**What is DSA?**

**Data Structures and Algorithms** (DSA) is a fundamental part of Computer Science that teaches you how to think and solve complex problems systematically. Using the right data structure and algorithm makes your program run faster, especially when working with lots of data.

**What Is Time Complexity?**

Time complexity is defined in terms of how many times it takes to run a given algorithm, based on the length of the input. Time complexity is not a measurement of how much time it takes to execute a particular algorithm because such factors as programming language, operating system, and processing power are also considered.

Time complexity is a type of computational complexity that describes the time required to execute an algorithm. The time complexity of an algorithm is the amount of time it takes for each statement to complete. As a result, it is highly dependent on the size of the [processed data](https://www.simplilearn.com/what-is-data-processing-article)**. It also aids in defining an algorithm's effectiveness and evaluating its performance.**

**What Is Space Complexity?**

When an algorithm is run on a computer, it necessitates a certain amount of memory space. The amount of memory used by a program to execute it is represented by its space complexity. Because a program requires memory to store input data and temporal values while running, the space complexity is auxiliary and input space.

### 1. Big-O Notation

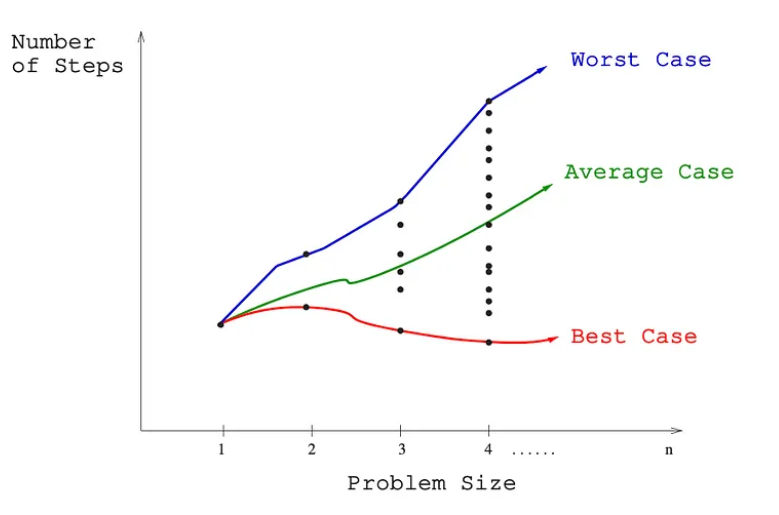
We define an algorithm’s **worst-case** time complexity by using the Big-O notation, which determines the set of functions grows slower than or at the same rate as the expression. Furthermore, it explains the maximum amount of time an algorithm requires to consider all input values.

### 2. Omega Notation

It defines the**best case** of an algorithm’s time complexity, the Omega notation defines whether the set of functions will grow faster or at the same rate as the expression. Furthermore, it explains the minimum amount of time an algorithm requires to consider all input values.

### 3. Theta Notation

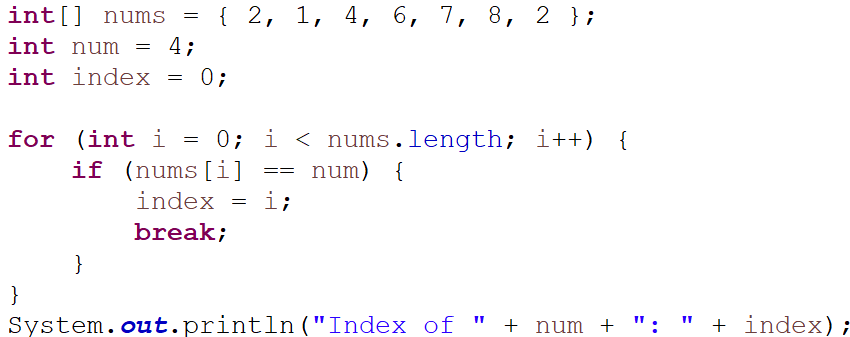
It defines the **average case** of an algorithm’s time complexity, the Theta notation defines when the set of functions lies in both **O(expression)** and **Omega(expression)**, then Theta notation is used. This is how we define a time complexity average case for an algorithm.



**Searching:**

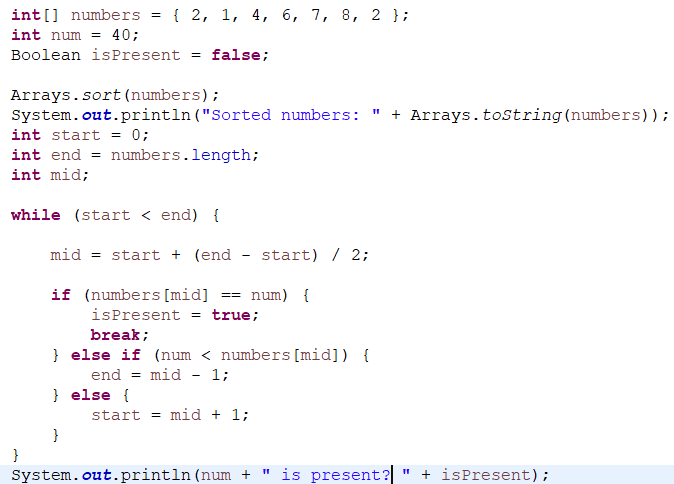
**Linear Search:**

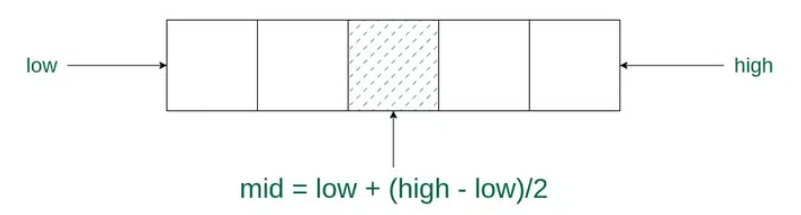
TimeComplexity: O(n)



**Binary Search:** It must be sorted

TimeComplexity: O(log(n))

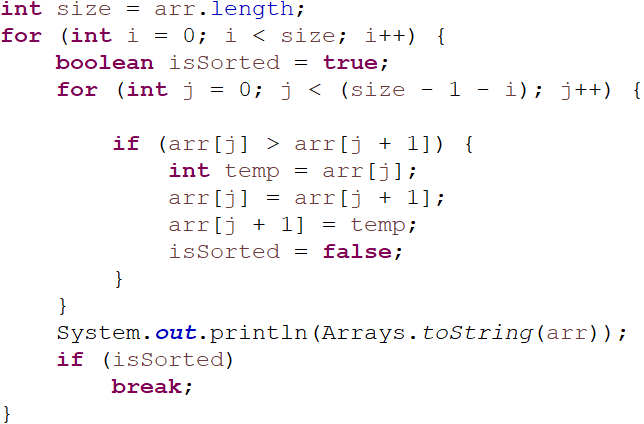
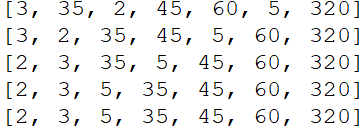




**Sorting:**

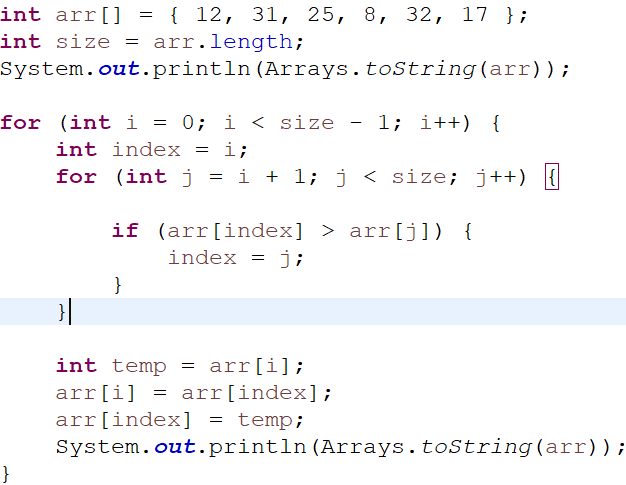
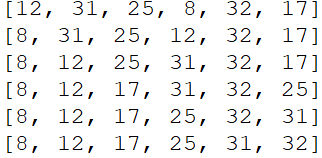
**BubbleSort: O(n2)**

In bubble sort algorithm, array is traversed from first element to last element. Here, current element is compared with the next element. If current element is greater than the next element, it is swapped.

**SelectionSort: O(n2)**

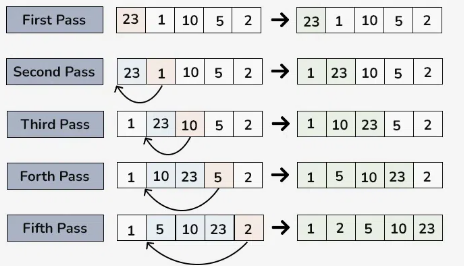
In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. It is an in-place comparison sorting algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

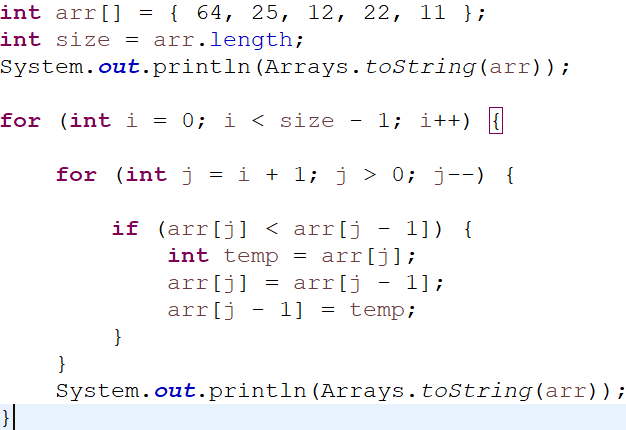
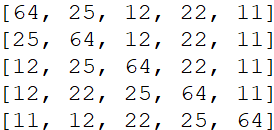
 

**InsertionSort: O(n2)**

**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 a **stable sorting** algorithm, meaning that elements with equal values maintain their relative order in the sorted output.

**Insertion sort**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.



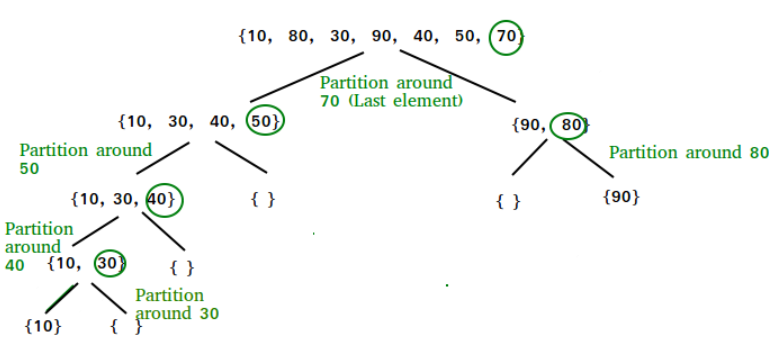
 

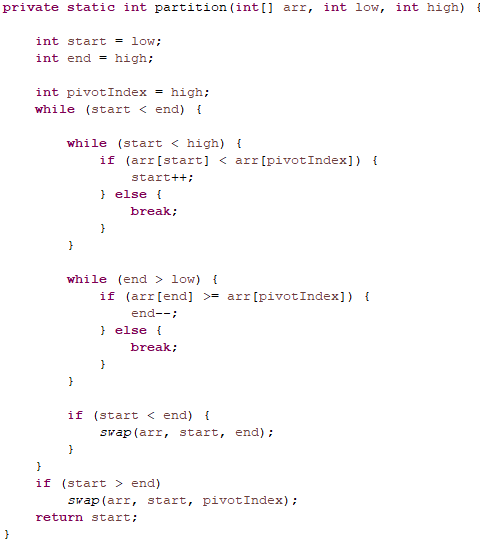
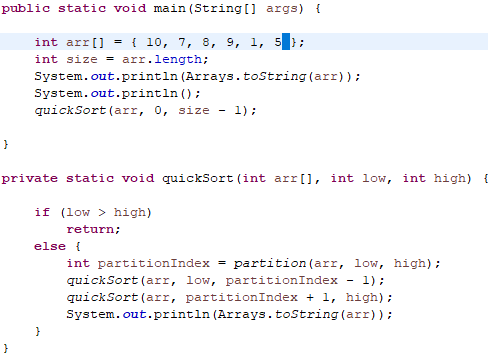
**QuickSort:**

**QuickSort** is a sorting algorithm based on the[Divide and Conquer algorithm](https://www.geeksforgeeks.org/divide-and-conquer-algorithm-introduction/) that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

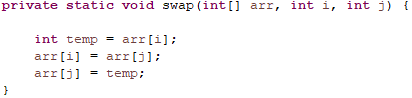
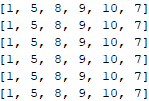
The key process in **quickSort**is a **partition()**. The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot.

Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



Output:

**Time Complexity:**

* **Best Case**: Ω (N log (N))  
  The best-case scenario for quicksort occur when the pivot chosen at the each step divides the array into roughly equal halves.  
  In this case, the algorithm will make balanced partitions, leading to efficient Sorting.
* **Average Case: θ ( N log (N))**Quicksort’s average-case performance is usually very good in practice, making it one of the fastest sorting Algorithm.
* **Worst Case: O(N2)**The worst-case Scenario for Quicksort occur when the pivot at each step consistently results in highly unbalanced partitions. When the array is already sorted and the pivot is always chosen as the smallest or largest element. To mitigate the worst-case Scenario, various techniques are used such as choosing a good pivot (e.g., median of three) and using Randomized algorithm (Randomized Quicksort ) to shuffle the element before sorting.
* **Auxiliary Space:** O(1), if we don’t consider the recursive stack space. If we consider the recursive stack space then, in the worst case quicksort could make *O*(*N*).

**MergeSort:**

**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.

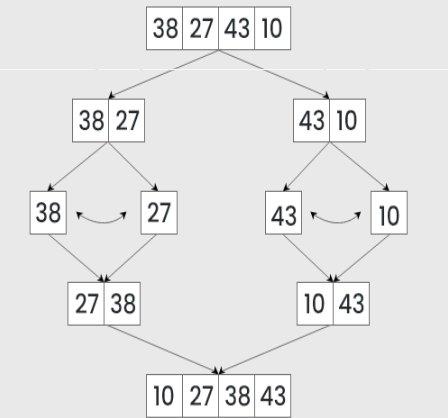
Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer**approach to sort a given array of elements.

Here’s a step-by-step explanation of how merge sort works:

**Divide:** Divide the list or array recursively into two halves until it can no more be divided.

**Conquer:**Each subarray is sorted individually using the merge sort algorithm.

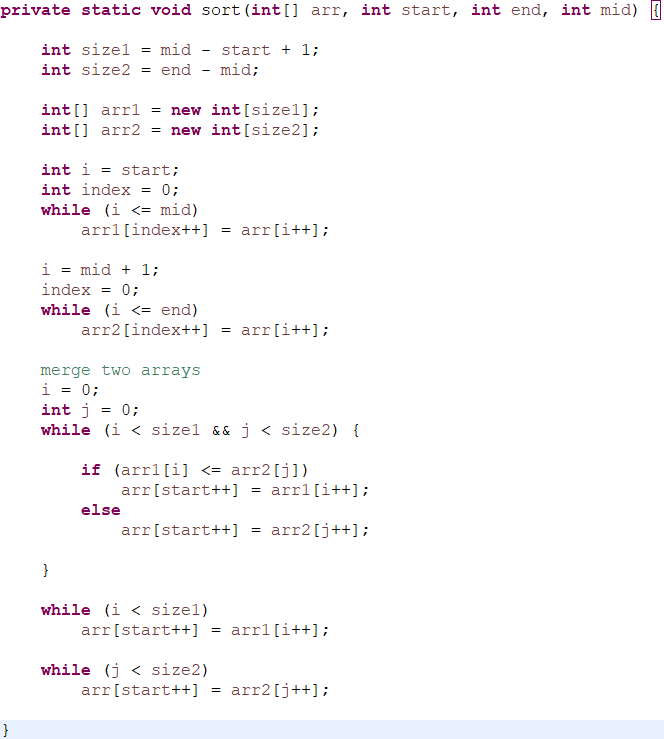
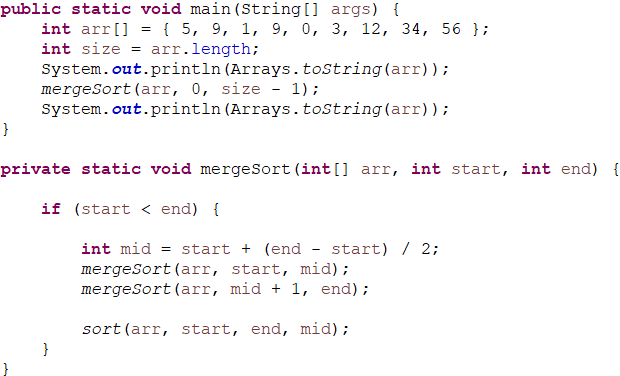
**Merge:**  The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.



**Time Complexity:**

* **Best Case:**O(n log n), When the array is already sorted or nearly sorted.
* **Average Case:** O(n log n), When the array is randomly ordered.
* **Worst Case:**O(n log n), When the array is sorted in reverse order.

**Space Complexity:**O(n), Additional space is required for the temporary array used during merging.

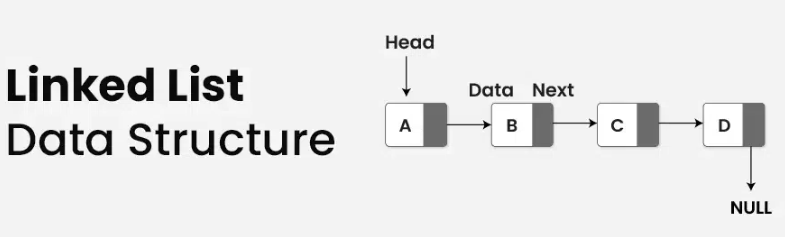
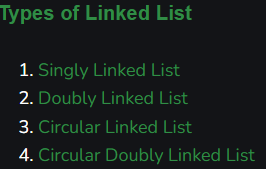
 

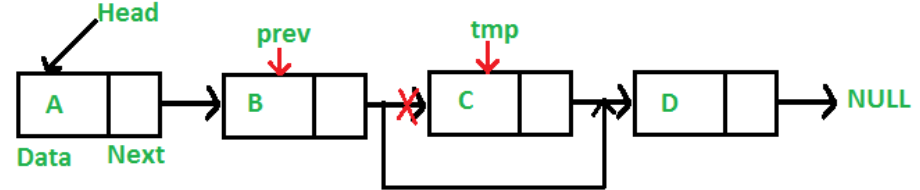
Output:



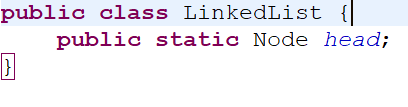
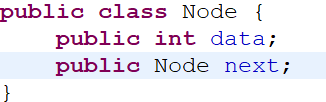
**LinkedList:**

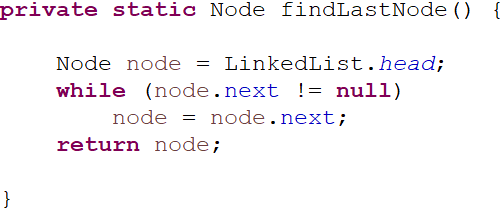
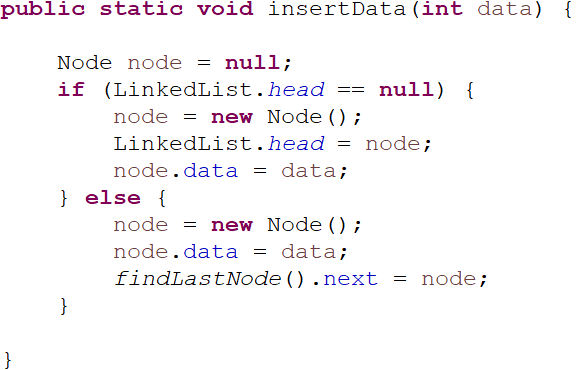
A **linked list** is a fundamental data structure in computer science. It consists of nodes where each node contains **data** and a**reference (link)** to the next node in the sequence. This allows for dynamic memory allocation and efficient **insertion** and **deletion** operations compared to arrays.

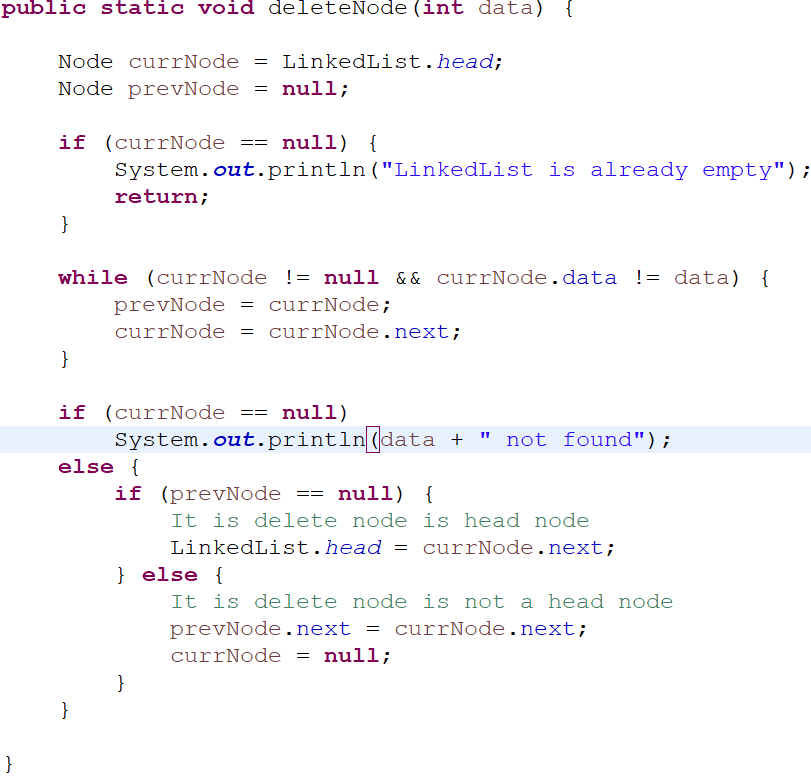
 

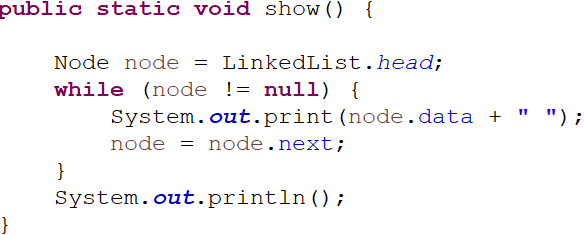
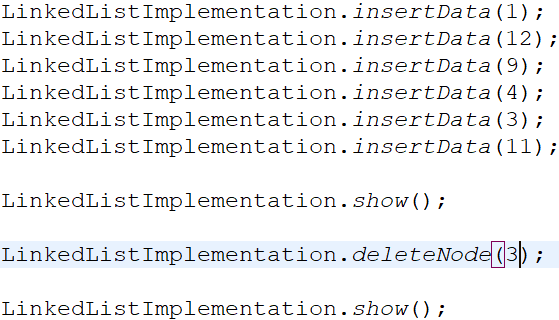


**Singly linked list:**



**Stack:**

A **Stack** is a linear data structure that follows a particular order in which the operations are performed. The order may be **LIFO(Last In First Out)** or **FILO(First In Last Out)**. **LIFO** implies that the element that is inserted last, comes out first and **FILO** implies that the element that is inserted first, comes out last.

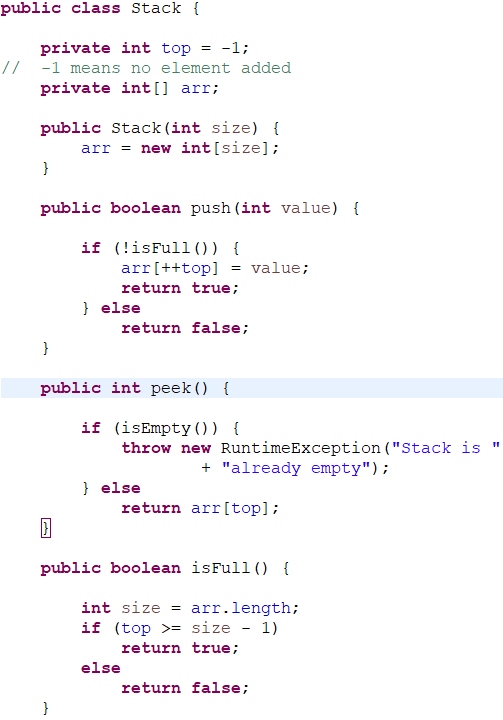
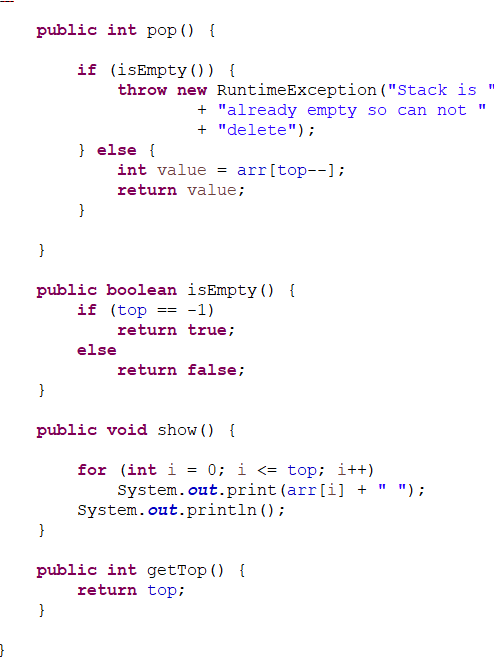
## Key Operations on Stack Data Structures

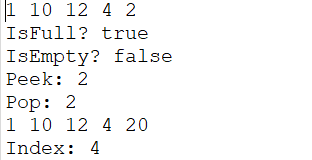
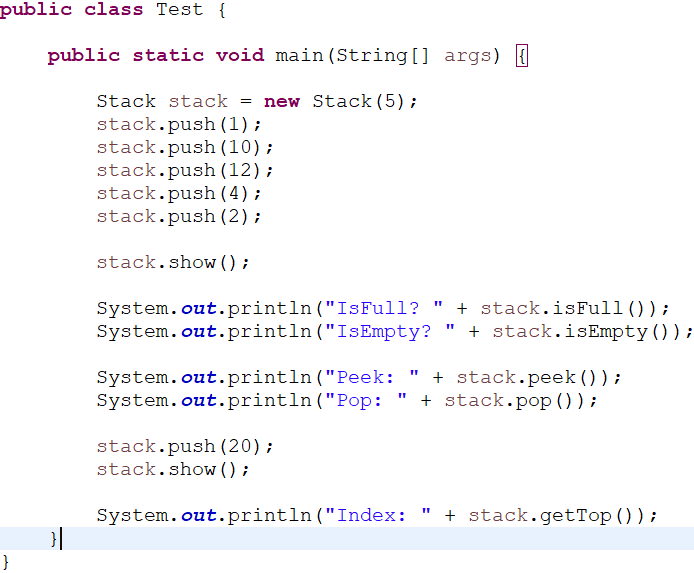
* **Push**: Adds an element to the top of the stack.
* **Pop**: Removes the top element from the stack.
* **Peek**: Returns the top element without removing it.
* **IsEmpty**: Checks if the stack is empty.
* **IsFull**: Checks if the stack is full (in case of fixed-size arrays).

Example:

Recursion, Undo/Redo Operations, Browser History, Function Calls

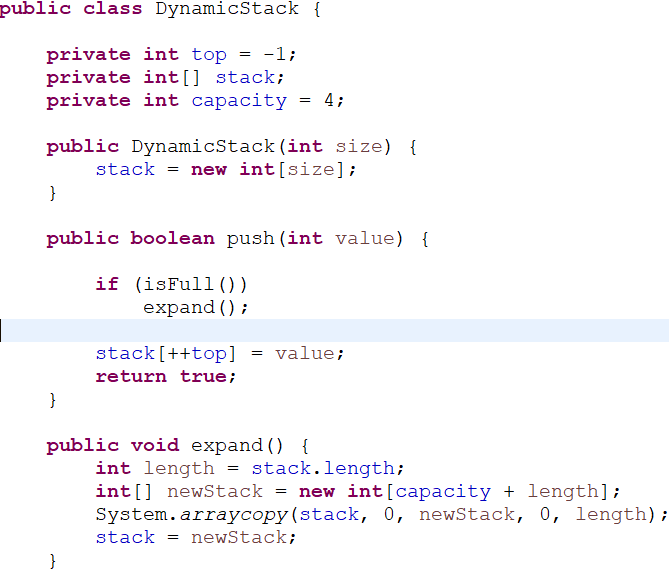
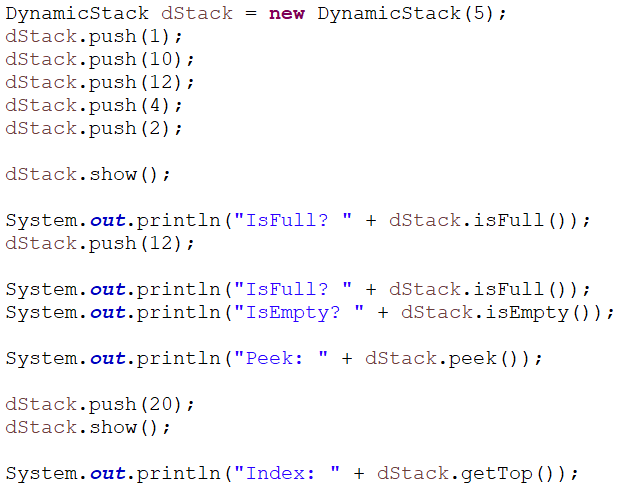
You can implement stack using arrya or linkedlist.

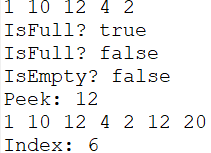


**Dynamic Stack:**

Dynamic stack is same fixed size stach, the difference is that when arr size is full it will create new array of size(capacity + arr.length) and copy all data from existing to new.

**Output::**

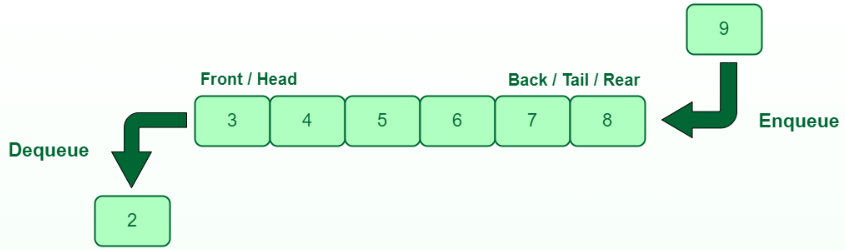


**Queue:**

It is a linear data structure that follows a particular order in which the operations are performed for storing data. The order is First In First Out **(FIFO)**. One can imagine a queue as a line of people waiting to receive something in sequential order which starts from the beginning of the line. It is an ordered list in which insertions are done at one end which is known as the rear and deletions are done from the other end known as the front. A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.

## Basic Operations on Queue:

* **enqueue():** Inserts an element at the end of the queue i.e. at the rear end.
* **dequeue():**This operation removes and returns an element that is at the front end of the queue.
* **front():**This operation returns the element at the front end without removing it.
* **rear():**This operation returns the element at the rear end without removing it.
* **isEmpty():**This operation indicates whether the queue is empty or not.
* **isFull():** This operation indicates whether the queue is full or not.
* **size():**This operation returns the size of the queue i.e. the total number of elements it contains.



The below code will rearrange data when we call dequeue method:

