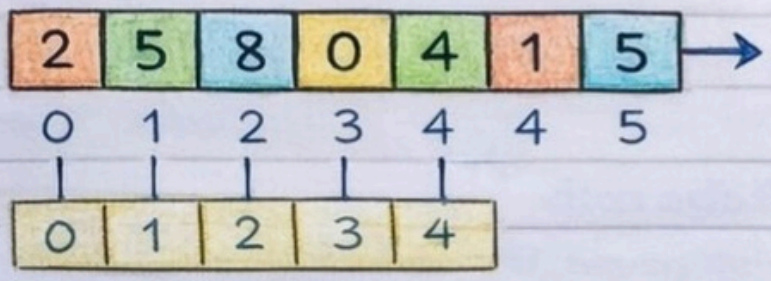
DSA NOTES

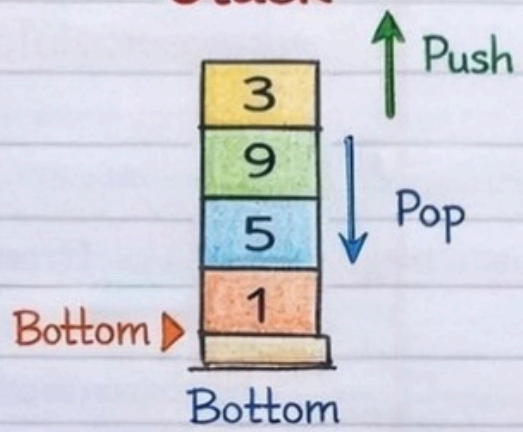
Complete DSA Notes - Part 1

@curious_programmer

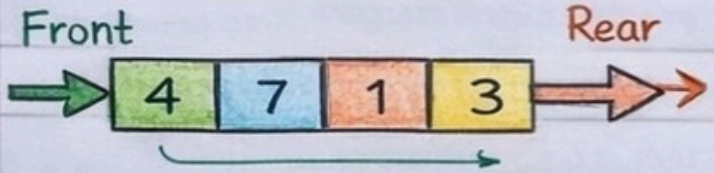
Array

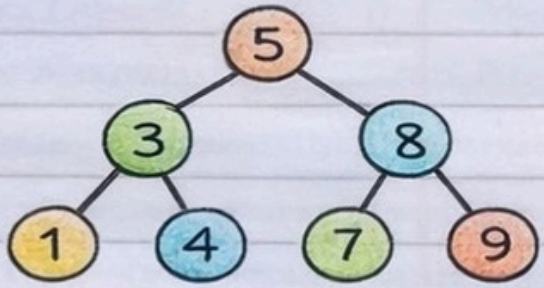


Stack

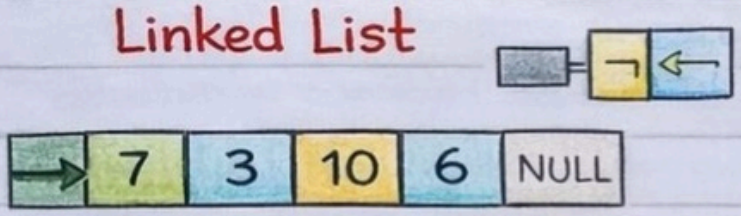


Queue



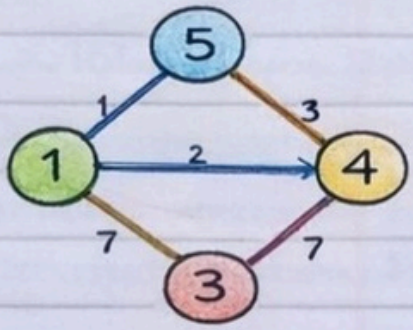


Linked List

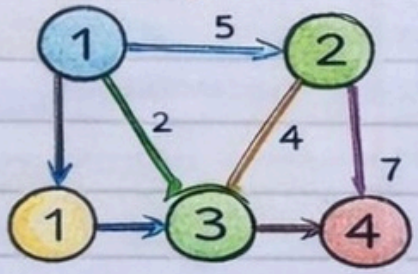


Linked List

Binary Tree



Graph

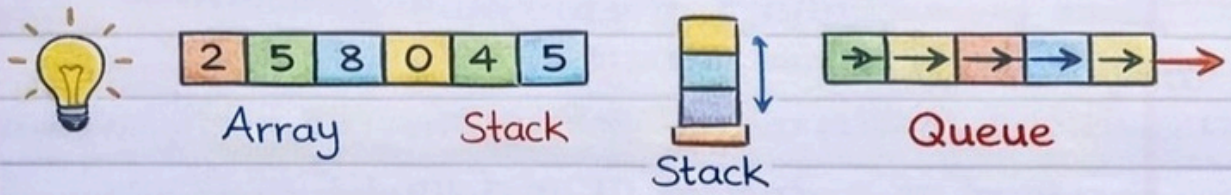


Chapter 1: Introduction to DSA

@curious_programmer

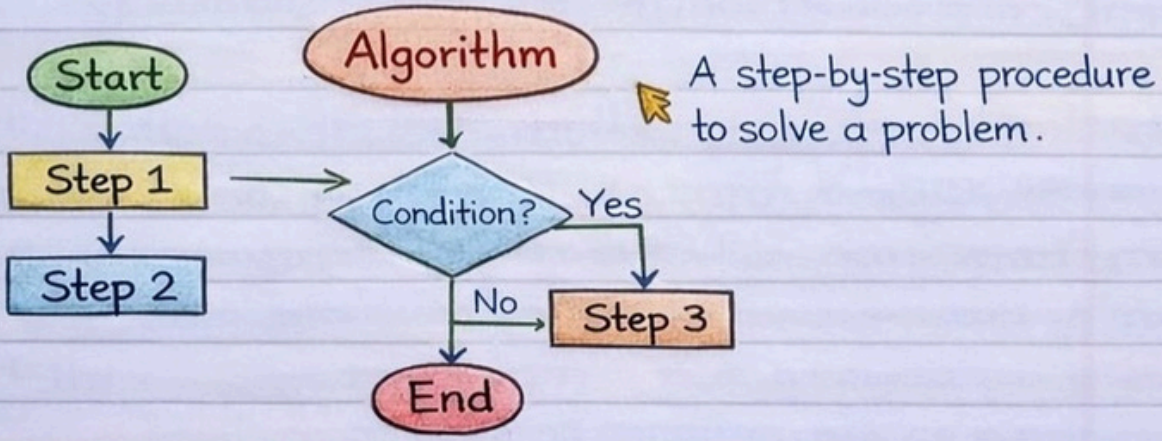
What is Data Structure?

A data structure is a way to store and organize data in a computer so that it can be used efficiently. Different data structures are designed for different kinds of data operations to make data processing more effective.



What is Algorithm?

An algorithm is a finite sequence of well-defined steps to solve a specific problem. It is like a recipe that defines the exact steps in a particular order.



Properties of Algorithms

- 1. **Finiteness** : Must terminate after a finite number of steps.
- 2 **Definiteness**: Steps must be precisely defined and unambiguous.
- 3. Input Output: Take zero or more inputs, produce at least one output.

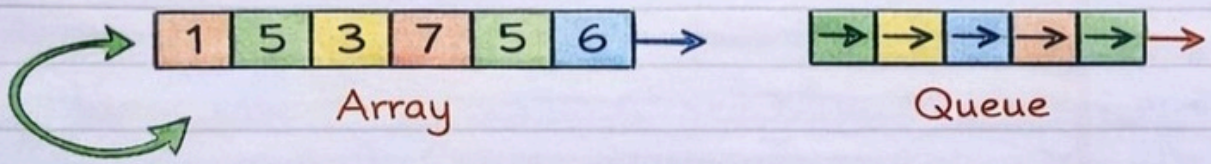
Why DSA is important?

@curious_programmer

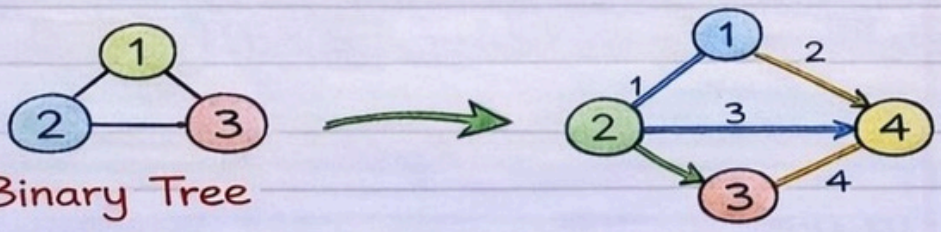
- **Efficient Problem Solving:** DSA provides tools to solve problems efficientty by choosing the right data structure and algorithm.
- **Optimal Resource Usage:** Helps use memory and time effectively by ensuring minimal resource consumption.
- **Coding Interviews & Competitions:** Crucial for coding interviews and programming contests as knowledge of DSA is essential for tackling challenging problems.

Types of Data Structures

- **Linear :** Linear data structures organize data in a sequential manner where each element is adjacent to the previous one.



- **Non-Linear :** Non-linear data structures organize data hierarchically, connecting elements in a more complex, branching manner.



- **Abstract Data Types (ADT)**

ADTs are defined by their operations, not by their implementation. Examples include:

List | Stack | Queue | Map | Set

Time & Space Complexity

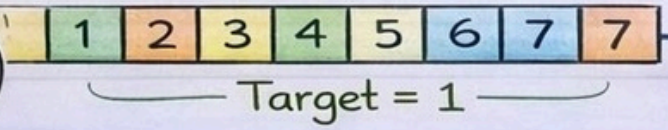
@curious_programmer

Why complexity analysis?

- **Evaluate Efficiency:** Complexity analysis helps in measuring an algorithm's efficiency by evaluating its running time (Time Complexity) and memory usage (Space Complexity).
- **Compare Algorithms:** Helps in comparing different algorithms to choose the most optimal one for a given problem.
- **Scalability:** Ensures that our algorithm can bandle large inputs efficientlty without degrading performance.

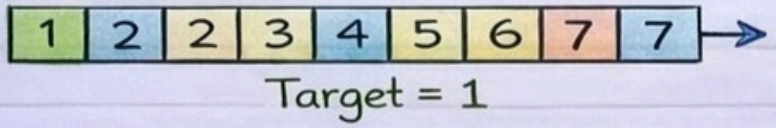
Best, Average & Worst Case

Algorithms are evaluated based on their performance in different scenarios:



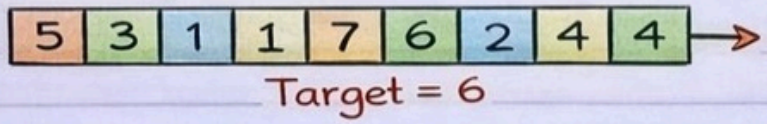
Ω(n)
Minimum time

Best Case



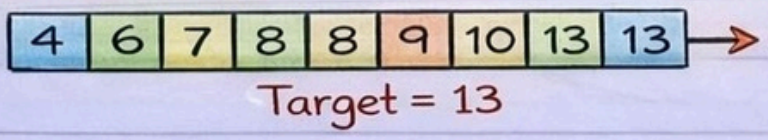
Ω(n)
Minimum time

Average Case



θ(n)
Expected time

Worst Case



O(n)
Maximum time

Big-O, Big-Ω, Big-Θ

@curious_programmer

✔ Why complexity analysis?

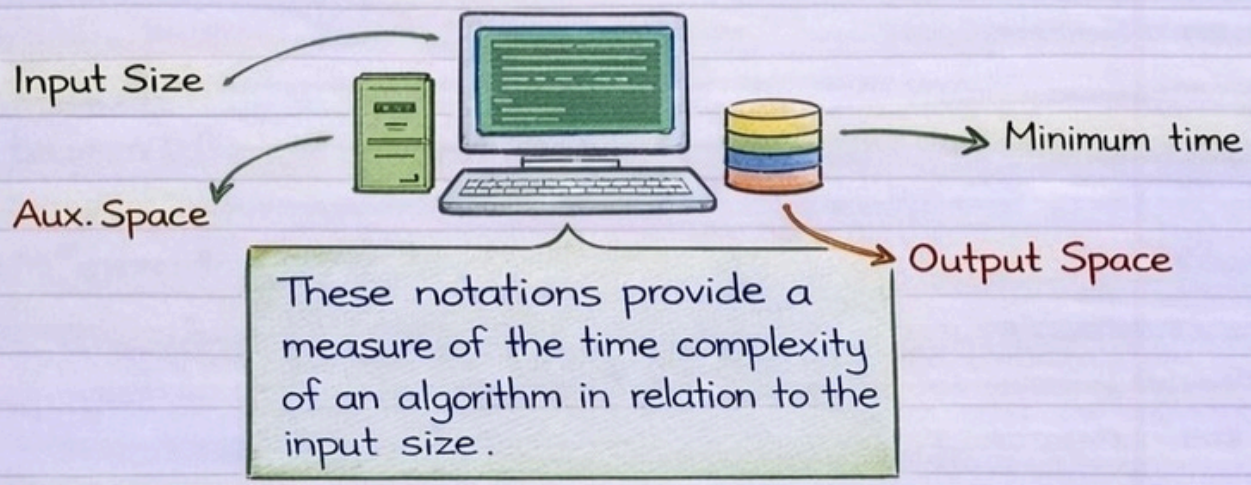
↑ Big-O	✔ Upper Bound	$f(n) \leq O(g(n)$
↓ Big-Ω	✔ Lower Bound	$f(n) \geq \Omega(g(n)$
↑ Big-Θ	✔ Tight Bound	$f(n) = \Theta(g(n)$

These notations provide a measure of the time complexity of an algorithm in relation to the input size.

Space Complexity

✔ What is Space Complexity?

Space Complexity measures the amount of memory an algorithm uses during its execution.



Space Complexity

✔ What is Space Complexity?

Space Complexity: measures the amount of memory an algorithm uses during its execution.

- Input Size: Memory needed to store the inputs.
- Aux. Space: (Aux.) Space: Extra memory needed apart
- Output Space: Memory used to store the output data.

Complexity of Common Loops & Recursion

@curious_programmer

✔ Complexity of Loops:

✔ Linear Loop:

```
for (int i = 0; i < n; i++)
    { ...
```

→ O(n)

✔ Nested Loop:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        { ...
    }
```

→ $O(n^2)$

✔ Logarithmic Loop:

```
for (int i = 1; i < n; i + = 2 )
    { ...
```

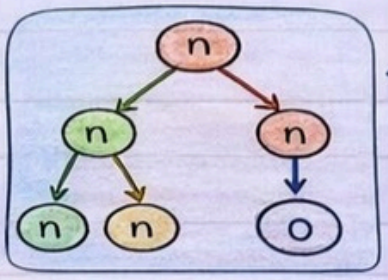
→ O(log n)

Complexity of Recursion

Algorithms are evaluated based on their performance in different scenarios:

✔ Linear Recursion

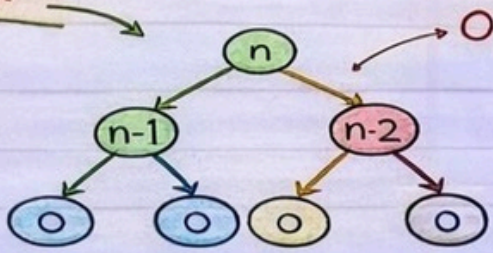
```
void recurse(int n)
    if (n ≤ 0) return;
        recurse(n - 1);
    }
}
```



- O(n)

✔ Binary (Exponential) Recursion

```
void recurse(int n)
    if (n ≤ 0) return;
        recurse(n - 1);
    }
}
```



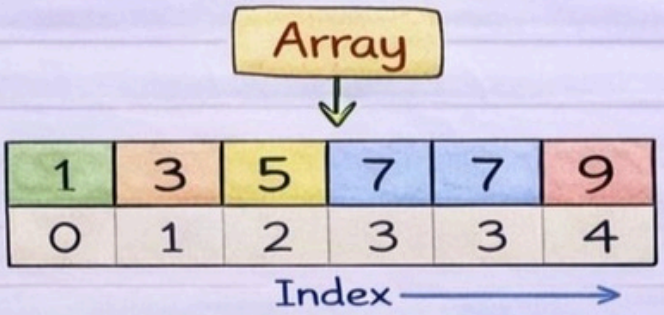
$O(2^n)$

Chapter 2: Arrays

@curious_programmer

2.1 Intrroduction to Arrays

- An array is a collection of elements of the same data type, stored in contiguous memory locations. It is a data structure that stores elements in a sequential manner.



Characteristics of Arrays

- **Fixed Size:** The size of an array is defined at the time of declaration and does not change.
- **Homogeneous Elements:** All elements in an array are of the same data type.
- **Contiguous (Sequential) Memory Locations:** Elements are stored in adjacent memory locations, allowing constant time access.

Properties of Arrays

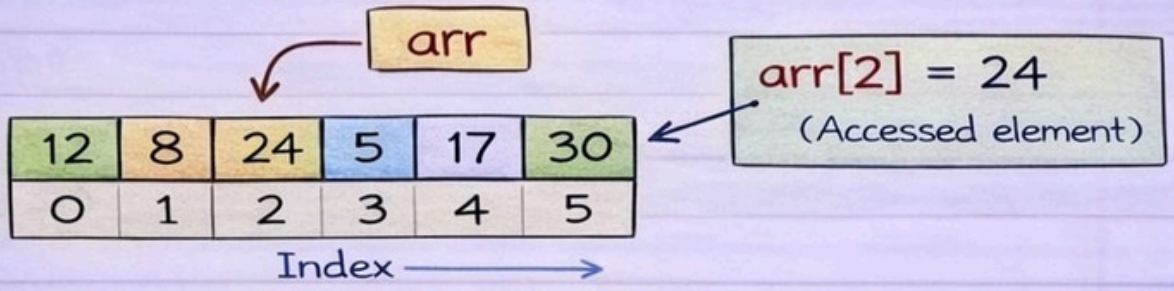
- **Constant Time Access → O(1):** Direct access to any element using its index.
- **Easy Traversal:** Simple to loop through all the elements using a for loop.
- **Effective Data Storage:** Useful for storing multiple values of the same type.

2.2 1D Arrays

@curious_programmer

2.2 1D Arrays

- A 1D array (or one-dimensional array) is a linear data structure that stores a collection of elements of the same data type in a single, contiguous block of memory.
- Each element in a 1D array is accessed using an index, which represents the element's position.



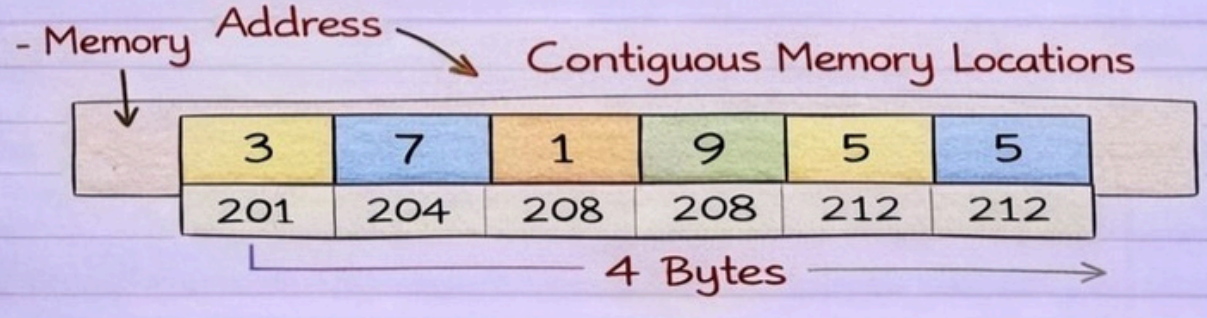
Declaration & Initialization

Syntax: `int arr[5];` Declares an integer array of size 5 without initializing elements.

Example: `int arr[5] = {3, 7, 1, 9, 5};` declares an integer array of size 5 and initializes it with the values 3, 7, 1, 9, 5.

Memory Representation

- The integer array is stored in a contiguous block of memory. Each element takes 4 bytes of memory (assuming 4 bytes for int).

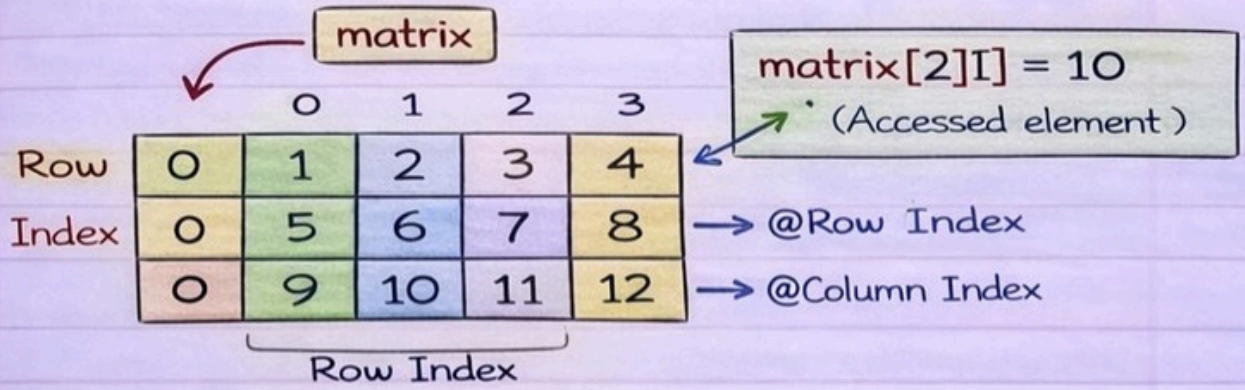


2.2 2D Arrays

@curious_programmer

2.2 2D Arrays

- A 2D array (or two-dimensional array) is a collection of elements arranged in rows and columns, forming a matrix-like structure.
- It allows storing data in a tabular format, with elements accessible using a pair of indices (row index and column index).



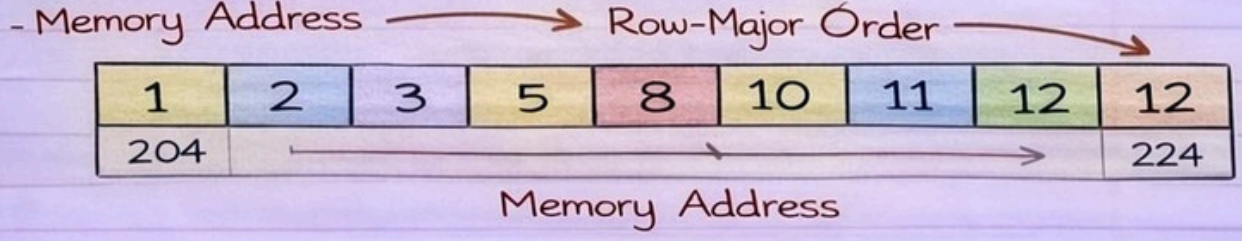
Declaration & Initialization

Syntax: `int matrix[3][4];` Declares a 3x4 integer array without initializing elements.

Example:
```
int matrix[3][4] = {1, 2, 3, 4},
                   {5, 6, 7, 8}, {9, 10, 11, 12};
```
Declares a 3x4 integer array initialized with the given values.

Memory Representation

- The 2D array is stored in row-major order, where rows are stored in contiguous memory locations.



2.4 Array Operations

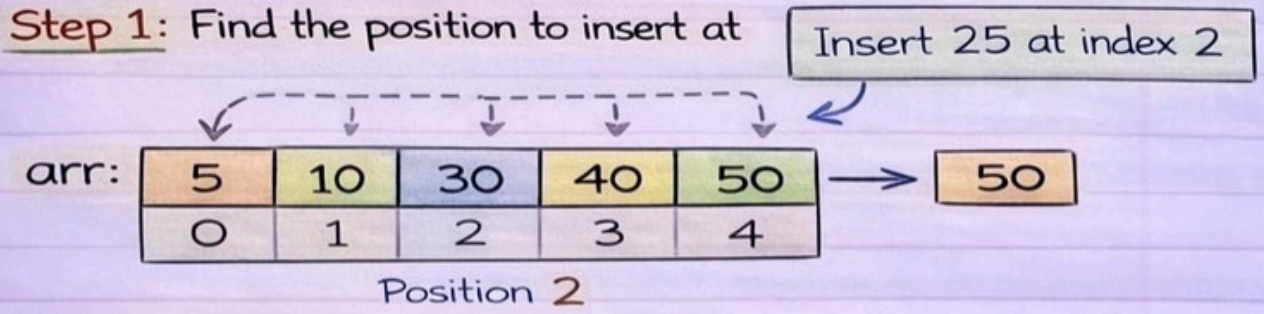
@curious_programmer

2.4.1 Insertion

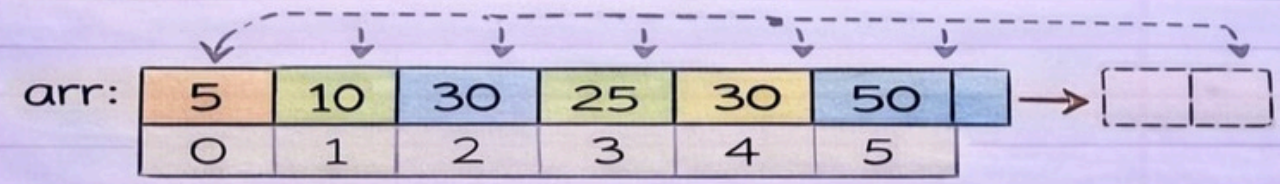
Insertion in 1D Arrays

- Array insertion involves adding a new element to a specific position within an array.
- This shifts the subsequent elements one position to the right to make space for the new element.

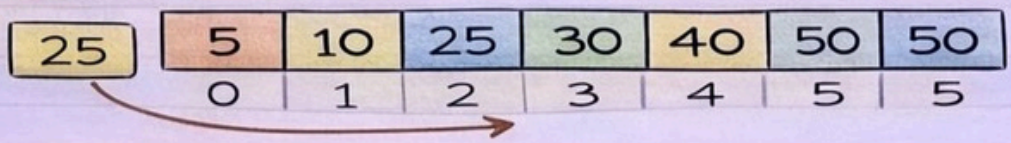
Step 1: Find the position to insert at



Step 2: Shift elements to the right



Step 3: Insert the new element at the found position



Properties of Insertion

- **Time Complexity → O(n):** In the worst case, all elements may need to be shifted, making it linear time.
- **Insertion at the end:** O(1) if there's space, as no shifting needed.
- **Full Array:** Insertion is not possible when the array is full and has no space for more elements.

2.4 Array Operations

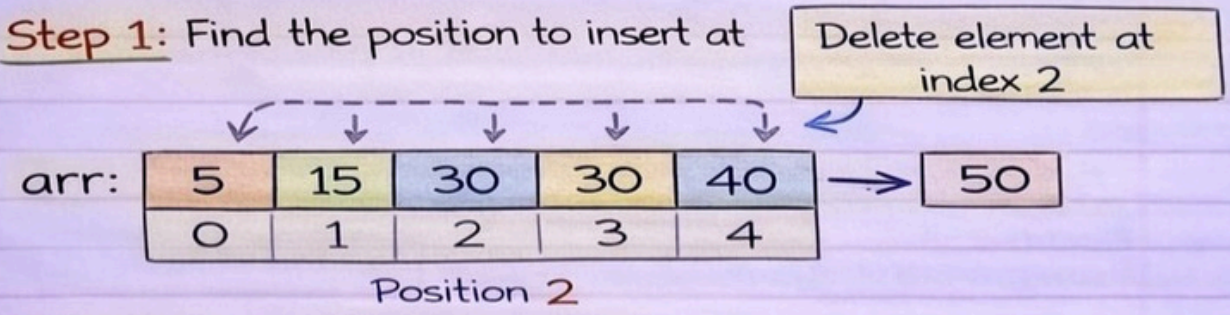
@curious_programmer

2.4.2 Deletion

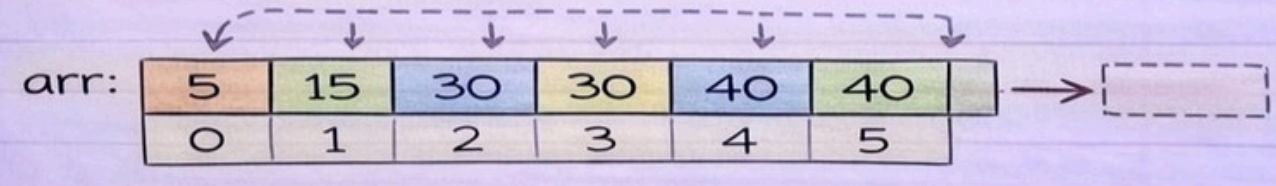
Deletion in 1D Arrays

- Array deletion involves removing an existing element from a specific position within an array.
- To fill the gap, the subsequent elements are shifted one position to the left.

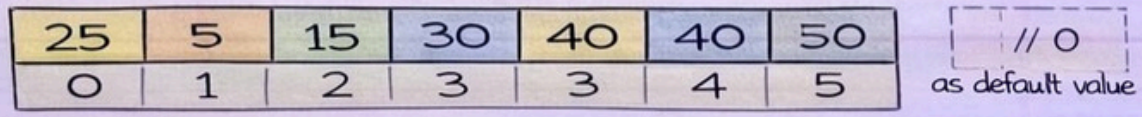
Step 1: Find the position to insert at



Step 2: Shift elements to the left



Step 3: Fill the last position with a default value (optional)



Properties of Deletion

- **Time Complexity → O(n):** In the worst case, all elements after the deleted position must be shifted.
- **Deletion from the end:** O(1) if no default value is needed, as no shifting is required.
- **Multiple Deletions:** Multiple deletions may cause fragmentation, leaving empty slots in the array.

2.5 Problems on Arrays

@curious_programmer

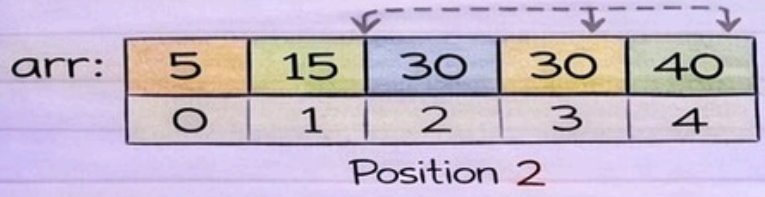
2.4.2 Deletion

Deletion in 1D Arrays

- Array deletion involves removing an existing element from a specific position within an array.
- To fill the gap, the subsequent elements are shifted one position to the left.

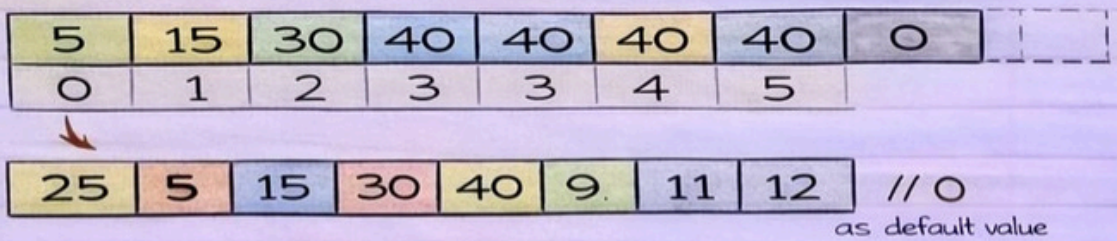
Step 1: Find the position to delete at

Delete element at index 2



Step 2: Shift elements to the left

Step 3: Merge Two Sorted Arrays



Tips for Solving Array Problems

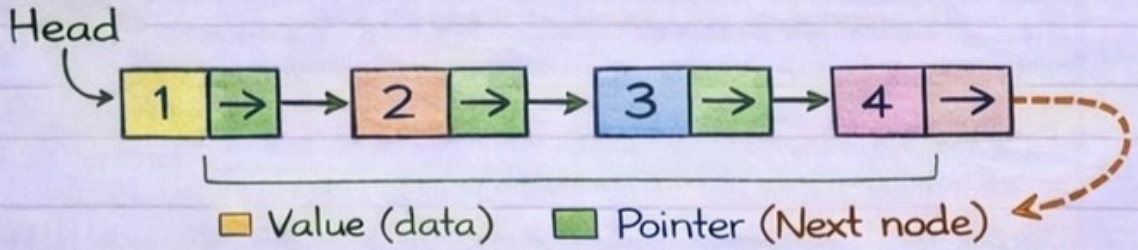
- Plan your approach before coding to understand the problem dearly.
- Consider edge cases, like an empty array or arrays with very large or very small values.
- Multiple Deletions: Multiple deletions may cause fragmentation, leaving empty slots in the array.

Chapter 3: Linked List

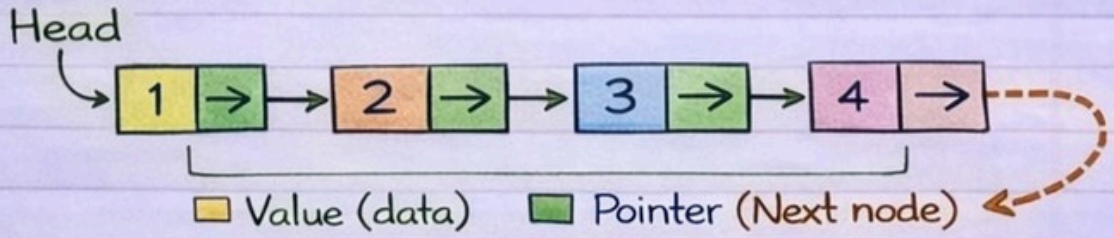
@curious_programmer

✔ 3.1 Introduction

A Linked List is a data structure used for storing a sequence of elements. Unlike arrays, linked lists consist of nodes where each node contains a value (or a pointer (or reference) to the next node in the sequence.



- ✔ **Dynamic Size:** Linked lists can easily grow and shrink in size, making them suitable for applications where the number of elements is unknown or varying.
- ✔ **Efficient Insertions/Deletions:** Elements can be easily inserted or removed without the need to shift other elements, unlike arrays.



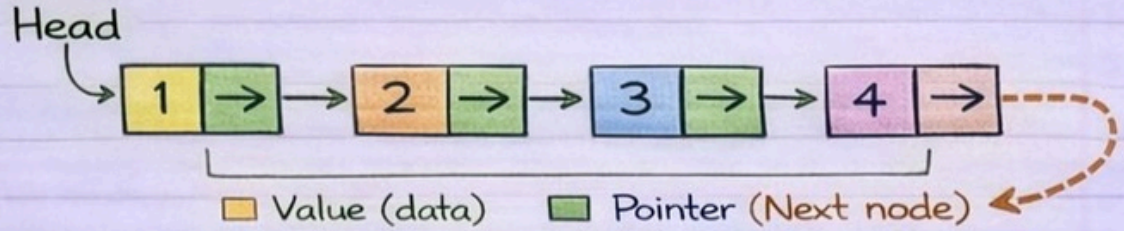
- ✔ **Dynamic Size:** Linked lists can easily grow and shrink in size, making them suitable for applications where the number of elements is unknown or varying.
- ✔ **Efficient Insertions/Deletions:** Elements can be easily inserted or removed without the need to shift other elements, unlike arrays.

Singly Linked List

@curious_programmer

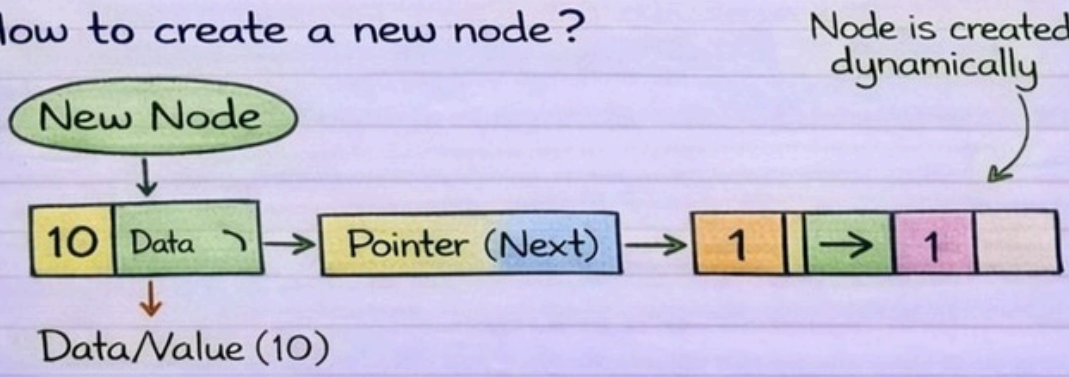
What is a Singly Linked List?

A Singly Linked List is a data structure used for storing a sequence of elements where each element (node) points to the next one through a pointer. It consists of a series of conneded nodes, where each node contains a piece of data and a pointer (reference) to the next node in the sequence.



- Each node consists of two parts:
  - Data/Value: Holds the actual value (e.g, 10)
  - Pointer: Stores the memory address of the next node.
- Unidirectional: It only allows traversal in one direction (from head to the last node).
- Memory Efficient: Nodes are dynamically created and memory is allocated as needed.
- Head Pointer: The list starts with a special pointer called "Head" that points to the first node in the list.

How to create a new node?

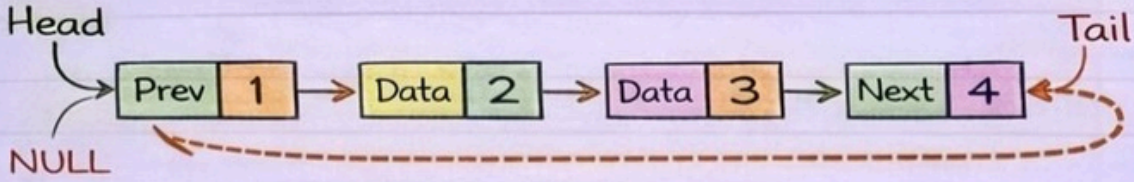


Doubly Linked List

@curious_programmer

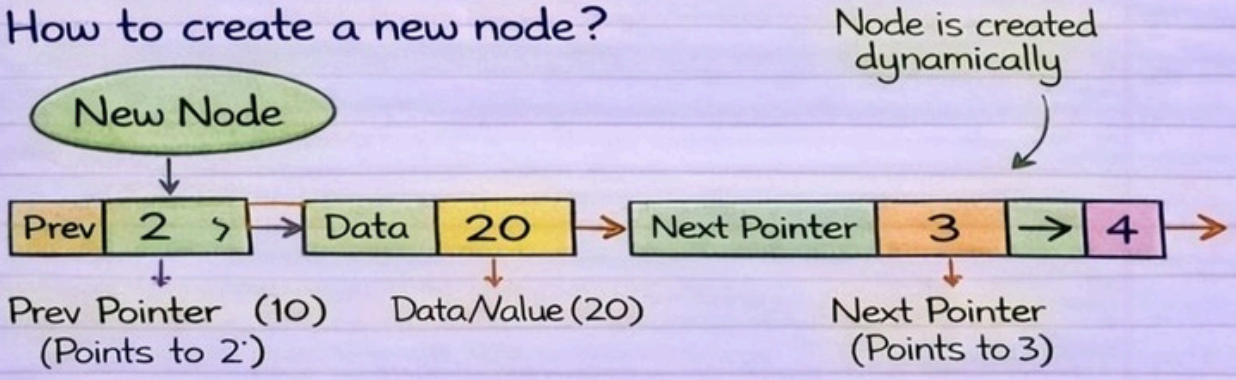
What is a Doubly Linked List?

A Doubly Linked List is a data structure used for storing a sequence of elements where each element (node) is connected to both its previous and next nodes through two pointers.



- Each node consists of three parts:
  - Prev Pointer: Stores the memory address of the previous node.
  - Data/Value: Holds the actual value (e.g, 20)
- Next Pointer: Stores the memory address of the next node.
- Bidirectional: Supports traversal in both forward and backward directions.
- More Memory: Requires extra memory for storing the previous pointer, unlike singly linked lists.
- Head & Tail Pointers: The list has two special pointers:
  - Head points to the first node.
  - Tail points to the last node.

How to create a new node?

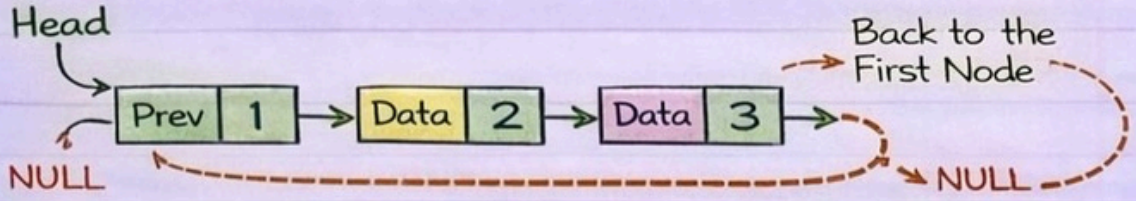


Circular Linked List

@curious_programmer

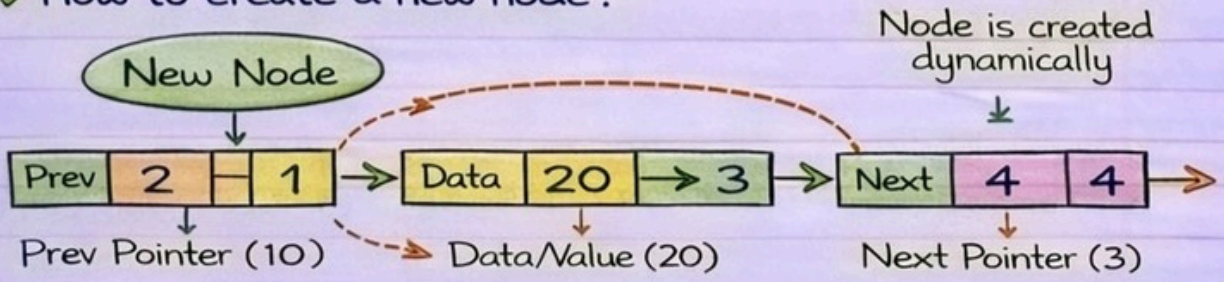
What is a Circular Linked List?

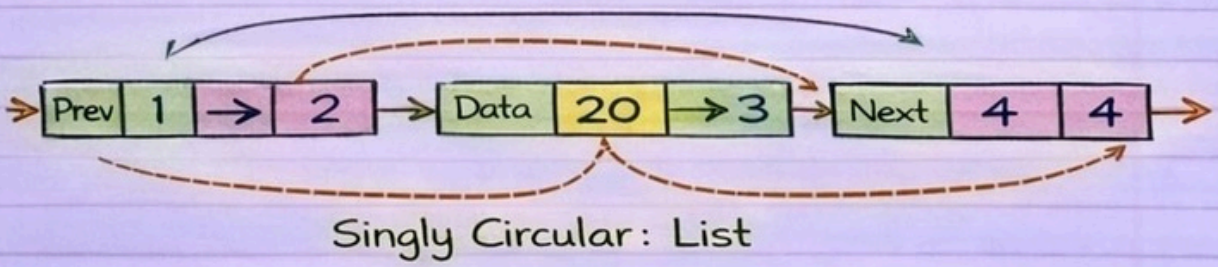
A Circular Linked List is a data structure used for storing a sequence of elements where each element (node) is connected to both its previous and next nodes through two pointers.



- **Forms a Circle:** The last node points back to the first node.
- **No NULL Value:** Unlike linear linked lists, there is no NULL pointer in a circular linked list because it forms a continuous loop.
- **Can be Singly or Doubly Linked:**
  - **Singly Circular:** Each node points to the next node, and the last node points back to the first node.
- **Doubly Circular:** Each node points to both the next and previous nodes, and the last node points back to the first node.
- **Head Pointer:** Points to the first node in the list.

How to create a new node?





Singly Circular: List

Insertion in Linked List

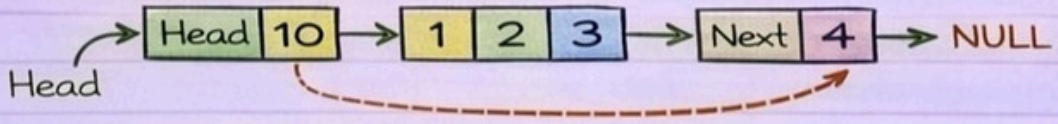
@curious_programmer

✔ How to insert an element into a Linked List?

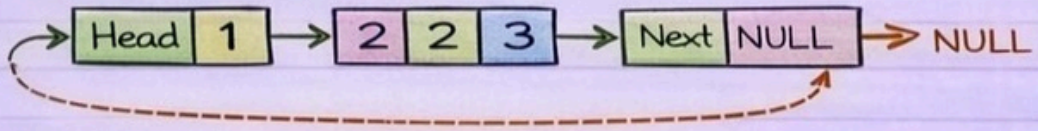
Inserting an element into a Linked List involves creating a new node and adjusting pointers to link it correctly. There are three common operations:

1. **Insert at the Beginning**: Add a new node as the first node of the linked list.
2. **Insert at the End**: Add a new node at the last node of the linked list.
3. **Insert at a Given Position**: Add a new node at a specific position in the linked list.

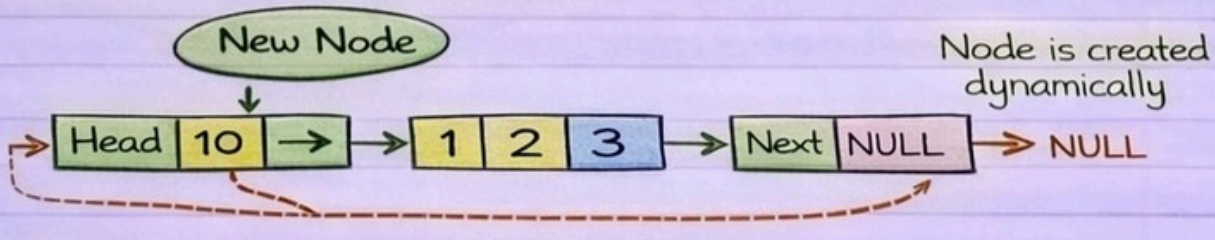
1. Insert at the Beginning:



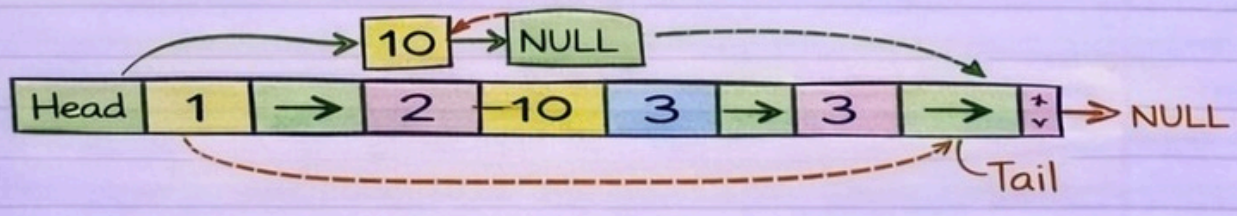
2. Insert at the End:

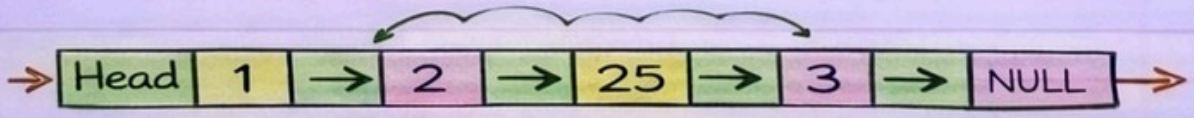


3. Insert at a Given Position:



✔ How to create a new node?





Singly Circular : List

Deletion in Linked List

@curious_programmer

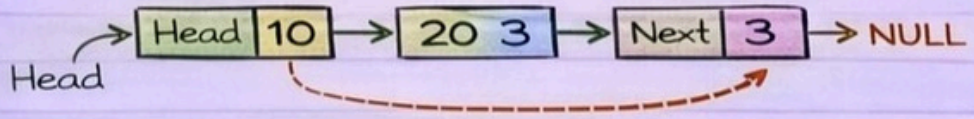
✔ How to delete an element from a Linked List?

Inserting an element into a Linked List involves adjusting pointers to bypass the node to be deleted. There are three common operations:

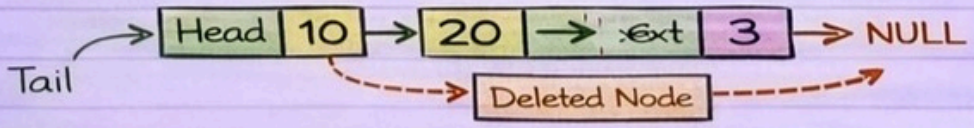
1. **Delete from the Beginning:** Remove the first node of the linked list.
2. **Delete from the End:** Remove the last node of the linked list.
3. **Delete at a Given Position:** Remove a node from a specific position in the linked list.

1. Delete from the Beginning:

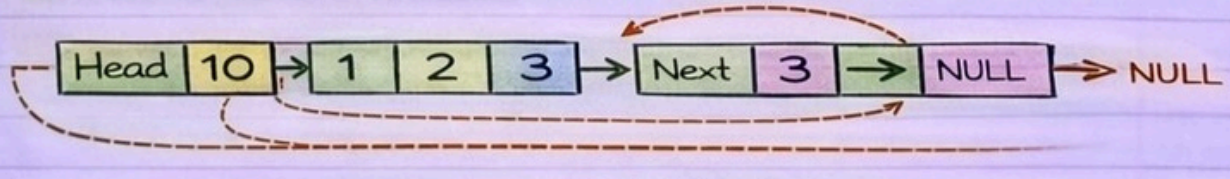
2. Delete From the End:

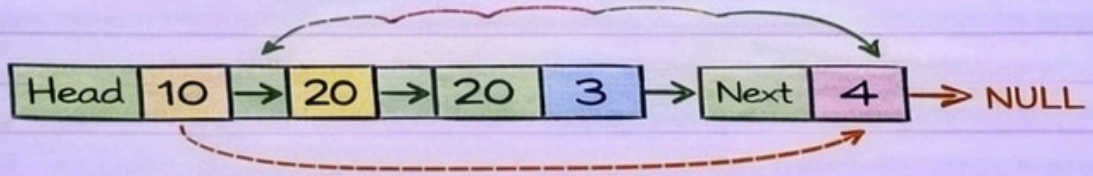


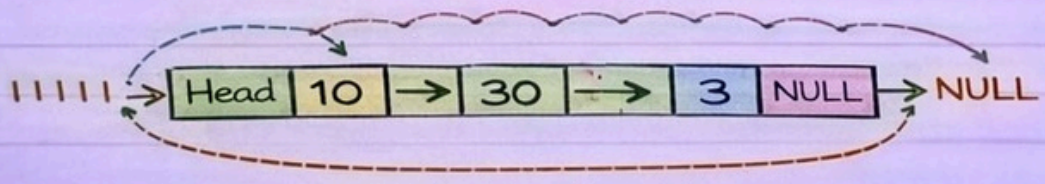
3. Delete at a Given Position:



✔ How to delete a node?







Singly Circular: List

Reverse Linked List

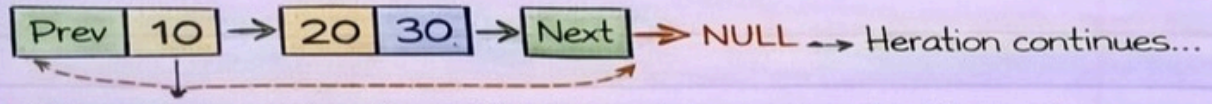
@curious_programmer

✔ How to reverse a Linked List ?

Reversing a linked list means changing the direction of the pointers so that the last node becomes the first node, reversing the list order.

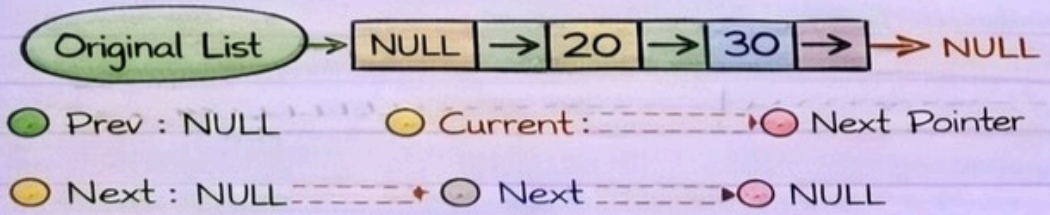
✔ Steps to Reverse a Linked List:

1. Initialize three pointers:
   - Prev : Initially set to NULL.
   - Current : Initially set to Head.
   - Next : Points to the node after Current.
2. Reverse pointers one by one:

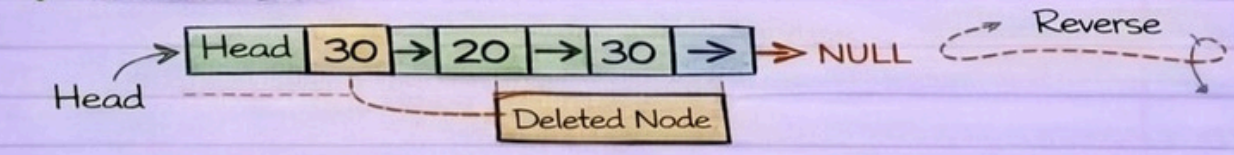


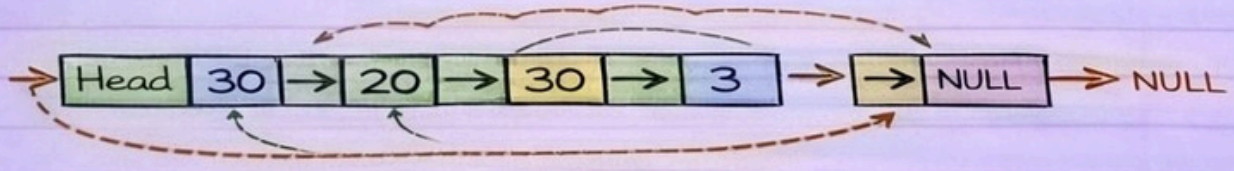
3. Move Prev, Current, and Next pointers one step forward.
4. Repeat until Current becomes NULL:

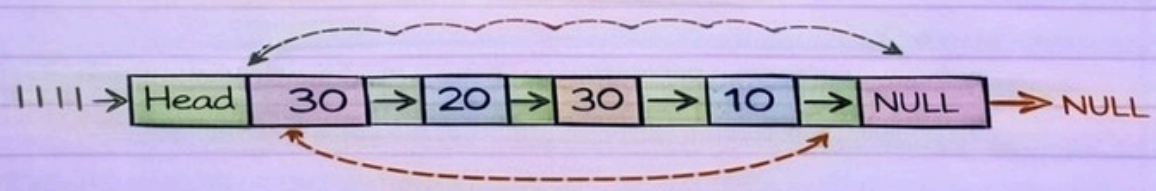
✔ Example:



✔ Reverse List:







Singly Circular: List

Chapter 4: Searching

@curious_programmer

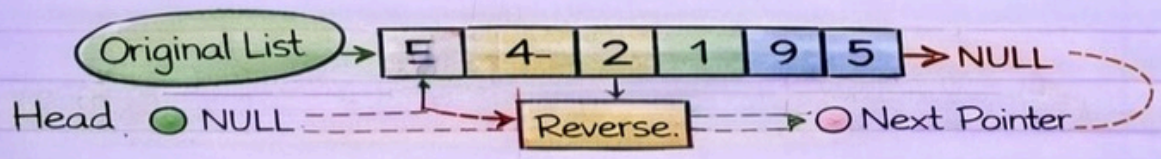
✔ 4.1 Linear Search

Linear Search is a simple searching algorithm that checks each element of a list sequentially until the desired element is found. It works with both unsorted and sorted arrays.

✔ Steps in Linear Search:

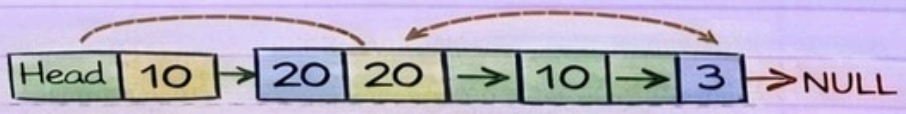
1. Start from the first element.
   - Prev : Initially set to NULL.
   - Current : Initially set to Head.
   - Next : Points to the node after Current.
2. Reverse pointers one by one:
3. Move Prev, Current, and Next pointers one step forward.
4. Repeat until Current becomes NULL:

✔ Example:



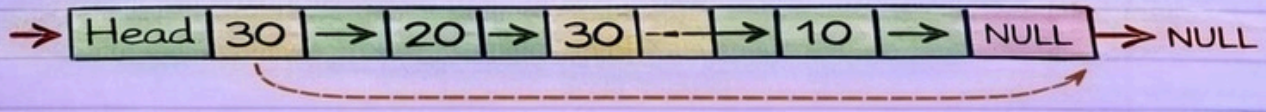
- Start with the first element and compare each element with the target (7)
- When element 7 is found at index 2, return 2.

✔ Example:



- ✔ Start with the first element and compare each element with the target (7).
- ✔ When element 7 is found at index 2, return 2. Result: 2

* Target = The value we're searching for "7"



Linear Search

@curious_programmer

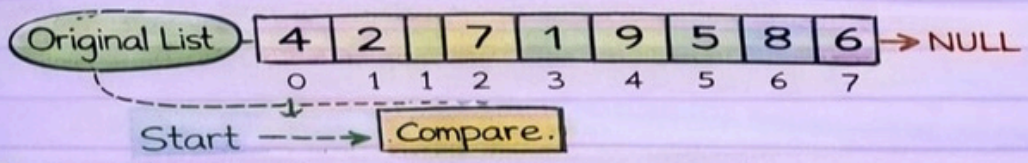
✔ 4.1 Linear Search

Linear Search is a simple searching algorithm that checks each element of a list sequentially until the desired element is found. It works with both unsorted and sorted arrays.

✔ Steps in Linear Search:

1. Start from the first element of the array (index 0).
2. Compare each element with the target value.
3. If the target value matches an element, return the index of the element.
4. If the target value is not found by the end of the list, return -1 (indicating not found).

✔ Example:



① Start from the first element of the array (index 0).

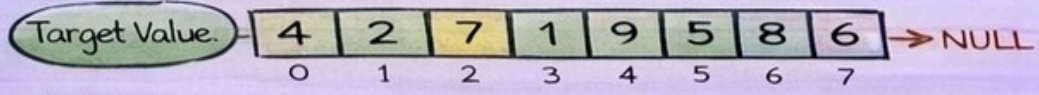
② Compare 4 with 7 → Not Found, move to next element...

③ If the target value matches an element.

④ If the target value is not found by the end of the list, return -1.)

Not Found: 1

✔ Step 1:



⑤ Step 1: Compare 4 with 7 → Not Found, move to next element...

① Step 2: Compare 2 with 7:

Match Found! Start at index 2

⑤ Step 3: 7 equals 7, target found at index 2

* Target = The value we're searching for "7"

Binary Search

@curious_programmer

✔4.1 Linear Search

Binary Search is an efficient searching algorithm that checks each element of a list sequentially until the desired element is found. It works with both unsorted and sorted arrays.

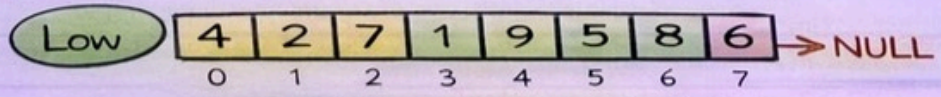
✔ Steps in Binary Search:

1. Initialize three pointers:
   - Low : Points to the first index of the array.
   - High : Points to the last index of the array.
   - Mid : (Low + High) / 2, points to the middle index.
2. Repeat until the target value is found, or the Low pointer exceeds the High pointer:
3. Calculate Mid:

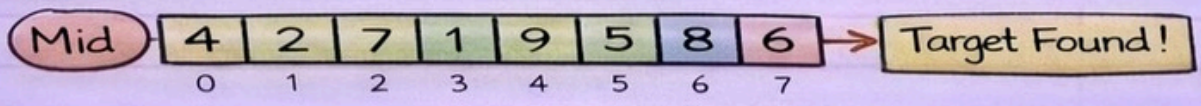
$$Mid = [Low + High) / 2]$$

4. Compare the middle element (array[Mid]) with the target value:

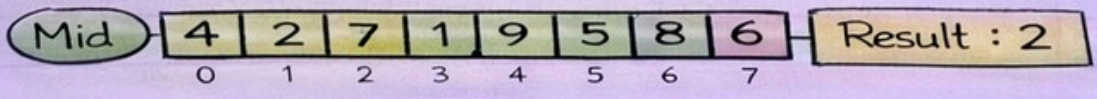
✔ Step 1:



② Step 1: Compare 4 with 7 → Not Found, move to next element...



② Step 2: Compare 2 with 7 → Search again 2.



→ Continue this process until the target value is found, or the search interval is empty (Low > High).

* **Important:** The array must be sorted for binary search to work

Example : Binary Search

@curious_programmer

✔ Example : Binary Search

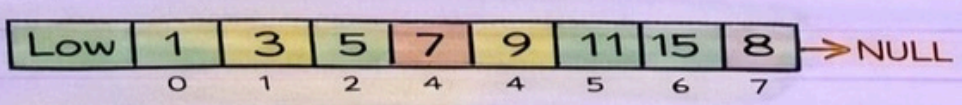
Binary Search is an efficient searching algorithm used to find a target value within a sorted array. It works by repeatedly dividing the search interval in half. focusing on the micidle element.

✔ Steps in Binary Search :

1. Initialize three pointers :
   - **Low** : Points to the first index of the array.
   - **High** : Points to the last index of the array.
   - **Mid** : (Low + High) / 2, points to the middle index.
2. Repeat until the target value is found, or the Low pointer exceeds the High pointer :

Calculate Mid Mid = [Low + High) / 2 = [0+7] = 3

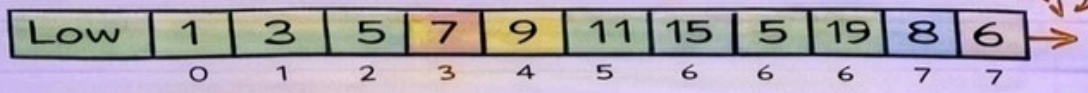
✔ Step 1: Compare 9 with array[3]



Mid = 3, array[Mid] = 7

③ Step 2: Compare 9 with array[5]

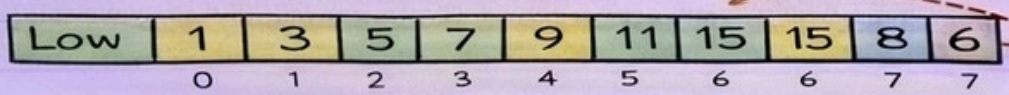
Search right half



Mid = 5, array[Mid] = 11

③ Step 3: Compare 9 with array[4]

Search left half



Mid = 4, array[Mid] = 9 ⟶ Target Found! 9 equals 9 at index 4

→ Continue this process until the target value is found, or the search interval (Low > High).

Result : 4

Example: Binary Search on Answer

@curious_programmer

✔ Find the smallest divisor for which the sum of all quotients is ≤ 7.

Array of num: [15, 21, 36, 45, 60]

✔ Steps in Binary Search:

1. Define the search space: Low = 1, High is the maximum element of the array.
2. Calculate the Mid value: Mid = (Low + High) / 2. Find the sum of the quotients by dividing each number by Mid and rounding up.
3. Check the condition: If the sum of quotients is ≤ 7, narrow the search to the left half (High = Mid).
   - ① Yes: narrow search to left half : (High = Mid)
   - ② No: search the right half : (Low = Mid + 1)
4. Repeat until Low exceeds High.
   The smallest divisor is the final value of Low.

✔ Example:

Numbers: [15, 21, 36, 45, 60]

Low = 1 | High = 60 | High = 60

① Calculate Mid: Mid = (Low + High) / 2 = [30]

Sum of Quotients = 1 + 1 + 2 + 2 + 2 = 8

Sum > 7 (No)

② Step 3: Compare 9 with array[4]

Number	1	3	5	7	9	11	15	15	15	8	6
	0	1	2	3	4	5	6	6	6	7	7

Mid = 4, array[Mid] = 9 ---------> Search the right half: [Low = 30 + 1

→ Sum of Quotients = 1 + 1 + 2 + 2 + 2 = 8 ←

Sum = 7

* **Quotient:** How many times a number can be divided by the divisor when rounding up to the nearest whole number

Low Pointer | Mid Pointer | High Pointer

Applications of Binary Search

@curious_programmer

✔ Find the smallest divisor for which the sum of all quotients is ≤ 7.

Array of [15, 21, 36, 45, 60]

✔ Quickly locate items in a sorted list by repeatingly dividing the list in half

2. Searching in infinite space:
   - Efficiently find solutions in problems with an infinite or continuous space; like finding a threshold or optimal value.
3. Finding the peak element in an array where elements are first increasing and then decreasing.
4. Check the condition: If the sum of quotients is ≤ 7, narrow the search to the left half (High = Mid).

① Yes: narrow search to left half (High = Mid)

② No: search the right half: (Low = Mid + 1)

④ Repeat until Low exceeds High. The smallest divisor is the final value of Low.

Example: Numbers: [15, 21, 36, 60]

Low = 1 | High = 60 | High = 60

① Calculate Mid: Mid = (Low + High) / 2 = [30]

Sum of Quotients = 1+1+2+2+2 = 8

Sum > 7 (No)

② Search the right half: Low = Mid + 1 → Low = 30 + 1 → Low = 31

Number	15	21	36	45	50	45	60	...
	0	1	2	3	4	5	6	7

→ Sum of Quotients = 1+1+2+2+2+2 = 8

Sum = 7 (No)

* **Quotient**: How many times a number can be divided by the divisor when rounding up to the nearest whole number.

Low Pointer | Mid Pointer | High Pointer

Chapter 5: Sorting Algorithms

@curious_programmer

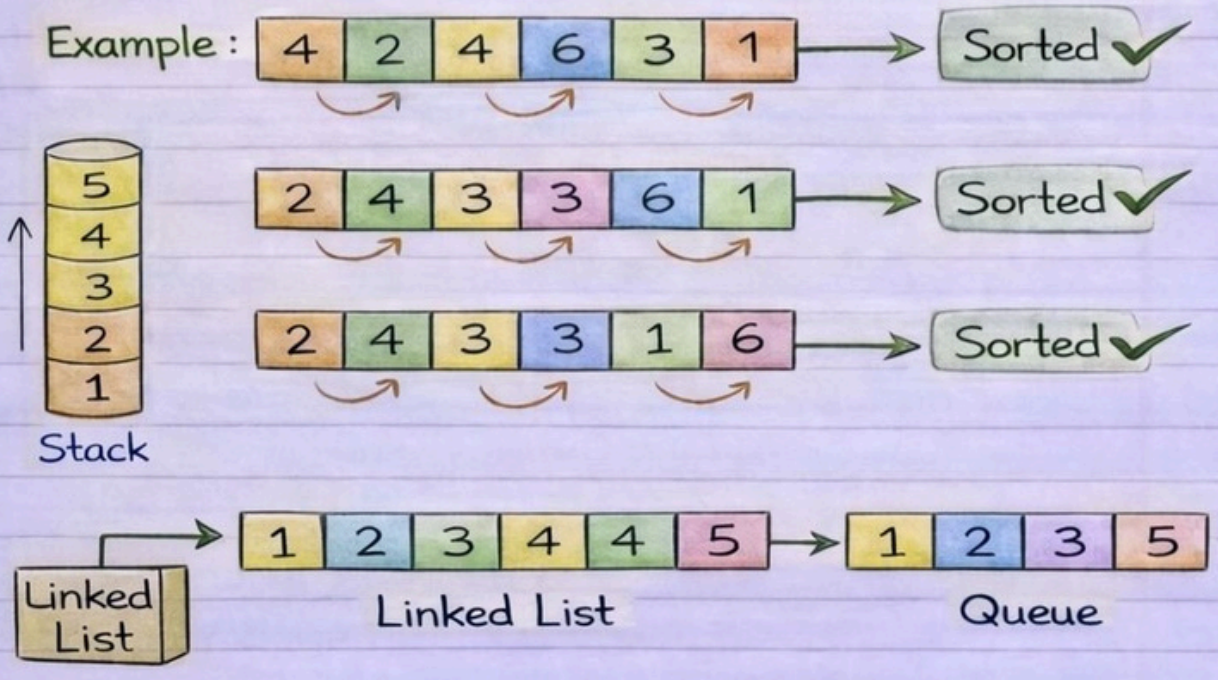
Introduction to Sorting Algorithms

Sorting algorithms are methods used to rearrange elements in a list or an array into a specific order (typically ascending or descending). They are fundamental in computer science for organizing data efficiently.

(1) Bubble Sort
(2) Selection Sort
(3) Insertion Sort
(4) Merge Sort
(5) Quick Sort
(6) Heap Sort
(6) Heap Sort
(7) Counting Sort
(8) Radix Sort
(8) Stability & in-place Sorting

Bubble Sort

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.



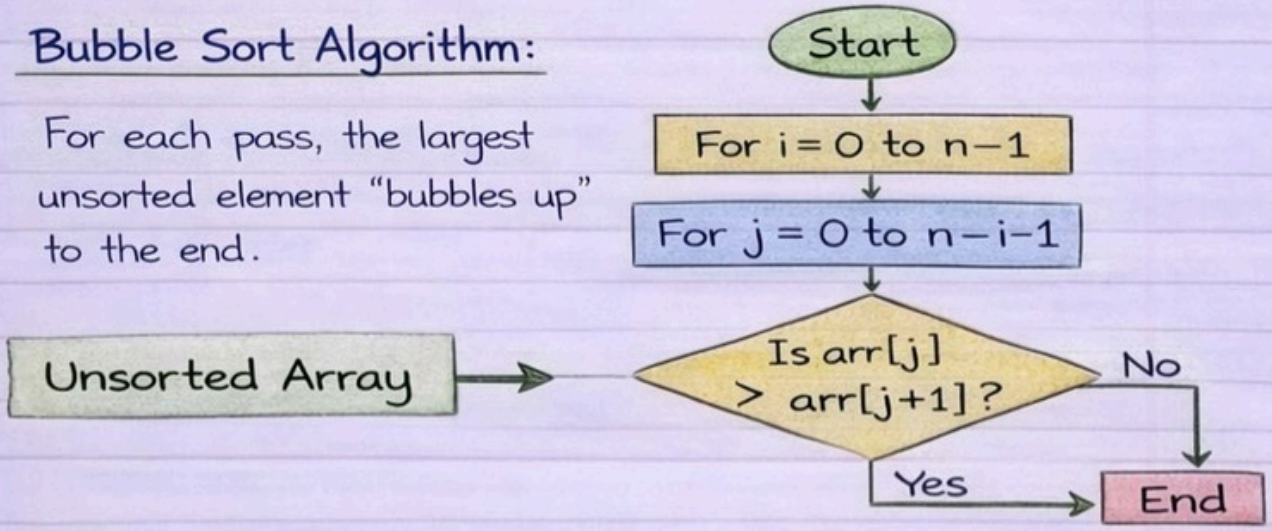
Bubble Sort

What is Bubble Sort?

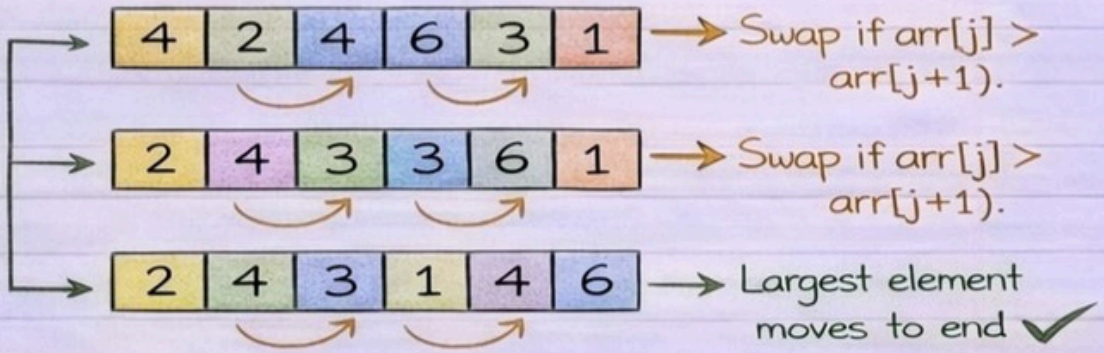
Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the entire list is sorted.

Bubble Sort Algorithm:

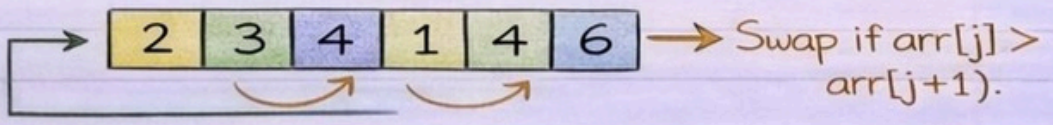
For each pass, the largest unsorted element "bubbles up" to the end.



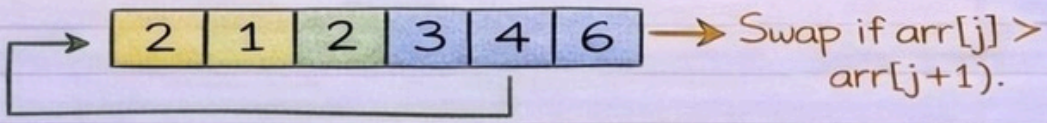
Pass 1: (i=0)



Pass 2: (i=1)



Pass 3: (i=2)



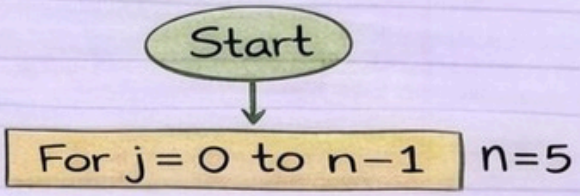
Final sorted list : 1 | 2 | 2 | 3 | 4 | 4 | 6

For each pass, the largest unsorted element "bubbles up" to the end.

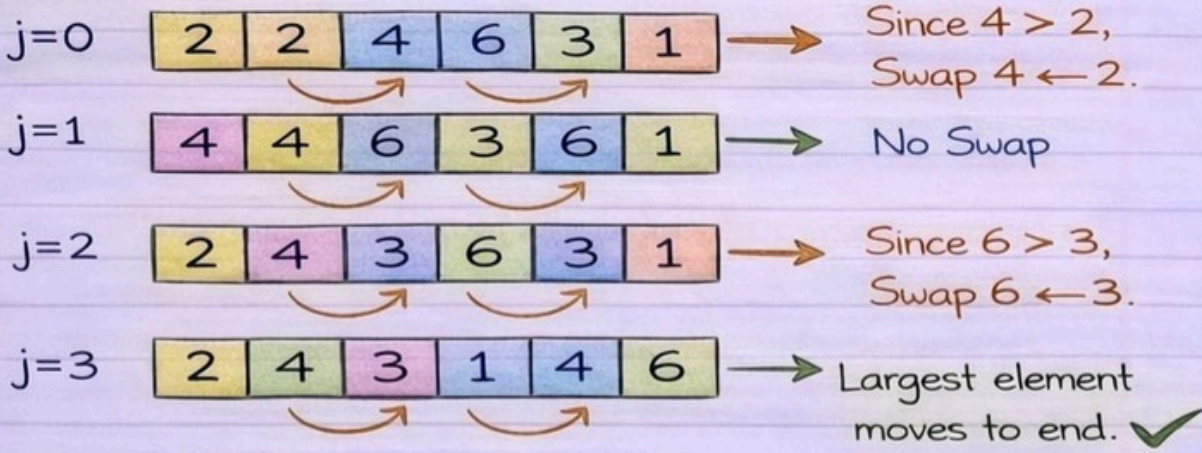
Bubble Sort Example

Start:

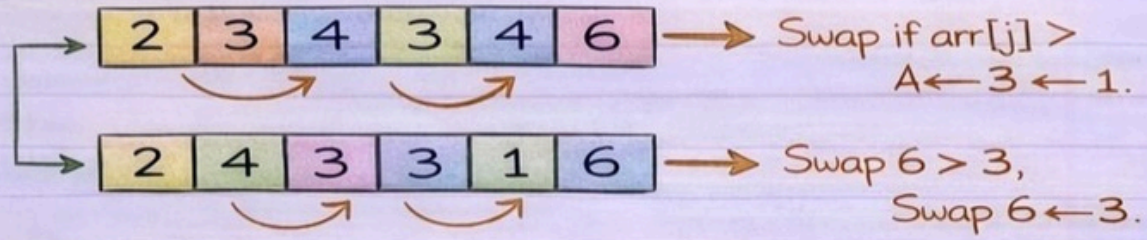
Unsorted Array = [4, 2, 6, 3, 1]



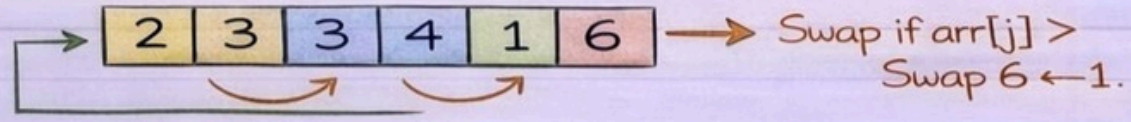
Pass 1: (i=0)



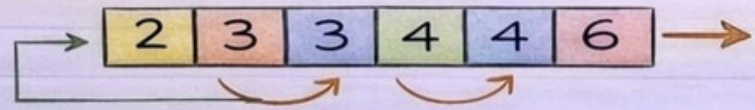
Pass 3: (i=2)



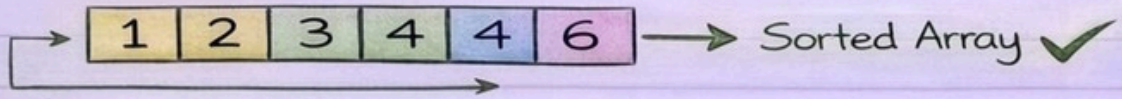
Pass 4: (i=3)



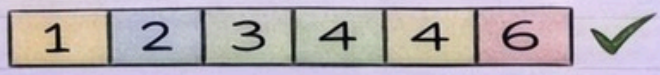
Pass 5: (i=4)



Pass 5: (i=4)



Pass 5: (i=4)



For each pass, the largest unsorted element "bubbles up" to the end.

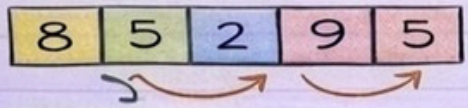
Selection Sort

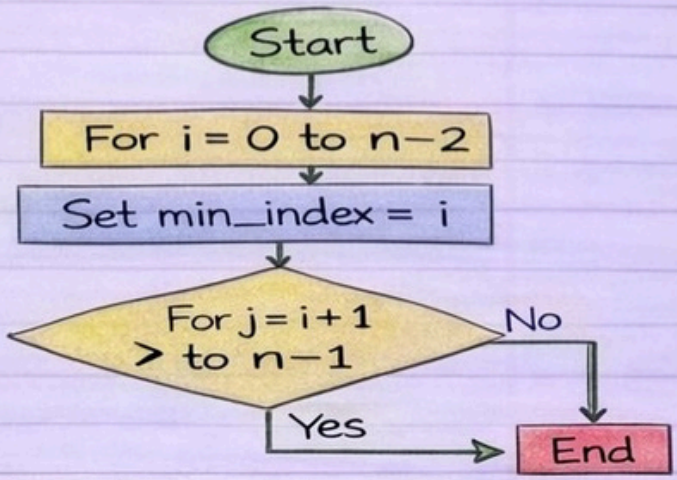
What is Selection Sort?

Selection Sort is a straighiforward sorting algorithm that repeatedly selects the minimum element from the unsorted portion of the list and swaps it with the first unsorted element. This process is repeated until the entire list is sorted.

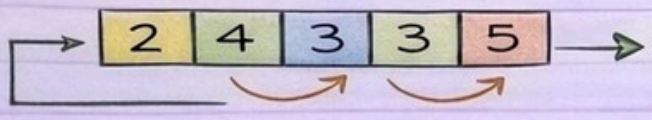
Selection Sort Algorithm:

For each pass,

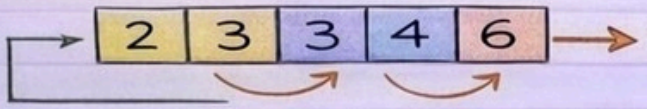




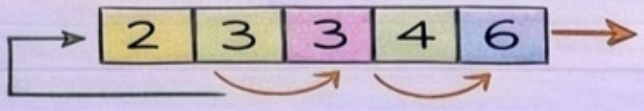
Pass 2 : (i=1)



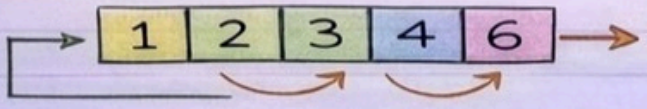
Pass 3 : (i=2)



Pass 4 : (i=3)



Pass 5 : (i=4)



Pass 5 : (i=4) 1 2 3 4 4 6 ✓

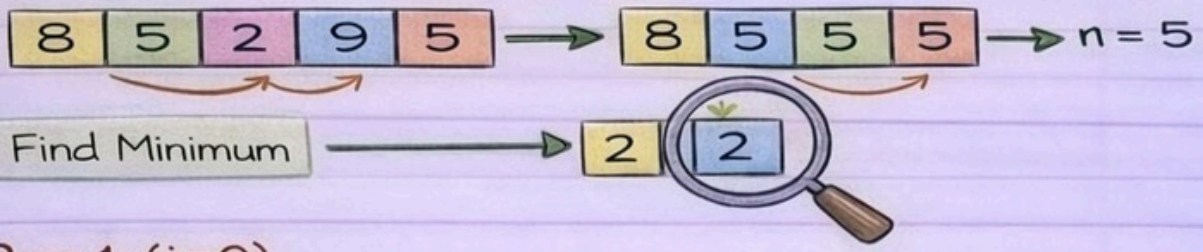
For each pass, the largest unsorted element "bubbles up" to the end.

Sorted Array = 1 2 3 4 4 6 ✓

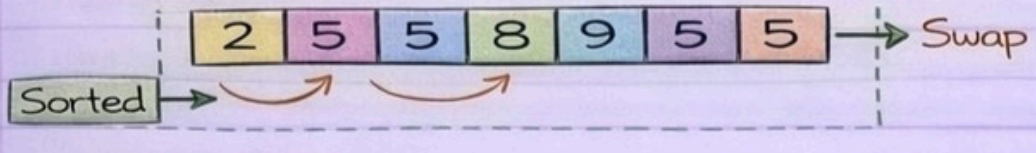
Selection Sort Example

Start:

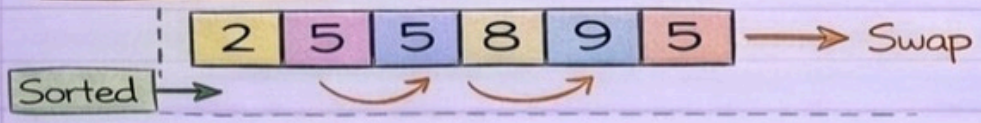
Unsorted Array = [8, 5, 2, 9, 5]



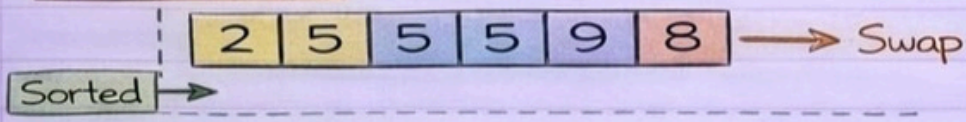
Pass 1: (i=0)



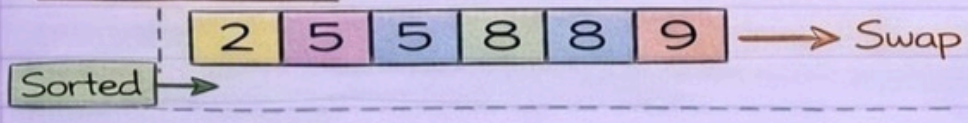
Pass 2: (i=1)



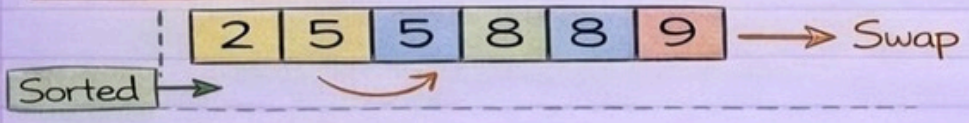
Pass 3: (i=2)



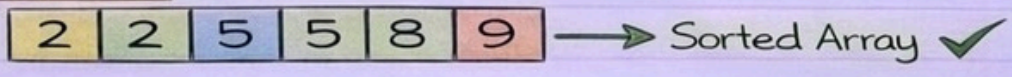
Pass 4: (i=3)



Pass 4: (i=3)



Pass 5: (i=4)



Sorted Array = 1 2 3 4 6 ✓

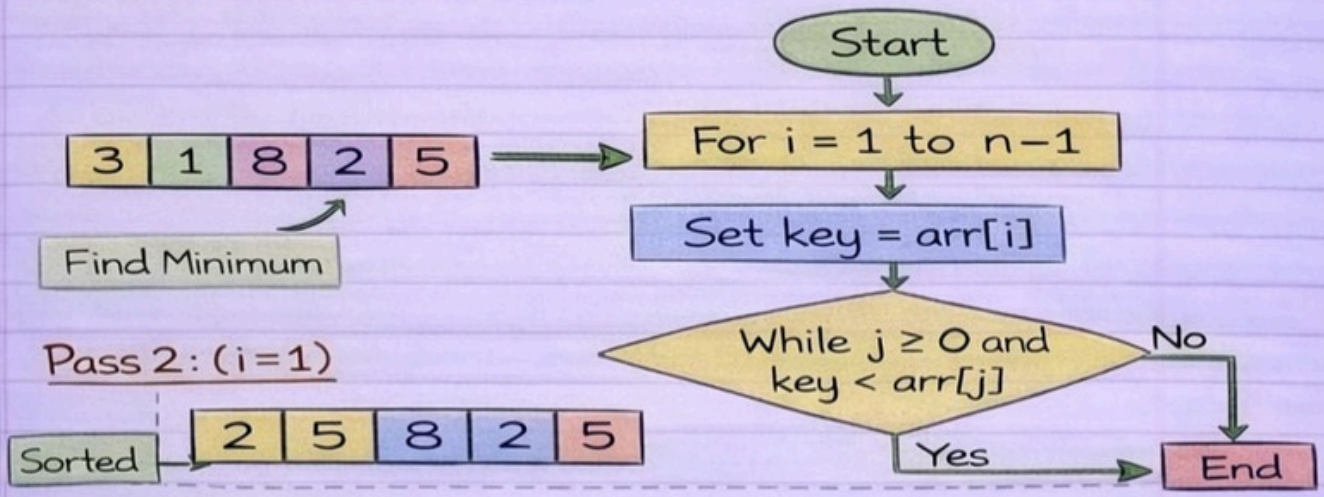
For each pass, select the minimum from the unsorted portion and place it at the sorted portion.

Insertion Sort

What is Insertion Sort?

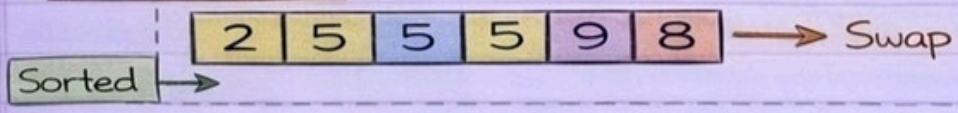
Insertion Sort is a simple sorting algorithm that builds the final sorted list one element at a time. It takes each element from the unsorted part and inserts it into the correct position in the sorted part of the list. This process is repeated until the entire list is sorted.

Insertion Sort Algorithm:

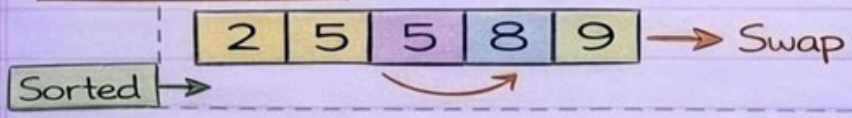


Pass 2: (i=1)

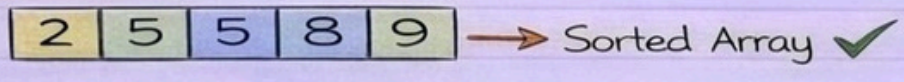
Pass 3: (i=2)

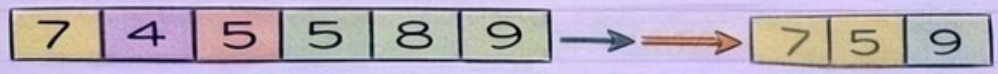


Pass 4: (i=3)



Pass 5: (i=4)





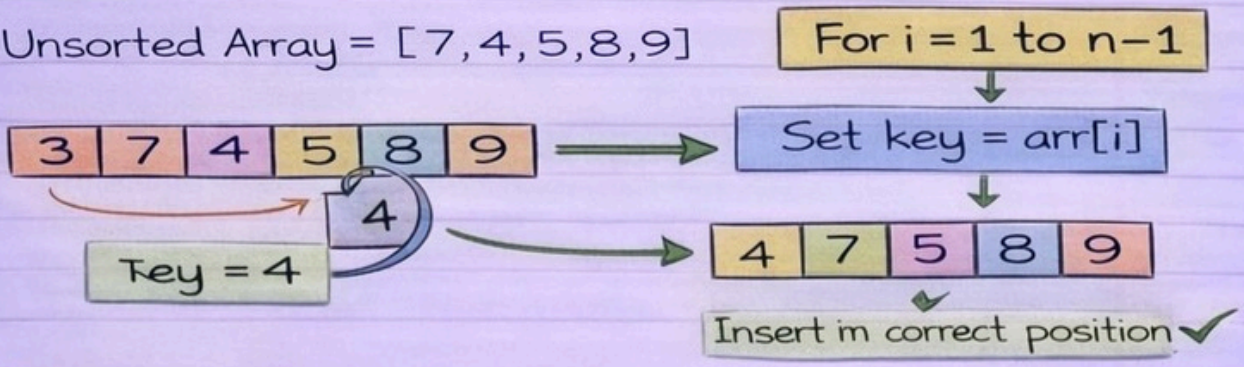
For each pass, seled the minimum from the unsorted portion and place it at the sorted portion.

Sorted Array = 2 2 5 5 8 9

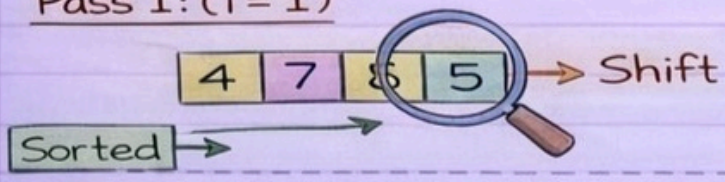
Insertion Sort Example

Start:

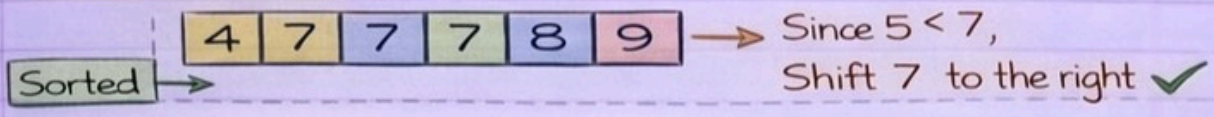
Unsorted Array = [7,4,5,8,9]



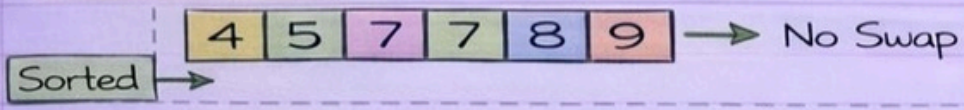
Pass 1: (i = 1)



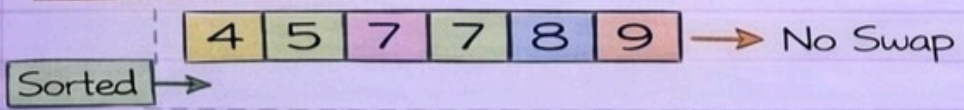
Pass 2: (i = 2)



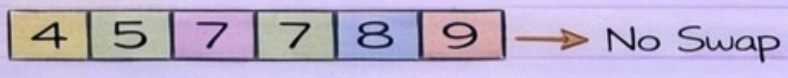
Pass 3: (i = 2)



Pass 4: (i = 4)



Pass 4: (i = 4)



Sorted Array = 4 5 7 7 8 9

In each pass, the key is compared and shifted left until it's placed at the correct sorted position.

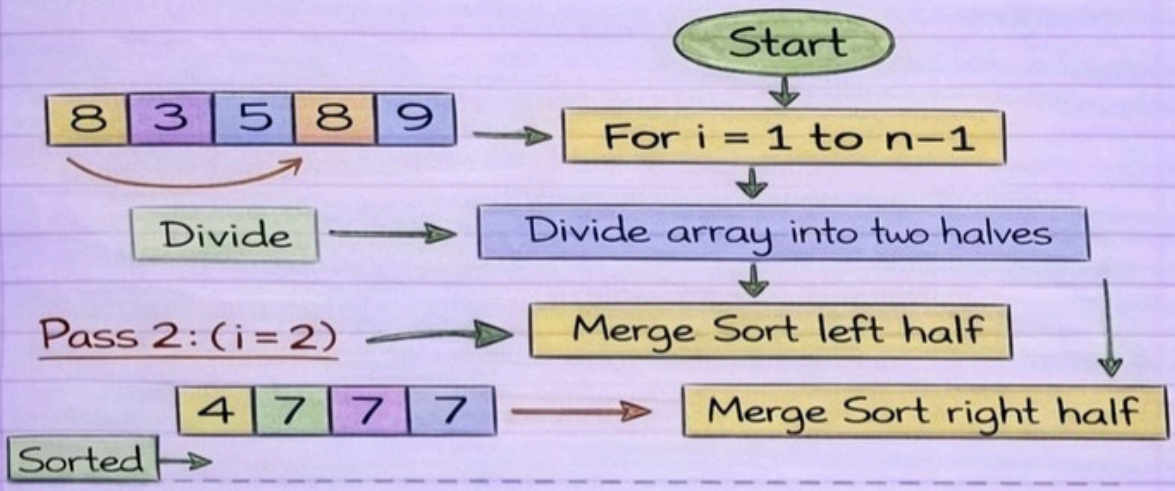
Sorted Array = 4 5 5 7 8 9

Merge Sort

What is Merge Sort?

Merge Sort is an efficient sorting algorithm based on the Divide and Conquer strategy. It works by recursively dividing the list into two halves until each half contains a single element. Then, it merges the halves back together in a sorted manner.

Merge Sort Algorithm:



Pass 3: (i = 3)

4 5 7 7 8 9 → No Swap

Sorted

Pass 4: (i = 4)

4 5 5 7 8 9 → No Swap

Key Properties:

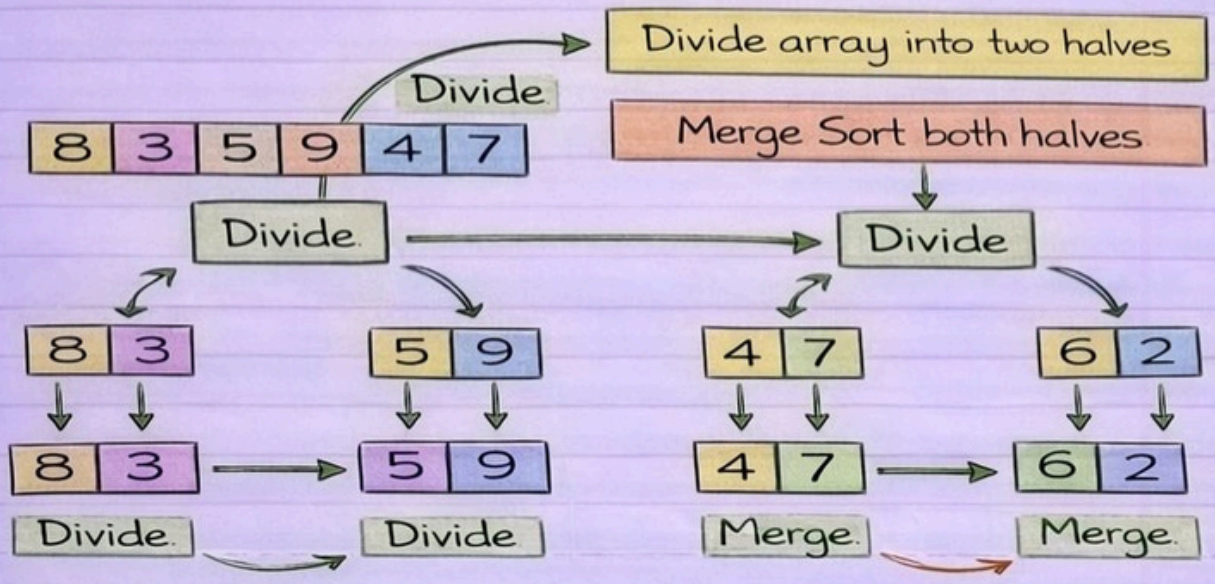
- **Stability**: Merge Sort maintains the relative order of equal elements.
- **Complexity**: $\Omega(n \log n)$: Best and Average Case.
  $O(n \log n)$: Worst Case.
- **Divide**: Recursively divides the array into two halves,
- **Conquer**: Combines the sorted halves into a single sorted list.

Sorted Array= 4 4 5 5 7 8 9 ✓

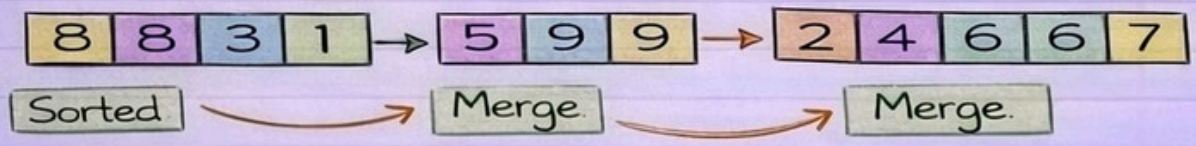
Merge Sort Example

Start:

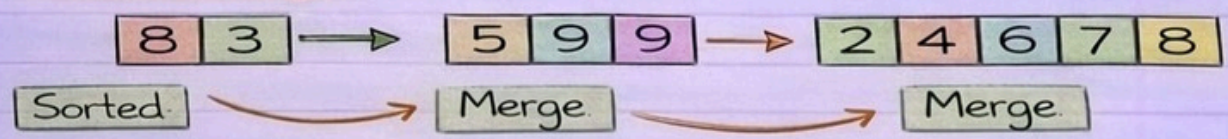
Unsorted Array = [8,3,5,9,4,7,6,2]



Pass 3: Merge



Pass 3: Merge



Sorted Array = 2 3 4 5 6 7 8 9 ✓

✔ Divide the array into two halves, sort each half, and then merge them to get the sorted array.

1. **Stability**: Merge Sort maintains the relative order of equal elements.
2. **Complexity**: $\Omega(n \log n)$: Best and Average Case.
   $O(n \log n)$: Worst Case.
3. **Divide**: Recursively divides the array into two halves.
4. **Conquer**: Combines the sorted halves into a single sorted list.

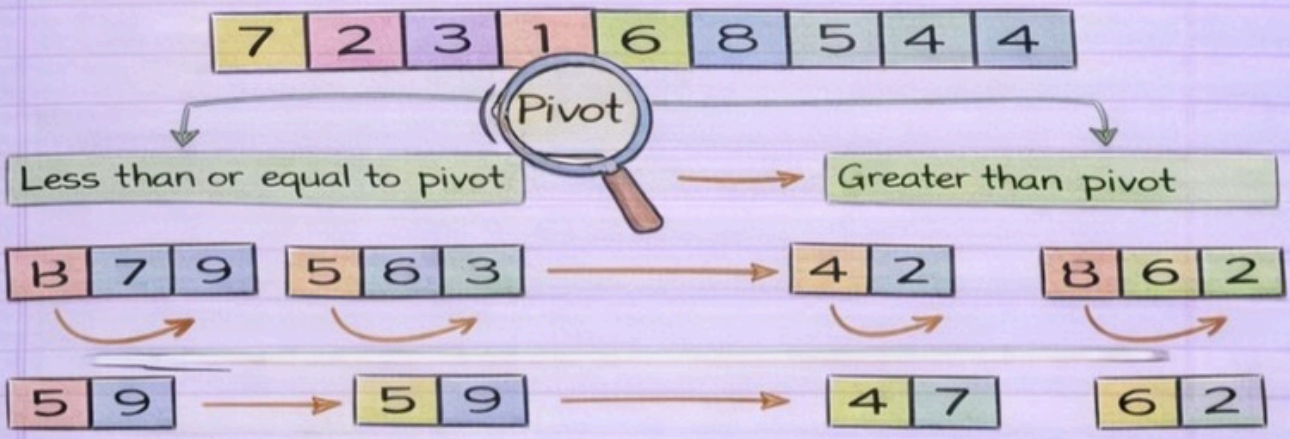
Sorted Array = 2 3 4 5 6 7 8 9 ✓

Quick Sort

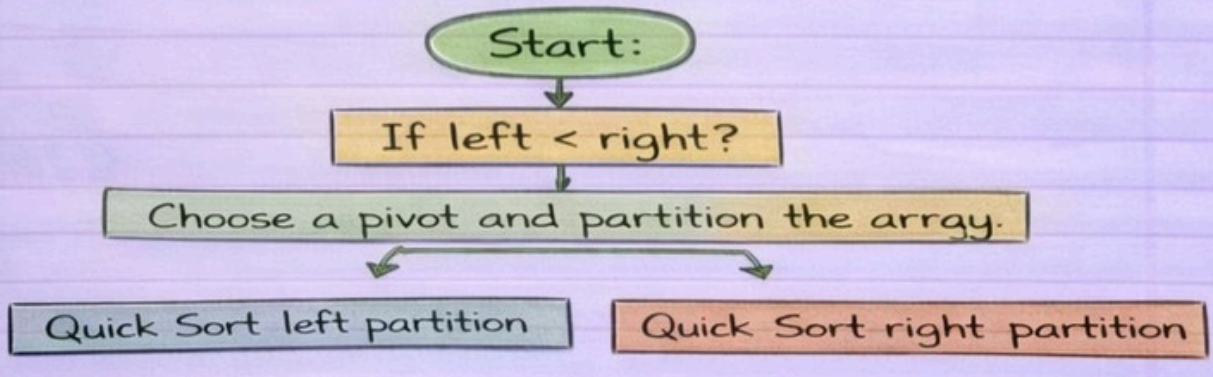
What is Quick Sort?

Quick Sort is an efficient sorting algorithm based on the Divide and Conquer strategy. It works by selecting a "pivot" element and partitioning the list into elements less than or equal to the pivot and elements greater than the pivot. The process is recursvely applied to the sublists.

Quick Sort Algorithm:



Pass 3: Merge.



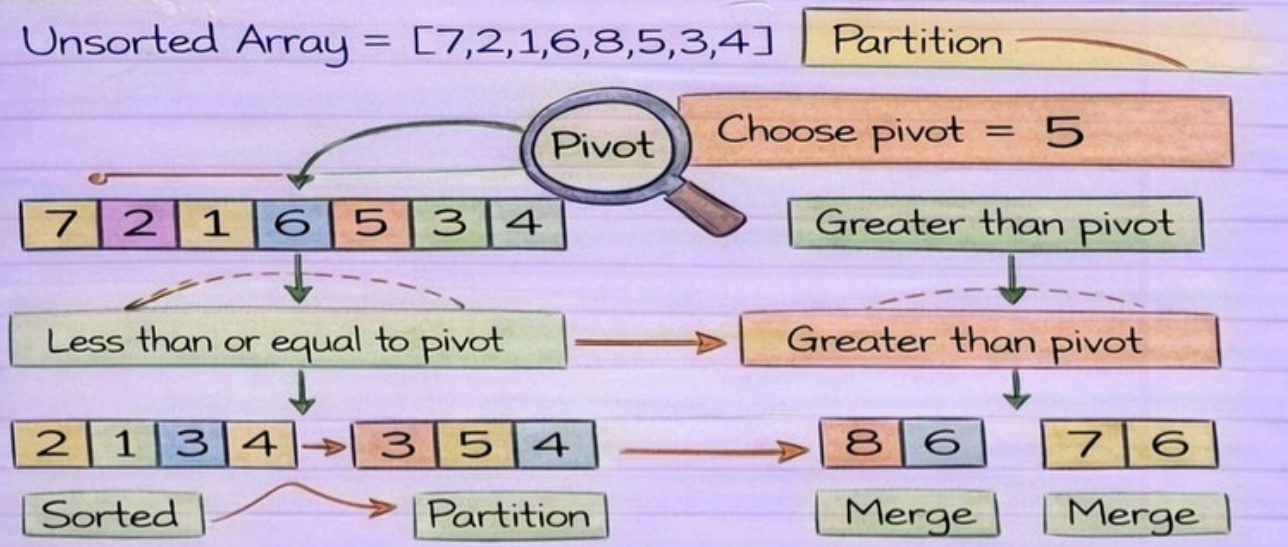
Key Properties:

- ✔ **Stability**: Quick Sort is not stable.
- ✔ **Complexity**: $O(n \log n)$: Best and Average Case.
  $O(n)$: Worst Case.
- ✔ **Divide**: Recursively selects a pivot and partitions the array.
- 4 **Conquer**: Recursively applics Quick Sort to partitions, combining the sorted

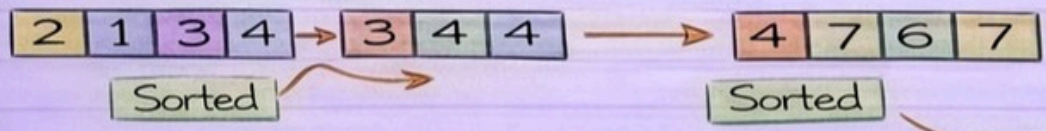
Quick Sort Example

Start:

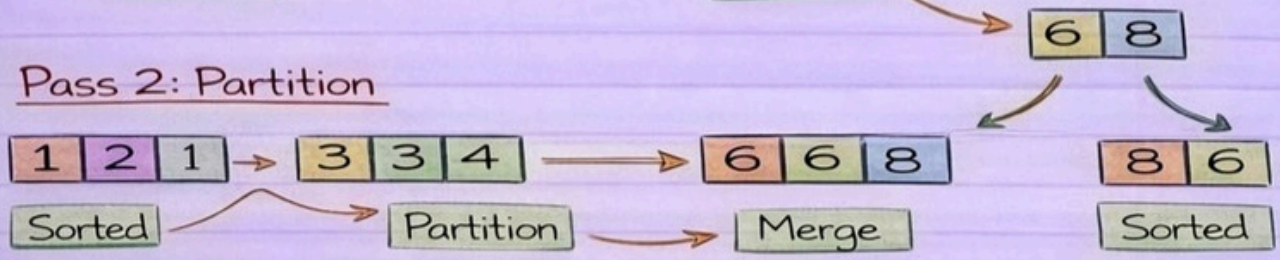
Unsorted Array = [7,2,1,6,8,5,3,4]



Pass 1: Partition



Pass 2: Partition



Sorted Array = 1 2 3 4 5 6 7 8 9 ✓

✓ Select a pivot, partition the array, and recursively apply Quick Sort to the partitions.

1. **Stability**: Quick Sort is not stable.
2. **Complexity**: $\Omega(n \log n)$: Best and Average Case.
   $O(n^2)$: Worst Case.
3. **Divide**: Recursively selects a pivot and partitions the array
4. **Conquer**: Recursively applies Quick Sort to partitions, combining the sorted

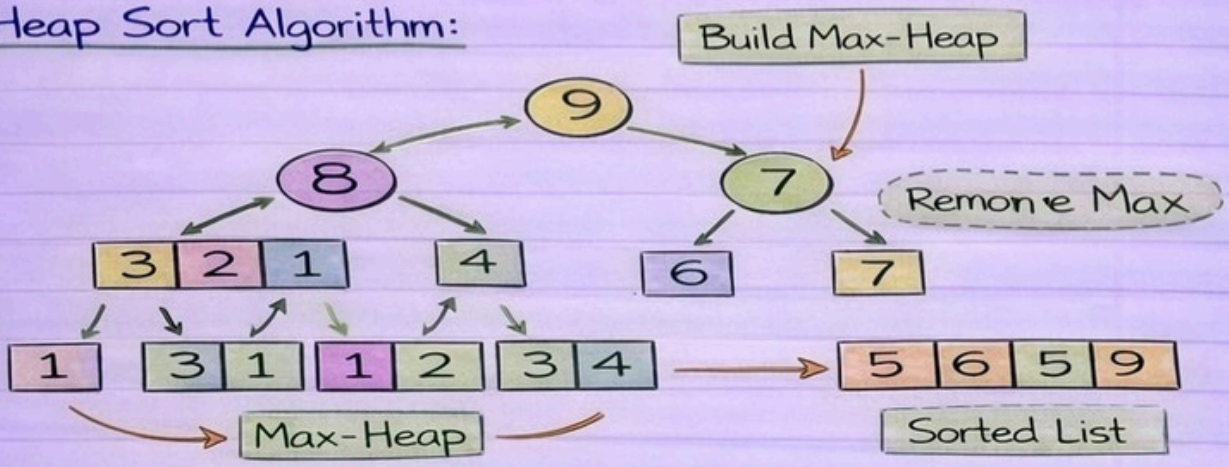
Sorted Array= 1 2 3 4 5 7 8 9 ✓

Heap Sort

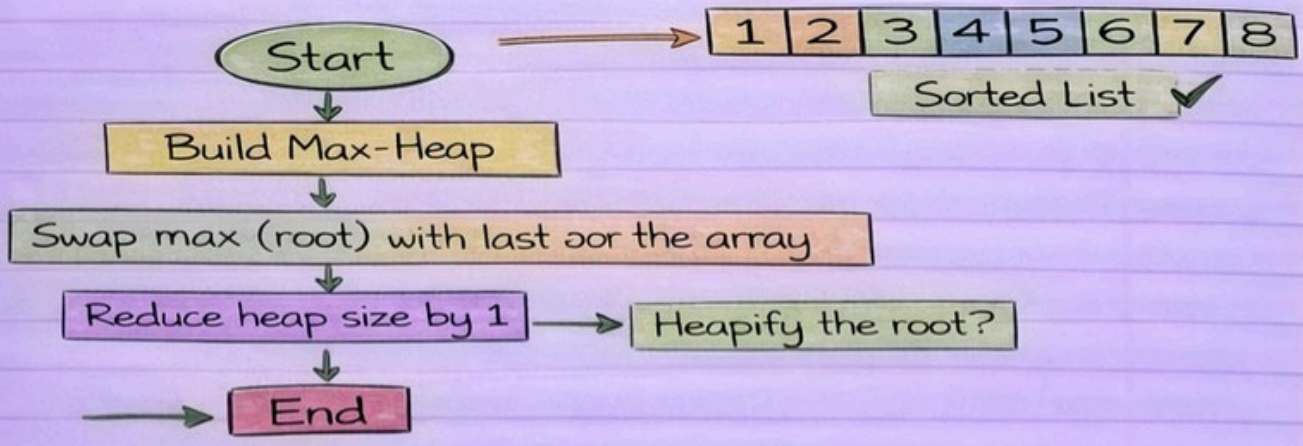
What is Heap Sort?

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to arrange elements in order. It first builds a Max-Heap from the list, then repeatedly removes the maximum element from the heap and places it at the end of the list, reducing the heap size by one. This process is repeated until the list is sorted.

Heap Sort Algorithm:



Pass 1: Partition



Key Properties:

- ✓ Stability: Heap Sort is not stable.
- ✓ Complexity: $\Omega(n \log n)$: Best and Average Case.
  $O(n^2)$: Worst Case.

3. Buide: Max-Heap: Builds a Max-Heap from the unsorted list.
4. Sort: Repeatedly removes the maximum element from the heap and appends it to the sorted portion.

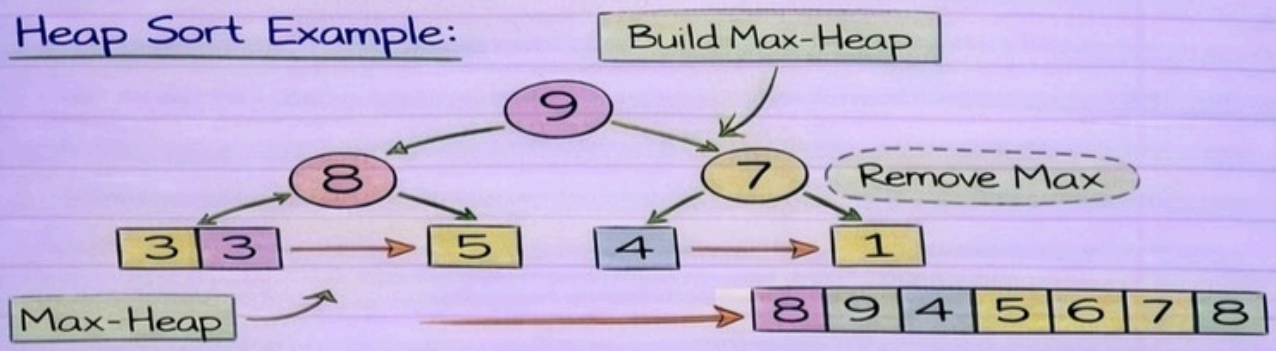
Heap Sort Example

Start:

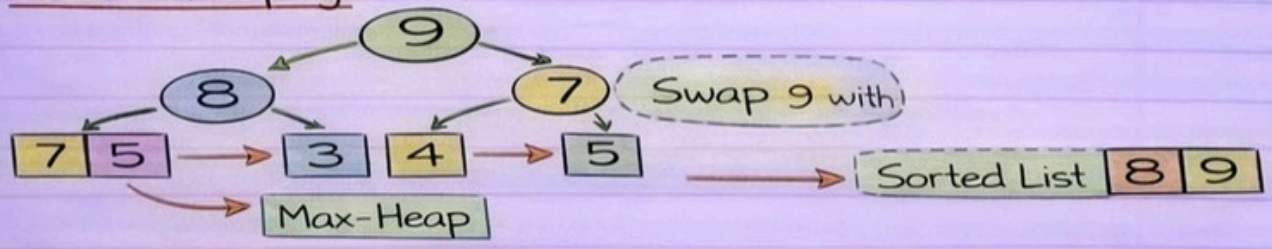
Unsorted Array = [3,9,2,1,4,5,7,8,8]

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to arrange elements in order. It first builds a Max-Heap from the list, then repeatedly removes the maximum element from the heap and places. it at the end of the list, reducing the heap size by one.

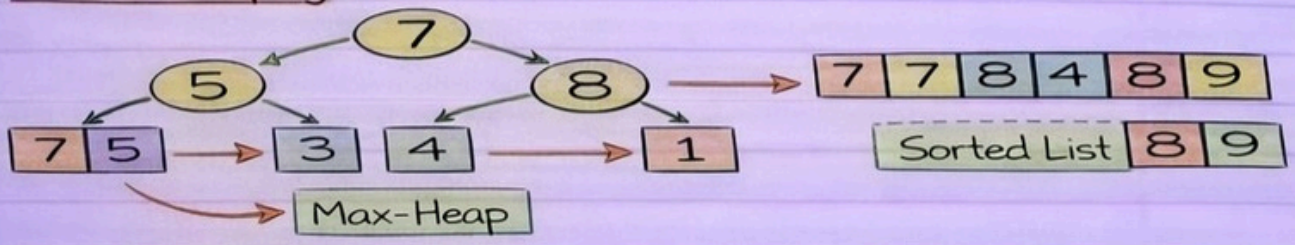
Heap Sort Example:



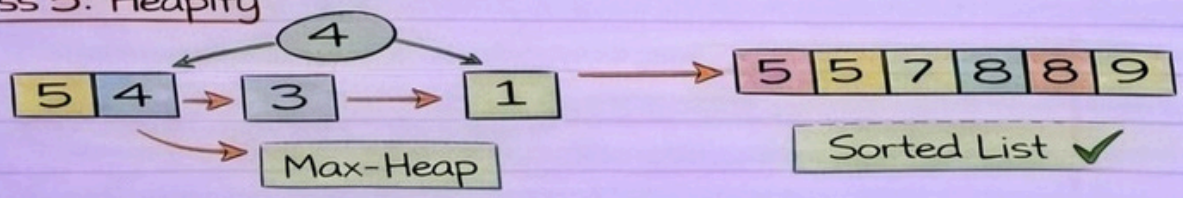
Pass 2: Heapify



Pass 3: Heapify



Pass 5: Heapify



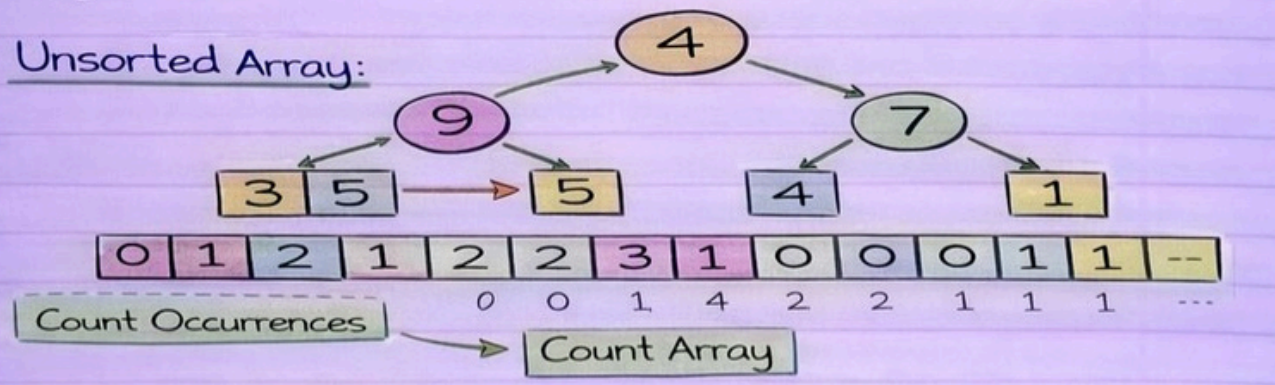
Sorted Array= 1 2 3 4 5 6 7 8 8 9 9 ✓

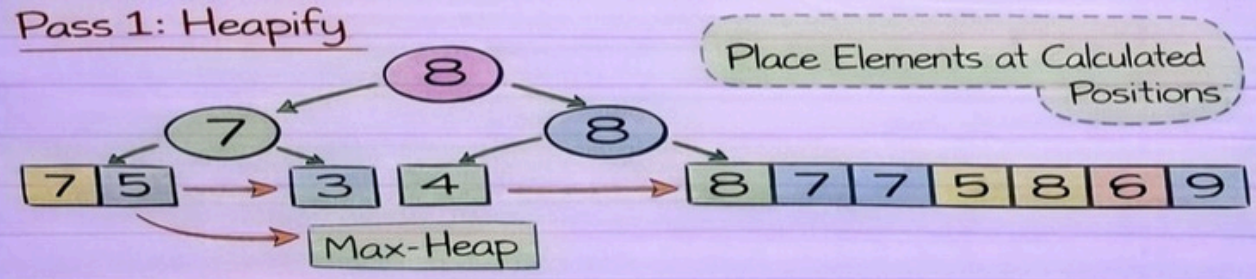
✓ Build a Max-Heap, remove the maximum repeatedly, and place it at the end of the list.

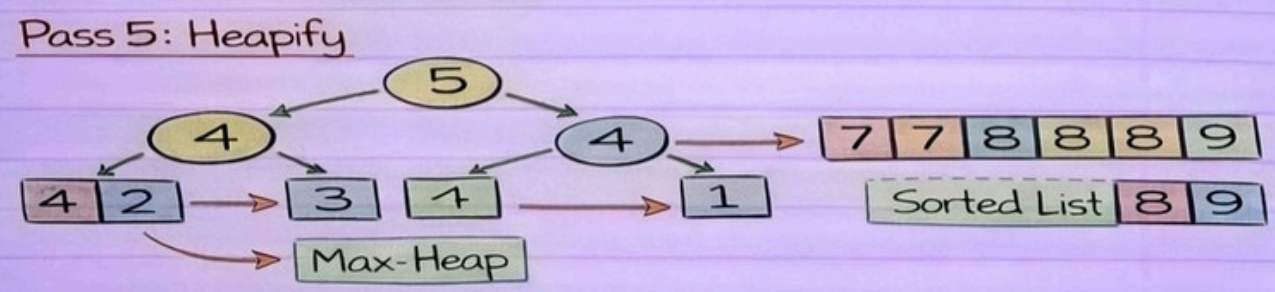
Counting Sort

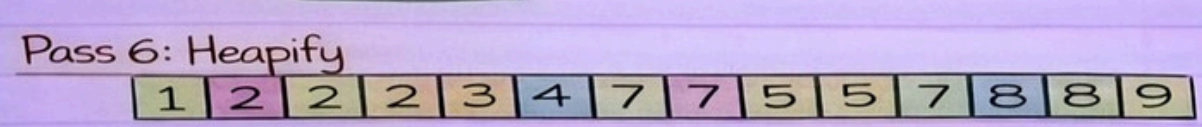
What is Counting Sort?

Counting Sort is a non-comparison-based sorting algorithm that works by counting the occurrences of each unique element in an input list. It creates a count array to store the frequency of each element. Using this count array, it then calculates the positions of each element in the sorted output array. Counting Sort is efficient for sorting integers within a known, limited range.









Key Properties:

- ✓ **Stability:** Counting Sort is stable.
- ✓ **Complexity:** $\Omega(n + k)$: Best, Worst and Average. Case,
  $O(n \log n)$: K'st roight for the snput array.

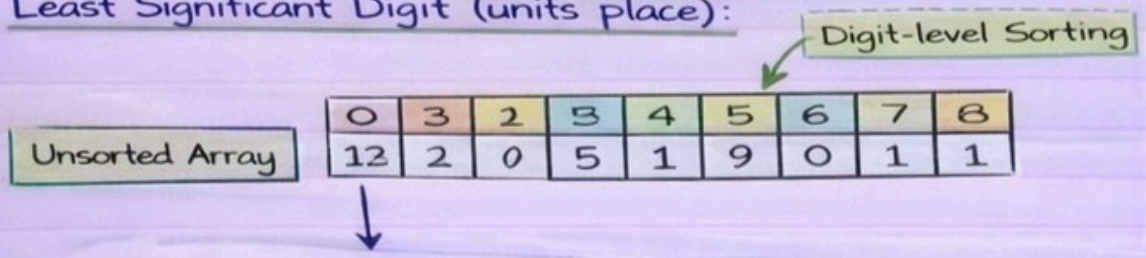
3. **Count Occurrences:** Counts the number of occurrences of each element.
4. **Sort:** Uses the count array to place the elements at their calculated positions in the sorted output array.

Radix Sort

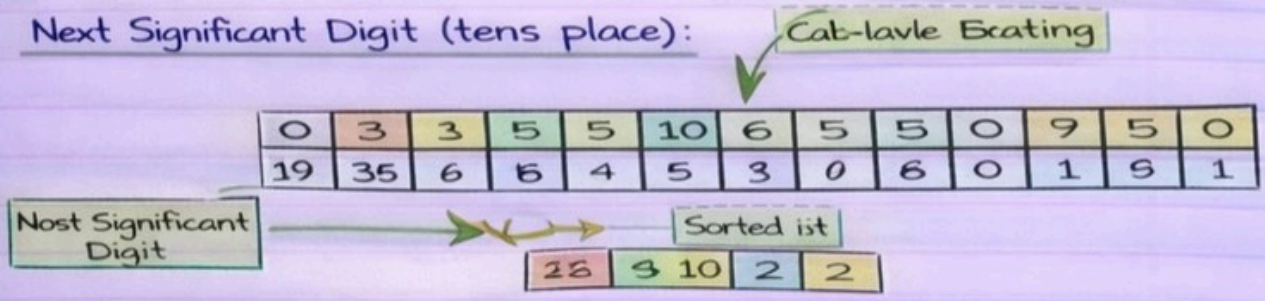
What is Radix Sort?

Radix Sort is a non-comparison-based sorting algorithm that sorts numbers by processing individual digits. It works by sorting the numbers digit by digit, starting from the least significant digit to the most significant digit (LSD to MSD). At each digit, Counting Sort is utilized as a stable sorting subroutine to sort the numbers. Radix Sort is effective for sorting integers. and can also be adapted for fixed-length strings.

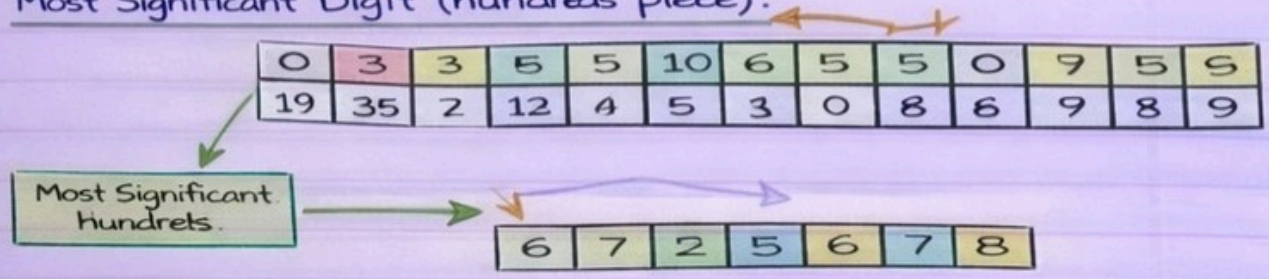
Least Significant Digit (units place):



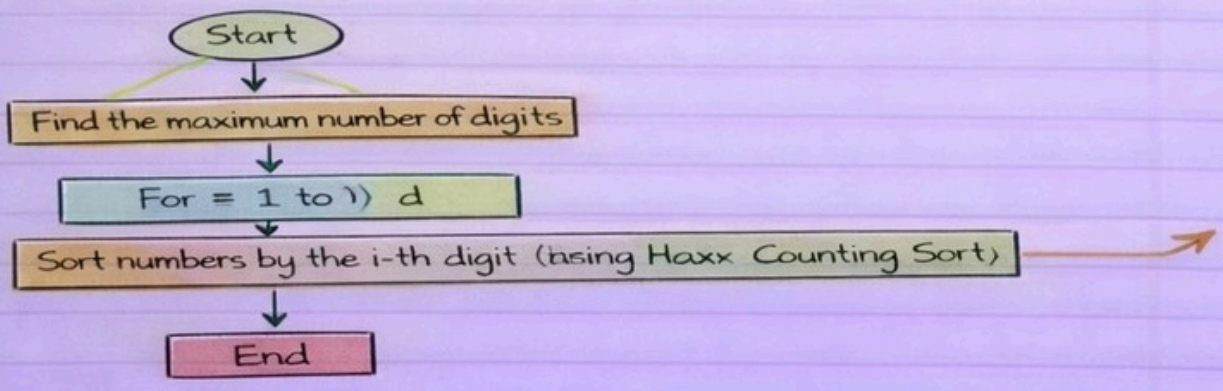
Next Significant Digit (tens place):



Most Significant Digit (hundreds plece):



Radix Sort Algorithm:



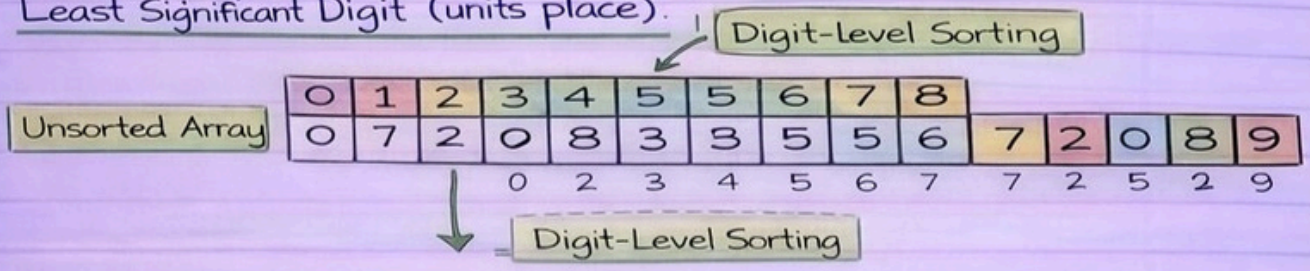
Radix Sort Example

Start:

Unsorted Array = [329, 457, 657, 839, 436, 720, 355]

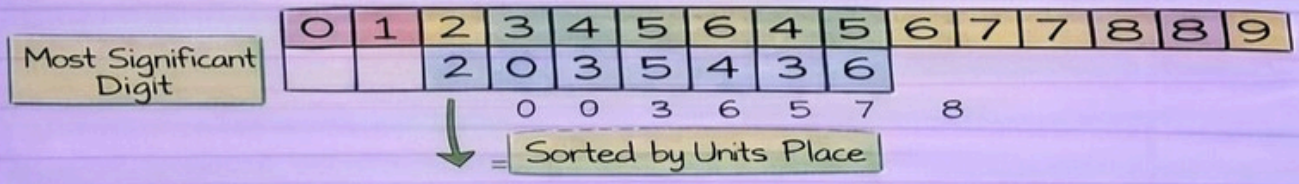
Radix Sort is a non-comparison-based sorting algorithm that sorts numbers by processing individual digits. It works by sorting the numbers digit by digit, storting from the least significant digit (LSD to MSD) At each digit, Counting Sort is utilized as a stable sorting subroutine to sort the numbers Radix Sort is effective for sorting integers and can also be adapted for fixed leiigth strings.

Least Significant Digit (units place).

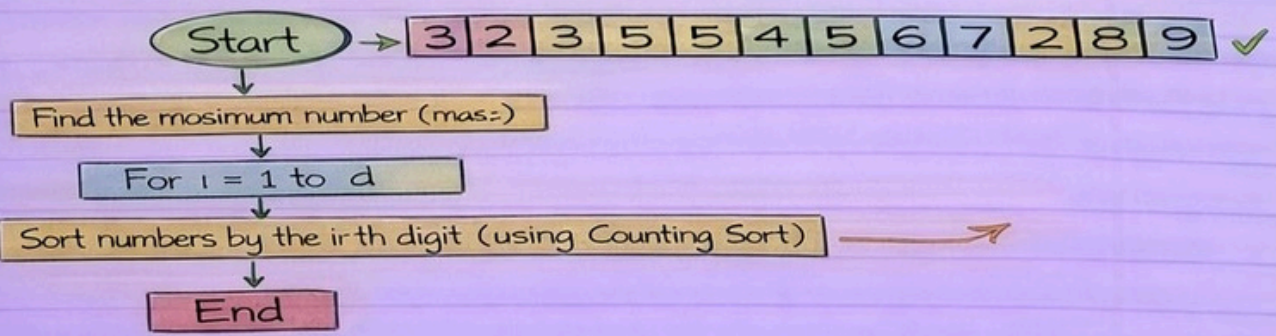


Next Significant Digit (tens place)

Most Significant Digit (hundreds place)



Radix Sort Algorithm:



✓ Start from the least significant digit and sort the numbers by each digit until the list is sorted.

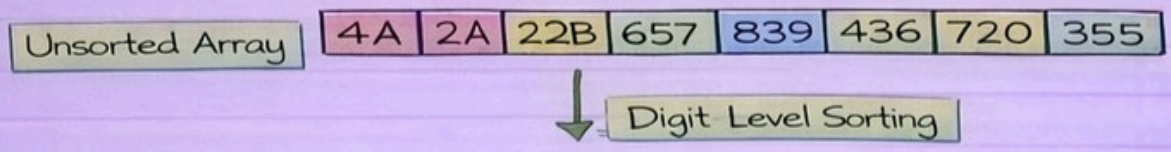
Stability & In-Place Sorting

Stability: @curious_programmer

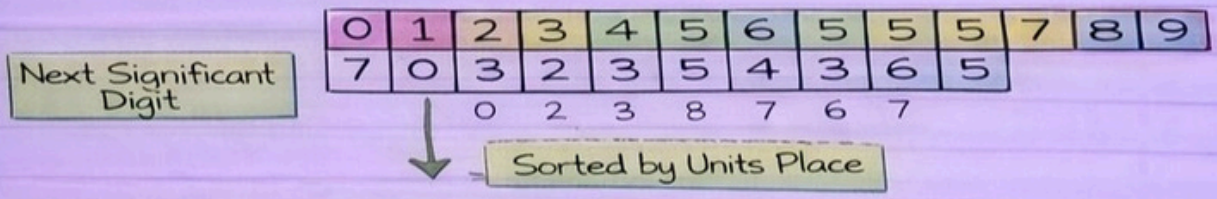
Stability:

Stability refers to the property of a sorting algorithm that preserves the relative order of equal elements in the sorted output as they appeared in the original array At examptar the Stability, Counting Sort is utilized. as a stable sorting subroutine.

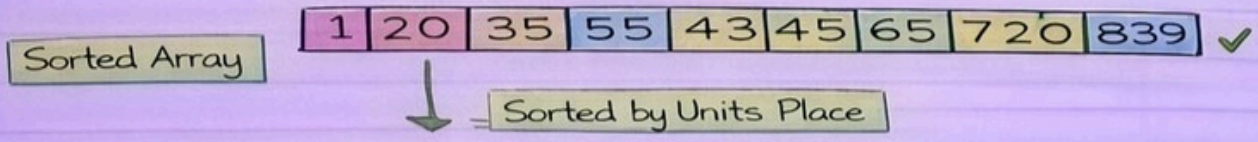
Radix Sort is effective for sorting integers and can also be adopted for fixed length strings.



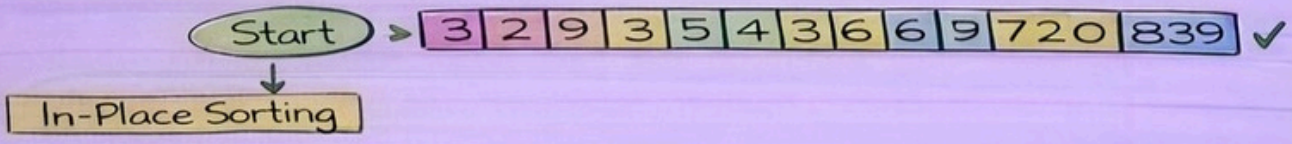
Sort by Next Significant Digit (tens place)



Sort by Hundreds Place:



Radix Sort Algorithm:



- Stort from the least significant digit and sort the numbers by each digit until the list is sorted.

Key Properties:

- **Stability:** Maintains the relative order of equal elements.
- **Complexity:** $O(n+k)$: Best and Average Case. $O(n+k)$: Woret Case, Where M is the number of elements, and K is the digits.
- **Stable Counting Sort:** Uses Counting Sort as a stable subroutine for sorting digits.

Chapter 6: Stack

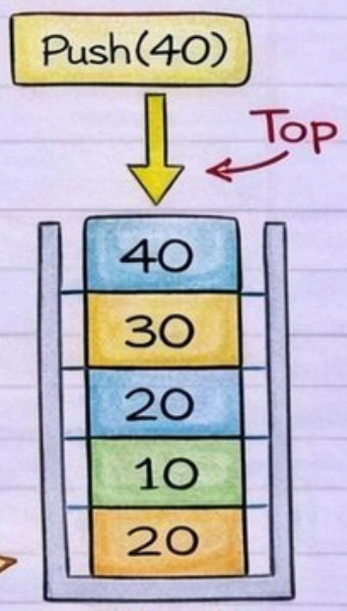
@curious_programmer

What is Stack?

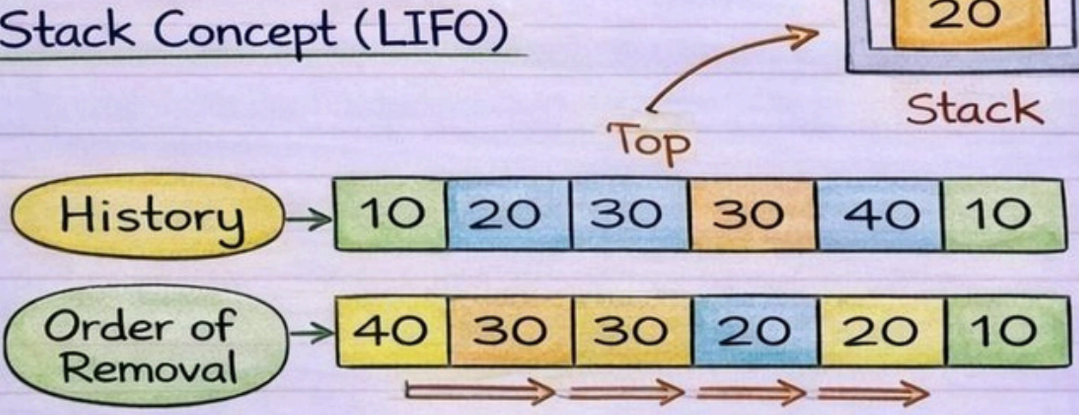
A stack is a linear data structure that follows the **Last In, First Out (LIFO)** principle. This means that the last element added to the stack will be the first to be removed.

Stack Concept (LIFO)

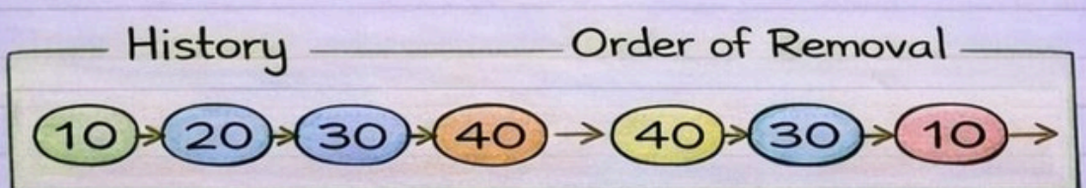
- **LIFO Principle:** The last item pushed onto the stack is the first item to be popped off.
- **Push:** Adding an element to the top of the stack is called "**Push**".
- **Pop:** Removing an element from the top of the stack is called "**Pop**".



Stack Concept (LIFO)



LIFO Order: Last In (40), First Out



Stack Implementation

@curious_programmer

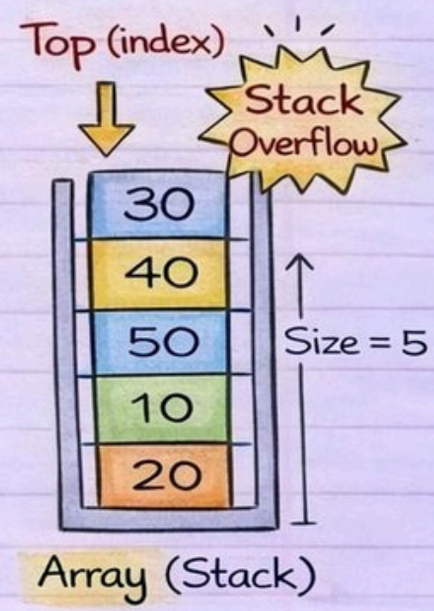
How to Implement a Stack?

Stacks can be implemented in two main ways: Using an Array

- Using a Linked List.

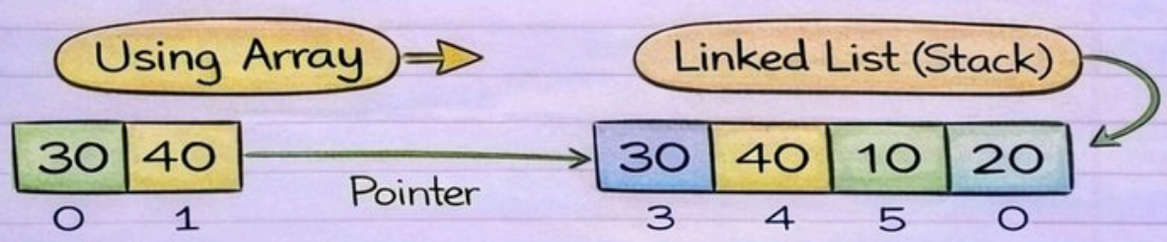
Using Array

- **Fixed Size:** Uses a fixed size array to store the stack elements.
- **Simple & Fast:** Easy to implement, requires less memory and provides fast access times.
- **Risk of Overflow:** Stack Overflow can occur if the stack becomes full and no more elements can be pushed.



Using Linked List

- **Dynamic Size:** Dynamically allocates memory as needed, grows and shrinks as elements are added or removed.
- **Memory Efficient:** No overflow issue as the size of the stack can grow as needed.
- **Slightly Slower:** Linked list implementation may be slightly slower due to memory allocation overhead.



LIFO Order: Last In (40), First Out

Infix, Prefix, Postfix

@ curious_programmer

✔ What are Infix, Prefix, and Postfix Notations?

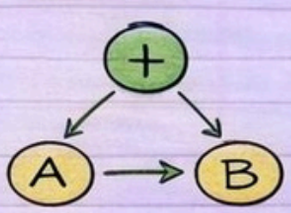
Infix, Prefix (Polish), and Postfix (Reverse Polish) notations are ways to write expressions involving operators and operands. They determine the position of the operator in relation to operands.

Infix Notation → Operator is between the operands.

A + B ——— A + B ——→ A + B

Prefix Notation → Operator is before the operands.

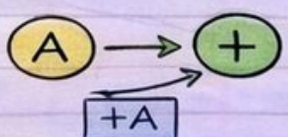
+ A B



Postfix Notation → Operator is after the operands.

A B + ——— A B ——→ A B +

Comparison of Notations



Notation	Notation Type	Example
Infix	Operator in Between	A + B
Prefix	Operator Before	+ A B
Postfix	Operator After	A B +

Notation	Notation Type	Example
Infix	Operator in Between	A + B
Prefix	Operator Before	+ A B
Postfix	Operator After	A B +

Expression Evaluation

@ curious_programmer

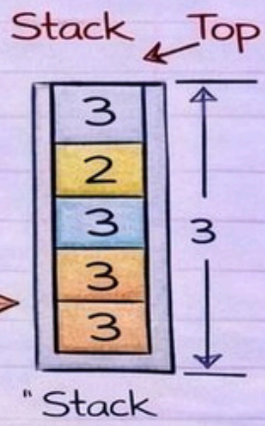
What is Expression Evaluation?

Expression evaluation refers to the process of computing the value of an expression.

- Stacks are used to evaluate expressions written in various notations, such as Infix, Postfix, and Prefix.

How to Evaluate an Expression Using Stack?

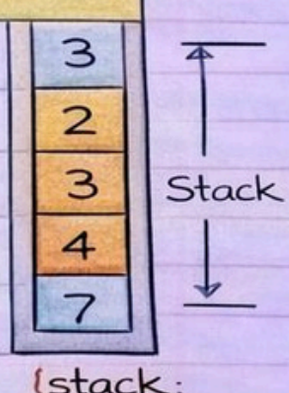
- Convert the expression to Postfix notation (if it is in Infix).
- Scan the Postfix expression from left to right.
- Push operands (numbers) onto the stack.
- When an operator is encountered, pop the top two operands from the stack, apply the operator, and push the result back to the stack.



Example: → 3 + 2 × 2 → 3 2 2 × +

Infix: 3 + 2 × 2 →

Step	Symbol	Stack
1	Push 3	3
2	Push 2	2 + 3
3	Push 2	2 + 3 → 4
4	Apply ×: 2 × 2 = 4	
5	Apply +: 3 + 4 = 7	



Postfix → 3 2 × + → Final Result = 7

Postfix → 7

Next Greater Element

@ curious_programmer

What is Next Greater Element (NGE)?

For each element in an array, the Next Greater Element (NGE) is the first greater element that is to the right.

- If there is no greater element, the NGE is -1.

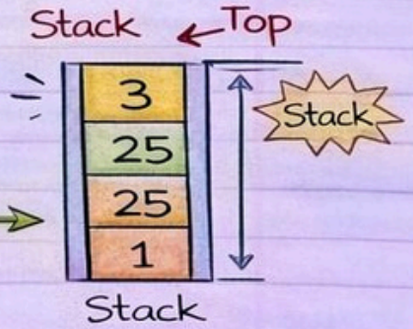
How to Evaluate an Expression Using Stack?

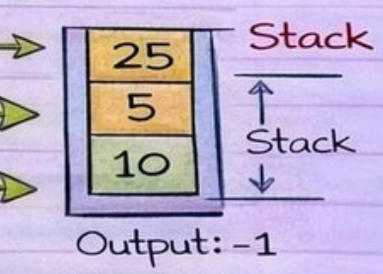
- Initialize an empty stack to keep track of elements to find their NGE.
- Scan the array from right to left.
- Pop elements from the stack untill you find an element greater than the current element.
- The NGE is the top element of the stack, or -1 if the stack is empty.

Example: → {3 + 5, 2 10, 8, 6, 25, 1 }

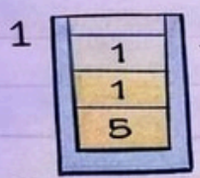
Infix : 3 + 2 · 2 → Postfix: 3 2 2 x +

Step		Symbol	Stack
①	1	Push 3	3
②	2	Push 2	2 | 2 | 3
③	3	Push 2	2 → 2 → 3
④	4	Apply x : 2 × 2 = 4	
⑤	5	Pop 10, Push 10 onto stack	
⑥	6	Pop 10, Push 10 onto stack	





Output: -1



Output: -1

Stack: 1 | 3 2 x 2 +

Stack:

Final Result = 7 → 7

Applications of Stack

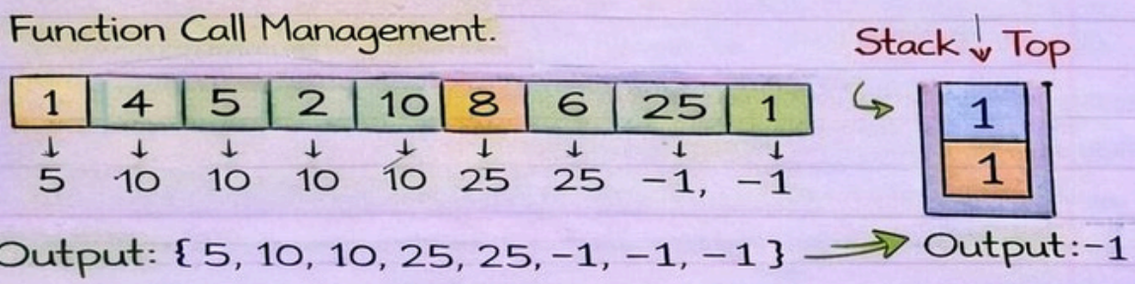
@ curious_programmer

✔ Where is Stack Used?

Stacks have a wide range of practical applications in computer science where order matters.

Common Applications of Stacks:

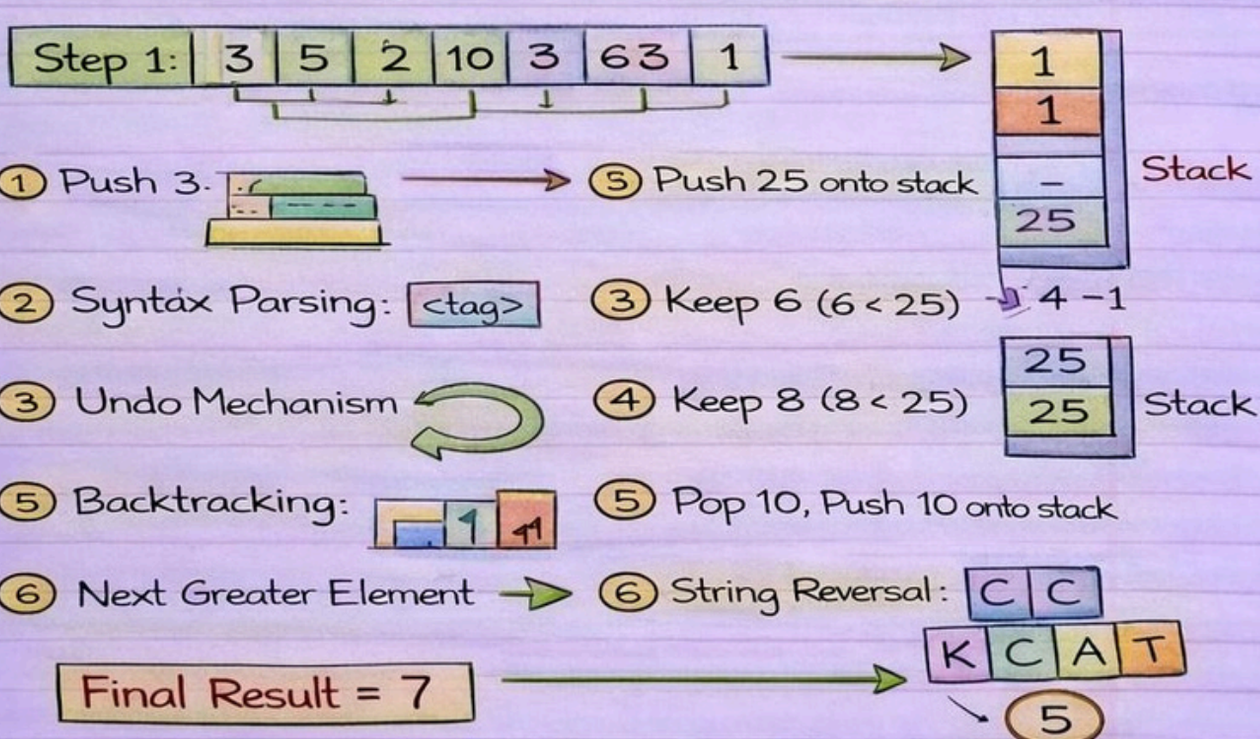
✔ Function Call Management.



Output: { 5, 10, 10, 25, 25, −1, −1, −1 } → Output: −1

✔ Common Applications of Stacks:

- ✔ Intialize an empty stack to keep track of elements to find their NGE.
- ✔ Scan the array from right to left.
- ✔ Pop elements from the stack until you find an element greater than the current element.
- ✔ The NGE is the top element of the stack, or −1 if the stack is empty.



Chapter 9: Queue

@curious_programmer

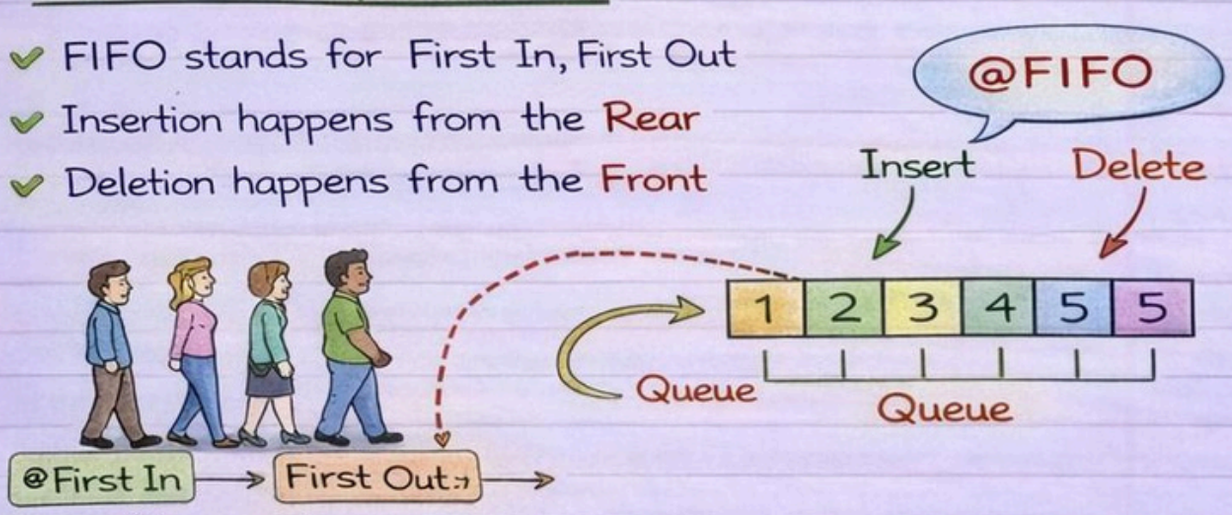
What is a Queue?

A Queue is a linear data structure that follows the FIFO (First In, First Out)) principle.
This means the element inserted first will be be removed first.

Example:
People standing in a queue – the person who comes first gets served first.

Queue Concept (FIFO)

- FIFO stands for First In, First Out
- Insertion happens from the Rear
- Deletion happens from the Front



Basic Operations:

- Enqueue : Insert an element at the rear
- Dequeue : Remove an element from the front
- Front / Peek : View the front element
- IsEmpty : Check if queue is empty
- IsFull : Check if queue is full

Simple Queue

@curious_programmer

What is a Simple Queue?

A Simple Queue is the basic implementation of a queue where:

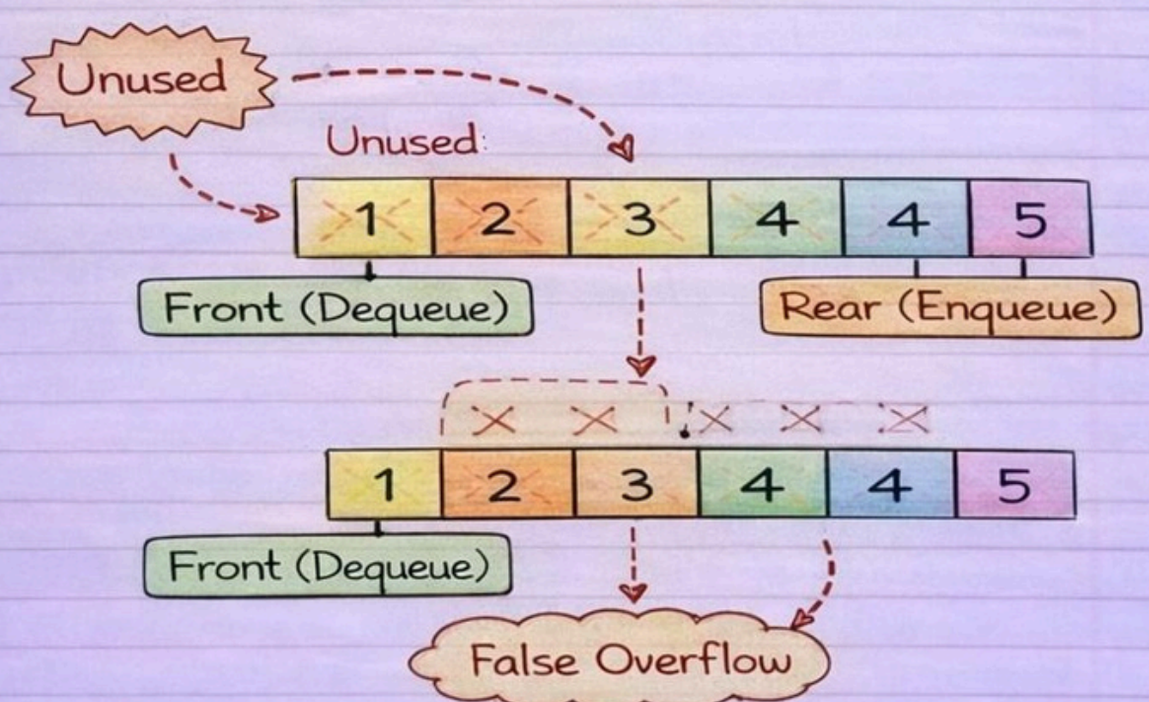
- Elements are inserted from the rear
- Elements are removed from the front

Characteristics:

- Follows FIFO
- Fixed size (in array implementation)
- Wastage of memory can occur

Limitation:

⚠ Once elements are removed from the front, the empty space cannot be reused, which leads to false overflow.



Circular Queue

@curious_programmer

What is a Circular Queue?

A Circular Queue is an improved version of a simple queue where the last position is connected back to the first position.

Why Circular Queue?

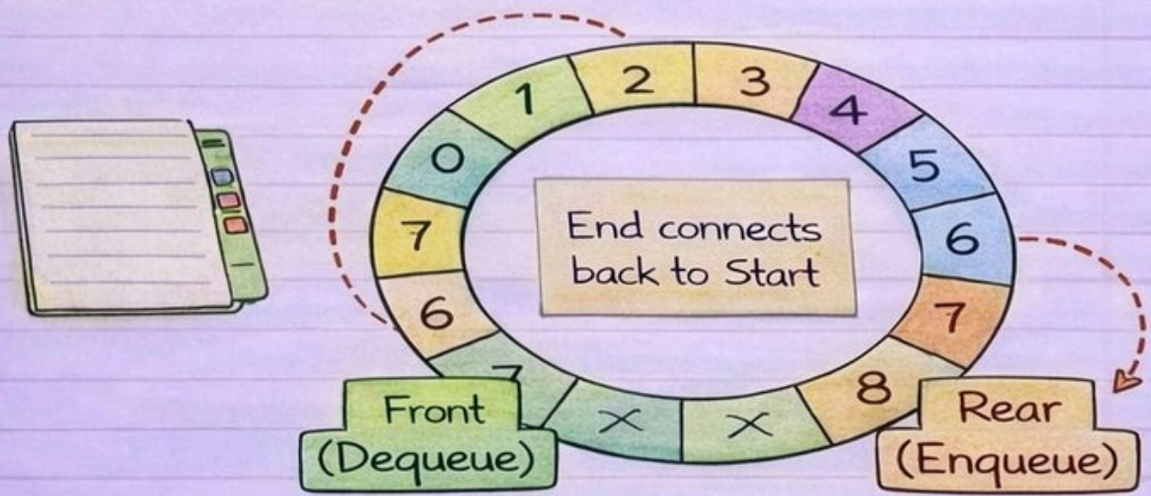
- Overcomes the memory wastage problem of simple queue
- Reuses empty spaces efficiently

Characteristics:

- Front and Rear move circularly
- Last index is connected to first index
- Better memory utilization

Condition to move circularly:

((index + 1) % size)



Condition to move circularly:

(index + 1) % size

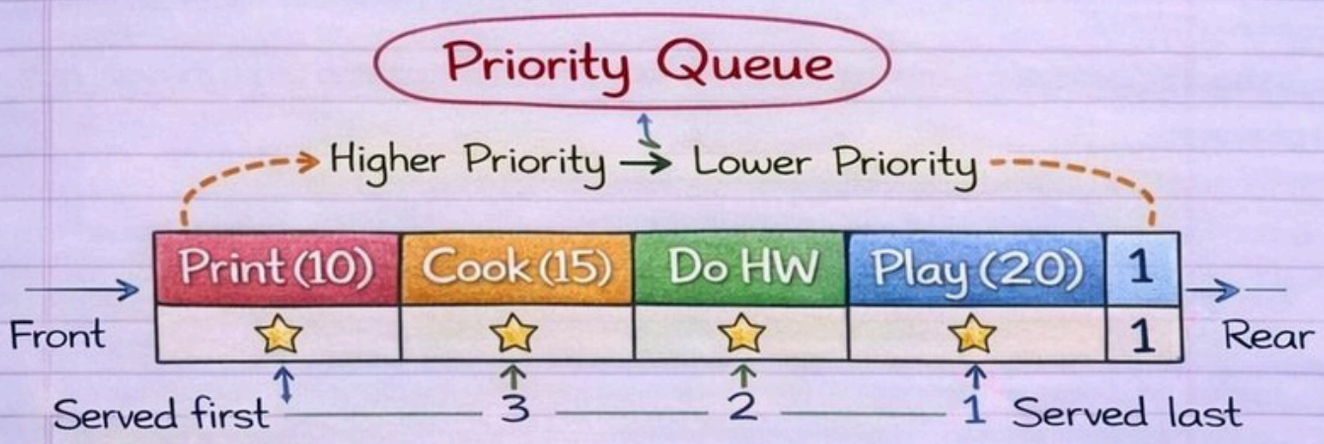
Chapter 9: Queue

@curious_programmer

✓ Priority Queue

✓ What is a Priority Queue?

A Priority Queue is a special type of queue where each element has a priority, Elements with higher priority are dequeued before elements with lower priority. If two elements have the same priority, they are dequeued in FIFO (First In, First Out) order.



✓ Properties of Priority Queue

- ✓ Each element has a priority.
- ✓ Element with higher priority is dequeued first.
- ✓ If priority is same → FIFO order is followed.

Used in:

- ✓ CPU Scheduling
- ✓ Dijkstra's Algorithm

✓ Properties of Priority Queue

- ✓ Each element has a priority.
- ✓ Element with higher priority is dequeued first.
- ✓ If priority is same → FIFO order is followed.

Deque (Double Ended Queue)

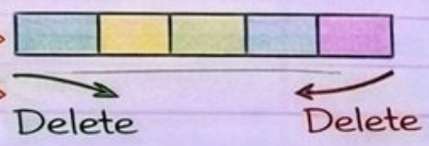
@curious_programmer

What is Deque?

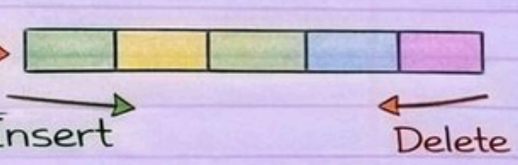
A Deque is a queue in which insertion and deletion can be done from both ends.

Types of Deque:

- Input Restricted Deque
  - Insertion allowed at one end only
  - Deletion allowed at both ends



- Output Restricted Deque
  - Insertion allowed at both ends
  - Deletion allowed at one end only



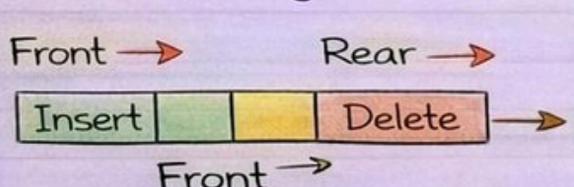
Applications:

- Undo/Redo operations
- Sliding window problems

Types of Deque

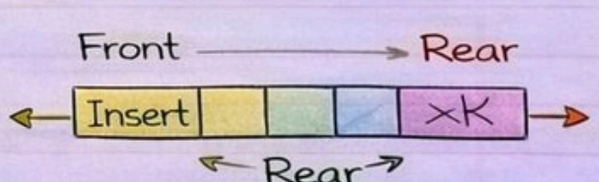
Input Restricted Deque

- Insertion allowed at one end only



Output Restricted Deque

- Insertion allowed at both ends
- Deletion allowed at one end



Chapter 9: Queue

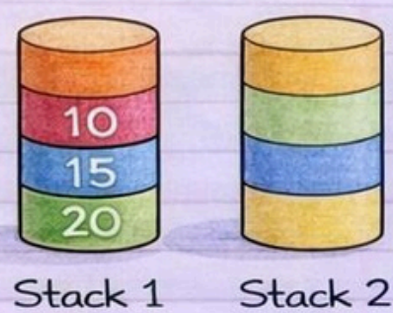
@curious_programmer

Queue Using Stack

How to implement a Queue Using Stacks?

A queue can be implemented using two stacks by applying logic to achieve FIFO (First In, First Out) order.

- ☑ **Stack 1** : used for enqueue
- ☑ **Stack 2** : used for dequeue

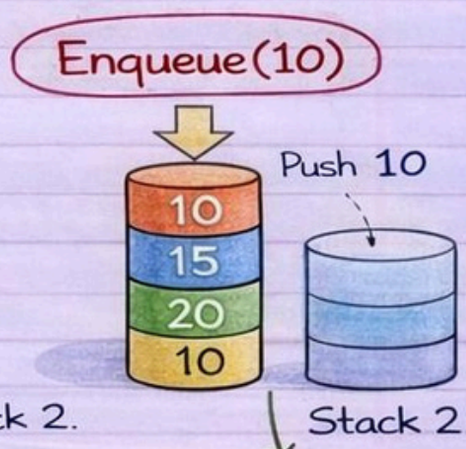


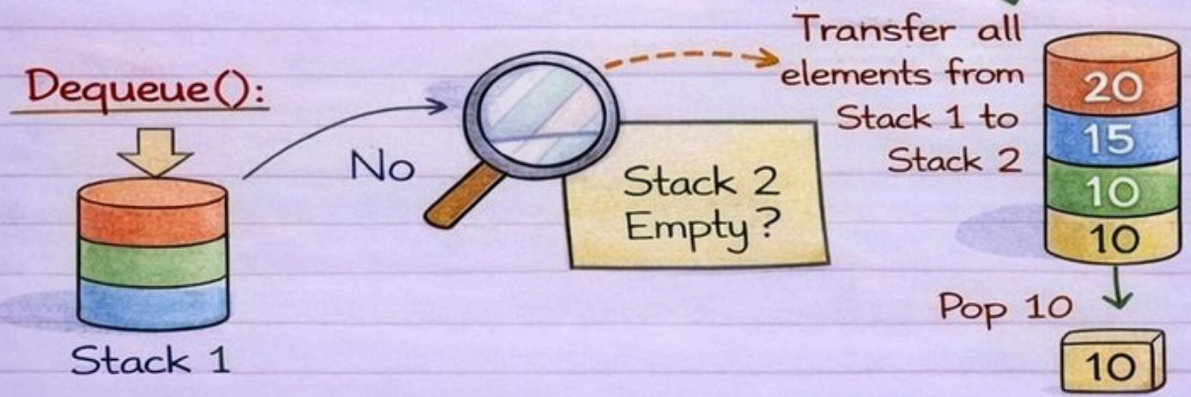
Approach:

- Stack 1 → used for enqueue
- Stack 2 → used for dequeue

Enqueue Operation:

1. If Stack 2 is empty:
   - Transfer all elements from Stack 1 to Stack 2.
2. Pop the element from the top of Stack 2.





Properties of Queue Using Stack:

- Two stacks are used to implement a queue.
- Stack follous LIFO, but logic is applied to achieve FIFO.
- Stack 1 for enqueueing : new elements are pushed onto Stack 1.

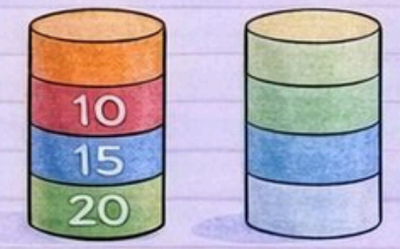
Stack Using Queue

How to implement a Stack Using Queues?

A stack can be implemented using two queues by applying logic to achieve LIFO (Last In, First Out) order.

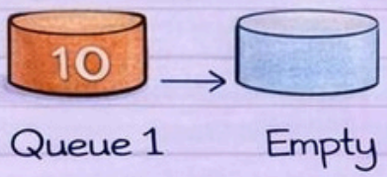
Approaches:

- Using two queues
- Using one queue (by rotation)



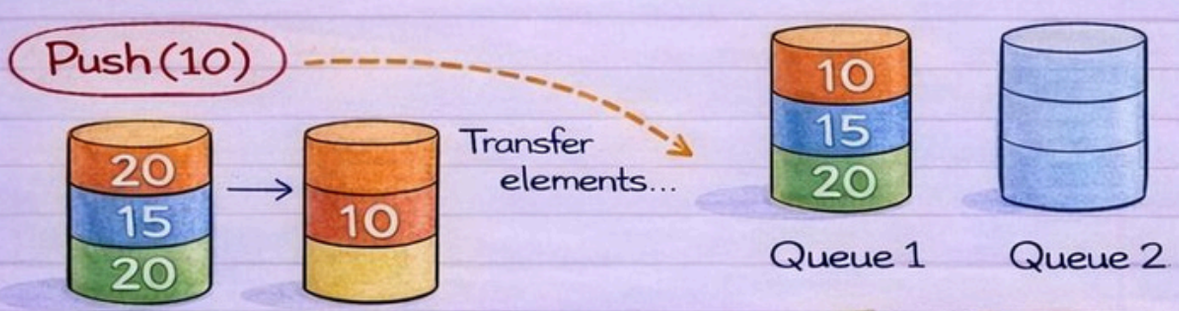
Approaches:

- Q1: used as queues
- Q2: used as auxiliary

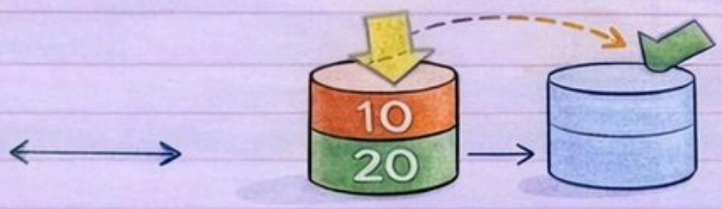


Push Operation:

1. Push new element to Q2.
2. Transfer all elements from Q1 to Q2.
3. Swap the names of Q1 and Q2.



- Swap queue names.



Properties of Stack Using Queue:

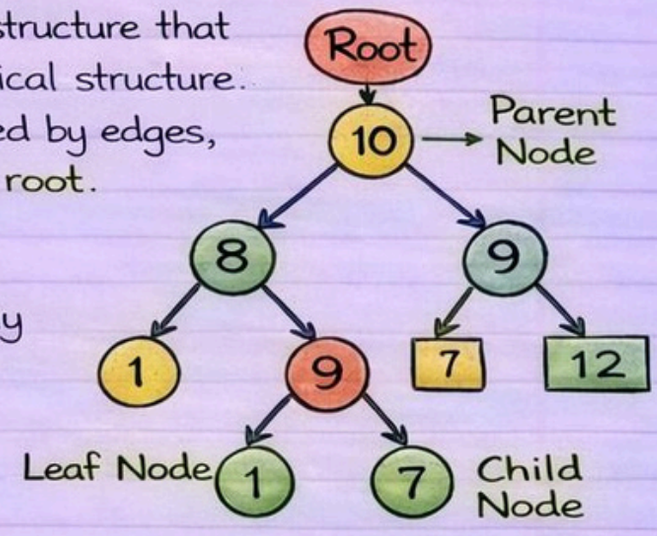
- Two queues are used to implement a stack.
- Queue follows FIFO, but logic is applied to acchieve LIFO.
- Approach 1: Q1 acts as the primory queue, Q2 acts as auxilary queue.

Chapter 10: Trees

@curious_programmer

What is a Tree?

A tree is a non-linear data structure that represents data in a hierarchical structure. It consists of nodes connected by edges, with a special node called the root.

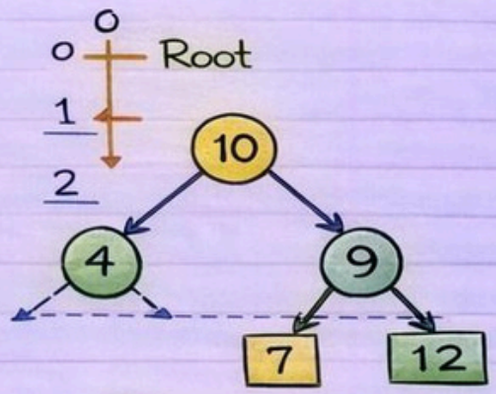


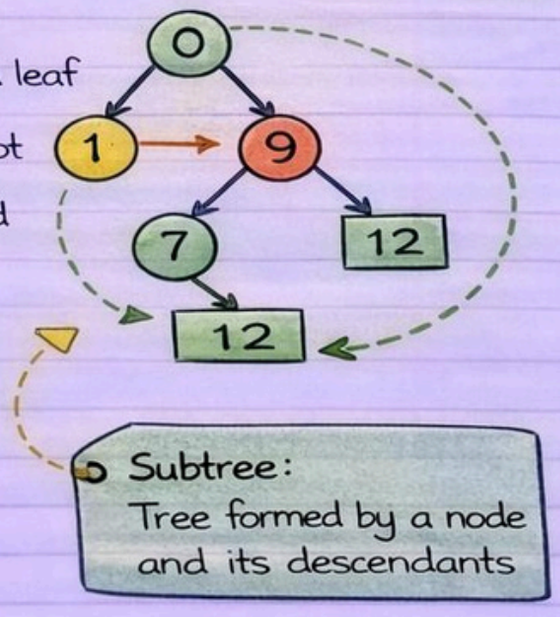
Key Points:

- Organizes data hierarchically
- No cycles
- One root node

Tree Terminology

- **Node:** Each element in a tree
- **Root:** Topmost node of the tree
- **Parent:** Node that has children
- **Child:** Node derived from a parent
- **Leaf Node:** Node with no children
- **Edge:** Connection between two nodes
- **Level:** Distance from root
- **Height:** Longest path from root to a leaf
- **Depth:** Distance of a node from root
- **Subtree:** Tree formed by a node and its descendants





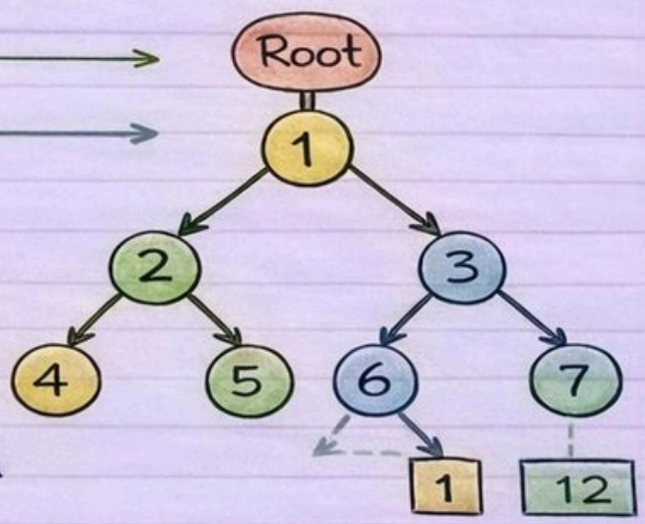
Binary Tree

@curious_programmer

What is a Binary Tree?

A Binary Tree is a tree in which each node has at most two children:

- Left Child
- Right Child

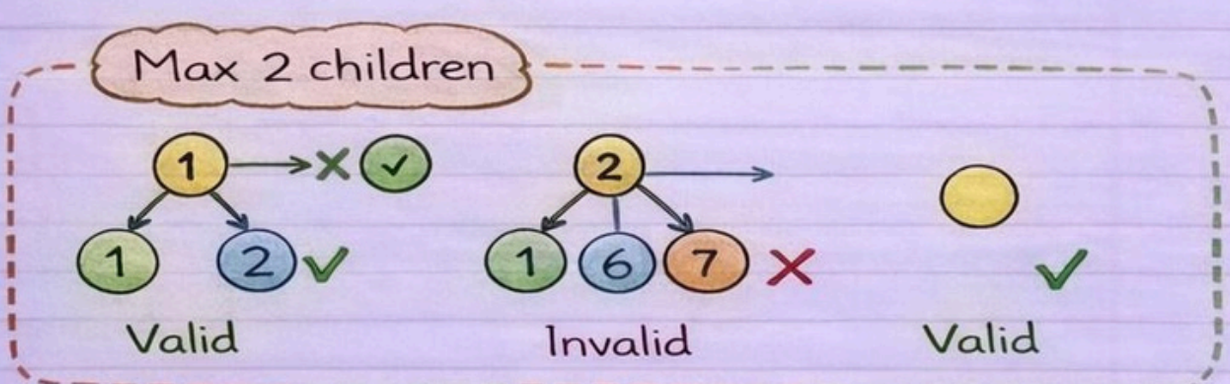


Properties:

- Maximum 2 children per node
- Children are ordered (left & right)
- Can be empty

Tree Terminology

- Maximum 2 children per node.
- Children are ordered (left & right)
- Can be empty



Binary Search Tree (BST)

@curious_programmer

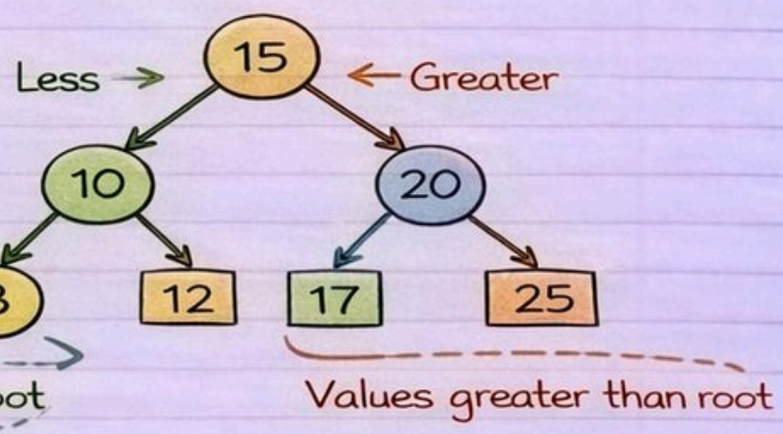
What is a Binary Search Tree?

A Binary Search Tree is a binary tree that follows a specific order:

- Left subtree → values less than root
- Right subtree → values greater than root

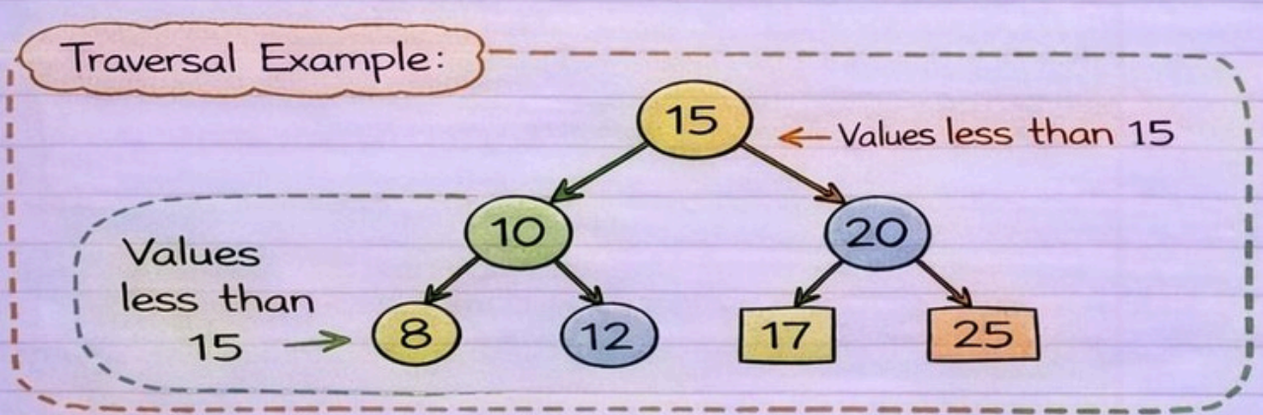
BST Property:

- Left subtree → values less than root



Advantages:

- Faster searching
- Efficient insertion & deletion



Output: – 8, 10, 12, 15, 17, 20, 25, ↑ ↑ ↑

Tree Traversals

@curious_programmer

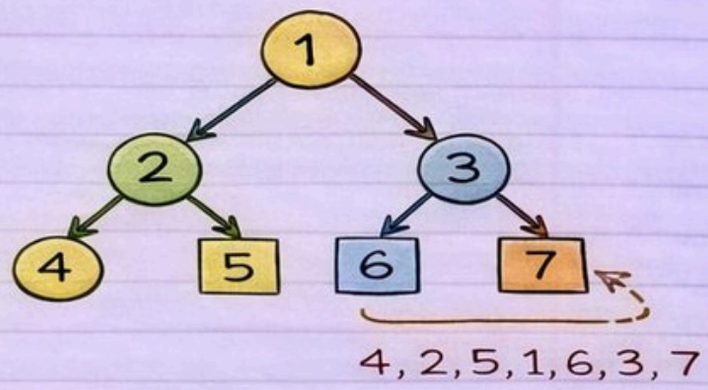
✔ Tree traversal means visiting all nodes of a tree exactly once in a specific order:

Inorder Traversal (L → Root → R)

✔ Steps :

1. Visit Left subtree
2. Visit Root
3. Visit Right subtree

Values less than root



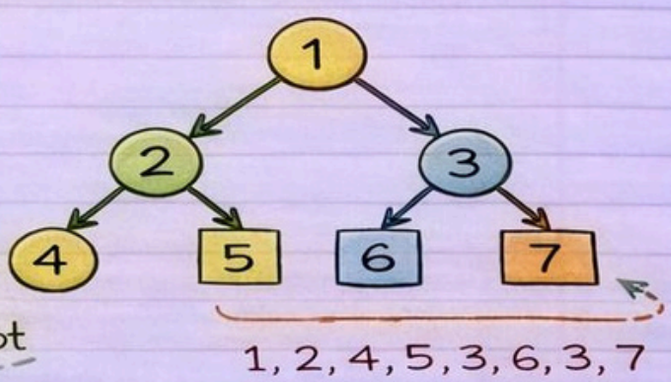
4, 2, 5, 1, 6, 3, 7

Preorder Traversal (Root → L → R)

✔ Steps :

1. Visit Root
2. Visit Left subtree
3. Visit Right subtree

Values greater than root

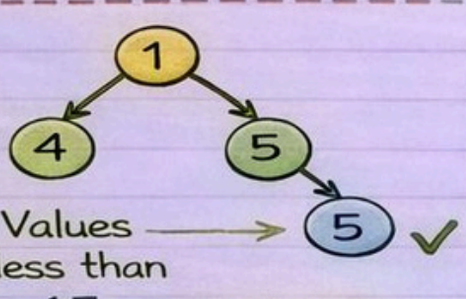


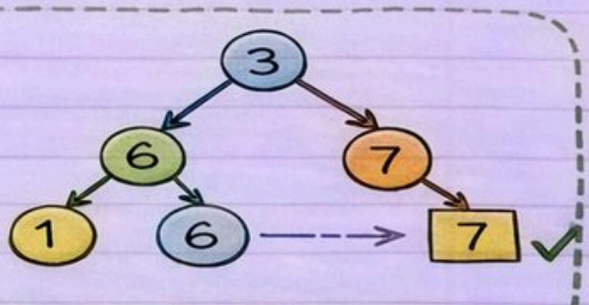
1, 2, 4, 5, 3, 6, 3, 7

Level Order Traversal

✔ Traversal is done level by level from left to right.

✔ Uses Queue data structure.





Output: 1, 2, 3, 4, 5, 7, 20, 25, ↑ ↑ ↑

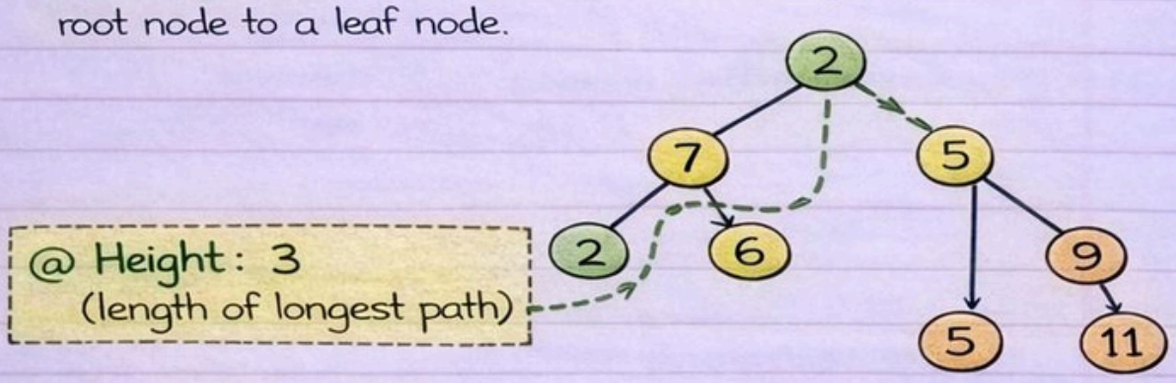
Height & Diameter, Lowest Common Ancestor

@ curious_programmer

✔ Height & Diameter

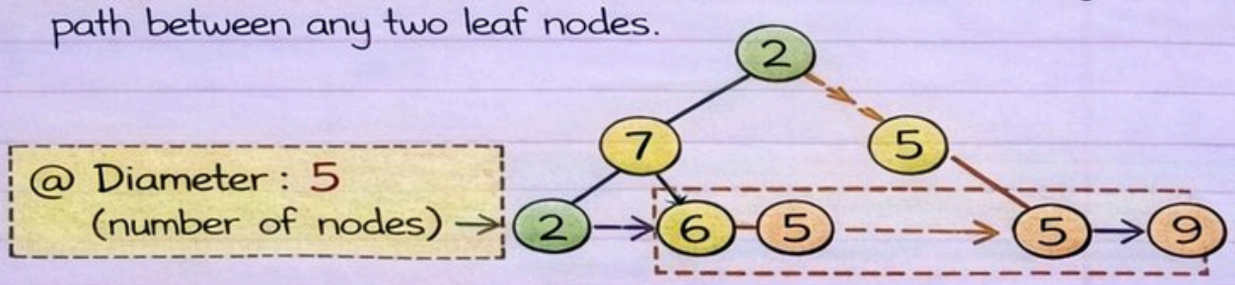
✔ What is Tree Height ?

- The height of a tree is the length of the longest path from the root node to a leaf node.



✔ What is Tree Diameter ?

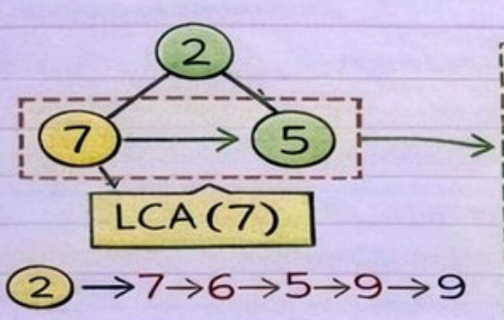
- The diameter of a tree is the number of nodes on the longest path between any two leaf nodes.



Lowest Common Ancestor

✔ What is Lowest Common Ancestor ?

- The Lowest Common Ancestor (LCA) of two nodes is the deepest node that is a common ancestor of both nodes.



@ Example:

- The LCA of nodes 5 and 11. So, 7 is the LCA.
- 5 is an ancestor of 5, and 11. So, 7 is the LCA.

Chapter 11: Heaps

@curious_programmer

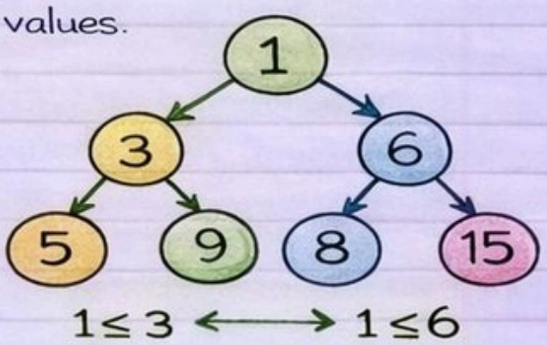
✔ What is a Heap?

- ✔ A heap is a complete binary tree that follows a specific heap property.
  - Heaps are mainly used to quickly access the minimum or maximum element.
  - Heaps can be of two types:

✔ Min Heap

- The minimum element is always at the root.
- Parent node value ≤ children values.
- Root = smallest element

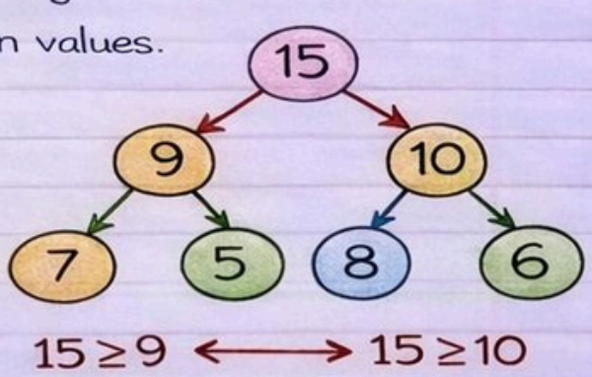
✔ Used in:
- Priority Queue (min priority)
- Dijkstra's Algorithm



✔ Max Heap

- The maximum element is always at the root.
- Parent node value ≥ children values.
- Root = largest element

Used in:
- Heap Sort
- Priority Queue (max priority)



Min Heap	Root →	1 ≤ 3	→	1 ≤ 6

Heap Operations

@curious_programmer

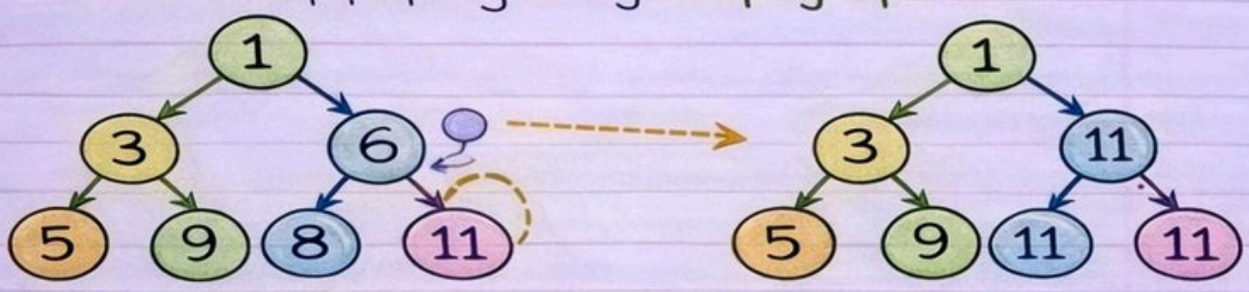
- Efficient processing of minimum or maximum element
- Commonly performed in O(log n) time

Heap Operations

1 Insertion:

Time Complexity: O(log n)

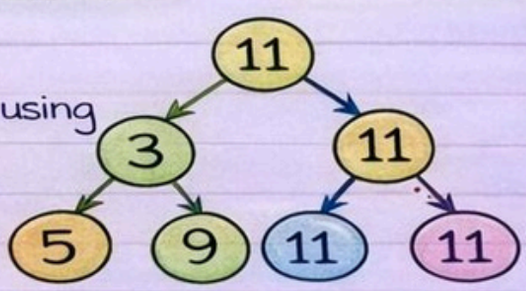
- Insert element at the end
- Restore heap property using heapify-up



2 Deletion (Remove Root):

Time Complexity: O(log n)

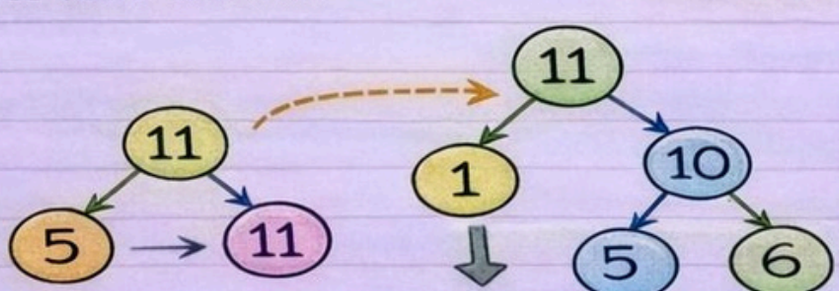
- Replace root with last element
- Reduce heap size
- Restore heap property using heapify-down



3 Peek (Get Min/Max):

Time Complexity: O(1)

- Access root element



Heapify

@curious_programmer

- Keep Heap valid
- Efficient rearranging in O(log n) time

What is Heapify?

Heapify is the process of rearranging elements to maintain the heap property.

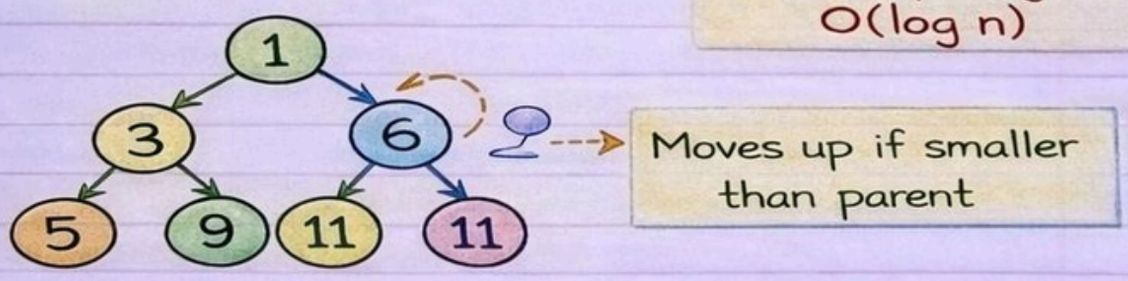
- It fixes the order of elements so that the heap is valid.

Types of Heapify

↑ Heapify-Up

✓ Used after insertion

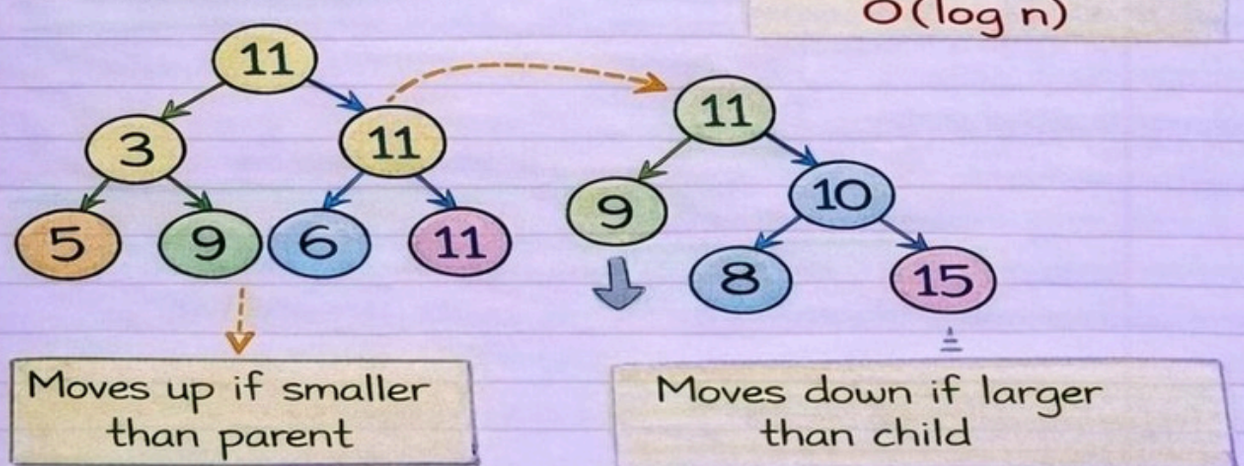
Time Complexity: O(log n)



② Heapify-Down

✓ Used after deletion

Time Complexity: O(log n)



Priority Queue using Heap

@ curious_programmer

✓ What is a Priority Queue?

A Priority Queue is an abstract data type where each element has a priority.

- Elements with the highest/ lowest priority are served first.

Priority Queue using Heap

✓ Heap is an efficient way to implement a Priority Queue.

↑ Min Heap

✓ Heap is an efficient way to implement a Priority Queue.

✓ Min Heap

- Lowest priority first
- Use Min Heap

Heap Operations:

- ✓ Insert: O(log n)
- ✓ Delete: O(log n)
- Peek: O(1)

✓ Min Heap

Priority	Element
1	10 10
2	20 20
3	30

Heaps priority first Served

✓ Max Heap

Priority	Element
3	30 10
2	20 10
1	10 15

Moves down if larger than child

Applications:

- ✓ Scheduling Tasks
- ✓ Dijkstra's Algorithm
- ✓ Load Balancing

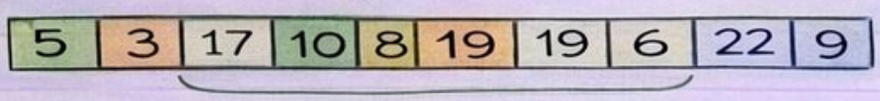
Heap Sort

@ curious_programmer

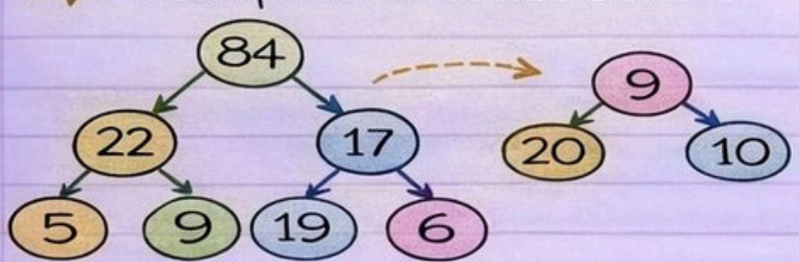
Heap Sort is a comparison-based sorting algorithm that uses a heap.

Steps:

1. Build a Max Heap

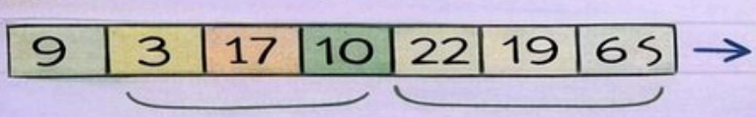


2. Swap root with last element



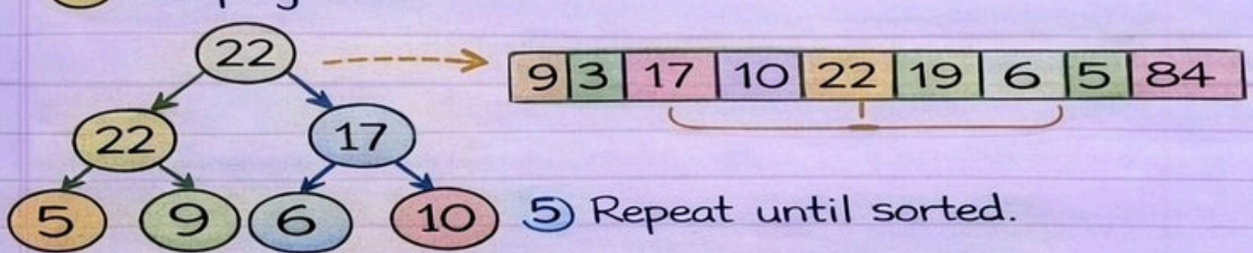
Time Complexity:
O(log n)

3. Reduce heap size:

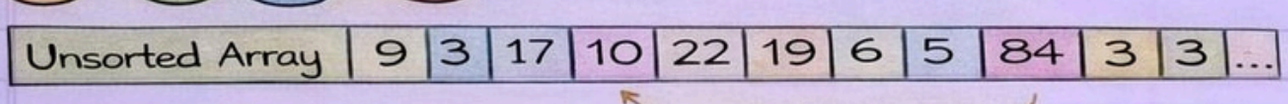


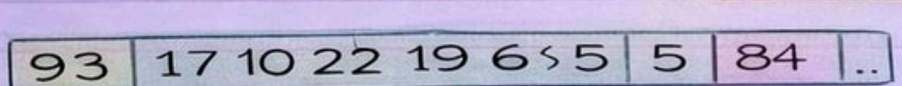
Heap Operations:
- ✔ Insert: O(log n)
- ✔ Delete: O(log n)
- ○ Peek: O(1)

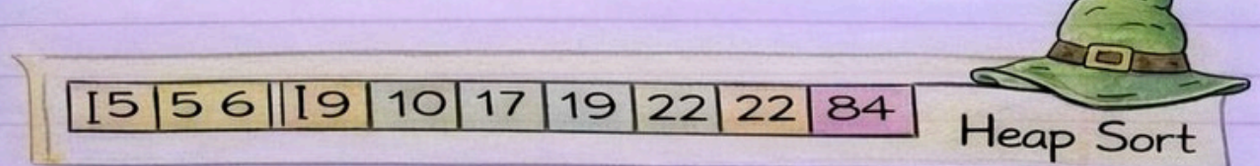
4. Heapify root



5. Repeat until sorted.







Heap Sort

Chapter 12: Hash Table

@curious_programmer

What is a Hash Table?

A Hash Table is a data structure that stores data in key-value pairs using a hash function.

- It allows fast insertion, deletion, and search
- Average time complexity is O(1)

Hash Function:

- A collision occurs when two different keys map to the same index.

Why collisions happen?

- Limited table size
- Poor hash function
- Fast to compute
- Minimizes collisions

Hash Function:

- Index = (ba + i)
- Index = (ba + b)

1 Build a Max Heap

101 = 532 Alex
532 = Bob
207 = 207 Cathy

Hash Table

Index	Value
0	532 : Bob
1	207 : Cathy
2	
3	
6	

Example:

hash(key) = key % table..size

Properties:

- Fast: Average O(1) time complexity
- Dynanic: Size of table grows as needed

Time Complexities:

- Insert O(1)
- Delete: O(1)
- Search: O(1)

Properties:

- Fast: Average O(1) time complexity
- Dynamic: Size of table grows as needed
- Unordered: No specific order of elements

Collision Handling & Chaining

@ curious_programmer

What is Collision Handling?

Collision Handling is the process of resolving hash table collisions (when two keys hash to the same index).

- It allows fast insertion, deletion, and search
- Average time complexity is O(1)

Chaining:

Chaining is a collision resolution technique where each index in the hash table points to a linked list of nodes (buckets).

Before:

Hash Table

0	152 : Bob
1	
2	
3	
4	

After Chaining

Index	Value
0	152: Amy
1	172: Bob
2	229: Sam
3	
5	

102: Bob → Collision! → 152 : Amy

2 After Chaining:

- Simple to implement
- Efficiently uses table size
- No overflow: can store more elements than table size

Time Complexities:

- Insert: O(1)
- Delete: O(1)
- Search: O(1)

Benefits of Chaining:

- Simple to implement
- Efficiently uses table size
- No overflow: can store more elements than table size

Open Addressing

@curious_programmer

What is Open Addressing?

In Open Addressing, all elements are stored inside the hash table itself:

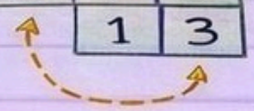
- It allows fast insertion, deletion, and search
- Average time complexity is O(1)

Types of Open Addressing:

Linear Probing

- index = (hash(key) + i) % table..size
- Checks next slot sequentially.

0	1	2	3	4	5	6	7	6
							1	3



O(1) Fast Lookups

Hash Function:

- Uses quadratic formula to reduce clustering

- Easy to implement
- Leads to clustering

Quadratic Probing

- index = (hash(key) + i^2) % table..size
- Uses quadratic formula to reduce clustering

HeapFunction:

- Reduces clustering
- Can still cause secondary clustering

Double Hashing

- index = (hash1(key) + i + hash2(key)) % table..size
- Best collision distribution

Properties:

- Fast: Average O(1) time complexity
- Dynamic: Size of table grows as needed

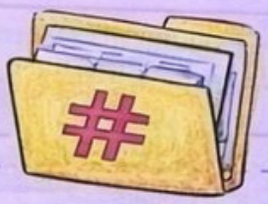
Time Complexities:

- Insert: O(1)
- Delete: O(1)
- Search: O(1)

15	6	9	10	17	19	22	84	..

Applications of Hashing

@curious_programmer



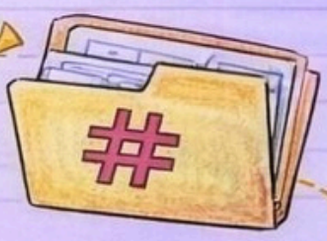
What is Open Addressing?

Collision Handling is the process of resolving hash table collisions (when two keys hash to the same index).

- Average time complexity is O(1)

1 Chaining

Chaining is a collision resolution technique where each index in the hash table points to a linked list.



Types of Open Addressing:

Linear Probing

- index = (hash(key) + i) % table..size
- Checks next slot sequentially

0	1	2	3	4	5	6	7	8	9	0	7	5

- Easy to implement
- Leads to clustering

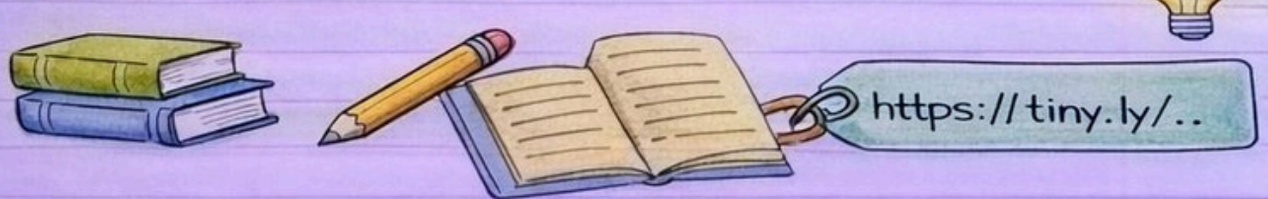
O(1) Fast Lookups

Quadratie Probing

- index = (hash(key) + i^2) % table..size
- Uses quadratic formula to reduce clustering

Double Hashing:
- hash((key) = key % 8
- hash2(key) -7 -fiey%7

- Pros:
- Reduces clustering
- Can still cause secondary clustering



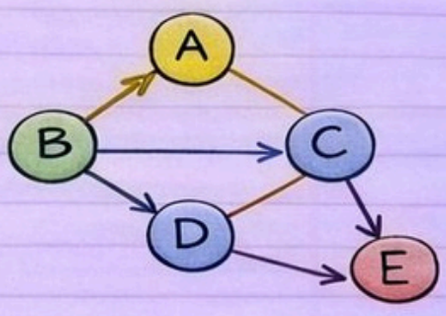
Chapter 13: Graphs

@curious_programmer

What is a Graph?

A Graph is a non-linear data structure that consists of:

- Vertices (Nodes)
- Edges (Connections)



- Graphs are used to represent real-world relationships like:
  - Social networks
  - Maps & routes
  - Computer networks

Graph Representation

Adjacency Matrix

An Adjacency Matrix is a 2D array where:

- Rows and columns represent vertices..
- Value 1 means edge exists
- Value 0 means no edge

2D array

	A	B	C	D	D
A	0	1	0	1	0
B	1	0	0	0	1
C	1	0	0	1	0
D	0	1	1	1	0

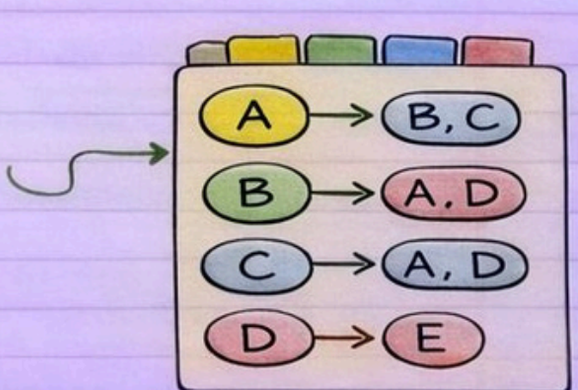
- Properties:
  - Uses more memory
  - Fast edge lookup
  - Best for dense graphs

Graph Traversal

Adjacency List

An Adjacency List stores:

- Each vertex with a list of its connected vertices



- Properties:
  - Memory efficient
  - Slower edge lookup
  - Best for sparse graphs

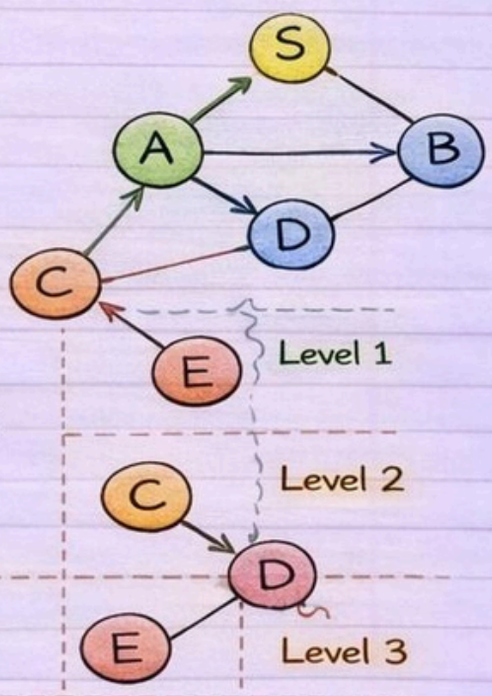
Breadth First Search (BFS)

@curious_programmer

What is BFS?

Breadth First Search (BFS) is a graph traversal algorithm that:

- Visits nodes level by level
- Uses Queue
- Best for finding shortest path in unweighted graphs

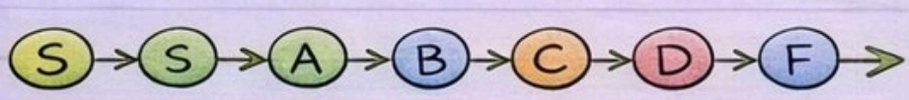


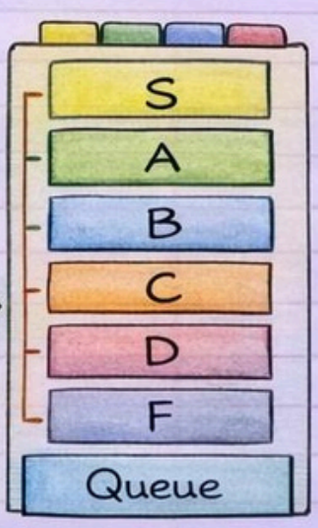
Steps of BFS

- **Step 1** – Start from the source node
- **Step 2** – Visit all adjacent nodes (one level at a time)
- **Step 3** – Move to the next level

Steps of BFS

- **Step 1**: Som the source node
- Visit all adjacent nodes (one level at a time)
- Move to the next level





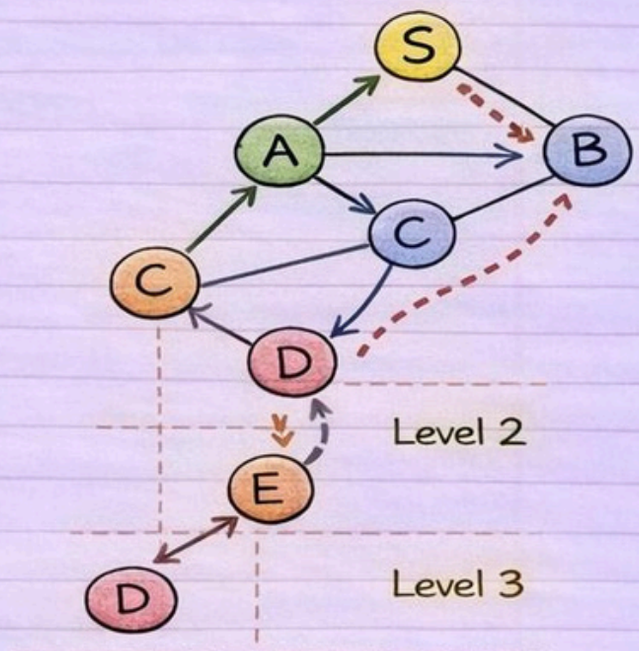
BFS Order: S → A → B → C → D → E → F

Depth First Search (DFS)

@curious_programmer

What is DFS?

- Goes deep before wide
- Uses Stack / Recursion
- Good for cycle detection

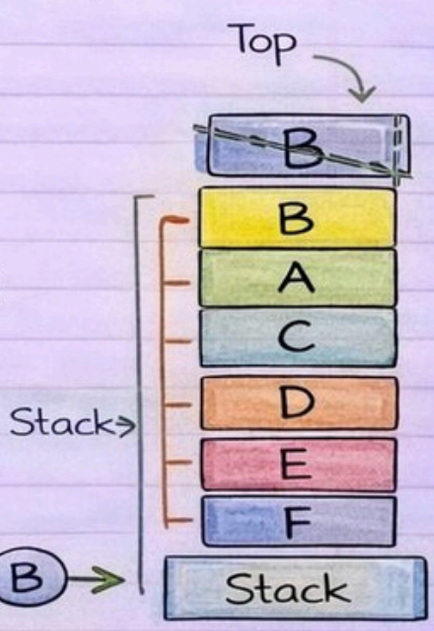


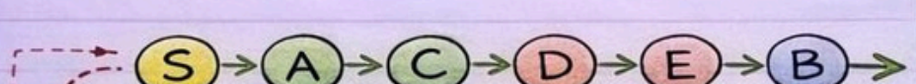
Steps of DFS

- Step 1 – Start from a node
- Step 2 – Visit an adjacent node fully
- Step 3 – Backtrack if no nodes left

Steps of DFS

- Step 1 – Start from a node
- Step 2 – Visit an adjacent node fully
- Step 3 – Backtrack if no nodes left





DFS Order: S → A → C → D → E → B

Connected Components

@curious_programmer

What are Connected Components?

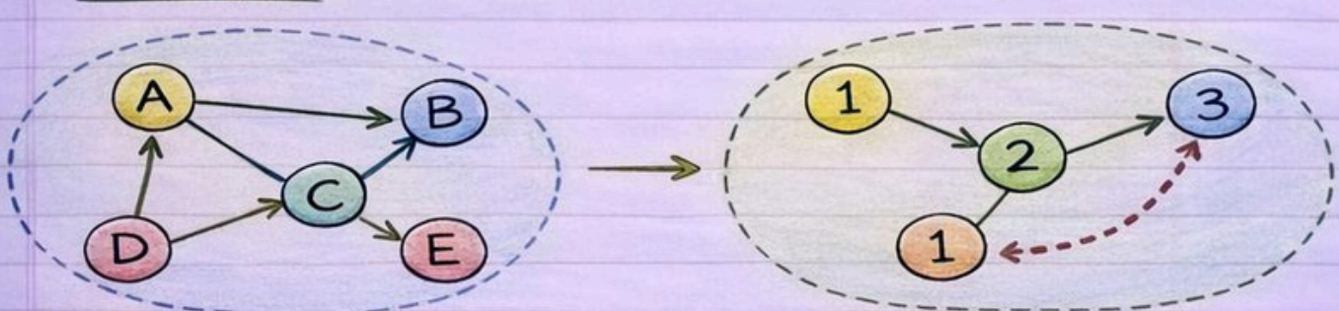
A Connected Component is a group of vertices where:

- Every vertex is reachable from any other vertex in the same group

Used to:

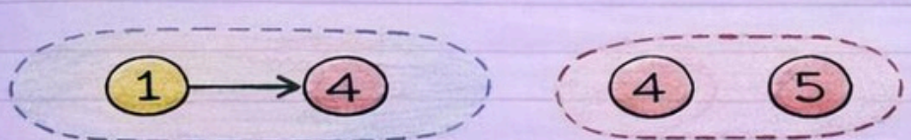
- Count isolated groups
- Check graph connectivity

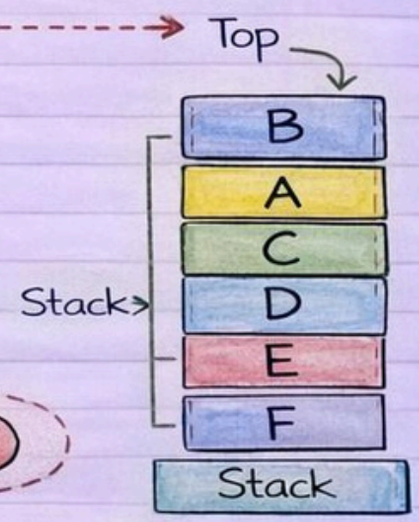
Example



3 Connected Components

- (1) A, B, C, D, E
- (2) 1, 2, 3
- (3) 4





3 Connected Components

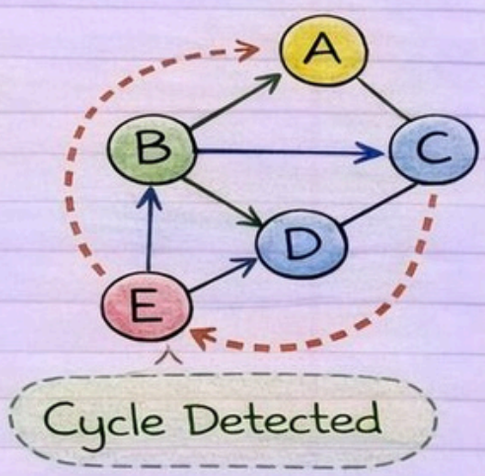
Cycle Detection

@curious_programmer

What are Connected Components?

A Cycle exists if:

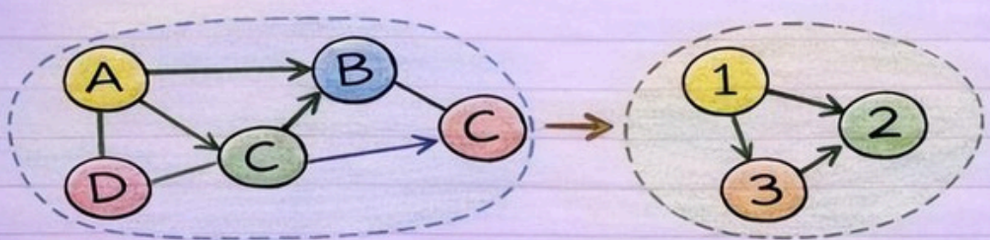
- You can start from a node and come back to it by following the edges of the graph



Used to:

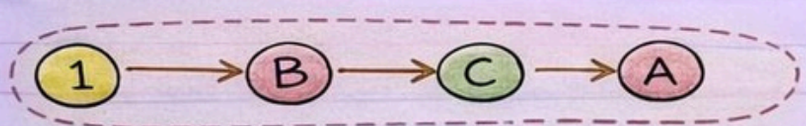
- Count isolated groups
- Check graph connectivity

Example:

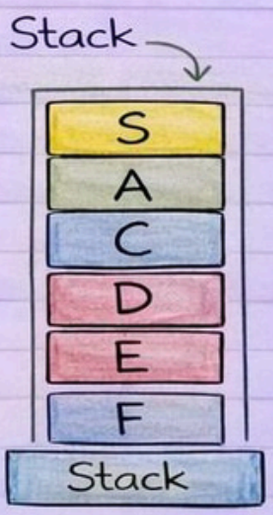


3 Connected Components

1. A, B, C, D, E
2. 1, 2, 3



Cycle: A → B → C → D → D → E → B



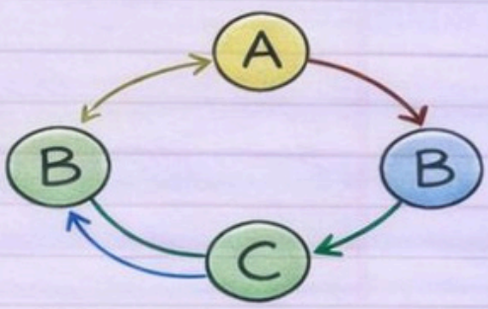
Cycle Detection

@curious_programmer

What is Cycle Detection?

A cycle exists if:

You can start from a node and come back to it



Cycle: A → B → C → D → A

A graph is cyclic if:

It has at least one cycle

Used in:

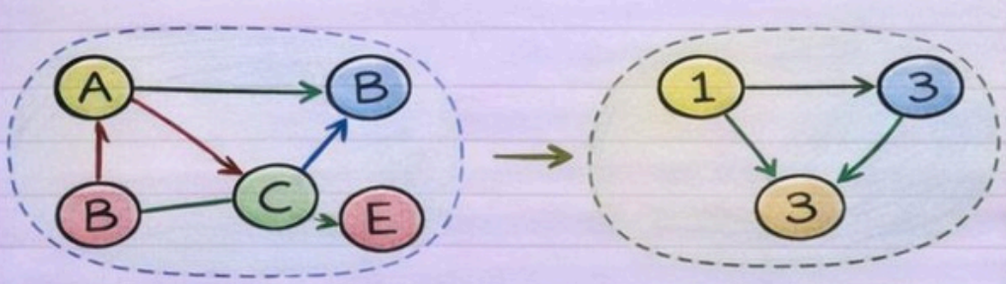
- Deadlock detection
- Dependency resolution

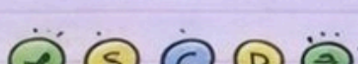
Topological Sorting

What is Topological Sorting?

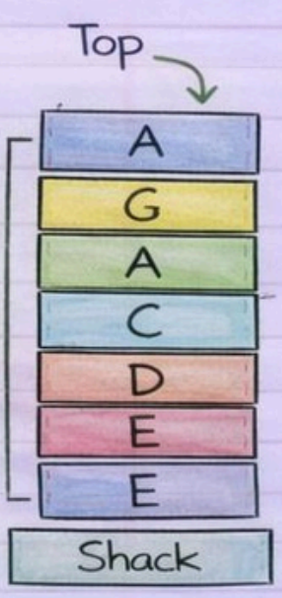
- Linear ordering of vertices
- Used only in Directed Acyclic Graph (DAG)

Conditions:





: A → C → D → E

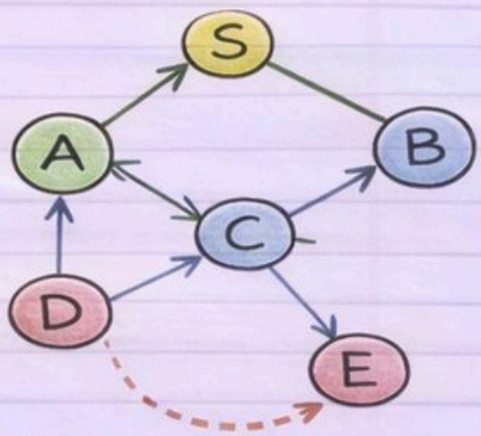


Topological Sorting

@curious_programmer

What is Topological Sorting?

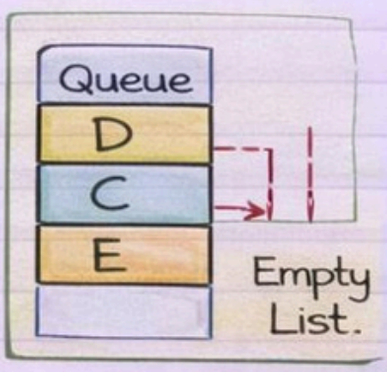
- Linearly orders vertices in a Directed Acyclic Graph (DAG)
- Ensures order based on dependencies
- Used in scheduling tasks, ordering courses, etc.



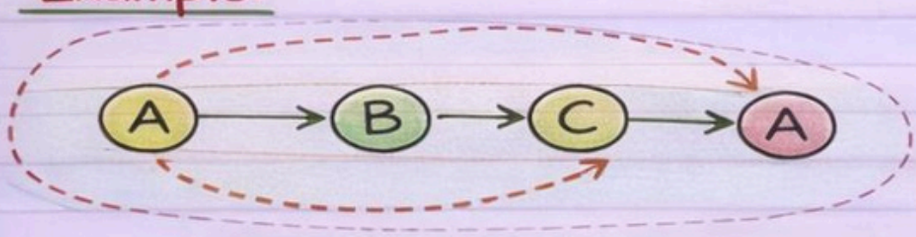
Tasks with dependencies

Steps of Topological Sort

1. Find nodes with no dependencies
2. Remove node and add to order
3. Update other nodes dependencies



Example:



Topological Order: S → A → B → C → D → E.

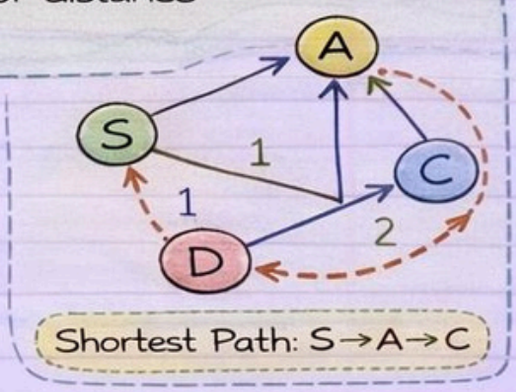
Order tasks such that dependencies are saitisfied.

Shortest Path (Basics)

@curious_programmer

What is Shortest Path?

Shortest path is the minimum cost or distance between two vertices.

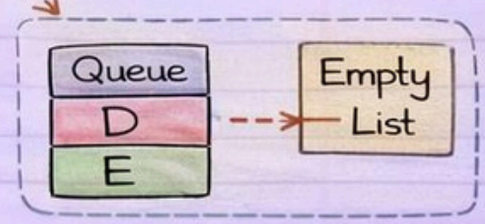


Common Algorithms:

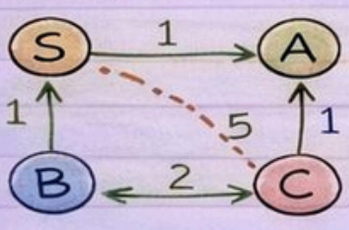
1. **BFS** (unweighted graph)
   - Uniform Cost: Treat all edges equally, find the quickest path.
2. **Dijkstra** (weighted graph–basics)
   - Non-uniform Cost: Accounts for edge weights, finds the least costly path.

Steps of Topological Sort

1. Find nodes with no dependencies
2. Remove node and add to order
3. Update other nodes' dependencies



Example:



Find shortest path from S to D:

Shortest Path: S→A→D→6

Uniform Cost: Add all weights equally: 1+5=6

Shortest Path: S→A→D→D→6