developer	90000
2	106	Analyst	80000
2	107	Intern	40000

Google Maps

SAMPLE PLACES (NODES)		
ID	Place	Lat, Long (Approx)
A	Home	12.9716, 77.5946
B	Office	12.9352, 77.6245
C	Mall	12.9340, 77.6100
D	Airport	12.9777, 77.5991
E	Railway St.	12.9629, 77.6408
F	Hospital	12.9165, 77.6100
G	Park	12.9460, 77.5726

GRAPH EXAMPLE – GOOGLE MAPS (SHORTEST PATH)

Google Maps finds the fastest/shortest route between two places using Graph algorithms.

We will use Dijkstra's Algorithm to find the shortest path from Home (A) to Airport (D).

GOAL

Find the shortest path (least time/distance) from Home (A) to Airport (D).

GRAPH MODEL (Google Maps View)

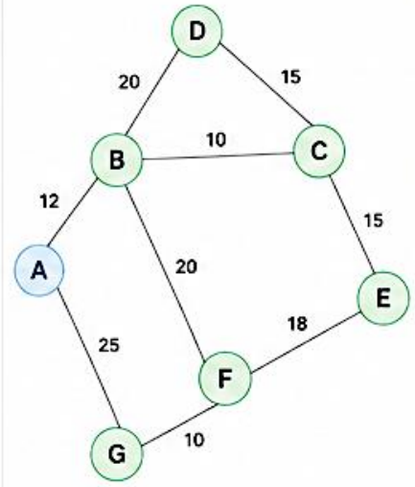
- Intersections / Places → Nodes (A, B, C, D, E, F, G)
- Roads / Connections → Edges (with weight: time in minutes)
- Traffic / Congestion → Affects weights

Example: A → B (12 min), B → D (20 min)

1. REAL MAP VIEW (Example)

Blue line shows one possible route from Home (A) to Airport (D).

2. GRAPH REPRESENTATION (Weighted)



Numbers on edges = Travel time (minutes)

3. DIJKSTRA'S ALGORITHM STEPS (A → D)

Step	Visited (Set)	Current Node	Update to Neighbors (Distance in min)
0	{A}	A (0)	B(12), G(25)
1	{A, B}	B (12)	D(32), C(22), F(32)
2	{A, B, C}	C (22)	E(37)
3	{A, B, C, D, F}	D (32)	(Destination reached) ✅

Shortest Time from A (Home) to D (Airport) = 32 min

Path: A → B → D

4. CODE EXAMPLE – DIJKSTRA'S ALGORITHM

C (GCC)

```c
#include <stdio.h>
#include <limits.h>
#define V 7  // Number of vertices

int minDistance(int dist[], int sptSet[]) {
   int min = INT_MAX, min_index;
   for (int v = 0; v < V; v++)
      if (!sptSet[v] && dist[v] <= min)
         min = dist[v], min_index = v;
   return min_index;
}

void dijkstra(int graph[V][V], int src) {
   int dist[V], sptSet[V];
   for (int i = 0; i < V; i++) {
      dist[i] = INT_MAX;
      sptSet[i] = 0;
   }
   dist[src] = 0;

   for (int count = 0; count < V - 1; count++) {
    int u = minDistance(dist, sptSet);
    sptSet[u] = 1;
    for (int v = 0; v < V; v++)
       if (!sptSet[v] && graph[u][v] &&
           dist[u] != INT_MAX &&
           dist[u] + graph[u][v] < dist[v])
          dist[v] = dist[u] + graph[u][v];
   }

   printf("Vertex\tDistance from Source\n");
   for (int i = 0; i < V; i++)
      printf("%d\t%d\n", i, dist[i]);
}
/* Call dijkstra(graph, 0); // 0 represents A */
```

Java

```java
import java.util.*;

public class Dijkstra {
   static final int V = 7;

   static int minDistance(int[] dist, boolean[] sptSet) {
      int min = Integer.MAX_VALUE, min_index = -1;
      for (int v = 0; v < V; v++)
         if (!sptSet[v] && dist[v] <= min)
            min = dist[v]; min_index = v;
      return min_index;
   }

   static void dijkstra(int[][] graph, int src) {
      int[] dist = new int[V];
      boolean[] sptSet = new boolean[V];
      Arrays.fill(dist, Integer.MAX_VALUE);
      dist[src] = 0;
      for (int count = 0; count < V - 1; count++) {
         int u = minDistance(dist, sptSet);
         sptSet[u] = true;
         for (int v = 0; v < V; v++)
            if (!sptSet[v] && graph[u][v] != 0 &&
                dist[u] != Integer.MAX_VALUE &&
                dist[u] + graph[u][v] < dist[v])
               dist[v] = dist[u] + graph[u][v];
      }

      System.out.println("Vertex\tDistance from Source");
      for (int i = 0; i < V; i++)
         System.out.println(i + "\t" + dist[i]);
   }
   // Call dijkstra(graph, 0); // 0 represents A
}
```

Python 3

```python
import heapq

def dijkstra(graph, src):
    V = len(graph)
    dist = [float('inf')] * V
    dist[src] = 0
    pq = [(0, src)]  # (distance, node)

    while pq:
        d, u = heapq.heappop(pq)
        if d > dist[u]:
            continue

        for v, w in enumerate(graph[u]):
            if w and dist[u] + w < dist[v]:
                dist[v] = dist[u] + w
                heapq.heappush(pq, (dist[v], v))

    print("Vertex\tDistance from Source")
    for i in range(V):
        print(f"{i}\t{dist[i]}")


# Call dijkstra(graph, 0)  # 0 represents A
```

In above code: graph[u][v] = 0 means no direct road between u and v.

5. GRAPH (ADJACENCY MATRIX)

From \ To	A	B	C	D	E	F	G
A	0	12	0	0	0	0	25
B	12	0	10	20	0	20	0
C	0	10	0	15	15	0	0
D	0	20	15	0	0	0	0
E	0	0	15	0	0	18	0
F	0	20	0	0	18	0	10
G	25	0	0	0	0	10	0

0 means no direct connection.

6. ROUTE OPTIONS (Google Maps Style)

Route 1 (Fastest)	A → B → D	32 min
Route 2	A → B → C → D	37 min
Route 3	A → G → F → B → D	52 min
Route 4	A → G → F → E → C → D	60 min

Google Maps suggests Route 1 as the fastest.

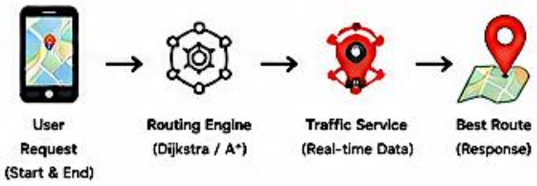
7. TIME COMPLEXITY

Algorithm	Time Complexity
Dijkstra (Adjacency Matrix)	O(V^2)
Dijkstra (Adjacency List + Min Heap)	O((V + E) log V)

V = Number of nodes (places)

E = Number of edges (roads)

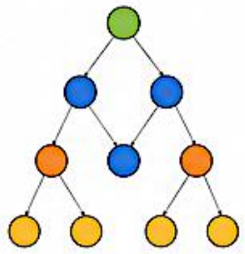
8. GOOGLE MAPS ARCHITECTURE (Simplified)



Uses Graph + Real-time data to provide best routes.

9. KEY TAKEAWAYS

- Google Maps models places as nodes and roads as edges.
- Edge weights represent travel time (changes with traffic).
- Dijkstra's Algorithm finds the shortest/fastest route.
- Efficient for non-negative weights (time/distance).
- Used in navigation, logistics, delivery apps, transport systems.



WORDS TO INSERT	
ID	Word
1	cat
2	car
3	care
4	card
5	cart
6	dog

TRIE (PREFIX TREE) EXAMPLE – STRING STORAGE & AUTOCOMPLETE

A Trie is a tree data structure used to store strings (words) in a way that allows fast search, insert and prefix based retrieval (perfect for Autocomplete).

GOAL

Store words and support:

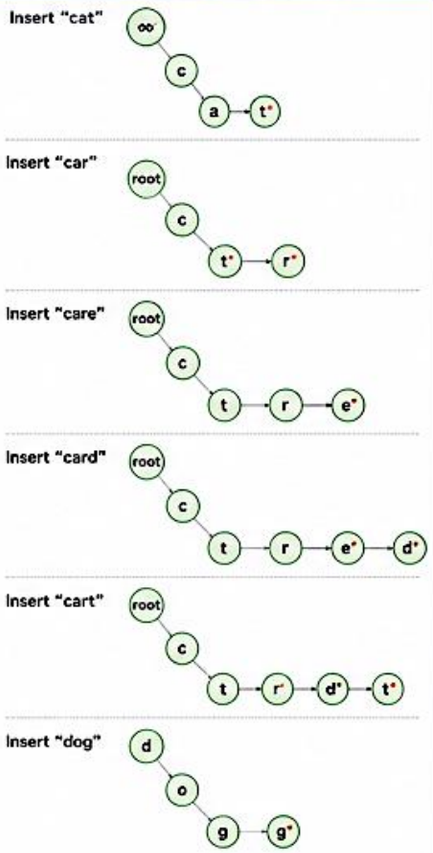
- ☑ Exact Search
- ☑ Prefix Search (Autocomplete)
- ☑ Efficient Insert

TRIE NODE STRUCTURE

Each node contains:

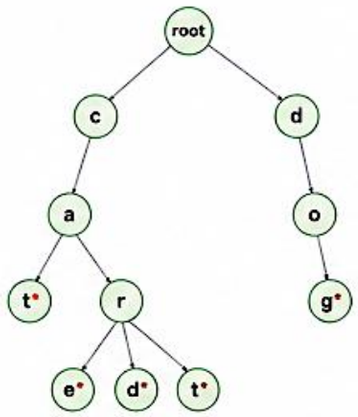
- Children pointers (a-z) / Map
- EndOfWord flag (true/false)
- Root node represents empty string

1. INSERTION (STEP BY STEP)



* = End of Word

2. TRIE AFTER ALL INSERTIONS



WORDS STORED

cat, car, care, card, cart, dog

3. OPERATIONS EXAMPLE

A. Search (Exact Match)

Search "care" → Found
Path: c → a → r → e (End = true)
Search "can" → Not Found
Path breaks at 'n' (No such child)

B. Prefix Search (Autoconoplete)

Prefix: "ca"
All words with prefix "ca" are:
cat, car, care, card, cart

C. Starts With

StartsWith("do") → True
StartsWith("de") → False

4. IMPLEMENTATION – ARRAY BASED TRIE

C (GCC)

```c
#include <stdio.h>
#include <string.h>
#define ALPHABET 26

typedef struct TrieNode {
  struct TrieNode *child[ALPHABET];
  int isEnd;
} TrieNode;

TrieNode* createNode() {
  TrieNode* node = (TrieNode*)malloc
  (sizeof(TrieNode));
  node->isEnd = 0;
  for (int i = 0; i < ALPHABET; i++)
     node->child[i] = NULL;
  return node;
}

void insert(TrieNode* root, char* key){
  TrieNode* p = root;
  for (int i = 0; key[i]; i++){
    int idx = key[i] - 'a';
    if (!p->child[idx])
      p->child[idx] = createNode();
    p = p->child[idx];
  }
  p->isEnd = 1;
}

int search(TrieNode* root, char* key){
  TrieNode* p = root;
  for (int i = 0; key[i]; i++){
    int idx = key[i] - 'a';
    if (!p->child[idx]) return 0;
    p = p->child[idx];
  }
) return p->isEnd;
```

Java

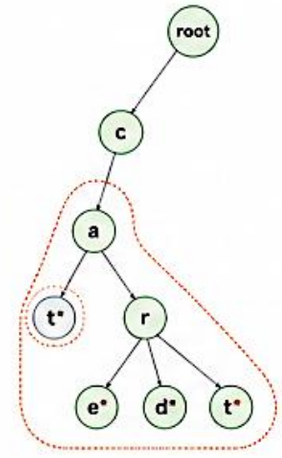
```java
class TrieNode {
  TrieNode[] child = new TrieNode[26];
  boolean isEnd;
}

class Trie {
  TrieNode root;

 Trie(){
    root = new TrieNode();
 }

 void insert(String key){
   TrieNode node = root;
   for (char ch : key.toCharArray()){
     int idx = ch - 'a';
     if (node.child[idx] == null)
       node.child[idx] = new TrieNode();
     node = node.child[idx];
   }
   node.isEnd = true;
 }

 boolean search(String key){
   TrieNode node = root;
   for (char ch : key.toCharArray()){
     int idx = ch - 'a';
     if (node.child[idx] == null) return false;
     node = node.child[idx];
   }
   return node.isEnd;
 }
```

Python 3

```python
class TrieNode:
  def __init__(self):
     self.children = {}
     self.isEnd = False

class Trie:
  def __init__(self):
     self.root = TrieNode()

  def insert(self, key):
     node = self.root
     for ch in key:
        if ch not in node.children:
           node.children[ch] = TrieNode()
           node = node.children[ch]
     node.isEnd = True

  def search(self, key):
     node = self.root
     for ch in key:
        if ch not in node.children:
           return False
        node = node.children[ch]
     return node.isEnd

  def startsWith(self, prefix):
     node = self.root
     for ch in prefix:
        if ch not in node.children:
           return False
        node = node.children[ch]
     return True
```

5. PREFIX SEARCH (AUTOCOMPLETE)

Prefix: "ca"



Suggestions (All words with prefix "ca")

cat
car
care
card
cart

How it works:

1. Traverse till prefix "ca".
2. Collect all words in the subtree.

6. TRIE VS OTHER DATA STRUCTURES (FOR SEARCH)

Data Structure	Exact Search	Prefix Search	Insert	Space	Best Use Case
Array / List	O(n)	O(n)	O(1)	Low	Small data, simple storage
Binary Search Tree	O(log n)*	Hard	O(log n)*	Medium	Ordered data
Hash Table	O(1)*	Not Possible	O(1)*	Medium	Exact match search
Trie (Prefix Tree)	O(L)	O(L + k)	O(L)	High	Autocomplete, Dictionary

n = number of words, L = length of word, k = number of results

7. TIME COMPLEXITY

Operation	Time Complexity
Insert	O(L)
Search (Exact)	O(L)
Search (Prefix)	O(L + k)
StartsWith	O(L)
Space Complexity	O(ALPHABET * N * L)

N = number of words, L = average length

8. REAL LIFE USE CASES

- Google Search Autocomplete (Type "ca" → cat, car, card, camera...)
- Mobile Keyboard Suggestions
- Spell Checker / Autocorrect
- Dictionary / Word Games
- IP Routing (Longest Prefix Match)
- File System / Path Lookup
- Contact List Search

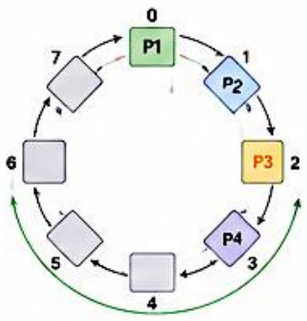
9. KEY TAKEAWAYS

- ✅ Trie stores strings character by character.
- ✅ Common prefixes are shared.
- ✅ Fast search and autocomplete.
- ✅ Ideal for prefix related operations.
- ✅ More memory, but much faster for prefix.

10. SAMPLE OUTPUT

After inserting: cat, car, care, card, cart, dog

Search "card" → Found
Search "can" → Not Found
Prefix "ca" → cat, car, care, card, cart
Prefix "do" → dog
StartsWith "ca" → True
StartsWith "de" → False



CIRCULAR QUEUE EXAMPLE – CPU SCHEDULING (ROUND ROBIN)

A Circular Queue is a linear data structure in which the last position is connected back to the first position.
It is widely used in CPU Scheduling (Round Robin Algorithm) to manage ready processes in a cyclic manner.

INITIAL PROCESSES

Process	Arrival Time	Burst Time
P1	0	8
P2	1	4
P3	2	9
P4	3	5
P5	4	2

GOAL

Efficiently manage ready processes using Circular Queue in Round Robin Scheduling (Time Quantum = 4) and compute Gantt Chart, Waiting Time and Turnaround Time.

WHY CIRCULAR QUEUE?

- ✔ Utilizes full space (no wastage of empty slots)
- ✔ Efficient enqueue (add process at rear)
- ✔ Efficient dequeue (remove process from front)
- ✔ Ideal for cyclic scheduling like Round Robin

CIRCULAR QUEUE OPERATIONS

- Enqueue (Insert) – Add process to rear
- Dequeue (Remove) – Remove process from front
- isEmpty() – Check if queue is empty
- isFull() – Check if queue is full

1. CIRCULAR QUEUE WORKING

1.1 Initial Empty Queue (Size = 5)

0	1	2	3	4
-	-	-	-	-

Front = -1 Rear = -1

1.2 After Enqueue P1, P2, P3

0	1	2	3	4
P1	P2	P3	-	-

Front = 0 Rear = 2

1.3 After Dequeue (Remove P1)

0	1	2	3	4
-	P2	P3	-	-

Front = 1 Rear = 2

1.4 After Enqueue P4, P5 (Wrap Around)

0	1	2	3	4
P5	P2	P3	P4	-

Front = 1 Rear = 4

1.5 After Enqueue P6 (Wrap to index 0)

0	1	2	3	4
P6	P2	P3	P4	P5

Front = 1 Rear = 0

Next Enqueue will check (rear + 1) % size == front
If true → Queue is Full

2. CPU SCHEDULING (ROUND ROBIN) USING CIRCULAR QUEUE

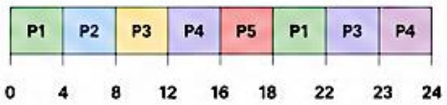
2.1 Round Robin (Time Quantum = 4)

- Add all arrived processes to the queue.
- Take process from front, execute for min(burst, quantum).
- If process not finished, add back to rear.
- Repeat until queue becomes empty.

2.2 Step by Step Execution

Step	Queue (Front → Rear)	Running Process	Exec Time	Remaining Burst
0	[P1]	-	-	-
1	[P1, P2]	P1	4	P1:4
2	[P2, P3]	P2	4	P2:0 (Done)
3	[P3, P4]	P3	4	P3:5
4	[P4, P5, P1]	P4	4	P4:1
5	[P5, P1, P3]	P5	2	P5:0 (Done)
6	[P1, P3, P4]	P1	4	P1:0 (Done)
7	[P3, P4]	P3	4	P3:1
8	[P4, P3]	P4	1	P4:0 (Done)
9	[P3]	P3	1	P3:0 (Done)

2.3 Gantt Chart



2.4 Final Statistics

Process	Burst Time	Completion Time	Turnaround Time (CT - AT)	Waiting Time (TAT - BT)
P1	8	18	18 - 0 = 18	18 - 8 = 10
P2	4	8	8 - 1 = 7	7 - 4 = 3
P3	9	24	24 - 2 = 22	22 - 9 = 13
P4	5	23	23 - 3 = 20	20 - 5 = 15
P5	2	18	18 - 4 = 14	14 - 2 = 12

Average Waiting Time = (10 + 3 + 13 + 15 + 12) / 5 = 10.6

Average Turnaround Time = (18 + 7 + 22 + 20 + 14) / 5 = 16.2

3. IMPLEMENTATION – CIRCULAR QUEUE

C (GCC)

```c
#include <stdio.h>
#define SIZE 5

typedef struct {
    int arr[SIZE];
    int front, rear;
} CircularQueue;

void init(CircularQueue *q){
    q->front = -1;
    q->rear = -1;
}

int isEmpty(CircularQueue *q){
  return (q->front == -1);
}

int isFull(CircularQueue *q){
  return ((q->rear + 1) % SIZE == q->front);
}

void enqueue(CircularQueue *q, int value){
  if (isFull(q)) return;
  if (isEmpty(q)) q->front = 0;
  q->rear = (q->rear + 1) % SIZE;
  q->arr[q->rear] = value;
}

int dequeue(CircularQueue *q){
  if (isEmpty(q)) return -1;
  int val = q->arr[q->front];
  if (q->front == q->rear) q->front = q->rear =
   -1;
  else q->front = (q->front + 1) % SIZE;
  return val;
}
```

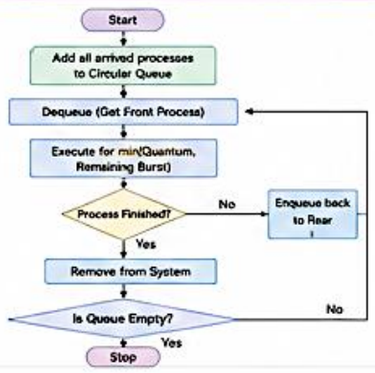
Java

```java
class CircularQueue {
    int arr[];
    int front, rear, size;
  CircularQueue(int n){
     size = n;
     arr = new int[size];
     front = rear = -1;
  }

  boolean isEmpty(){
     return front == -1;}
  boolean isFull(){
     return (rear + 1) % size == front;
  }

  void enqueue(int x){
    if (isFull()) return;
    if (isEmpty()) front = 0;
    rear = (rear + 1) % size;
     arr[rear] = x;
  }

  int dequeue(){
    if (isEmpty()) return -1;
    int x = arr[front];
    if (front == rear) front = rear = -1;
    else front = (front + 1) % size;
    return x;

  }
```

Python 3

```python
class CircularQueue:
  def __init__(self, n):
      self.arr = [0]*n
      self.size = n
      self.front = -1
      self.rear = -1

  def is_empty(self):
      return self.front == -1

  def is_full(self):
      return (self.rear + 1) % self.size ==
      self.front

  def enqueue(self, x):
      if self.is_full(): return
      if self.is_empty():
          self.front = 0
      self.rear = (self.rear + 1) % self.size
      self.arr[self.rear] = x

  def dequeue(self):
      if self.is_empty(): return -1
      x = self.arr[self.front]
      if self.front == self.rear:
          self.front = self.rear = -1
      else:
          self.front = (self.front + 1) % self.size
      return x
```

4. CPU SCHEDULING FLOW (ROUND ROBIN)



5. EXAMPLE WITH DIFFERENT TIME QUANTUM

Time Quantum = 2

Process	Gantt Chart	Avg Waiting Time	Avg Turnaround Time
P1	P1 P2 P3 P4 P5 P1 P3 P4 P3 P1 P3	12.2	17.8
P1	0 2 4 6 8 10 12 14 16 18 20 22 24		

Time Quantum = 3

Process	Gantt Chart	Avg Waiting Time	Avg Turnaround Time
P1	P1 P2 P3 P4 P5 P1 P3 P4 P3	11.0	16.0
0	3 6 9 12 14 17 20 23 24		

Time Quantum = 5

Process	Gantt Chart	Avg Waiting Time	Avg Turnaround Time
P1	P1 P2 P3 P4 P5 P3 P3	10.4	15.6
0	5 5 9 14 16 21 24		

6. TIME COMPLEXITY

Operation	Time Complexity
Enqueue (Insert)	O(1)
Dequeue (Remove)	O(1)
Peek (Front Element)	O(1)
isEmpty()	O(1)
isFull()	O(1)

All operations are constant time making it efficient for scheduling.

7. REAL LIFE USE CASES

- CPU Scheduling (Round Robin)
- Memory Management (Page Replacement)
- I/O Buffering in Operating Systems
- Network Packet Scheduling
- Printer Spooling
- Real-time Systems

"Circular Queue helps in cyclic management of resources fairly and efficiently.

8. KEY TAKEAWAYS

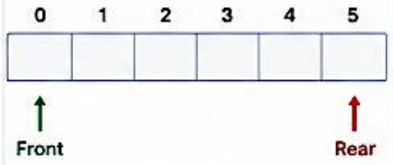
- Circular Queue is an extension of Queue.
- It uses modulo (%) to wrap around.
- No wasted space like simple queue.
- Perfect for cyclic processes like Round Robin Scheduling.
- All operations are O(1) (constant time).

DEQUE (DOUBLE ENDED QUEUE) EXAMPLE

A Deque (Double Ended Queue) is a linear data structure that allows insertion and deletion at both front and rear ends.

Examples: Task Scheduler, Sliding Window Maximum, Undo-Redo, Palindrome Checker

DEQUE VISUAL (Capacity = 6)



WHY DEQUE?

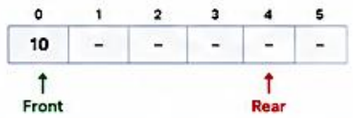
- Insertion/Deletion at both ends – O(1)
- More flexible than Queue & Stack
- Efficient for many real life problems

COMMON OPERATIONS

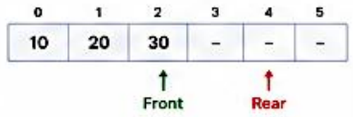
Operation	Description
insertFront(x)	Add element at front
insertRear(x)	Add element at rear
deleteFront()	Remove element from front
deleteRear()	Remove element from rear
getFront()	Get front element
getRear()	Get rear element
isEmpty()	Check if deque is empty
isFull()	Check if deque is full

1. BASIC OPERATIONS EXAMPLE

1.1 insertRear(10)



1.2 insertRear(20), insertRear(30)

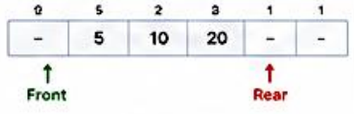


1.3 insertFront(5)

1.4 insertFront(2)

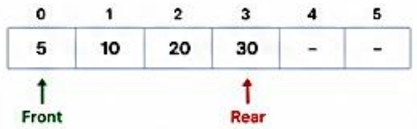
1.5 deleteRear() → removes 30

1.6 deleteFront() → removes 2



2. WRAP AROUND IN DEQUE (Circular Deque)

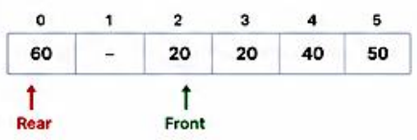
2.1 Suppose capacity = 6



2.2 deleteFront() twice

2.3 insertRear(40), insertRear(50), insertRear(60)

2.4 insertRear(60) → Wrap around to index 0



Now deque is full (Front at 2, Rear at 0 in circular manner)

3. REAL LIFE EXAMPLE – TASK SCHEDULER

A CPU Task Scheduler keeps tasks in a deque.
High priority tasks are added at front, normal tasks at rear.

Step by Step Simulation

Step	Operation	Deque (Front → Rear)	Explanation
1	insertRear(T1)	[T1]	Add T1 to rear
2	insertRear(T2)	[T1, T2]	Add T2 to rear
3	insertFront(H1)	[H1, T1, T2]	High priority H1 added at front
4	deleteFront()	[T1, T2]	Execute H1
5	insertFront(H2)	[H2, T1, T2]	High priority H2 added Last
6	deleteFront()	[T1, T2]	Execute H2
7	deleteFront()	[T2]	Execute T1
8	deleteFront()	[]	Execute T2

Deque helps in handling both high priority (front) and normal (rear) tasks efficiently.

4. IMPLEMENTATION – ARRAY DEQUE (Circular)

C (GCC)

```c
#include <stdio.h>
#define MAX 6

typedef struct {
   int arr[MAX];
   int front, rear, size;
} Deque;

void initDeque(Deque *dq){
    dq->front = -1;
    dq->rear  = -1;
    dq->size  = 0;
    dq->size
}

int isEmpty(Deque *dq){
    return (dq->size == 0);
}

int isFull(Deque *dq){
    return (dq->size == MAX);
}

void insertRear(Deque *dq,  int x){
   if(isFull(dq)) return;
   if(dq->rear == MAX-1) dq->rear = 0;
   else dq->rear++;
   if(dq->front == -1) dq->front = dq->rear;
   dq->arr[dq->rear] = x;
   dq->size++;
}

void insertFront(Deque *dq,  int x){
   if(isFull(dq)) return;
   if(dq->front == 0) dq->front = MAX-1;
   else dq->front--;
   dq->arr[dq->front] = x;
   dq->size++;
   if(dq->rear == -1) dq->rear = dq->front;
}

int deleteFront(Deque *dq){
   if(isEmpty(dq)) return -1;
   int x = dq->arr[dq->front];
   if(dq->front == dq->rear{
      dq->front = dq->rear = -1;
   }else{
      dq->front = (dq->freat + 1) % MAX;
   }
   dq->size--;
   return x;
}
```

Java

```java
class Deque {
   int arr[];
   int front, rear, size, cap;
   Deque(int c){
      cap = c;
      arr = new  int[cap];
      front = rear = -1;
      size = 0;
   }

boolean isEmpty(){ return size==0; }
boolean isFull(){ return size==cap; }

void insertRear(int x){
   if(isFull()) return;
   if(rear==cap-1) rear=0;
   else rear++;
   if(front==-1) front=rear;
   arr[rear]=x; size++;
}

void insertFront(int x){
   if(isFull()) return;
   if(front==0) front=cap-1;
   else front--;
   arr[front]=x; size++;
   if(rear==-1) rear=front;
}

int deleteFront(){
   if(isEmpty()) return -1;
   int x = arr[front];
   if(front==rear{ front=rear=-1; }
   else front=(front+1)%cap;
   size--; return x;
}

int deleteRear(){
   if(isEmpty()) return -1;
   int x = arr[rear];
   if(front==rear{ front=rear=-1; }
   else rear=(rear-1+cap)%cap;
   size--; return x;

}
```

Python 3

```python
class Deque:
   def __init__(self, n):
      self.arr = [0]*n
      self.front = -1
      self.rear = -1
      self.size = 0
      self.n = n

   def isEmpty(self):
      return  self.size == 0

   def isFull(self):
      return  self.size == self.n

   def insertRear(self, x):
      if self.isFull():  return
      if self.rear == self.n-1:
         self.rear = 0
      else:
         self.rear  += 1
      if self.front == -1:
         self.front = self.rear
      self.arr[self.rear] = x
      self.size += 1

   def insertFront(self, x):
      if self.isFull():  return
      if self.front == 0:
         self.front = self.n-1
      else:
         self.front -= 1
      self.arr[self.front] = x
      self.size += 1
      if self.rear == -1:
         self.rear = self.front

   def deleteFront(self):
      if self.isEmpty():  return -1
      x = self.arr[self.front]
      if self.front == self.rear:
         self.front = self.rear = -1
      else:
         self.front = (self.front + 1) % self.n
      self.size -= 1
      return x

   def deleteRear(self):
      if self.isEmpty():  return -1
      x = self.arr[self.rear]
      if self.front == self.rear:
         self.front = self.rear = -1
      else:
         self.rear = (self.rear - 1 + self.n) % self.n
      self.size -= 1
      return x
```

5. APPLICATIONS OF DEQUE

- **CPU Task Scheduling (Round Robin)** – Add new tasks at rear, urgent tasks at front.
- **Browser History** – Back/Forward navigation
- **Undo / Redo Operation** – Undo at front, Redo at rear
- **Sliding Window Maximum/Minimum**
- **Palindrome Checking** – Compare from both ends

6. COMPARISON – QUEUE, STACK vs DEQUE

Feature	Queue	Stack	Deque
Insertion	Rear only	Top only	Both Front & Rear
Deletion	Front only	Top only	Both Front & Rear
Type	FIFO	LIFO	Both directions
Flexibility	Low	Low	High
Implementation	Array / Linked List	Array / Linked List	Array / Doubly Linked List
Applications	Simple scheduling	Function calls, undo	Scheduling, Undo-Redo, Sliding Window

7. TIME COMPLEXITY

Operation	Time Complexity
insertFront(x)	O(1)
insertRear(x)	O(1)
deleteFront()	O(1)
deleteRear()	O(1)
getFront()	O(1)
getRear()	O(1)

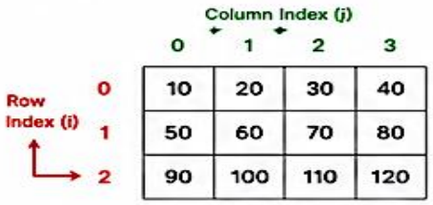
All operations are O(1) (constant time)

8. KEY TAKEAWAYS

- Deque allows operations at both ends.
- Efficient with circular array (no shifting).
- Ideal for problems requiring access from both ends.
- Used in many real world and interview problems.

9. SAMPLE I/O (DEQUE OPERATIONS)

Operation	Deque (Front → Rear)	Output
insertRear(10)	[10]	-
insertRear(20)	[10, 20]	-
insertFront(5)	[5, 10, 20]	-
deleteRear()	[5, 10]	20
deleteFront()	[10]	5
insertFront(8)	[8, 10]	-



Example:

A[1][2] = 70

A[2][1] = 100

MATRIX (2D ARRAY) EXAMPLES

A Matrix is a 2-Dimensional Array arranged in rows and columns.
Element at row i and column j is represented as **A[i][j]**.

KEY POINTS

- Matrix = m rows × n columns
- Total Elements = m × n
- 0 ≤ i < m (rows) and 0 ≤ j < n (columns)
- Used in real life for tables, images, graphs, games, spreadsheets, etc.

1. MATRIX REPRESENTATION

A matrix of 3 rows and 4 columns (3x4)

$$A = \begin{bmatrix} 10 & 20 & 30 & 40 \\ 50 & 60 & 70 & 80 \\ 90 & 100 & 110 & 120 \end{bmatrix}$$

Row 0 → 10 20 30 40
Row 1 → 50 60 70 80
Row 2 → 90 100 110 120

Column 0 → 10 50 90
Column 1 → 20 60 100
Column 2 → 30 70 110
Column 3 → 40 80 120

2. MEMORY LAYOUT (ROW MAJOR ORDER)

Matrix A (3x4)

10	20	30	40
50	60	70	80
90	100	110	120

↓

Stored in Memory (Row Major)

10	20	30	40	50	60	70	80	90	100	110	120
Index 0	2	3	4	5	6	7	8	9	10	11	

Address Formula (Row Major):
Address of A[i][j] = Base + (i × N + j) × Size
where N = number of columns

3. TYPES OF MATRICES (EXAMPLES)

3.1 Square Matrix (3x3)

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

3.2 Diagonal Matrix

$$\begin{bmatrix} 5 & 0 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 3 \end{bmatrix}$$

3.3 Identity Matrix (I)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

3.4 Upper Triangular

$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{bmatrix}$$

3.5 Lower Triangular

$$\begin{bmatrix} 1 & 0 & 0 \\ 2 & 3 & 0 \\ 4 & 5 & 6 \end{bmatrix}$$

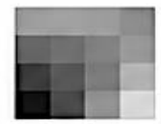
3.6 Symmetric Matrix ($A^T = A$)

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 4 & 5 \\ 3 & 5 & 6 \end{bmatrix}$$

4. REAL LIFE EXAMPLES

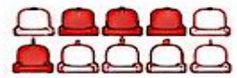
4.1 Image (Pixel Matrix - Grayscale 4x4)

120	130	140	150
110	125	135	145
100	115	128	138
90	105	118	130



4.2 Seats in a Cinema (0 = Empty, 1 = Booked)

0	1	0	0	1	0
0	0	1	1	0	0
1	0	0	0	1	1



4.3 Chess Board (Piece Codes)

(0=Empty, 1=Pawn, 2=Rook, 3=Knight, 4=Bishop, 5=Queen, 6=King)

2	3	4	5	6	4	3	2
1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
2	3	4	5	6	4	3	2

5. IMPLEMENTATION (2D ARRAY)

C

```c
#include <stdio.h>
#define R 3
#define C 4
int main() {
    int A[R][C], i, j;
    printf("Enter elements:\n");
    for(i=0;i<R;i++)
        for(j=0;j<C;j++)
            scanf("%d", &A[i][j]);
    printf("Matrix:\n");
    for(i=0;i<R;i++){
        for(j=0;j<C;j++)
            printf("%d\t", A[i][j]);
        printf("\n");
    }
    return 0;
}
```

Java

```java
import java.util.*;
class Main {
    public static void main(String[] args){
        int R=3, C=4;
        int A[][] = new int[R][C];
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter elements:");
        for(int i=0;i<R;i++)
            for(int j=0;j<C;j++)
                A[i][j] = sc.nextInt();
        System.out.println("Matrix:");
        for(int i=0;i<R;i++){
            for(int j=0;j<C;j++)
                System.out.print(A[i][j]+"\t");
            System.out.println();
        }
    }
}
```

Python

```python
R, C = 3, 4
A = []
print("Enter elements:")
for i in range(R):
    row = []
    for j in range(C):
        row.append(int(input()))
    A.append(row)
print("Matrix:")
for i in range(R):
    for j in range(C):
        print(A[i][j], end="\t")
    print()
```

6. COMMON OPERATIONS

Operation	Description	Example (A is 3x3)
Travesse	Visit each element	for i and j loops
Sum of Elements	Add all elements	Total = $\sum A[i][j]$
Row Sum	Sum of each row	Row 1 = $\sum_j A[1][j]$
Column Sum	Sum of each column	Col 2 = $\sum A[i][2]$
Transpose	Swap rows and columns	B[j][i] = A[i][j]
Matrix Addition	Add two matrices	C[i][j] = A[i][j] + B[i][j]
Matrix Multiplication	Multiply two matrices	C[i][j] = $\sum A[i][k] \times B[k][j]$

7. EXAMPLE – MATRIX ADDITION

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \quad B = \begin{bmatrix} 9 & 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix}$$

$$C = A + B = \begin{bmatrix} 10 & 10 & 10 \\ 10 & 10 & 10 \\ 10 & 10 & 10 \end{bmatrix}$$

8. EXAMPLE – TRANSPOSE

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

9. EXAMPLE – MATRIX MULTIPLICATION

A (2x3) = $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ B (3x2) = $\begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix}$

C = A × B (2x2) = $\begin{bmatrix} 58 & 64 \\ 139 & 154 \end{bmatrix}$

10. APPLICATIONS

- Image Processing (Pixel Matrices)
- Computer Graphics (Transformations)
- Scientific Computations
- Data Analysis & Machine Learning
- Spreadsheets (Rows & Columns)
- Game Development (Boards, Maps)

11. KEY TAKEAWAYS

- Matrix is a 2D array of size m x n.
- Elements are accessed using A[i][j].
- Used to represent real world data.
- Important in many algorithms and real life applications.