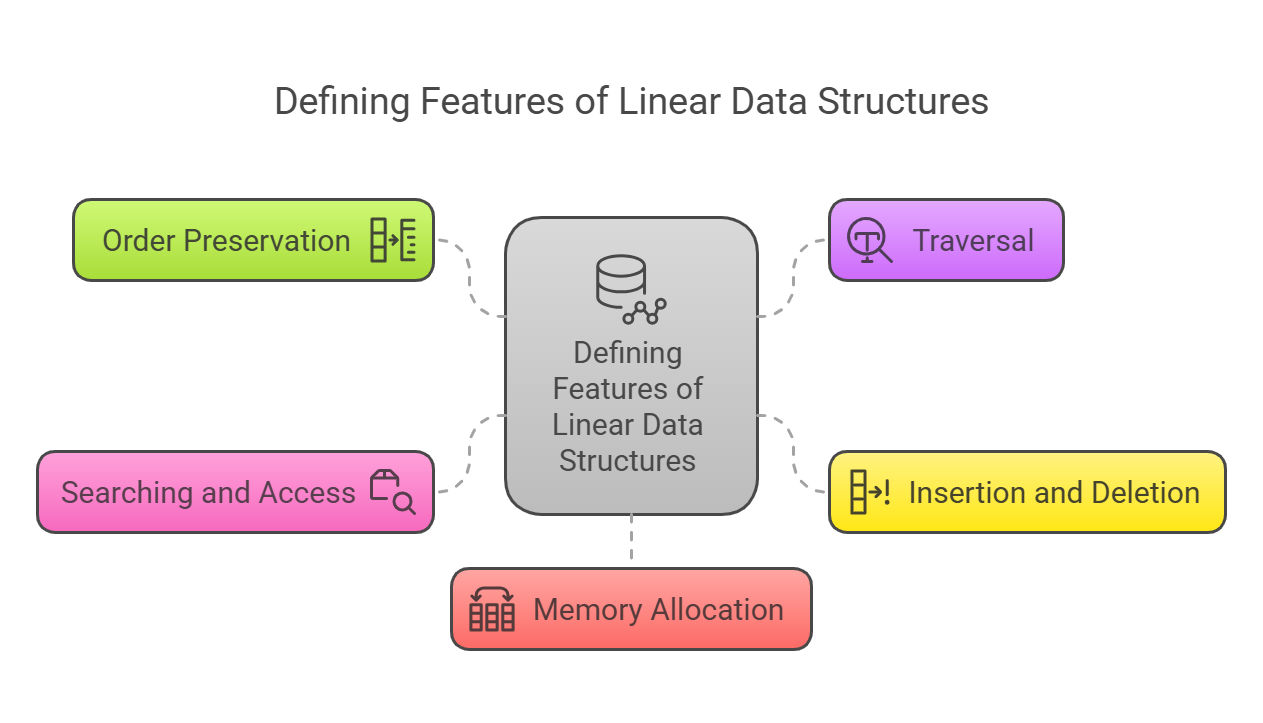
**UNIT -1**

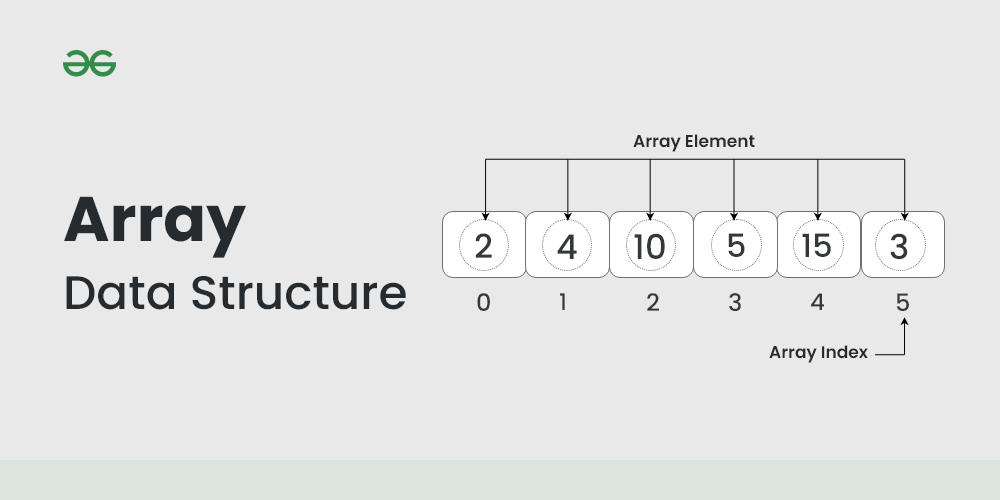
DATA STRUCTURES

1. **Definition and importance of linear data structures.**

A linear data structure is a type of data structure where elements are arranged in a sequential order, one after the other. Each element is connected to its previous and next element, forming a linear sequence. This arrangement allows for straightforward traversal, insertion, and deletion operations.

Common examples of linear data structures include:

* **Arrays**
* **Linked Lists**
* **Stacks**
* **Queues**
* **1. Arrays:**

**Definition:**  
An **array** is a collection of elements, all of the same type, arranged in a contiguous block of memory. The elements are identified by an index or position, starting from zero (0) for the first element. Arrays allow for efficient access to elements by their index.

**2. Linked Lists:**

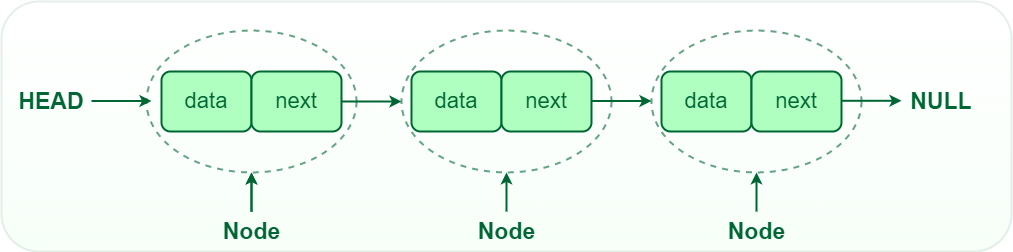
**Definition:**  
A **linked list** is a linear data structure consisting of a sequence of elements called nodes. Each node contains two parts:

1. **Data:** The actual value or information stored in the node.
2. **Next Pointer:** A reference (or link) to the next node in the sequence.

The last node's "next" pointer points to null, indicating the end of the list.

**Types of Linked Lists:**

* **Singly Linked List:** Each node points to the next node in the sequence.
* **Doubly Linked List:** Each node points to both the next and the previous nodes.
* **Circular Linked List:** The last node points back to the first node.



**3. Stacks:**

**Definition:**

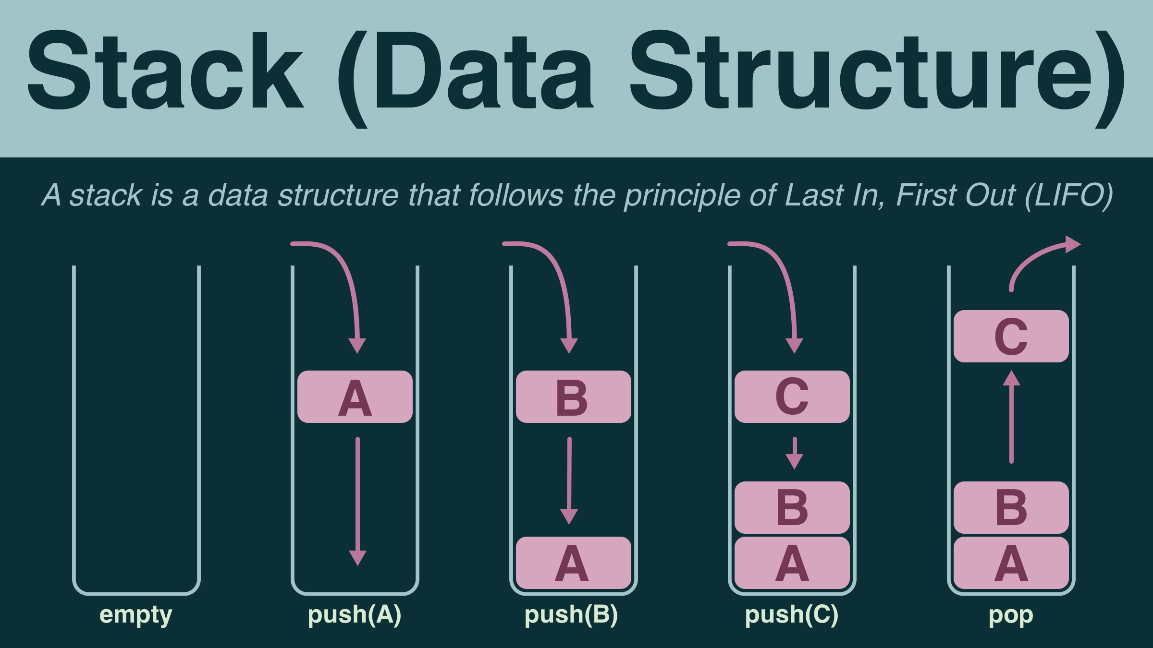
A stack is a linear data structure that follows the Last in First Out (LIFO) principle. This means the most recently added element is the first one to be removed. It works like a stack of plates — you add plates to the top and also remove from the top.

**Operations:**

**Push:** Adds an element to the top of the stack.

**Pop:** Removes and returns the element from the top of the stack.

**Peek/Top:** Returns the element at the top without removing it.

**Is Empty:** Checks whether the stack is empty.

**4. Queues:**

**Definition:**

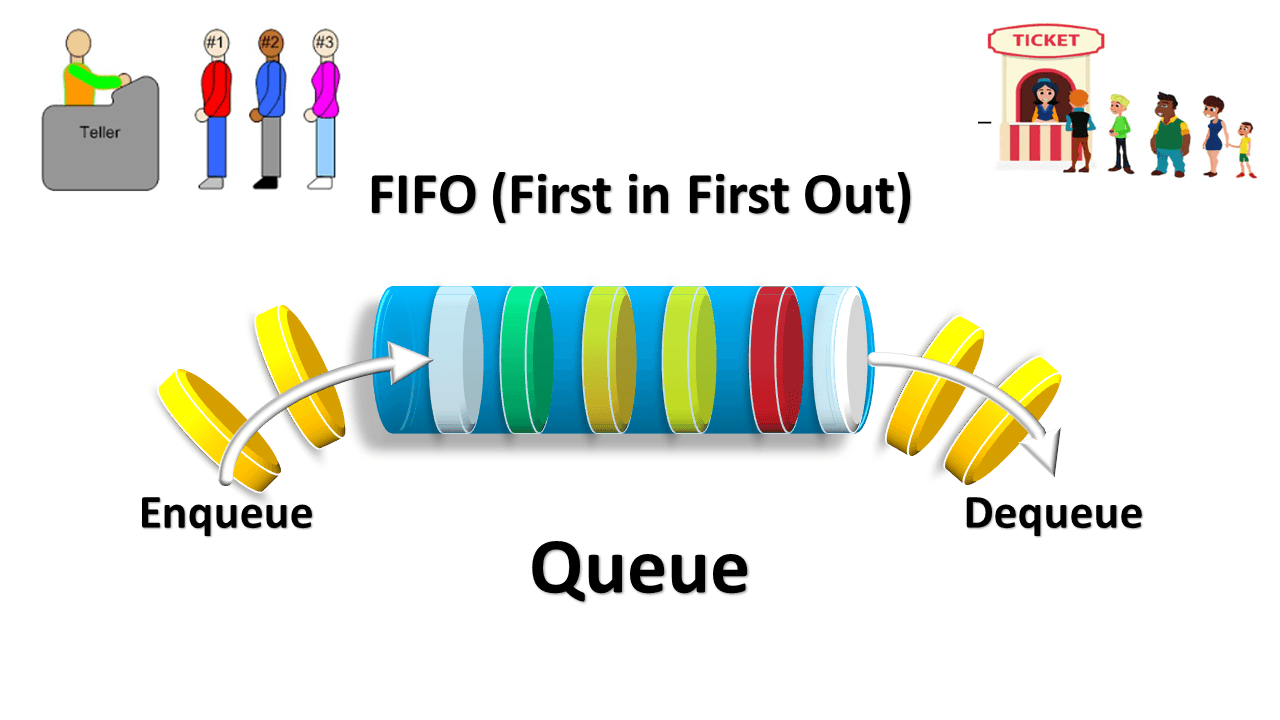
A queue is a linear data structure that follows the First In First Out (FIFO) principle. The first element added to the queue will be the first one to be removed, like a queue of people at a checkout line.

**Operations:**

**Enqueue**: Adds an element to the end (rear) of the queue.

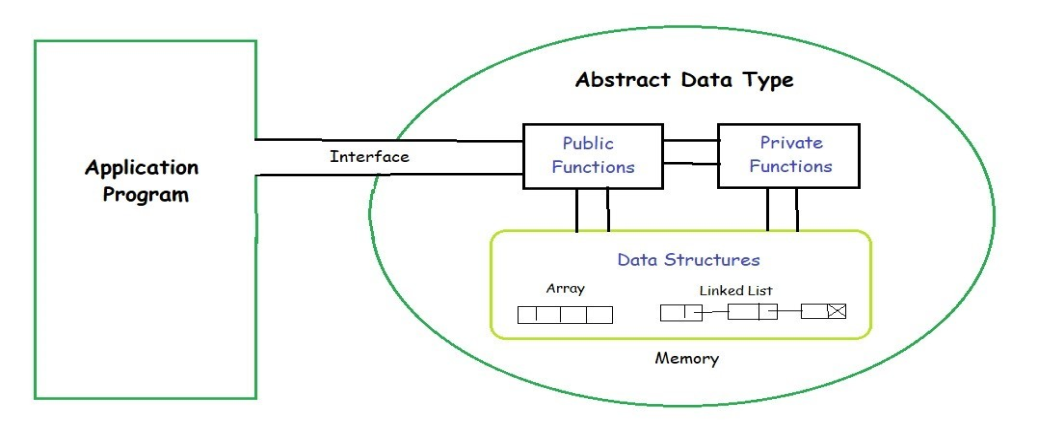
**Dequeue:** Removes and returns the element from the front of the queue.

**Front:** Returns the element at the front without removing it.

**Is Empty:** Checks whether the queue is empty.  
**2. Abstract data types (ADTs) and their implementation.**

An Abstract Data Type (ADT) is a conceptual model that defines a set of operations and behaviours for a data structure, without specifying how these operations are implemented or how data is organized in memory. The definition of ADT only mentions what operations are to be performed but not how these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. It is called “abstract” because it provides an implementation-independent view.

The process of providing only the essentials and hiding the details is known as abstraction.



1. **Overview of time and space complexity analysis for linear data structure.**

**Time Complexity:**

* **Definition:**

Measures the execution time of an algorithm as a function of the input size (n).

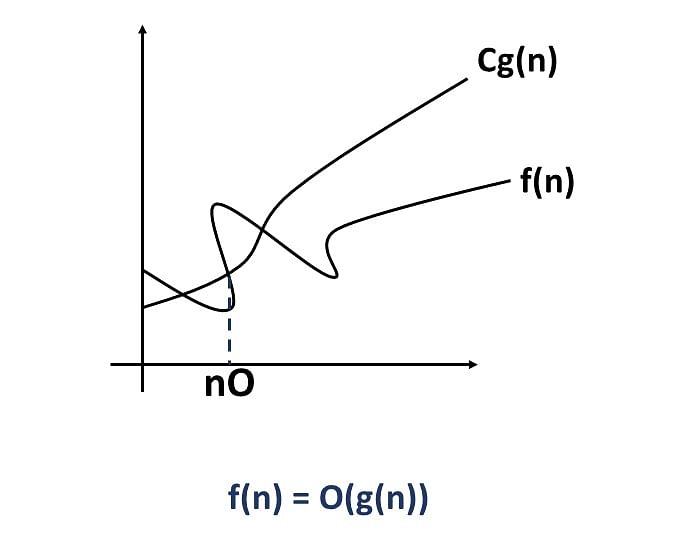
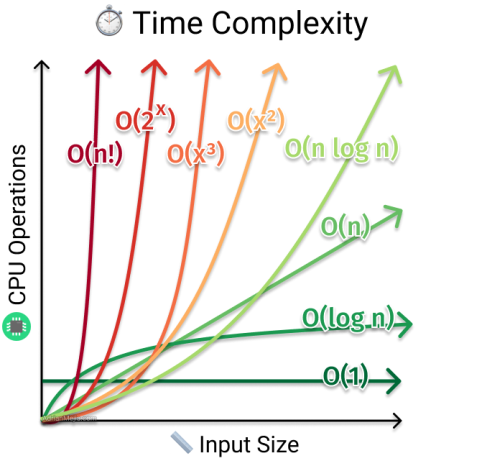
* **Common Time Complexities:**
  + **O (1) (Constant Time):**The execution time remains the same regardless of the input size.
  + **O (n) (Linear Time):**The execution time grows proportionally to the input size.
  + **O (n^2) (Quadratic Time):**The execution time grows proportionally to the square of the input size.
* **Examples in Linear Data Structures:**
  + **Array Access:**Accessing an element by its index in an array is typically O (1).
  + **Array Traversal/Search:**Iterating through each element in an array is typically O (n).
  + **Insertion/Deletion (Unsorted Array):**Inserting or deleting an element in an unsorted array can be O (n) in the worst case (if the element needs to be shifted).
  + **Insertion/Deletion (Linked List):**Inserting or deleting an element in a linked list can be O (1) if you know the position or O (n) if you need to traverse.

**Space Complexity:**

* **Definition:**

Measures the amount of memory an algorithm uses as a function of the input size.

* **Common Space Complexities:**
  + **O (1) (Constant Space):**The algorithm uses a fixed amount of memory, regardless of the input size.
  + **O (n) (Linear Space):**The amount of memory used grows proportionally to the input size.
* **Examples in Linear Data Structures:**
  + **Array:**An array stores elements in contiguous memory locations, so its space complexity is O (n).
  + **Linked List:**a linked list stores elements in non-contiguous memory locations and its space complexity is also O (n).
  + **Stack:**A stack, implemented using an array or linked list, has a space complexity of O (n) in the worst case.
  + **Queue:**A queue, implemented using an array or linked list, has a space complexity of O (n) in the worst case.



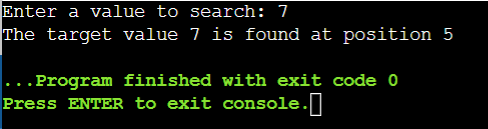
**Space complexity**

1. **Searching techniques: linear & binary search**

**What is Linear Search?**

A linear search, also called as sequential search, simply scans each element one at a time. In this search, an array is sequentially traversed from the first element until the element is found or the end of the array is reached. This method can only be suitable for searching over a small array or an unsorted array.

**Algorithm:**

1. Assume we need to find an element that is in an array in random order.
2. Start with the first position and compare it to the target in order to search for a target element.
3. If the current element matches the target element, return the position of the current element.
4. If not, move on to the next one until we reach the very end of an array.
5. If still unable to find the target, return -1.
6. // Linear Search Implementation in C Language
7. #include <stdio.h>
8. int LinearSearch(int arr[], int target, int Length)
9. {
10. int i;
11. for(i=0;i<Length; i++)
12. {
13. if (target == arr[i])
14. return (i + 1);
15. }
16. return -1;
17. }
18. int main()
19. {
20. int elements count;
21. int n, result;
22. //Create an array and initialize it
23. int search\_list[] = { 3, 1, 9, 8, 7, 12, 56, 23, 89 };
24. elements\_count=sizeof(search\_list)/sizeof(int);
26. //Read the target value to search
27. printf("Enter a value to search: ");
28. scanf("%d", &n);
29. result = LinearSearch(search\_list, n, elements\_count);
30. if (n != -1)
31. printf("The target value %d is found at position %d", n, result);
32. else
33. printf("Target not found!");
34. return 0;
35. }
36. **Output**
37. 

**BINARY SEARCH:**

Binary search is a searching algorithm that operates on a sorted dataset. Instead of scanning every element sequentially, like linear search, binary search divides the dataset into halves, narrowing down the search range systematically. This "divide and conquer" approach makes it significantly faster than other search methods.

Let's dissect it. Consider using a dictionary to look up a word. You don’t flip through every page. Instead, you start somewhere in the middle, determine if the word lies before or after the current page, and repeat until you find it. That’s essentially how the binary search in C works.

Binary search requires:

1. A sorted dataset.
2. Efficient handling of midpoints.
3. Repeated halving of the search range.

This simplicity and speed make binary search a preferred choice in competitive programming and real-world applications.

#**include <stdio.h>**

**int binarySearch(int arr[], int size, int target) {**

**int low = 0, high = size - 1;**

**while (low <= high) {**

**int mid = (low + high) / 2;**

**if (arr[mid] == target)**

**return mid;**

**else if (arr[mid] < target)**

**low = mid + 1;**

**else**

**high = mid - 1;**

**}**

**return -1;**

**}**

**int main() {**

**int arr[] = {10, 20, 30, 40, 50};**

**int size = sizeof(arr) / sizeof(arr[0]);**

**int target = 40;**

**int result = binarySearch(arr, size, target);**

**if (result != -1)**

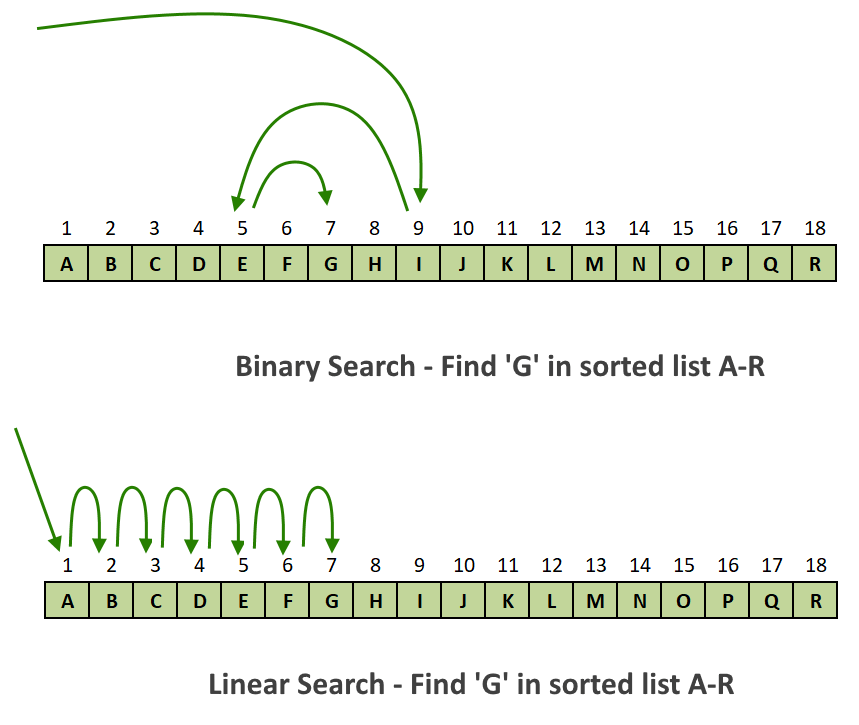
**printf("Element found at index %d\n", result);**

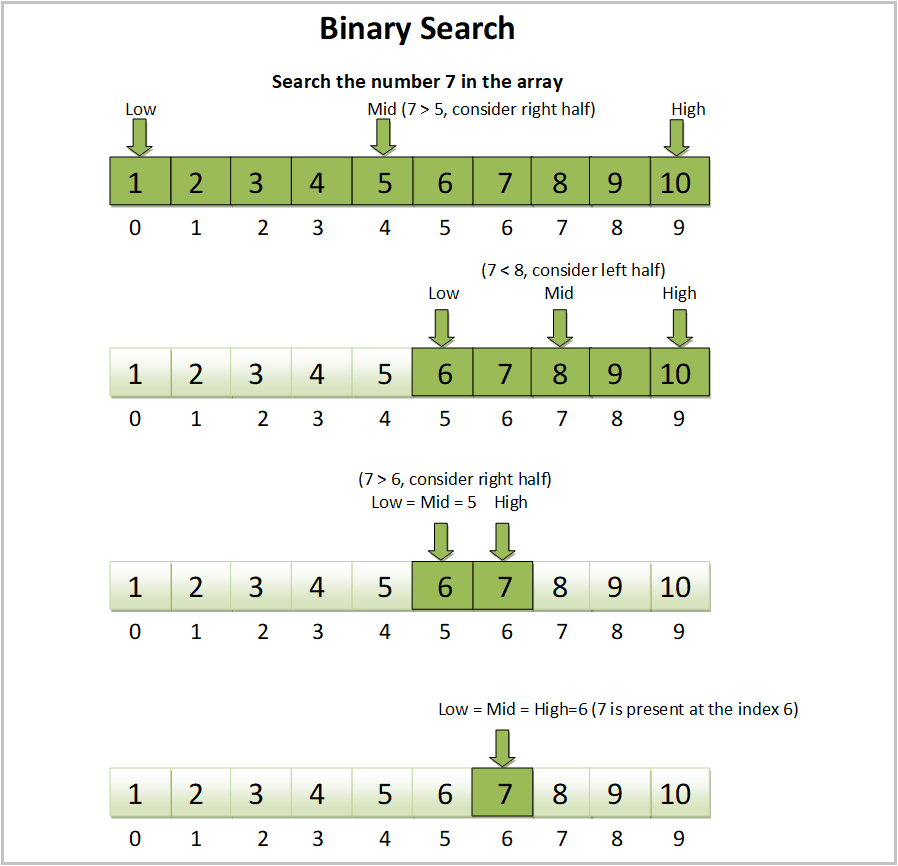
**else**

**printf("Element not found\n");**

**return 0;**

**}**





**5. SORTING TECHNIQUES: BUBBLE SORT, SELECTION SORT, and INSERTION SORT.**

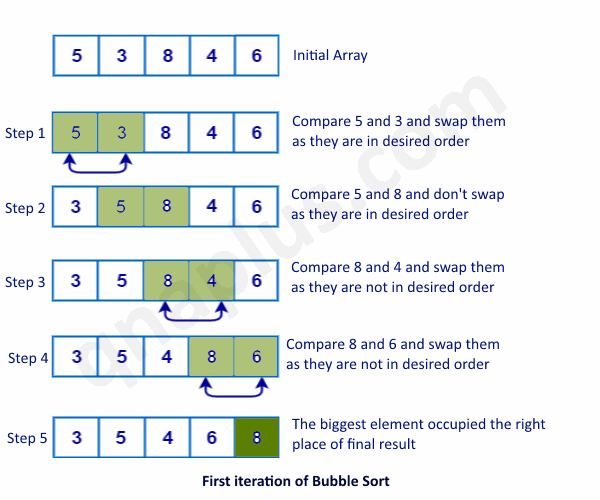
**1. BUBBLE SORT:**

**Bubble Sort Algorithm**

1. Start at the beginning of the list.
2. Compare each pair of adjacent elements.
3. If the elements are in the wrong order, swap them.
4. Move to the next pair and repeat step 3 until the end of the list.
5. After each pass through the list, the largest element moves to its correct position.
6. Repeat the process for the remaining elements (excluding the last sorted elements) until no swaps are needed.

## Bubble Sort Example (Step by Step)

Let's sort the list [4, 2, 7, 1] using Bubble Sort.



**EXAMPLE CODE:**

**#include <stdio.h>**

**// Function to perform bubble sort**

**void bubbleSort(int arr[], int n) {**

**int i, j, temp;**

**int swapped;**

**// Outer loop to iterate through the array**

**for (i = 0; i < n - 1; i++) {**

**swapped = 0; // Flag to check if any swap occurred in this pass**

**// Inner loop to compare and swap adjacent elements**

**for (j = 0; j < n - i - 1; j++) {**

**// Compare adjacent elements**

**if (arr[j] > arr[j + 1]) {**

**// Swap elements if they are in the wrong order**

**temp = arr[j];**

**arr[j] = arr[j + 1];**

**arr[j + 1] = temp;**

**swapped = 1; // Set flag to indicate a swap occurred**

**}**

**}**

**// If no swaps occurred in this pass, the array is already sorted**

**if (swapped == 0) {**

**break;**

**}**

**}**

**}**

**// Function to print an array**

**void printArray(int arr[], int n) {**

**int i;**

**for (i = 0; i < n; i++) {**

**printf("%d ", arr[i]);**

**}**

**printf("\n");**

**}**

**int main() {**

**int arr[] = {64, 34, 25, 12, 22, 11, 90};**

**int n = sizeof(arr) / sizeof(arr[0]);**

**printf("Unsorted array: \n");**

**printArray(arr, n);**

**bubbleSort(arr, n);**

**printf("Sorted array: \n");**

**printArray(arr, n);**

**return 0;**

**}**

**Explanation:**

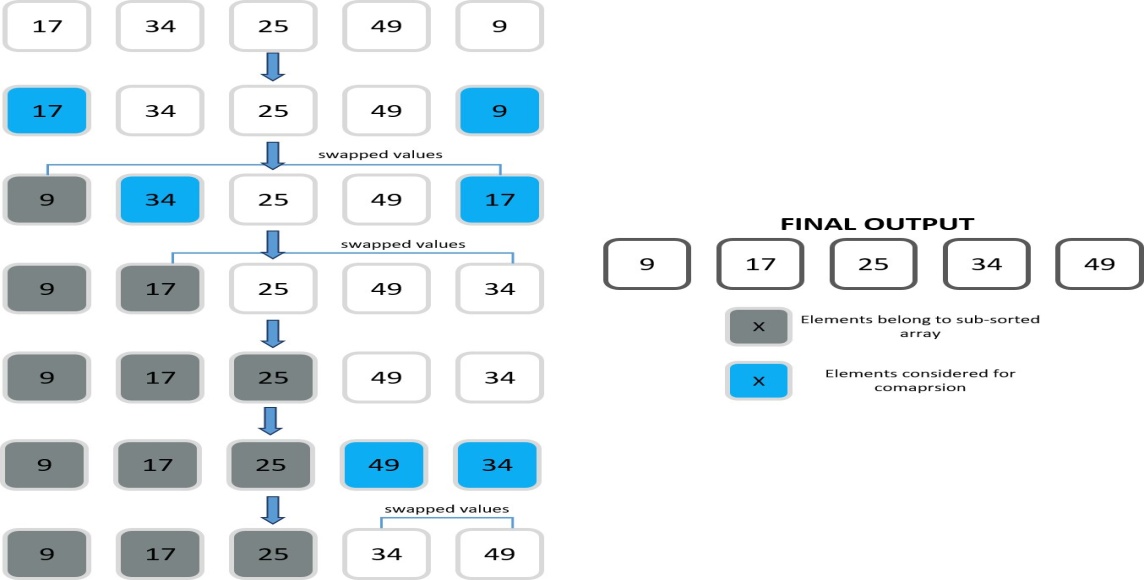
1. 1. #include <stdio.h>:

Includes the standard input/output library for functions like printf.

1. 2. bubbleSort(int arr[], int n):
   * Takes an integer array arr and its size n as input.
   * i and j are loop counters.
   * temp is a temporary variable used for swapping.
   * swapped is a flag to optimize the algorithm. If no swaps occur in a pass, it means the array is already sorted, and we can terminate early.
   * Outer loop (for (i = 0; i < n - 1; i++)): Iterates through the array n-1 times.
   * Inner loop (for (j = 0; j < n - i - 1; j++)): Compares adjacent elements.
   * if (arr[j] > arr[j + 1]): Checks if the current element is greater than the next element.
   * Swapping: If the elements are out of order, they are swapped using a temporary variable.
   * swapped = 1;: Sets the flag to indicate that a swap occurred.
   * if (swapped == 0) break;: If no swaps occurred in a pass, the array is sorted, and the algorithm terminates.
2. 3. printArray(int arr[], int n):
   * Takes an integer array arr and its size n as input.
   * Prints the elements of the array to the console.
3. 4. main():
   * Creates an unsorted array arr.
   * Calculates the size of the array n.
   * Prints the unsorted array.
   * Calls the bubbleSort function to sort the array.
   * Prints the sorted array.

**2. SELECTION SORT**

Selection Sort is a simple comparison-based sorting algorithm. Its primary idea is to divide the input list into two parts: a sorted sublist and an unsorted sublist. Initially, the sorted sublist is empty, and the unsorted sublist contains all the elements. The algorithm repeatedly selects the smallest (or largest, depending on the sorting order) element from the unsorted sublist and moves it to the end of the sorted sublist. This process continues until the unsorted sublist is empty and the sorted sublist contains all the elements in the desired order.



**EXAMPLE CODE:**

**#include <stdio.h>**

**void selectionSort(int arr[], int n) {**

**int i, j; // Outer, Inner loop counters**

**int minIndex; // Smallest element index**

**int temp; // Swap variable**

**// Outer loop: up to second-last element**

**for (i = 0; i < n - 1; i++) {**

**minIndex = i; // Start with current index**

**// Inner loop: find smallest in unsorted part**

**for (j = i + 1; j < n; j++) {**

**if (arr[j] < arr[minIndex]) {**

**minIndex = j; // Update smallest index**

**}**

**}**

**// Swap**

**temp = arr[i];**

**arr[i] = arr[minIndex];**

**arr[minIndex] = temp;**

**}**

**}**

**int main() {**

**int arr[] = {17, 34, 25, 49, 9}; // Initial array**

**int n = sizeof(arr) / sizeof(arr[0]); // Array length**

**// Display original array**

**printf("Original Array: ");**

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

**printf("%d ", arr[i]);**

**}**

**printf("\n");**

**selectionSort(arr, n); // Sort the array**

**// Display sorted array**

**printf("Sorted Array: ");**

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

**printf("%d ", arr[i]);**

**}**

**printf("\n");**

**}**

**Output**

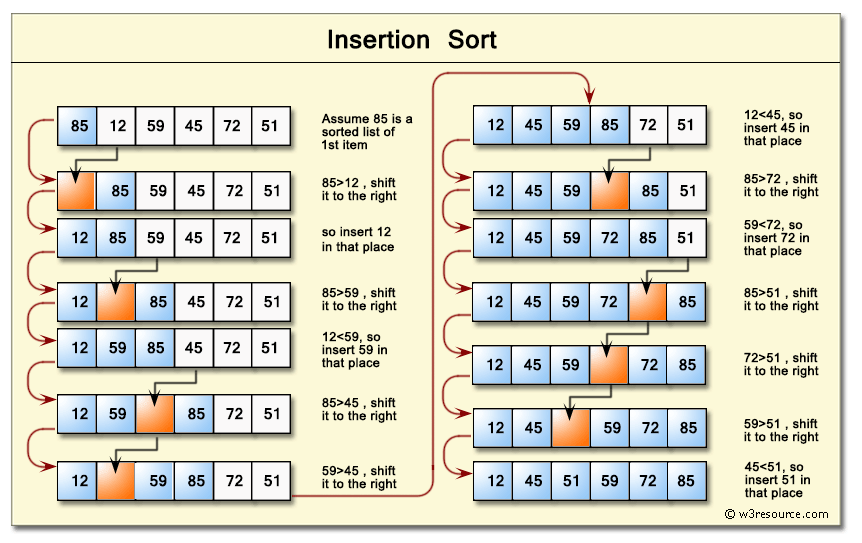
Original Array: 17 34 25 49 9

Sorted Array: 9 17 25 34 49

**3. INSERTION SORT:**

To implement the Insertion Sort algorithm in a programming language, we need:

1. An array with values to sort.
2. An outer loop that picks a value to be sorted. For an array with nn values, this outer loop skips the first value, and must run n−1n−1 times.
3. An inner loop that goes through the sorted part of the array, to find where to insert the value. If the value to be sorted is at index ii, the sorted part of the array starts at index 00 and ends at index i−1i−1.
4. def insertion\_sort(arr):
5. for i in range(1, len(arr)):
6. key = arr[i]
7. j = i - 1
8. while j >= 0 and key < arr[j]:
9. arr[j + 1] = arr[j]
10. j -= 1
11. arr[j + 1] = key



**Advantages:**

* **Simple to implement: Insertion sort is one of the easiest sorting algorithms to understand and implement.**
* **Efficient for small datasets: It performs well on small arrays or nearly sorted arrays.**
* **Stable: It maintains the relative order of equal elements.**
* **In-place: It sorts the array without requiring extra memory.**

**Disadvantages:**

* **Inefficient for large datasets: Its time complexity is O(n^2), making it slow for large arrays.**
* **Not suitable for large datasets: Its time complexity is O(n^2), making it slow for large arrays.**
* **Not as efficient as other sorting algorithms: Other algorithms like merge sort or quicksort are generally more efficient for large datasets.**

**THE END UNIT-1**