DSA

Data Structures (DS): These are ways of organizing and storing data to perform operations efficiently. Examples include arrays, linked lists, stacks, queues, trees, graphs, and hash tables.

Algorithms (A): These are step-by-step procedures or formulas for solving a particular problem. Examples include sorting algorithms (Bubble Sort, Quick Sort) and searching algorithms (Binary Search, DFS, BFS).

What is Data Structure?

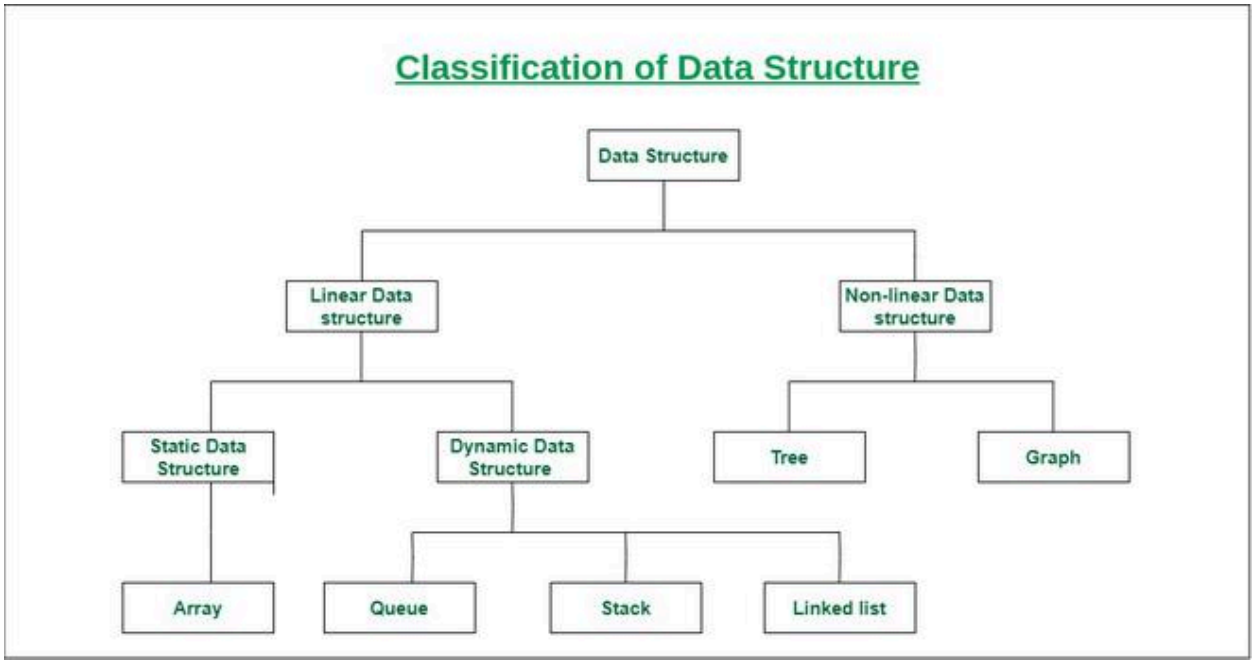
A **data structure** is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge about data structures.

Why Use DSA?

DSA is crucial for optimizing software performance, reducing execution time, and improving memory efficiency. Here are key reasons for its usage:

- **Efficient Problem Solving:** It helps in solving complex problems efficiently.
- **Memory Optimization:** Proper data structures help in utilizing memory effectively.
- **Code Scalability:** Ensures that the system can handle increased data efficiently.
- **Faster Execution:** Optimized algorithms improve program execution time.



1. Linear Data Structure

A **linear data structure** is a type of data structure where elements are arranged in a **sequential order** and each element is connected to its previous and next element. These structures are easy to implement and allow for efficient traversal.

- **Examples:**
  - **Array** – A fixed-size sequential collection of elements of the same type.
  - **Stack** – Follows **LIFO (Last In, First Out)** principle.
  - **Queue** – Follows **FIFO (First In, First Out)** principle.
  - **Linked List** – A dynamic list where each element (node) points to the next.
- **Advantages:**
  - Simple and easy to implement.
  - Memory is allocated sequentially.
  - Suitable for problems requiring ordered access.
- **Disadvantages:**
  - May require shifting elements (e.g., insertion and deletion in arrays).
  - Can have memory wastage if a static structure (like arrays) is used.

A). Static Data Structure

A **static data structure** has a **fixed memory size**, meaning memory is allocated during **compile-time** and cannot be changed later. It allows **faster access** but can lead to memory wastage if not fully utilized.

- **Example:**
  - **Array** – Has a fixed size allocated at the beginning.
- **Advantages:**
  - Quick access to elements using indexing.
  - Predictable memory usage.
- **Disadvantages:**
  - Cannot resize dynamically.
  - Can lead to inefficient memory usage.

B). Dynamic Data Structure

A **dynamic data structure** does not have a fixed size; it can **grow or shrink** during program execution. Memory is allocated dynamically, making it **memory-efficient**.

- **Examples:**
  - **Linked List** – Grows dynamically by creating new nodes when needed.
  - **Queue** – Implemented dynamically using linked lists.
  - **Stack** – Can grow dynamically in memory.
- **Advantages:**
  - Efficient memory usage since it only allocates memory when needed.
  - Allows flexibility in size adjustment.
- **Disadvantages:**
  - Access time can be slower compared to static structures.
  - Extra memory is required for storing pointers in linked structures.

2) Non-Linear Data Structure

A **non-linear data structure** is a type where elements **do not follow a sequential order**. The elements are connected in a **hierarchical manner**, making them suitable for complex relationships.

- **Examples:**
  - **Trees** – Used in hierarchical data representation (e.g., file systems, databases).
  - **Graphs** – Used in networking, social media, and AI pathfinding algorithms.
- **Advantages:**
  - Suitable for representing complex relationships between data.
  - More efficient than linear structures in many scenarios (e.g., searching in trees).
- **Disadvantages:**
  - More complex to implement compared to linear structures.
  - Traversal can be tricky and may require recursion or advanced algorithms.

Here are some commonly used data structures and algorithms:

1. Arrays

Example: Storing a list of student names.

Use case: Used in applications that require constant-time access to elements.

Static declaration of array

```
String[] daysOfWeek = {"Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday", "Saturday", "Sunday"};

// Can also be done as:


String daysOfWeek_declareFirst[];

daysOfWeek_declareFirst= new String[] {"Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday",
        "Sunday"};
```

Dynamic declaration of array

```
String[] daysOfWeek = new String[7];

daysOfWeek[0] = "Sunday";
daysOfWeek[1] = "Monday";
daysOfWeek[2] = "Tuesday";
daysOfWeek[3] = "Wednesday";
daysOfWeek[4] = "Thursday";
daysOfWeek[5] = "Friday";
daysOfWeek[6] = "Saturday";
```

2. Linked Lists

- **Example:** Implementing a browser's back and forward navigation.
- **Use case:** Used in dynamic memory allocation.

3. Stacks

- **Example:** Undo/Redo feature in text editors.
- **Use case:** Used in function call recursion.

4. Queues

- **Example:** Ticket booking systems.
- **Use case:** Used in CPU scheduling and print spooling.

5. Trees

- **Example:** File system structure.
- **Use case:** Used in search algorithms like BST (Binary Search Tree).

6. Graphs

- **Example:** Google Maps uses graphs for route navigation.
- **Use case:** Used in social media friend recommendations.

7. Sorting Algorithms

- **Example:** Online shopping websites sort products by price.
- **Use case:** Used in data analysis and reporting.

8. Searching Algorithms

- **Example:** Searching for a contact in your phone.
- **Use case:** Used in databases for fast retrieval.

Time and Space Complexity are crucial for analyzing the efficiency of algorithms.

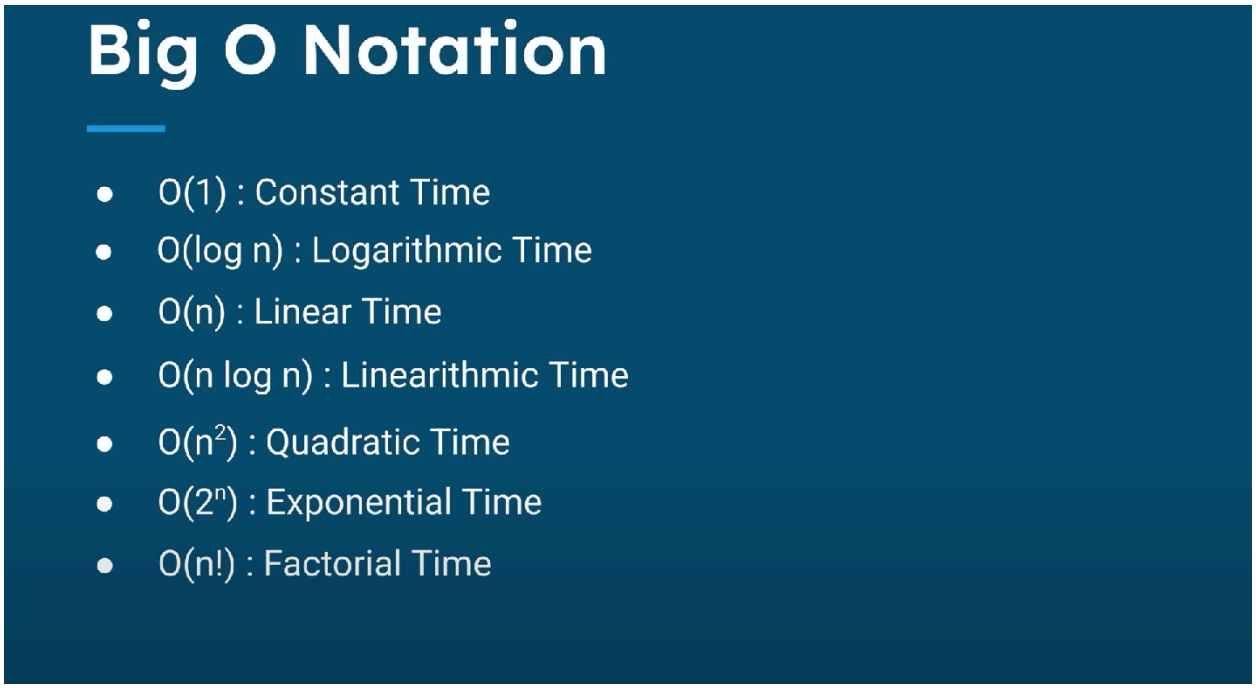
What is Time Complexity?

Time Complexity measures **how fast** an algorithm runs as the input size (n) grows. It helps us determine which algorithm is **more efficient** when handling large inputs.

Big-O Notation

We use **Big-O notation** to describe the **upper bound** of an algorithm's time complexity.

Big-O Notation	Name	Example
O(1)	Constant Time	Accessing an array element
O(log n)	Logarithmic Time	Binary Search
O(n)	Linear Time	Looping through an array
O(n log n)	Linearithmic Time	Merge Sort, Quick Sort
$O(n^2)$	Quadratic Time	Nested loops (Bubble Sort)
$O(2^n)$	Exponential Time	Fibonacci (Recursive)
O(n!)	Factorial Time	Traveling Salesman Problem



Types of Algorithms Based on Big-O Complexity

Big-O Notation	Algorithm Type	Common Examples
O(1) - Constant Time	Direct Access / Hashing	HashMap lookup, Array indexing
O(log n) - Logarithmic Time	Divide & Conquer	Binary Search, Balanced Trees (Red-Black Tree, AVL Tree)
O(n) - Linear Time	Iterative / Scanning	Single loop traversal, Finding min/max in an array
O(n log n) - Linearithmic Time	Efficient Sorting & Divide & Conquer	Merge Sort, Quick Sort, Heap Sort
$O(n^2)$ - Quadratic Time	Brute-Force / Naïve	Bubble Sort, Selection Sort, Insertion Sort, Checking all pairs
$O(2^n)$ - Exponential Time	Backtracking / Exhaustive Search	Recursive Fibonacci, Subset Sum, Traveling Salesman (Brute-force)

1️⃣ O(1) – Constant Time Complexity

No matter how large n is, the function takes the same amount of time.

Used in: Direct access operations, Hashing

Example: Looking up an element in an array or HashMap

Algorithm Type: Direct Access

```
public class ConstantTimeExample {
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        System.out.println(arr[2]); // Always takes the same time
    }
}
```

✅ **Best Case:** No loops, direct access.

2️⃣ O(n) – Linear Time Complexity

Used in: Scanning or searching in an array

Example: Finding the maximum value in an array

Algorithm Type: Iterative / Sequential Search

Time increases **proportionally** with n.

```
public class LinearTimeExample {
   public static void main(String[] args) {
      int[] arr = {10, 20, 30, 40, 50};
      for (int num : arr) {  // Iterates through all elements
         System.out.print(num + " ");
      }
   }
}
```

✅ **Best for:** Searching an element in an unsorted array.

Searching

Linear Search Explanation (O(n))

Linear Search works by checking **each element one by one** from the start of the array.

How Time Complexity Varies in Linear Search

Case	Scenario	Time Complexity
Best Case	Target is the **first element**	**O(1) (Instantly found)**
Worst Case	Target is the **last element** or **not in the array**	**O(n) (Checks all elements)**
Average Case	Target is **somewhere in the middle**	**O(n/2) ≈ O(n)**

Example: Linear Search on [10, 20, 30, 40, 50], Target = 50

🔍 **Search process:**

- Check 10 ❌
- Check 20 ❌

- Check 30 ❌
- Check 40 ❌
- Check 50 ✅ **(Found at last index, worst case)**

Total comparisons: 5 (O(n))

Example: Target = 10 (First Element)

- Check 10 ✅ **(Found at first index, best case O(1))**

Linear Search is Inefficient When:

- The array is **large** (e.g., 1 million elements).
- The target is **at the end** or **not present**.

Binary Search Explanation (O(log n))

Binary Search **only works on sorted arrays** and repeatedly divides the search space in half.

How Time Complexity Varies in Binary Search

Case	Scenario	Time Complexity
Best Case	Target is the **middle element**	**O(1) (Instantly found)**
Worst Case	Target is at the **end** of search space or not present	**O(log n) (Keeps dividing)**
Average Case	Target is **somewhere in the array**	**O(log n)**

Example: Binary Search on [10, 20, 30, 40, 50], Target = 50

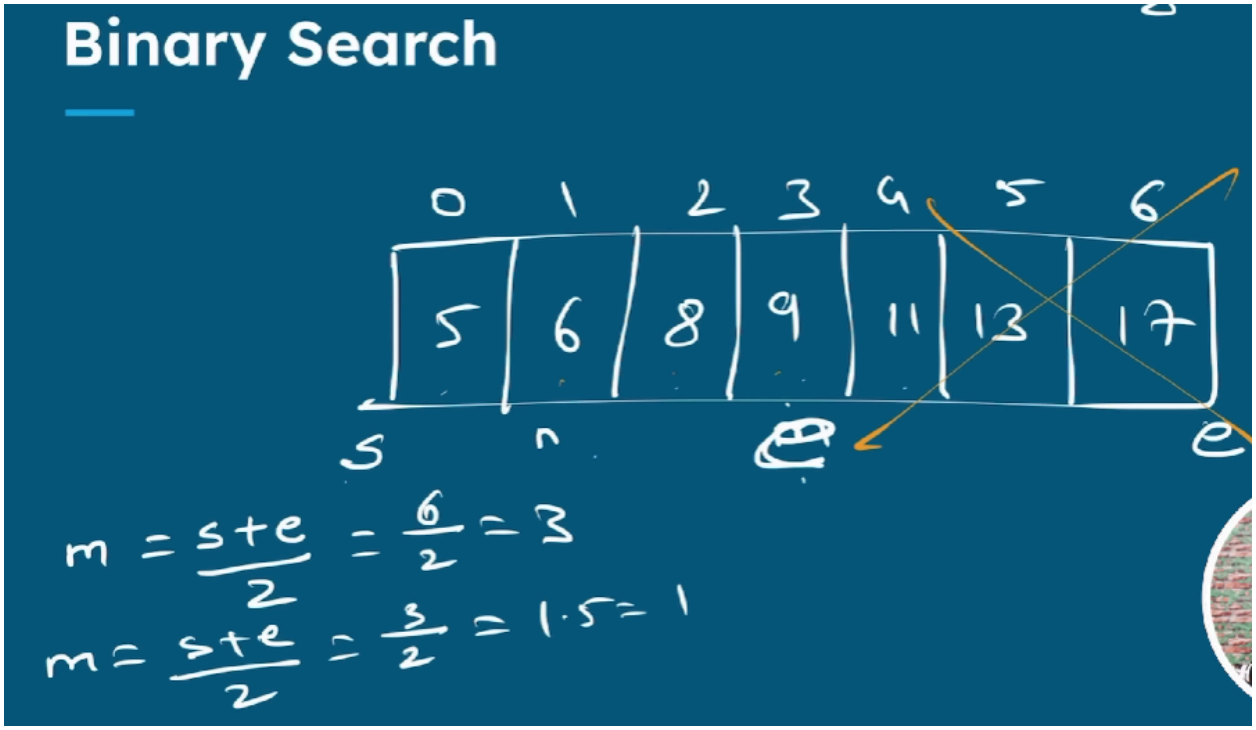
🔍 **Search process:**

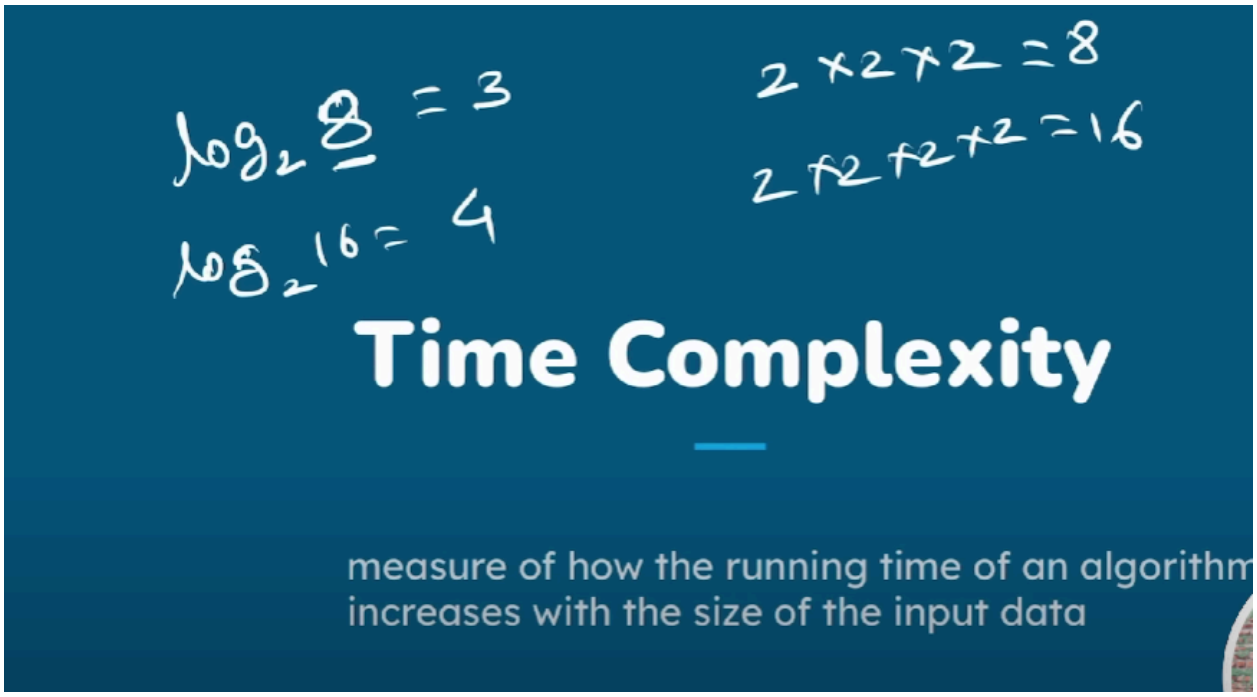
- **Middle = 30**, target is greater → search right half.
- **Middle = 40**, target is greater → search right half.
- **Middle = 50**, ✅ **(Found, 3 comparisons instead of 5!)**

Total comparisons: 3 (O(log n)) instead of 5 (O(n)) in Linear Search.

Example: Target = 10 (First Element)

- **Middle = 30**, target is smaller → search left half.
- **Middle = 10**, ✅ **(Found in 2 steps, O(log n))**





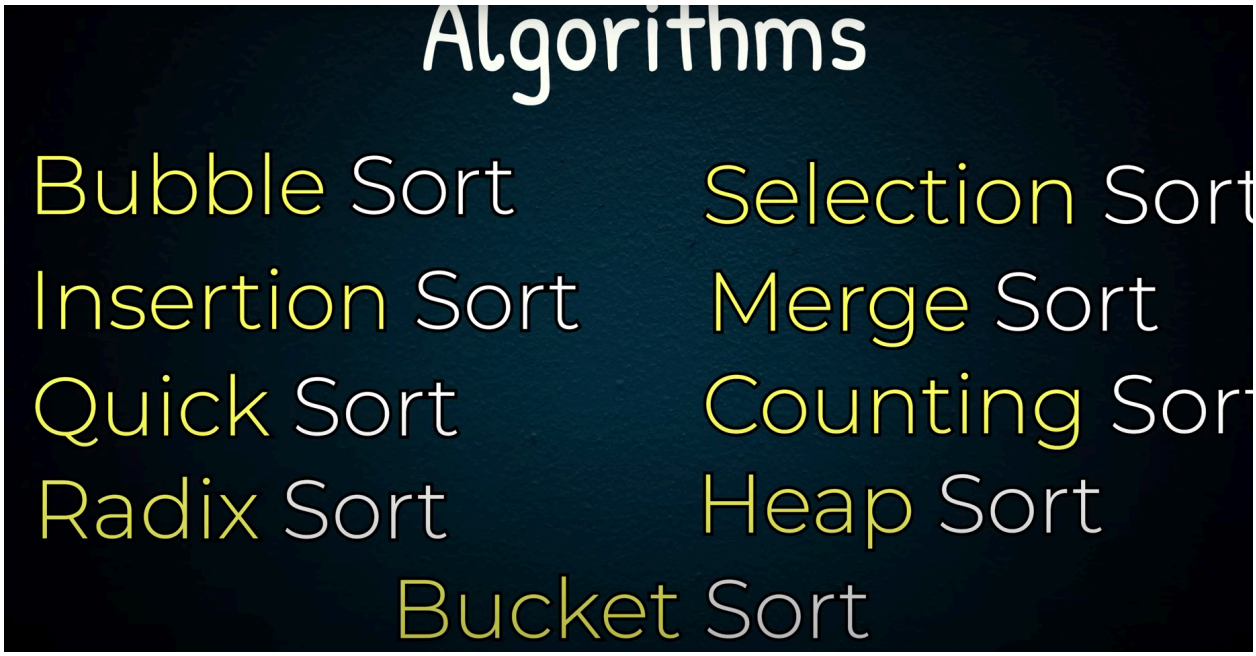
Ex:

```
package dsa;
public class Searching {
	public static void main(String[] args) {
```

```
		int num[] = { 5, 6, 7, 8, 9, 10 };
		int target = 17;
		// using linear search , it accept the where to search and target
		// int result = linearSearch(num, target);
		// int result = binarySearch(num, target);
		int result = binarySearch(num, target, 0, num.length - 1);
		if (result != -1)
			System.out.println("Element found at Index: " + result);
		else
			System.out.println("Element found not Found:");
	}
	private static int linearSearch(int[] num, int target) {
		for (int i = 0; i < num.length; i++) {
			if (num[i] == target) {
				return i;
			}
		}
		return -1;
	}
	// we have to pass only sorted array
//	private static int binarySearch(int[] num, int target) {
//		int left = 0;
//		int right = num.length - 1;
//		while (left <= right) {
//			int mid = (left + right) / 2;
//			if (num[mid] == target) {
//				return mid;
//			} else if (num[mid] < target) {
//				left = mid + 1;
//			} else
//				right = mid - 1;
//		}
//		return -1;
//	}
	private static int binarySearch(int[] num, int target, int left, int right) {
		if (left <= right) {
			int mid = (left + right) / 2; // Find middle index
			if (num[mid] == target) {
				return mid; // Target found
			} else if (num[mid] < target) {
```

```
                    return binarySearch(num, target, mid + 1, right); // Search right
half
                } else {
                    return binarySearch(num, target, left, mid - 1); // Search left half
                }
            }
            return -1;
        }
}
```

Sorting



1. Bubble Sort

Repeatedly swaps adjacent elements if they are in the wrong order.

Time Complexity: **O(n²)**

Algorithm Type: Brute-Force

Time Complexity: O(n2) ,as there are two nested loops:

- One loop to select an element of Array one by one = O(n)
- Another loop to compare that element with every other Array element = O(n)
- Therefore overall complexity = O(n) * O(n) = O(n*n) = O(n2)

Space Complexity: **O(1)**

Stable: **Yes**

Example:
Input: [5, 3, 8, 1, 2]
Output: [1, 2, 3, 5, 8]

Bubble Sort Algorithm

1. Compare adjacent elements: If the first is greater than the second, swap them.
2. Repeat this for the entire list, moving the largest element to the last position.
3. Repeat the process for the remaining elements until the list is sorted.

Example:

Unsorted Array:

[6, 5, 2, 8, 9, 4]

Pass 1 (i = 0):

- Compare 6 and 5 → 6 > 5, so **swap** → [5, 6, 2, 8, 9, 4]
- Compare 6 and 2 → 6 > 2, so **swap** → [5, 2, 6, 8, 9, 4]
- Compare 6 and 8 → No swap
- Compare 8 and 9 → No swap
- Compare 9 and 4 → 9 > 4, so **swap** → [5, 2, 6, 8, 4, 9]
- **Largest element (9) is placed at the last position** ✅

Pass 2 (i = 1):

- Compare 5 and 2 → 5 > 2, so **swap** → [2, 5, 6, 8, 4, 9]
- Compare 5 and 6 → No swap
- Compare 6 and 8 → No swap
- Compare 8 and 4 → 8 > 4, so **swap** → [2, 5, 6, 4, 8, 9]
- **Second-largest element (8) is placed correctly** ✅

Pass 3 (i = 2):

- Compare 2 and 5 → No swap
- Compare 5 and 6 → No swap
- Compare 6 and 4 → 6 > 4, so **swap** → [2, 5, 4, 6, 8, 9]
- **Third-largest element (6) is placed correctly** ✅

Pass 4 (i = 3):

- Compare 2 and 5 → No swap
- Compare 5 and 4 → 5 > 4, so **swap** → [2, 4, 5, 6, 8, 9]
- **Fourth-largest element (5) is placed correctly** ✅

Pass 5 (i = 4):

- Compare 2 and 4 → No swap
- **Array is sorted!** ✅ [2, 4, 5, 6, 8, 9]

```
package dsa.sorting;
public class Sorting {
	public static void main(String[] args) {
		int nums[] = { 6, 5, 2, 8, 9, 4 };
		int temp = 0;
		System.out.println("Before sorting");
		for (int num : nums) {
			System.out.print(num + " ");
		}
		for (int i = 0; i < nums.length; i++) {
			for (int j = 0; j < nums.length - i - 1; j++) {
				if (nums[j] > nums[j + 1]) {
					temp = nums[j];
					nums[j] = nums[j + 1];
					nums[j + 1] = temp;
				}
			}
			System.out.println(" ");
			for (int num : nums) {
				System.out.print(num + " ");
			}
		}
		System.out.println(" ");
		System.out.println("After sorting");
		for (int num : nums) {
			System.out.print(num + " ");
		}
```

```
        }
}
```

Before sorting
6 5 2 8 9 4
5 2 6 8 4 9
2 5 6 4 8 9
2 5 4 6 8 9
2 4 5 6 8 9
2 4 5 6 8 9
2 4 5 6 8 9
After sorting
2 4 5 6 8 9

for (int j = 0; j < nums.length - i - 1; j++)
In the **Bubble Sort algorithm**, with each pass, the largest element moves to its correct position at the end.
Thus, after every iteration of the outer loop, we don't need to check the last i elements because they are already sorted.

Drawbacks of Bubble Sort

Bubble Sort is simple but inefficient for large datasets due to its **high time complexity**.

1. **Time Complexity is Poor:**
    - **Worst Case ($O(n^2)$)**: When the array is sorted in reverse order, Bubble Sort has to go through every pair of elements multiple times.
    - **Best Case ($O(n)$)**: If the array is already sorted, only one pass is needed (with an optimized version using a swap flag).
    - **Average Case ($O(n^2)$)**: Even for random data, it takes quadratic time, making it inefficient for large datasets.
2. **Too Many Swaps:**
    - Bubble Sort swaps elements in every iteration, even if they are already in the correct place. This increases execution time.

2. Selection Sort

- Selects the smallest element and swaps it with the first element, then the second smallest with the second element, and so on.

Time Complexity: O(n2) ,as there are two nested loops:

- One loop to select an element of Array one by one = O(n)
- Another loop to compare that element with every other Array element = O(n)
- Therefore overall complexity = O(n) * O(n) = O(n*n) = O(n2)

Space Complexity: **O(1)**

Key Difference Between Bubble Sort and Selection Sort

- **Bubble Sort:** Swaps adjacent elements multiple times in each iteration.
- **Selection Sort: Finds the smallest (or largest) element in each pass and swaps only once per iteration.**

How Selection Sort Works?

1. Find the **smallest/largest** element in the array.
2. Swap it with the **first** element if it is small or swap it with the **last** if it is largest.
3. Move to the next position and repeat until the entire array is sorted.

Step-by-Step Explanation

Unsorted Array:

[6, 5, 2, 8, 9, 4]

Pass 1 (i = 0):

- **Initially, we assume the first element is the smallest** → nums[0] = 6.
- Compare 6 with 5 → 5 is smaller, update minIndex = 1.
- Compare 5 with 2 → 2 is smaller, update minIndex = 2.
- Compare 2 with 8, 9, 4 → No smaller element found.
- **Swap 6 with 2**.
- **Array after Pass 1:** [2, 5, 6, 8, 9, 4]

Pass 2 (i = 1):

- **Assume nums[1] = 5 is the smallest**.
- Compare 5 with 6, 8, 9, 4.
- 4 is smaller, update minIndex = 5.
- **Swap 5 with 4**.
- **Array after Pass 2:** [2, 4, 6, 8, 9, 5]

Pass 3 (i = 2):

- **Assume nums[2] = 6 is the smallest**.
- Compare 6 with 8, 9, 5.
- 5 is smaller, update minIndex = 5.
- **Swap 6 with 5**.
- **Array after Pass 3:** [2, 4, 5, 8, 9, 6]

Pass 4 (i = 3):

- **Assume nums[3] = 8 is the smallest**.
- Compare 8 with 9, 6.
- 6 is smaller, update minIndex = 5.
- **Swap 8 with 6**.
- **Array after Pass 4:** [2, 4, 5, 6, 9, 8]

Pass 5 (i = 4):

- **Assume nums[4] = 9 is the smallest**.
- Compare 9 with 8.
- 8 is smaller, update minIndex = 5.
- **Swap 9 with 8**.
- **Array after Pass 5:** `[2, 4, 5, 6, 8, 9] ✅ Sorted!**

```
package dsa.sorting;
public class SelectionSorting {
	public static void main(String[] args) {
```

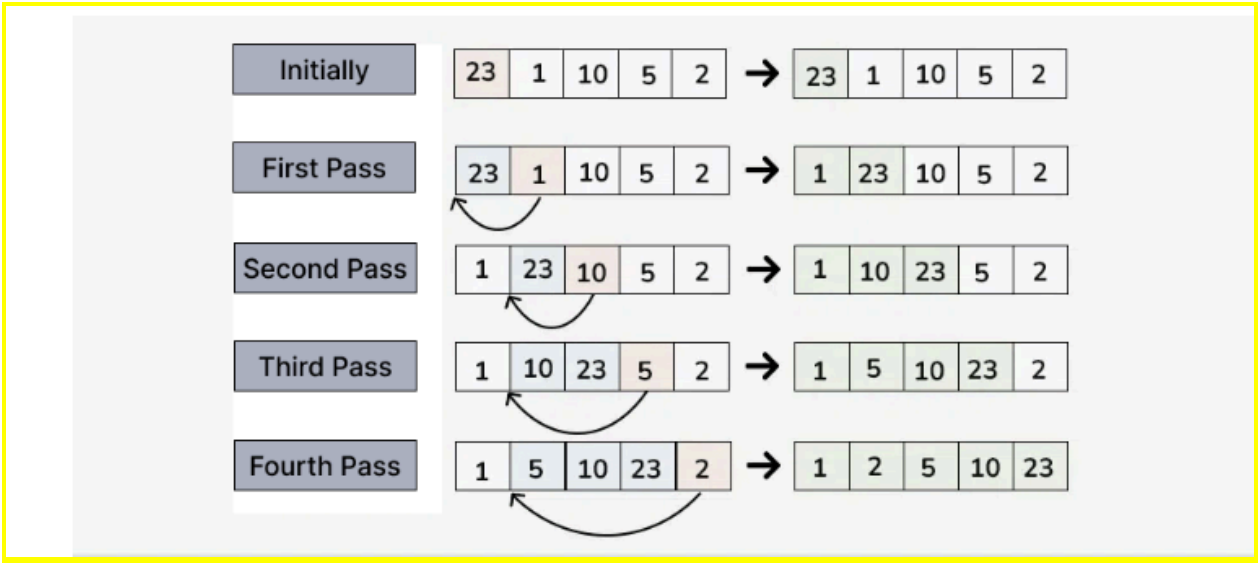
```
		int nums[] = { 6, 5, 2, 8, 9, 4 };
		int temp = 0;
		int minIndex = -1;
		int size = nums.length;
		System.out.println("Before sorting");
		for (int num : nums) {
			System.out.print(num + " ");
		}
		for (int i = 0; i < size; i++) {
			minIndex = i;
			for (int j = i + 1; j < size; j++) {
				if (nums[minIndex] > nums[j])
					minIndex = j;
			}
			temp = nums[minIndex];
			nums[minIndex] = nums[i];
			nums[i] = temp;
			System.out.println(" ");
			for (int num : nums) {
				System.out.print(num + " ");
			}
		}
		System.out.println(" ");
		System.out.println("After sorting ");
		for (int num : nums) {
			System.out.print(num + " ");
		}
	}
}
```

```
Before sorting
6 5 2 8 9 4
2 5 6 8 9 4
2 4 6 8 9 5
2 4 5 8 9 6
2 4 5 6 9 8
2 4 5 6 8 9
2 4 5 6 8 9
After sorting
2 4 5 6 8 9
```

Insertion sort

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- We start with the second element of the array as the first element in the array is assumed to be sorted.
- Compare the second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the first two elements and put at its correct position
- Repeat until the entire array is sorted.



Step-by-Step Example

Consider sorting the array:
Unsorted Array: [7, 3, 5, 1, 9]

Iteration 1 (i = 1, key = 3)

- Compare 3 with 7 → Shift 7 to the right.
- Insert 3 in its correct position.
- **Array:** [3, 7, 5, 1, 9]

Iteration 2 (i = 2, key = 5)

- Compare 5 with 7 → Shift 7 to the right.
- Insert 5 in its correct position.
- **Array:** [3, 5, 7, 1, 9]

Iteration 3 (i = 3, key = 1)

- Compare 1 with 7, 5, 3 → Shift all three elements to the right.
- Insert 1 in its correct position.
- **Array:** [1, 3, 5, 7, 9]

Iteration 4 (i = 4, key = 9)

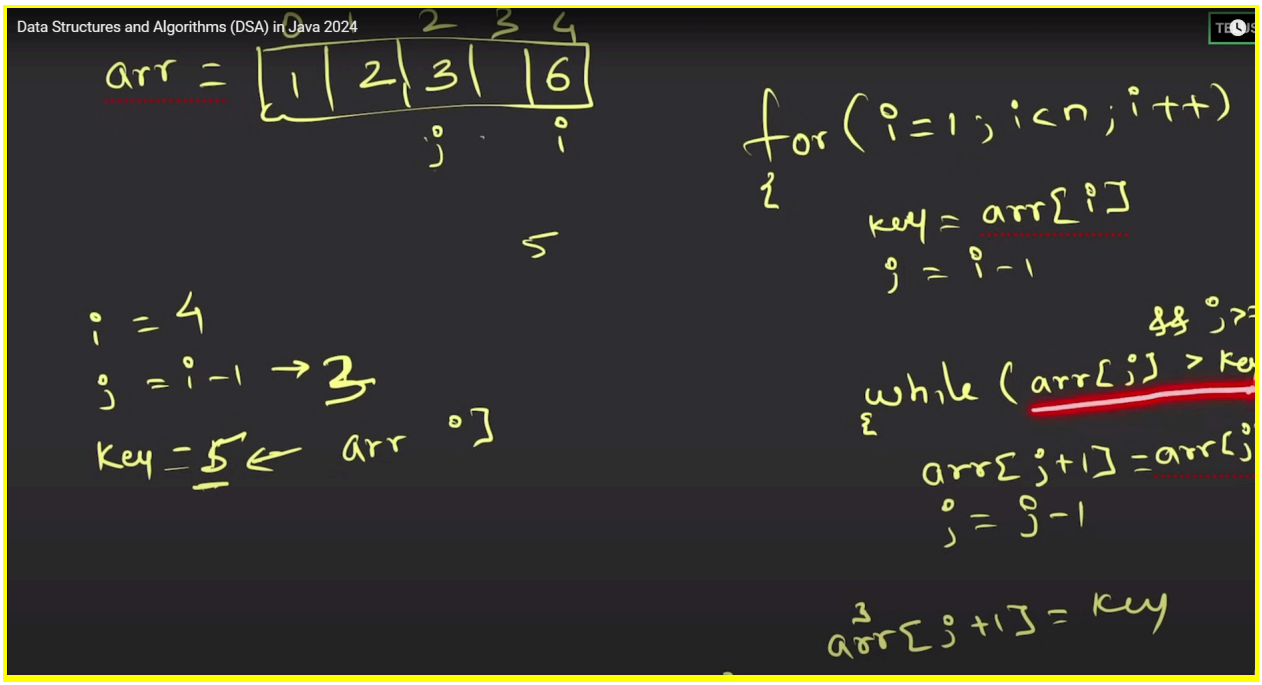
- Compare 9 with 7 → No shifting needed, as 9 is already in the correct position.
- **Final Sorted Array:** [1, 3, 5, 7, 9]

Key Features of Insertion Sort

✔️ **Stable Sorting Algorithm** (Preserves order of equal elements)
✔️ **Efficient for Small or Nearly Sorted Data**
❌ **Not Suitable for Large Data Sets** (Due to O(n2)O(n^2)O(n2) complexity)

Time Complexity Analysis

Case	Time Complexity	Explanation
Best Case	$O(n)$	Already sorted array (only comparisons, no shifts).
Worst Case	$O(n^2)$	Reverse-sorted array (maximum shifts required).
Average Case	$O(n^2)$	Randomly ordered elements.




```
package dsa.sorting;
public class InsertionSort {
	public static void main(String[] args) {
		int arr[] = { 5, 6, 2, 3, 1 };
		for (int i = 1; i < arr.length; i++) {
			int key = arr[i];
			int j = i - 1;
			// Shift elements to the right to make space for key
			while (j >= 0 && arr[j] > key) {
				arr[j + 1] = arr[j];
				j--;
			}
			// Insert key at the correct position
			arr[j + 1] = key;
		}
		for (int num : arr) {
			System.out.print(num + " ");
		}
	}
}
```

1 2 3 5 6

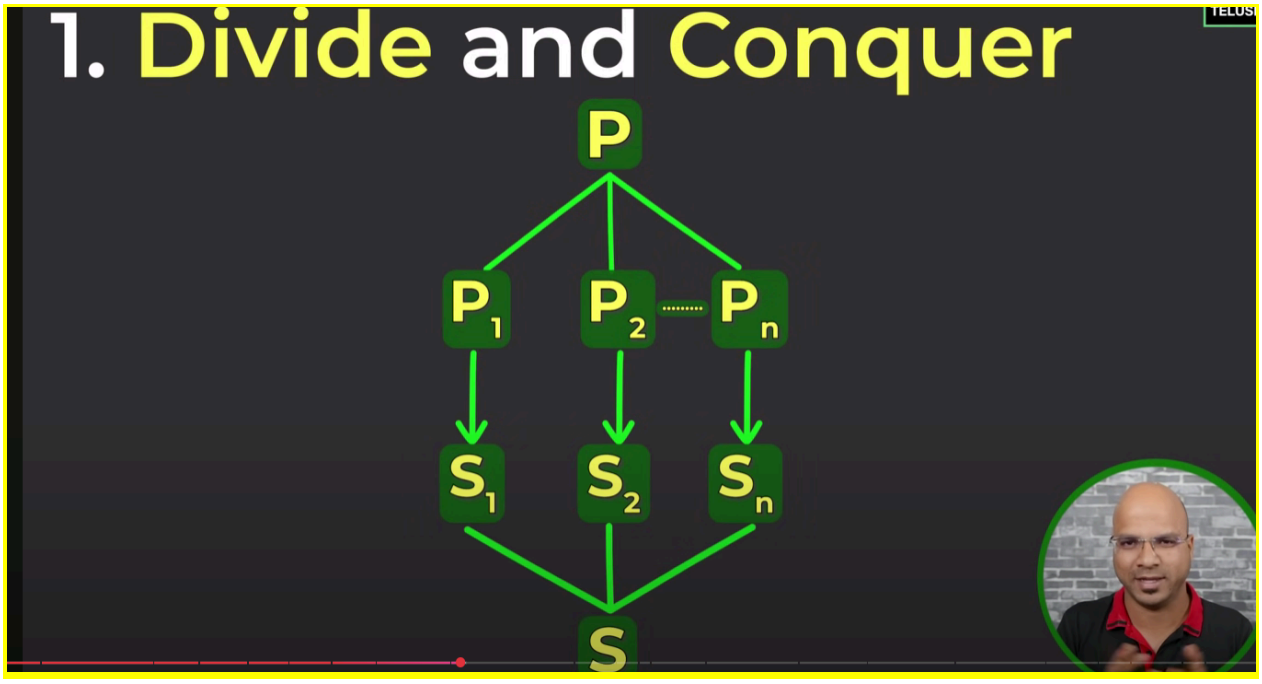
We have covered **Bubble Sort, Selection Sort, and Insertion Sort**, and all three have the following time complexities:

Sorting Algorithm	Best Case	Worst Case	Average Case
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$

Now, Let's Focus on More Efficient Sorting Algorithms!

Quick Sort

Quick Sort is a **divide and conquer** algorithm that sorts an array efficiently in **O(n log n)** time for most cases.



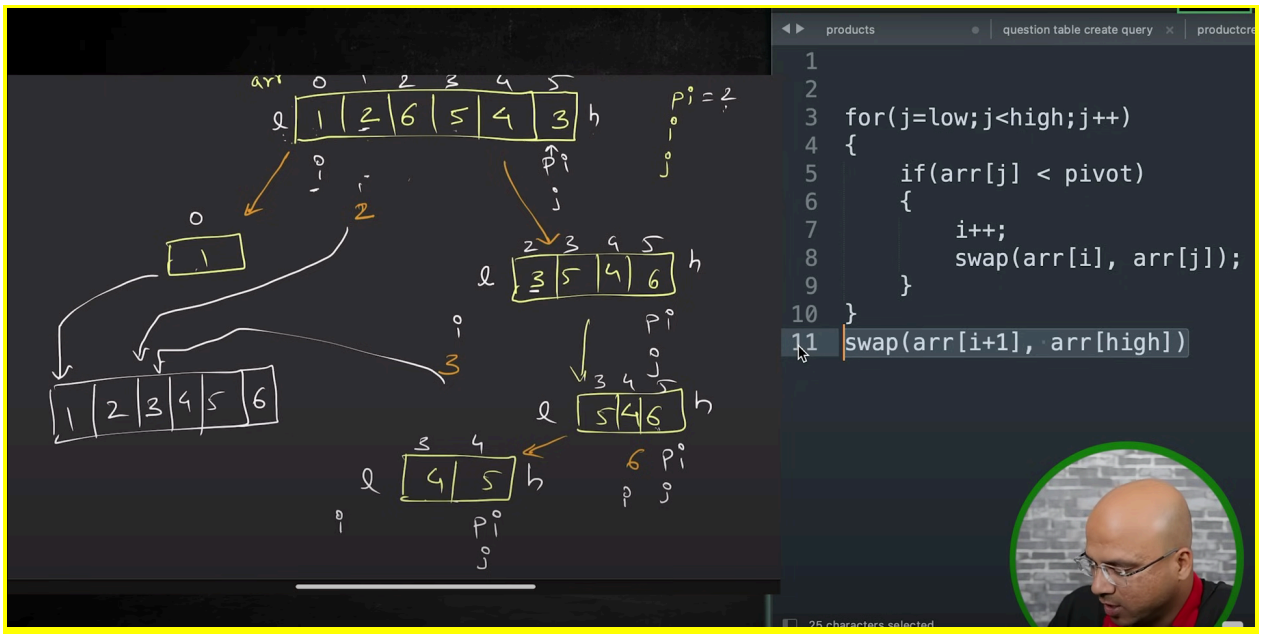
How does the QuickSort Algorithm work?
QuickSort works on the principle of **divide and conquer**, breaking down the problem into smaller sub-problems.

There are mainly three steps in the algorithm:

1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array:** Rearrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Complexity Analysis of Quick Sort
Time Complexity:

- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).




```
package dsa.sorting;
public class QuickSortEx {
	public static void main(String[] args) {
		int arr[] = { 5, 6, 2, 3, 1 };
		quickSort(arr, 0, arr.length - 1);// (arr,low,high)
		for (int val : arr) {
			System.out.print(val + " ");
		}
	}
	private static void quickSort(int[] arr, int low, int high) {
		if (low < high) {
			// pi is the partition return index of pivot
			int pi = partition(arr, low, high);
			// Recursion calls for smaller elements
			// and greater or equals elements
			quickSort(arr, low, pi - 1);
			quickSort(arr, pi + 1, high);
		}
	}
	private static int partition(int[] arr, int low, int high) {
		int pivot = arr[high]; // Choose the pivot
		// Index of smaller element and indicates
		// the right position of pivot found so far
		int i = low - 1;
```

```
                for (int j = low; j < high; j++) {
                        if (arr[j] < pivot) {
                                i++;
                                int temp = arr[i];
                                arr[i] = arr[j];
                                arr[j] = temp;
                        }
                }
                int temp = arr[i + 1];
                arr[i + 1] = arr[high];
                arr[high] = temp;
                return i + 1;
        }
}
```

1 2 3 5 6

Understanding the Given Java Code

The provided **Java program sorts the array {5, 6, 2, 3, 1} using Quick Sort**.

Step-by-Step Execution Flow

1️⃣ Main Method Execution

```
int arr[] = { 5, 6, 2, 3, 1 };
quickSort(arr, 0, arr.length - 1);
```

- The array {5, 6, 2, 3, 1} is passed to quickSort(arr, 0, 4).
- **Low = 0**, **High = 4** (entire array).

2️⃣ First Call to quickSort(arr, 0, 4)

- Call **partition(arr, 0, 4)** to find the pivot position.

3️⃣ First Partitioning (partition(arr, 0, 4))

```
int pivot = arr[4]; // pivot = 1 (last element)
int i = -1;
```

- **Pivot** is 1.
- **Initial index i = -1** (elements less than pivot will be stored before i).

Comparisons and Swaps

j (Index)	arr[j] Value	Condition arr[j] < pivot	Swap arr[i] ↔ arr[j]	Updated i
0	5	No	No Swap	-1
1	6	No	No Swap	-1
2	2	No	No Swap	-1
3	3	No	No Swap	-1

✅ **Final Swap**

- Swap arr[i + 1] = arr[0] with arr[4] (pivot).
- **Array after partition**: {1, 6, 2, 3, 5}.
- **Pivot Index** = 0, so partition() **returns 0**.

4️⃣ **Recursive Calls after First Partition**

```
quickSort(arr, 0, -1);  // Left part (empty, stops)
quickSort(arr, 1, 4);   // Right part {6, 2, 3, 5}
```

5️⃣ **Second Partitioning (partition(arr, 1, 4))**

```
int pivot = arr[4]; // pivot = 5
int i = 0;
```

- **Pivot** is 5.

Comparisons and Swaps

j (Index)	arr[j] Value	Condition arr[j] < pivot	Swap arr[i] ↔ arr[j]	Updated i
1	6	No	No Swap	0

2	2	Yes	Swap arr[1] ↔ arr[2]	1
3	3	Yes	Swap arr[2] ↔ arr[3]	2

✅ **Final Swap**

- Swap arr[i + 1] = arr[3] with arr[4] (pivot).
- **Array after partition**: {1, 2, 3, 5, 6}.
- **Pivot Index** = 3, so partition() **returns 3**.

6️⃣ **Recursive Calls after Second Partition**

```
quickSort(arr, 1, 2);  // Left part {2, 3}
quickSort(arr, 4, 4);  // Right part {6} (single element, stops)
```

7️⃣ **Third Partitioning (partition(arr, 1, 2))**

```
int pivot = arr[2]; // pivot = 3
int i = 0;
```

- **Pivot** is 3.

Comparisons and Swaps

j (Index)	arr[j] Value	Condition arr[j] < pivot	Swap arr[i] ↔ arr[j]	Updated i
1	2	Yes	Swap arr[1] ↔ arr[1]	1

✅ **Final Swap**

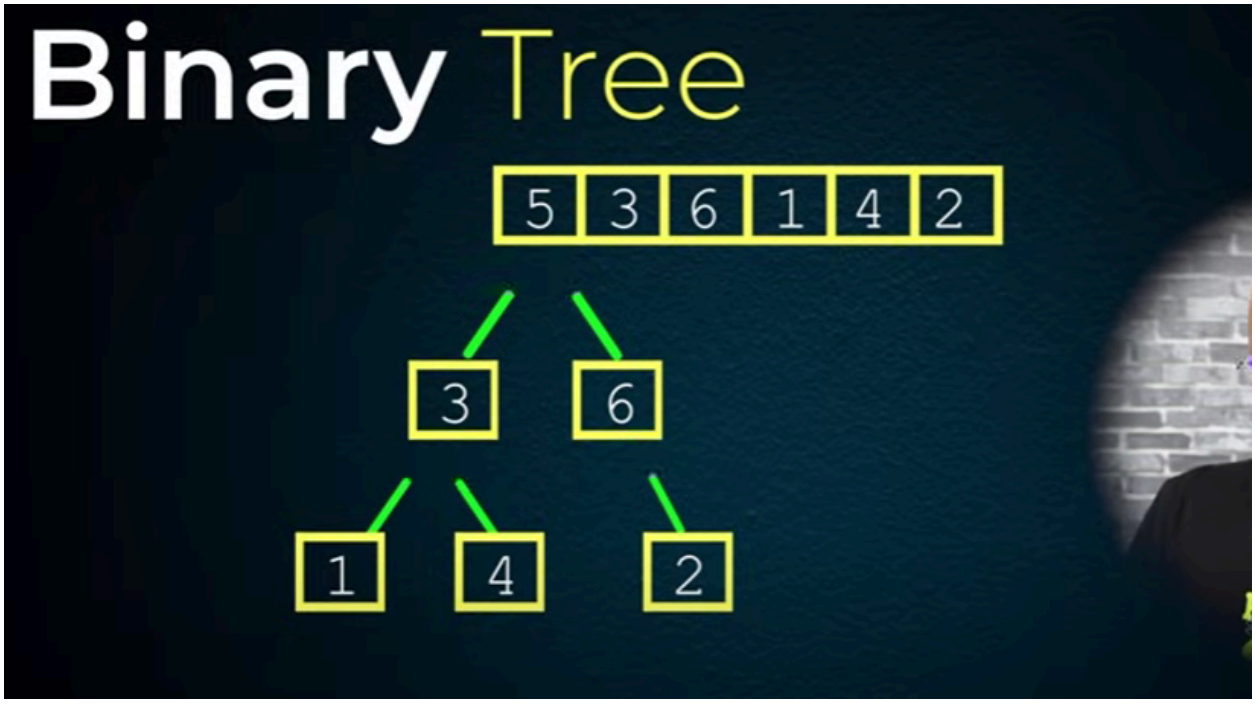
- Swap arr[i + 1] = arr[2] with arr[2] (pivot stays in place).
- **Array remains** {1, 2, 3, 5, 6}.
- **Pivot Index** = 2, so partition() **returns 2**.

8️⃣ **Final Recursive Calls**

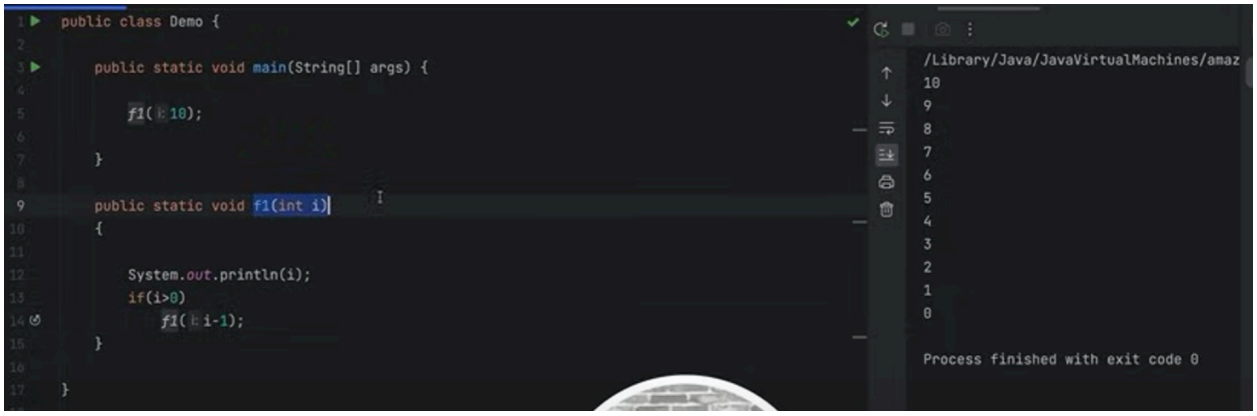
```
quickSort(arr, 1, 1);  // Stops (single element)
quickSort(arr, 3, 2);  // Stops (invalid range)
```

✅ **Final Sorted Array**

1 2 3 5 6



Recursion



What is Recursion?

Recursion is a programming concept where a function calls itself to solve a problem. It breaks down a problem into smaller subproblems of the same type until reaching a base case (a condition where the recursion stops).

Example: Factorial Calculation

-

The factorial of a number n (denoted as $n!$) is defined as:

n!=n×(n−1)×(n−2)×...×1n! = n \times (n-1) \times (n-2) \times ... \times 1n!=n×(n−1)×(n−2)×...×1

For example,

5!=5×4×3×2×1=1205! = 5 \times 4 \times 3 \times 2 \times 1 = 1205!=5×4×3×2×1=120

Recursive Implementation in Java

```
public class RecursionExample {
   // Recursive function to calculate factorial
   public static int factorial(int n) {
      if (n == 0 || n == 1) { // Base case
         return 1;
      }
      return n * factorial(n - 1); // Recursive call
   }

   public static void main(String[] args) {
      int number = 5;
      System.out.println("Factorial of " + number + " is: " + factorial(number));
   }
}
```

How It Works?

1. factorial(5) calls factorial(4).
2. factorial(4) calls factorial(3), and so on.
3. Once factorial(1) is reached (base case), it returns 1.
4. The recursive calls start returning results back up the call stack.

Merge sort

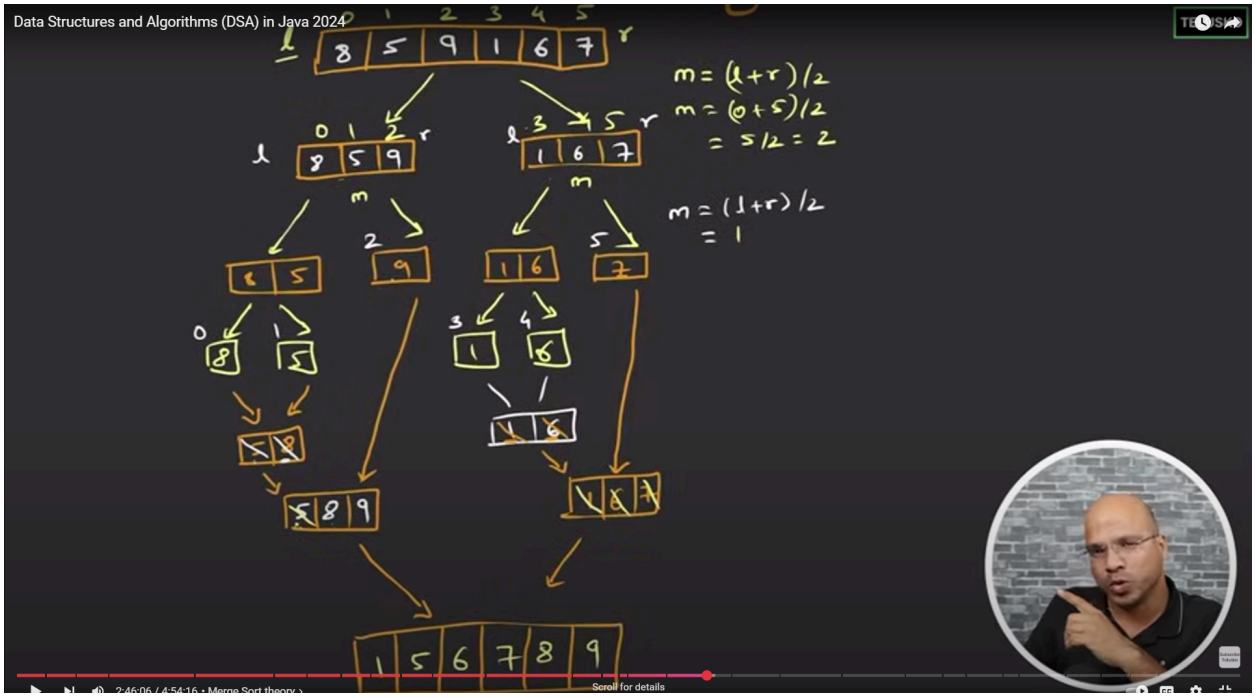
Merge sort is a sorting algorithm that follows the **divide-and-conquer** approach. It works by recursively dividing the input array into smaller subarrays and sorting those subarrays then merging them back together to obtain the sorted array.

Algorithm Type: Sorting (Divide & Conquer)

In simple terms, we can say that the process of **merge sort** is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

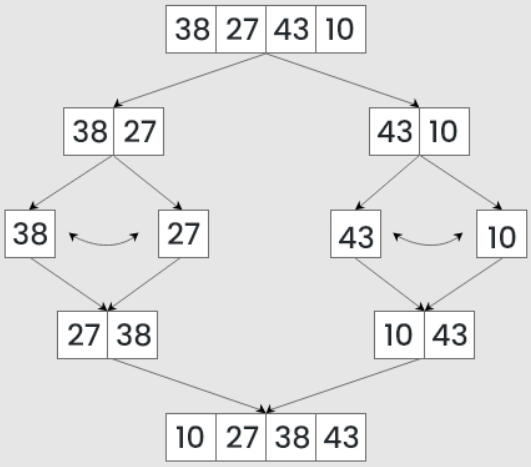
Time Complexity of Merge Sort

- Best Case: $O(n \log n)$
- Average Case: $O(n \log n)$
- Worst Case: $O(n \log n)$



Merge Sort

Algorithm



Basis for comparison	Quick Sort	Merge Sort
The partition of elements in the array	The splitting of a array of elements is in any ratio, not necessarily divided into half.	In the merge sort, the array is parted into just 2 halves (i.e. n/2).
Worst case complexity	O(n^2)	O(n log n)
Works well on	It works well on smaller array	It operates fine on any size of array
Speed of execution	It work faster than other sorting algorithms for small data set like Selection sort etc	It has a consistent speed on any size of data
Additional storage space requirement	Less(In-place)	More(not In-place)
Efficiency	Inefficient for larger arrays	More efficient
Sorting method	Internal	External
Stability	Not Stable	Stable
Preferred for	for Arrays	for Linked Lists
Locality of reference	good	poor
Major work	The major work is to partition the array into two sub-arrays before sorting them recursively.	Major work is to combine the two sub-arrays after sorting them recursively.
Division of array	Division of an array into sub-arrays may or may not be balanced as the array is partitioned around the pivot.	Division of an array into sub array is always balanced as it divides the array exactly at the middle.
Method	Quick sort is in- place sorting method.	Merge sort is not in – place sorting method.
Space	Quicksort does not require additional array space.	For merging of sorted sub-arrays, it needs a temporary array with the size equal to the number of input elements.

LinkedList

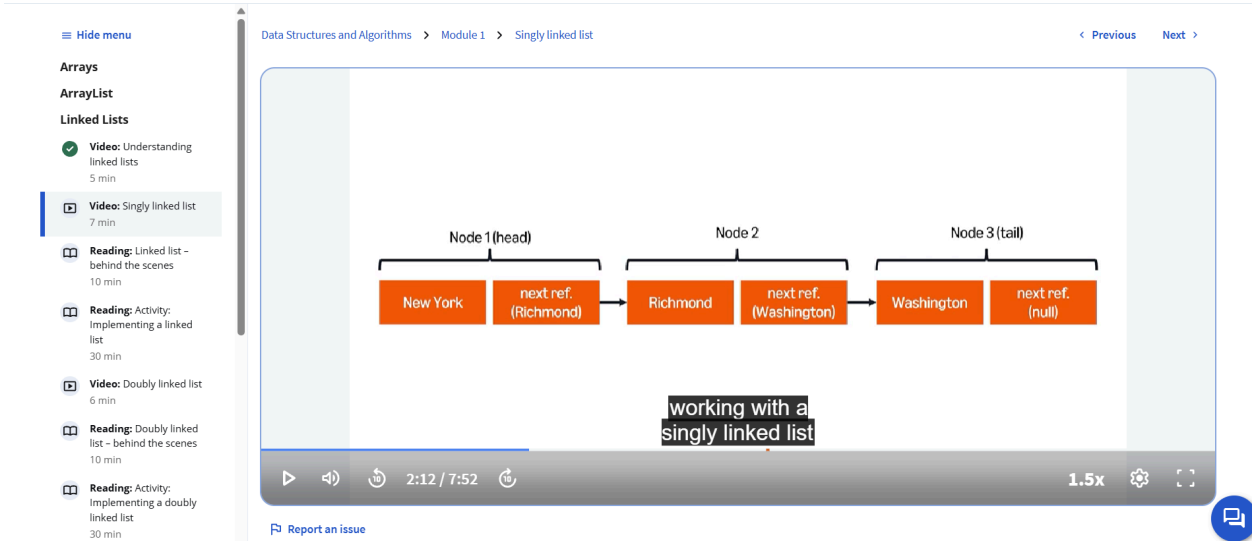
What is a Linked List?

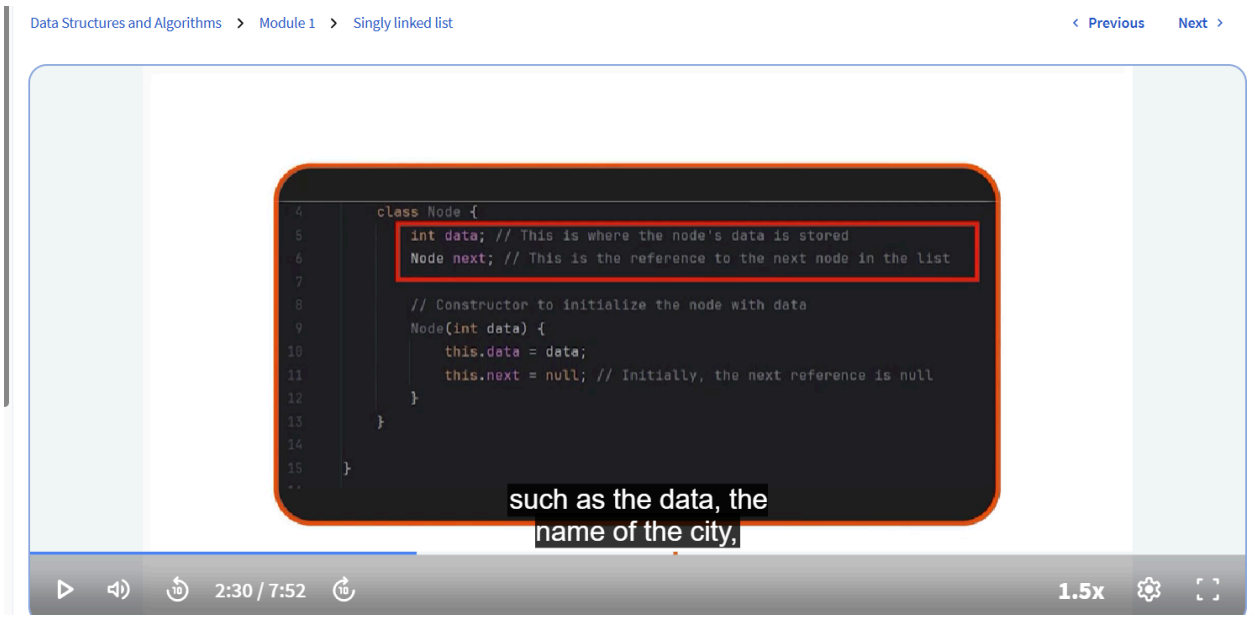
A **Linked List** is a linear data structure where elements (nodes) are connected using **pointers**. Each node contains:

1. **Data** – Stores the value.

2. **Next** – A pointer/reference to the next node in the list.

Unlike arrays, linked lists **do not require contiguous memory** allocation, making insertions and deletions more efficient.





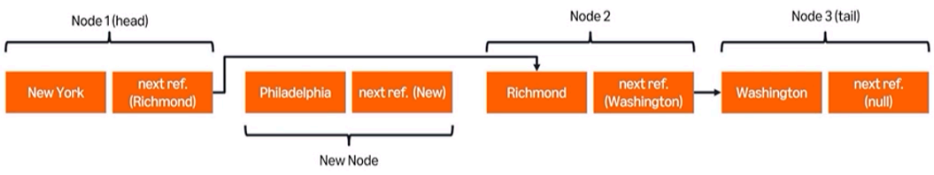
```
Node head; // This is the head of the list, the entry point
// Constructor to initialize the linked
SinglyLinkedList() {
    this.head = null; // Initially, the list is empty, so head is null
}
```

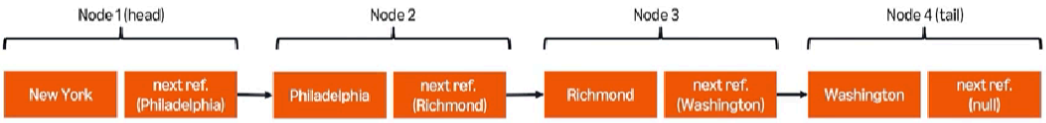
create an instance of

```
3  public class Main {
4      public static void main(String[] args) {
5          SinglyLinkedList list = new SinglyLinkedList();
6      }
7  }
8
9
```

So you can set the head node to

Add new node





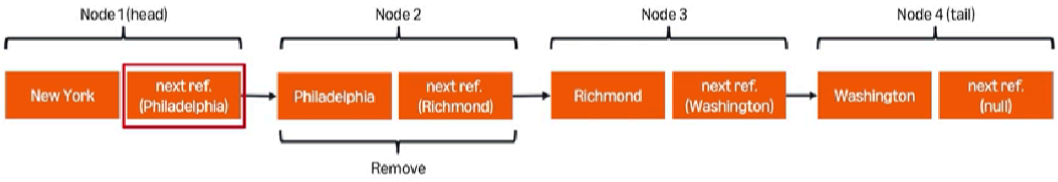
You're updating a train route by inserting a new station between two existing ones in a singly linked list. What is the correct action to take?

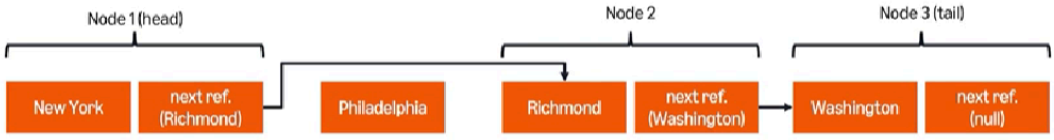
- ○ Reassign the 'head' to the new station.
- ◉ Update the 'next' reference of the previous station to point to the new station, and the new station's 'next' to the following station.
- ○ Remove the last station and insert the new one.
- ○ Add the new station at the end of the list without changing any other references.

✓ Correct

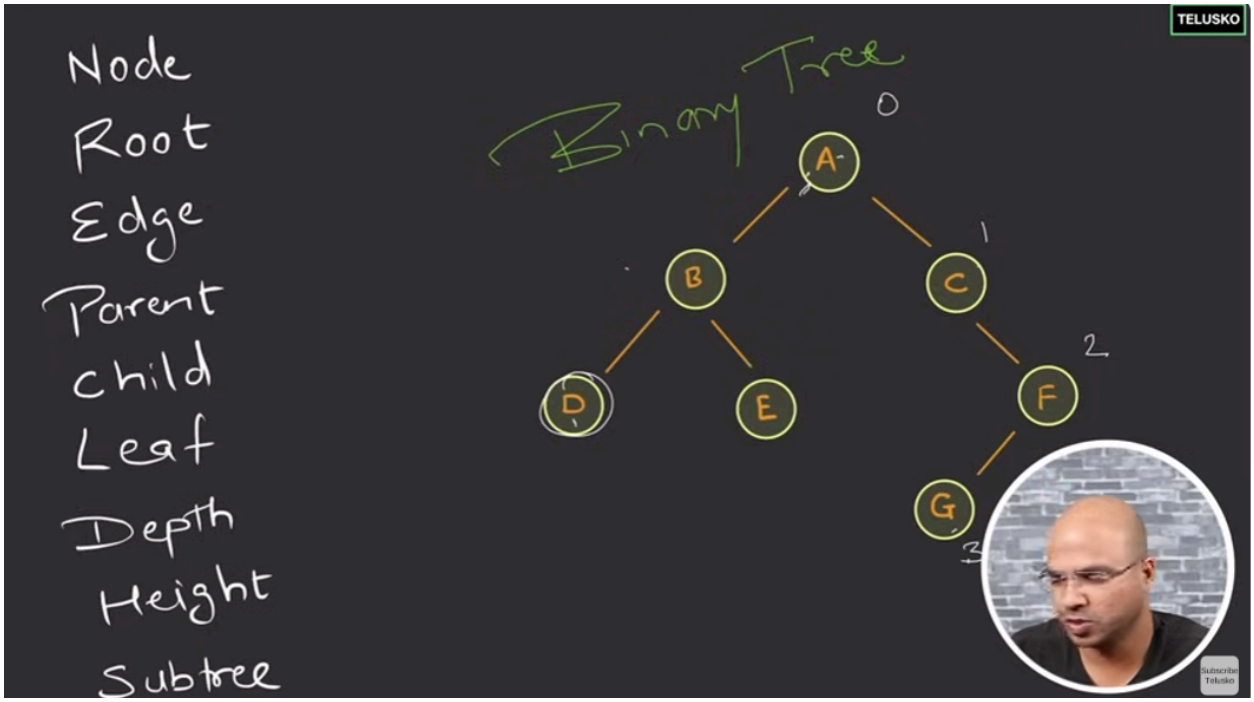
That's correct. This ensures the new station is properly inserted in the sequence.

Removing node





TRee



Binary Tree and a **Binary Search Tree (BST)**

A **Binary Tree** and a **Binary Search Tree (BST)** are both types of tree data structures in computer science, but they have different properties and use-cases. Let's break them down:

Binary Tree

Definition:

A **Binary Tree** is a tree data structure where each node has **at most two children**: a left child and a right child.

Properties:

- Can be **complete**, **full**, **perfect**, or **skewed**.
- No ordering is required among the nodes.
- Used in various hierarchical data representations (e.g., expression trees, parse trees).

Example:

1

```
  / \
 2   3
/ \
4   5
```

🌲 Binary Search Tree (BST)

Definition:

A **Binary Search Tree** is a special type of binary tree where:

- **Left subtree** of a node contains only nodes with **keys less than** the node's key.
- **Right subtree** of a node contains only nodes with **keys greater than** the node's key.
- No duplicate nodes (usually).

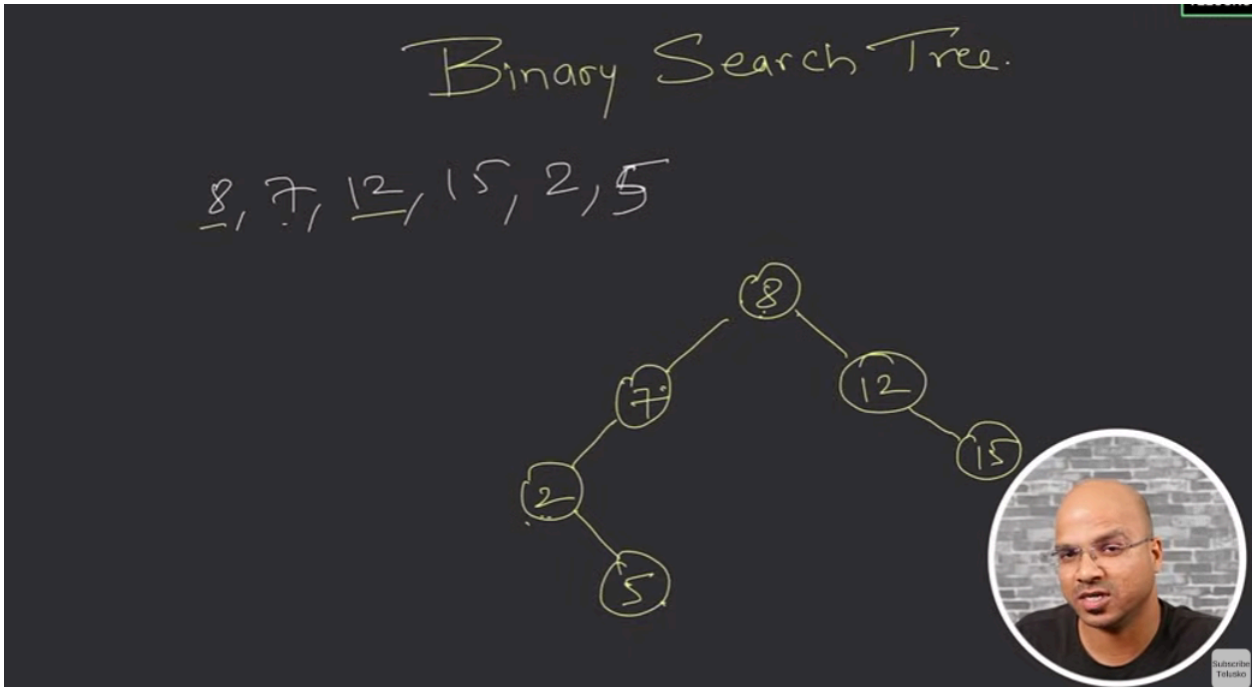
Properties:

- Enables **efficient searching**, **insertion**, and **deletion** (average time complexity: O(log n), worst: O(n)).
- **In-order traversal** of a BST yields **sorted order** of elements.

Example:

```
   8
  / \
 3   10
/ \   \
```

```
1   6   14
 /\  /
4  7 13
```



Feature	Binary Tree	Binary Search Tree (BST)
Structure	Any structure with ≤2 children	Ordered structure
Left/Right Rule	No rule	Left < Node < Right
Use Case	Generic trees (e.g., expression)	Fast lookup, insertion, deletion
Sorted Elements	Not necessarily	Yes (via in-order traversal)

Both **Binary Trees** and **Binary Search Trees (BSTs)** can be **traversed using DFS (Depth-First Search)** and **BFS (Breadth-First Search)** techniques. These are not types of trees — they are **tree traversal algorithms** used to explore the nodes of a tree.

DFS (Depth First Search) and **BFS (Breadth First Search)**, two fundamental algorithms used in **graph and tree traversal**.

What is DFS?

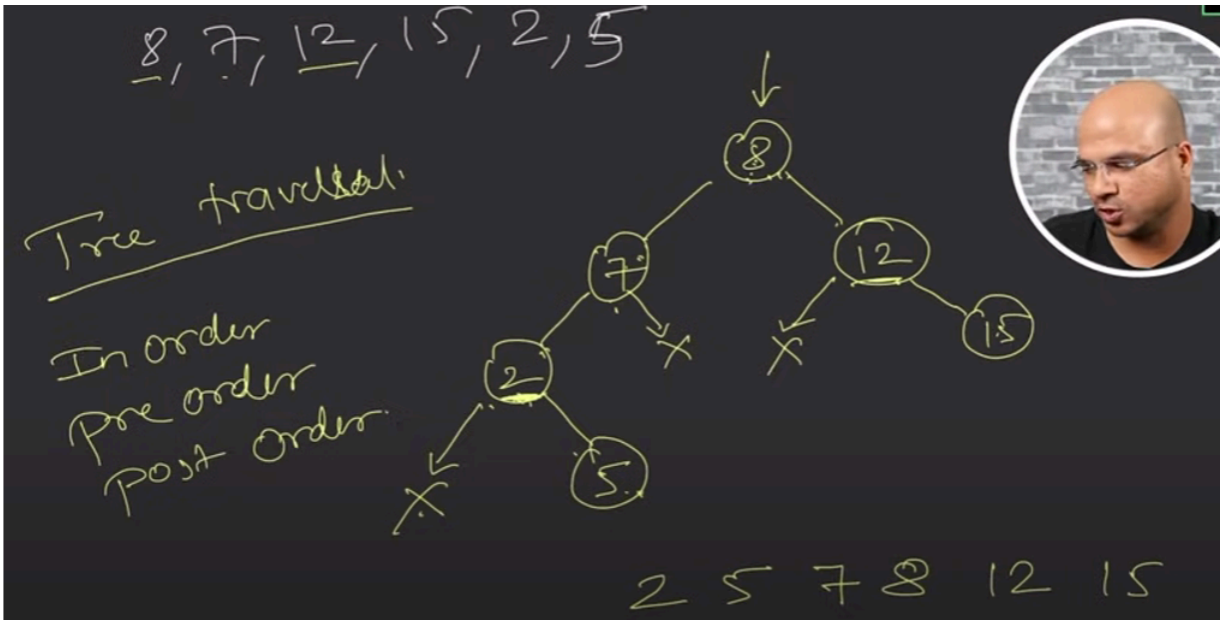
DFS (Depth-First Search) explores **as deep as possible** along each branch before backtracking.

It has **3 common traversal techniques (for trees):**

Traversal	Order	Example Use
Inorder	Left → Root → Right	Used in **Binary Search Trees** to get sorted order
Preorder	Root → Left → Right	Used to **clone/copy** a tree or to **serialize** it
Postorder	Left → Right → Root	Used to **delete** a tree or evaluate **postfix expressions**

These are **DFS-style traversals** because they go deep into the tree before visiting siblings.

Data structure used:

- **Stack** (can be explicit using a stack or implicit with recursion)
- 

✅ What is BFS?

BFS (Breadth-First Search) explores all **nodes at the current level** before going to the next level.

It has **one main traversal technique**:

Traversal	Order	Example Use
Level Order	Level by level (top to bottom, left to right)	Used in **shortest path**, **printing levels**, or **tree visualization**

It uses a **queue** to keep track of nodes.

Data structure used:

- **Queue**

Red-black tree in Data Structure

The red-Black tree is a binary search tree.It is a self-balancing Binary Search tree. Here, self-balancing means that it balances the tree itself by either doing the rotations or recoloring the nodes.

This tree data structure is named as a Red-Black tree as each node is either Red or Black in color.

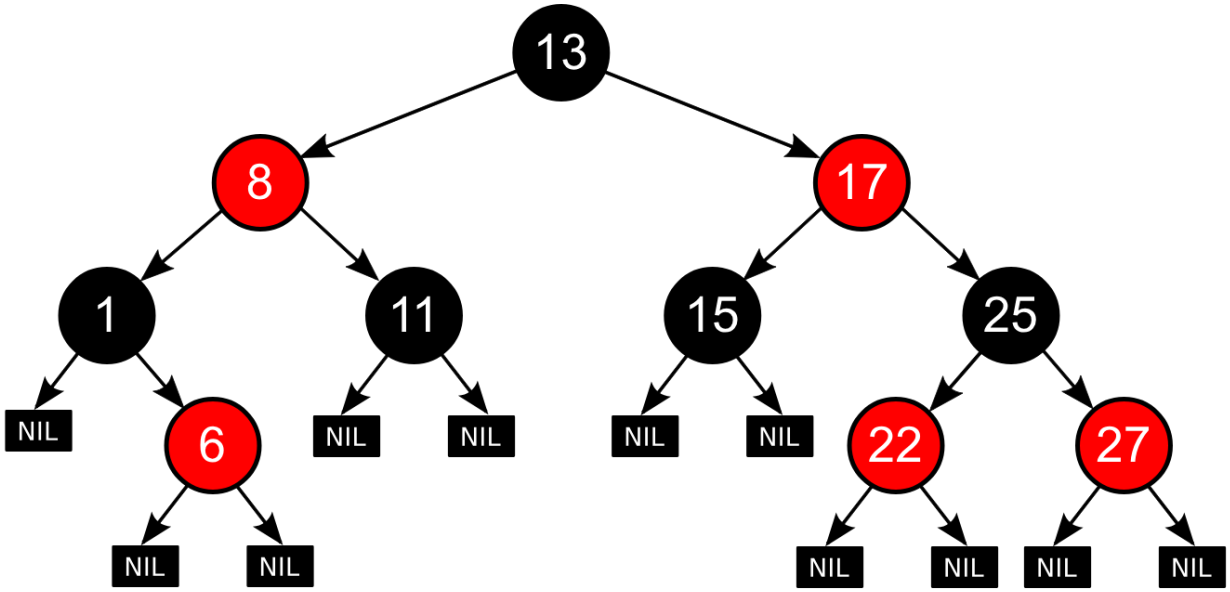
In a binary search tree, the values of the nodes in the left subtree should be less than the value of the root node, and the values of the nodes in the right subtree should be greater than the value of the root node.

🔴⚫ Red-Black Tree Properties

Each node in the tree contains an extra bit for color: **red** or **black**. The tree maintains these properties:

1. **Every node is either red or black**.
2. **The root is always black**.
3. **All leaves (nulls or NILs) are black**.
4. **Red nodes can't have red children** (no two reds in a row).

5. **Every path from a node to its descendant NIL nodes has the same number of black nodes** (called **black-height**).



Why Use a Red-Black Tree?

- Guarantees **balanced** tree height: maximum height is about **2 × log(n)**.
- Ensures **fast operations** even in the worst case:
    - **Search:** O(log n)
    - **Insert:** O(log n)
    - **Delete:** O(log n)

Rotations and Recoloring

To maintain its properties during insertions and deletions, a red-black tree may perform:

- **Left Rotation**
- **Right Rotation**
- **Recoloring**

Insert these elements in order:(Ex: Left Rotation)

10, 20, 30

🔽 **Step-by-step:**

1. Insert 10

- The tree is empty, so 10 becomes the **root**.
- Property: The **root is always black**.

[10B]

✔ Balanced, all red-black properties are satisfied.

---

2. Insert 20

- 20 > 10 → goes to the **right of 10**.
- New nodes are **always red** by default.

```
[10B]
   \
  [20R]
```

✔ No issues! Red child under black parent is allowed.

---

3. Insert 30

- 30 > 20 > 10 → goes to the **right of 20**.

```
[10B]
   \
  [20R]
     \
    [30R]
```

❌ Problem: Two **red nodes in a row** (20R → 30R), which violates **property #4** (No red-red).

🔁 **Fix: Left Rotation + Recoloring**

- **Left rotate** at 10 to bring 20 up.
- Recolor: 20 becomes **black**, 10 and 30 become **red**.

```
   [20B]
   /   \
 [10R]  [30R]
```

✔ Now all red-black properties are satisfied again!

✅ **Final Red-Black Tree after inserting 10, 20, 30:**

```
   [20B]
   /   \
 [10R]  [30R]
```

Insert these elements in order:(Ex: right Rotation)

Insert:

10, 5, 1

🔽 **Step-by-step:**

1. Insert 10

- Tree is empty → 10 becomes root and is colored **black**.

[10B]

2. Insert 5

- 5 < 10 → goes to the **left** of 10.
- New node 5 is **red** by default.

```
 [10B]
  /
[5R]
```

✔ No problem, red child under black parent.

3. Insert 1

- 1 < 5 < 10 → goes to the **left** of 5.
- New node 1 is **red**.

```
  [10B]
   /
 [5R]
 /
[1R]
```

❌ Problem: **Red parent (5R) and red child (1R)** – violates red-black rule #4 (no two red nodes in a row).

🔁 **Fix: Right Rotation + Recoloring**

- Perform a **right rotation** at 10.
- Make 5 the new root.
- Recolor: 5 becomes **black**, 10 and 1 become **red**.

```
  [5B]
  /   \
[1R]  [10R]
```

✔ All red-black properties are satisfied!

✅ **Final Red-Black Tree after inserting 10, 5, 1:**

```
  [5B]
  /   \
[1R]  [10R]
```

Ex:

```
    [10B]
    /    \
```

```
  [7B]    [18B]
       /
    [16R]
    /
  [15R]
```

⚠️ **Still a Problem:**

- Node 16 is **red**, and its child 15 is also **red**.
- 🔴 **Red-Red violation** again.

✅ **Fix Step: Right Rotation on 18**

We perform a **right rotation** at node 18 to move 16 up:

Rotation Breakdown:

Before:

```
    [18B]
     /
  [16R]
  /
[15R]
```

After Right Rotation on 18:

```
    [16R]
    /   \
 [15R]  [18B]
```

Now plug this into the full tree:

```
     [10B]
     /   \
```

```
[7B]    [16R]
      /    \
   [15R]  [18B]
```

⚠️ **One Last Problem:**

- 16 is red
- It's now the **child of 10**, which is **black** → this is fine.
- But 16 has a red child 15 → 🔴 still a **red-red violation**.

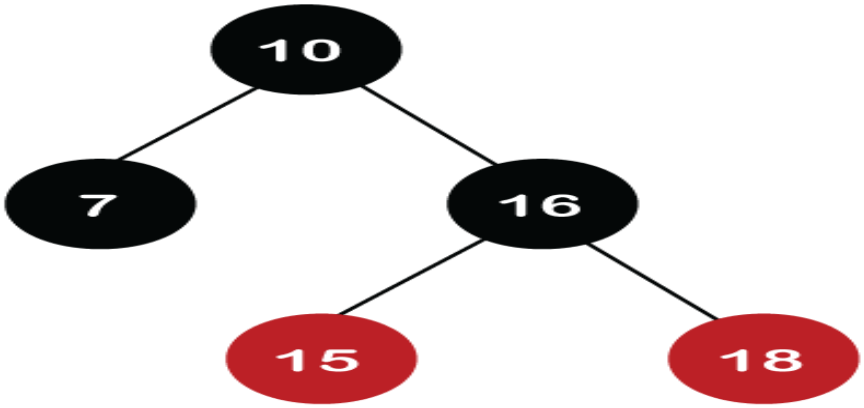
✅ **Final Fix: Recoloring**

We recolor:

- 16 → black
- 15 and 18 → red

Now the tree looks like this:

```
    [10B]
    /    \
 [7B]    [16B]
       /    \
    [15R]  [18R]
```



All Red-Black Tree properties are now satisfied!