ADVANCED DATA STRUCTURES AND ALGORITHMS

Course Code: 243MC007	**L**	**T**	**P**	**C**
	2	**0**	**1**	**3**

Course Outcomes:

At the end of the Course, Student will be able to:

- **CO1:** Implement fundamental data structures such as stacks, queues, and matrices
- **CO2:** Apply linked list implementations of stacks and queues to solve problems
- **CO3:** Analyze the asymptotic performance of the algorithm
- **CO4:** Categorize various tree algorithms for efficient data retrieval and manipulation
- **CO5:** Apply greedy and dynamic programming approaches to solve optimization problems

Mapping of Course Outcomes with Program Outcomes:

CO/PO	PO 1	PO 2	PO 3	PO 4	PO 5	PO 6	PO 7	PO 8
CO1	3	-	-	2	-	-	-	-
CO2	3	-	-	2	-	-	-	-
CO3	3	3	-	3	-	-	-	-
CO4	3	2	-	3	-	-	-	-
CO5	3	-	2	3	-	-	-	-

Mapping of Course Outcomes with Program Specific Outcomes:

CO/PSO	PSO1	PSO2	PSO3
CO1	3	-	-
CO2	-	2	2
CO3	-	3	3
CO4	-	-	-
CO5	-	3	3

UNIT – I

Introduction to Data Structures And Data Types: Overview of Arrays**,** Classification of Data Structures, Storage structure of linear array,Implementation of Stacks, Queues and circular queues using arrays, storage representation of matrices and sparsematrices, deques and priority queues

Practice:

1. Implement and analyze the operations of stacks, queues, and circular queues using arrays.
2. Implement and analyze the operations of circular queues using arrays.
3. Implement storage representations of matrices and sparse matrices.

UNIT – II

Stacks, Queues and Circular Queues: Implementation using singly linked, doubly linked list. Applications of stacks – Evaluation of postfix expression, infix to postfix, recursion

Practice:

1. Implement and analyze the operations of stacks using singly and doubly linked lists.
2. Implement and analyze the operations of queues using singly and doubly linked lists.
3. Write a program to evaluate postfix expression

UNIT – III

Notion of Algorithm: Asymptotic Notations, Mathematical Analysis of Non Recursive algorithms: Linear Search, Selection Sort, Bubble Sort, Insertion Sort

Mathematical Analysis of Recursive Algorithms: Factorial of a number, Tower of Hanoi, nth Fibonacci number Divide and Conquer Algorithms: Binary Search, Merge sort, Quicksort.

Practice:

1. Analyze the time complexity of sorting algorithms
2. Analyze the time complexity of recursive algorithms

UNIT – IV

INTRODUCTION TO TREES: Representation of Binary trees using array and list, Binary tree traversals, Binary search trees, Transform & Conquer: AVL tress, 2-3 trees, Heaps and Heap sort

Practice:

1. Implement and analyze operations on Binary Search Trees
2. Implement a Heap data structure and Heap sort algorithm.

UNIT – V

Graph Representations, Decrease & Conquer Algorithms: Graph Traversal Algorithms, Topological Ordering

Greedy Algorithms: To Find Minimum Spanning Tree, Single Source Shortest Path algorithm

Dynamic Programming: Binomial Coefficient, transitive closure and All pair shortest path algorithms

Practice:

1. Implement DFS graph traversal algorithm
2. Implement Kruskal's algorithms for finding Minimum Spanning Tree

Text Books:

1. "Data Structures", Lipschutz Seymour ,Revised First Edition, Schaum's Outline Series, McGraw Hill Education India, ISBN-10**:** 0070605188
2. "Introduction to the Design and Analysis of Algorithms", Anany Levitin, Third Edition, Pearson Education**.** ISBN-10: 0132316817

Reference Books:

1. Data structures-A pseudocode approach with C, Richard F. Gilberg and Behrouz A. Fourouzan, 2nd Edition, 2005,Cengage Learning, ISBN-10**:** 0619216490
2. An Introduction to Data Structures With Application, Jean-Paul Tremblay, Paul G. Sorenson, 2nd Edition, Mcgraw Hill , ISBN-10: 0072943790
3. "Introduction to Algorithms", Thomas H.Cormen, Charles E.Leiserson, Ronald L. Rivest and Clifford Stein, Third Edition, PHI Learning**,** ISBN-10**:** 0262033844

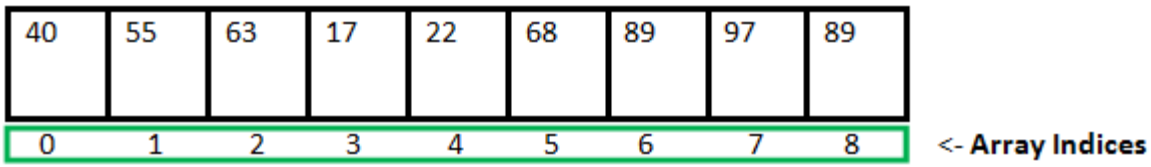
Web Links:

1. http://freevideolectures.com/Course/2519/C-Programming-and-Data-Structures
2. http://www.nptelvideos.in/2012/11/data-structures-and-algorithms.html
3. https://onlinecourses.nptel.ac.in/noc16_cs06/preview
4. http://nptel.ac.in/courses/106101060
5. https://nptel.ac.in/courses/106/106/106106145

UNIT-I

1.1 Arrays:

- An Array is a group of **contiguous or related data items** that share a common name.
- In Java, all arrays are dynamically allocated.
- Arrays may be stored in contiguous memory.
- Since arrays are objects in Java, we can find their length using the object property "length".
- A Java array variable can also be declared like other variables with [] after the data type.
- The variables in the array are ordered, and each has an index beginning with 0.
- The **size** of an array must be specified by int or short value and not long.
- The direct superclass of an array type is Object.
- The size of the array cannot be altered(once initialized).
- Java supports **One Dimensional** Arrays and **Multi Dimensional** Arrays



Array Length = 9
First Index = 0
Last Index = 8

One Dimensional Array : If an array variable have only one subscript then that array is called as One-Dimensional Array.

Declare the Array: The arrays in java may be declared in two forms

Datatype Arrayname [] or Datatype [] Arrayname

Ex : int number[] or int [] number

Creating memory for an Array: After declaring an array variable, we need to create memory. Java allows it by using "new" operator only.

Arrayname = new Datatype[size]

Ex : number = new int[5];

Initialization of array variable: we can initialize the array same as other variables

Type Arrayname []= { list of values }

Ex: int number [] = {34, 25, 67, 1 ,3};

Two Dimensional Array: If an array variable have two subscript then that array is called as Two Dimensional Array.

Datatype Arrayname [][] or Datatype[] [] Arrayname

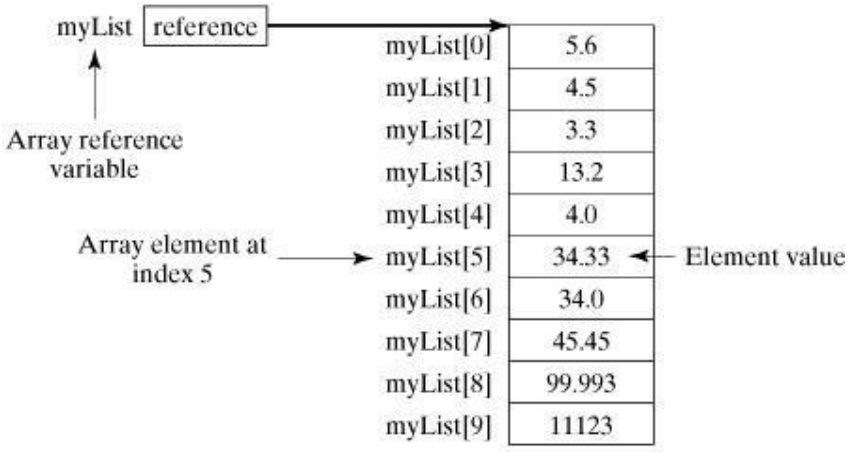
Ex : int number[][] or int [][] number

Example:

Following statement declares an array variable, myList, creates an array of 10 elements of double type and assigns its reference to myList:

```
double[] myList = new double[10];
```

Following picture represents array myList. Here, myList holds ten double values and the indices are from 0 to 9.



Processing Arrays:

When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

Example:

Here is a complete example of showing how to create, initialize and process arrays:

```
public class TestArray {


   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};


      // Print all the array elements

      for (int i = 0; i < myList.length; i++) {
         System.out.println(myList[i] + " ");

      }

      // Summing all elements double
      total = 0;

      for (int i = 0; i < myList.length; i++) {
         total += myList[i];

      }

      System.out.println("Total is " + total);

      // Finding the largest element
      double max = myList[0];

      for (int i = 1; i < myList.length; i++) { if
         (myList[i] > max) max = myList[i];

      }

      System.out.println("Max is " + max);

   }

}
```

The foreach Loops:

JDK 1.5 introduced a new for loop known as foreach loop or enhanced for loop, which enables you to traverse the complete array sequentially without using an index variable.

Example:

The following code displays all the elements in the array myList:

```
public class TestArray {


   public static void main(String[] args) {
      double[] myList = {1.9, 2.9, 3.4, 3.5};


      // Print all the array elements
      for (double element: myList) {

         System.out.println(element);
```

This would produce the following result:

```
1.9

2.9
```

Passing Arrays to Methods:

Just as you can pass primitive type values to methods, you can also pass arrays to methods. For example, the following method displays the elements in an int array:

```
public static void printArray(int[] array) {
  for (int i = 0; i < array.length; i++) {

    System.out.print(array[i] + " ");

  }
```

You can invoke it by passing an array. For example, the following statement invokes the printArray method to display 3, 1, 2, 6, 4, and 2:

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Returning an Array from a Method:

A method may also return an array. For example, the method shown below returns an array that is the reversal of another array:

```
public static int[] reverse(int[] list) {
  int[] result = new int[list.length];


  for (int i = 0, j = result.length - 1; i < list.length; i++, j--) {
    result[j] = list[i];

  }
```

The Arrays Class:

The java.util.Arrays class contains various static methods for sorting and searching arrays, comparing arrays, and filling array elements. The methods enhance array manipulation, contributing to cleaner and more efficient code. Following are the some methods of Array class.

asList()	The asList() method in Java is a static method defined in the Arrays class that converts an array into a fixed-size list. The method returns a list with the original array as its backing. It indicates that any changes to the list will be reflected in the original array as well
binarySearch()	Utilizes the binary search algorithm to efficiently locate a specified value within a sorted array. Returns the index of the target value if found, or a negative value indicating the insertion point if not found.
compare(array 1, array 2)	The compare() method in Java is a static method defined in the Arrays class that compares two arrays of the same data type lexicographically.
copyOf(originalArray, newLength)	The copyOf method in java.util.Arrays duplicates an array, adjusting its length to a specified size by truncating or padding with default values as necessary.
copyOfRange(originalArray, fromIndex, endIndex)	The copyOfRange(originalArray, fromIndex, toIndex) method in java.util.Arrays creates a new array by copying elements from the original array within the specified range, starting from fromIndex (inclusive) and ending at toIndex (exclusive).
equals(array1, array2)	The Arrays.equals() method in Java checks if two arrays, whether containing objects or primitive types, are equal. It looks at each pair of elements in the arrays, considering them equal only if the arrays have the same length and each pair of elements matches
sort(originalArray)	The sort(originalArray) method in java.util.Arrays performs an ascending-order sort on the entire array (originalArray).

Example program for methods of Array class.

```
import java.util.Arrays;
import java.util.Comparator;
public class ArrayMethodsExample
{
   public static void main(String[] args)
   {
        // Method 1: asList()
      Integer[] array = {1, 4, 7, 10, 13};
      System.out.println("Original Array: " + Arrays.asList(array));
        // Method 2: binarySearch()
      int keyToSearch = 7;
      int index = Arrays.binarySearch(array, keyToSearch);
      System.out.println("Index of " + keyToSearch + ": " + index);
        // Method 3: compare(array1, array2)
      Integer[] array1 = {1, 2, 3};
```

```
        Integer[] array2 = {1, 2, 3};
        int comparisonResult = Arrays.compare(array1, array2);
        System.out.println("Comparison result between array1 and array2: " + comparisonResult);
        int[] originalArray = {3, 8, 4, 1, 5};
          // Method 4: copyOf(originalArray, newLength)
        int newLength = 7;
        int[] copyArray = Arrays.copyOf(originalArray, newLength);
        System.out.println("Copy of originalArray with new length: " + Arrays.toString(copyArray));
          // Method 5: copyOfRange(originalArray, fromIndex, endIndex)
        int fromIndex = 2;
        int toIndex = 4;
        int[] copyRangeArray = Arrays.copyOfRange(originalArray, fromIndex, toIndex);
        System.out.println("Copy of originalArray from index " + fromIndex + " to " + toIndex + ": " +
Arrays.toString(copyRangeArray));
          // Method 6: sort(originalArray)
            Arrays.sort(originalArray);
        System.out.println("Sorted array using sort(): " + Arrays.toString(originalArray));
         // Method 7: sort(originalArray, fromIndex, endIndex)
        Arrays.sort(originalArray, fromIndex, toIndex);
        System.out.println("Partially sorted array from index " + fromIndex + " to " + toIndex + ": " +
Arrays.toString(originalArray));
    }
}
```

Output:

```
F:\>java ArrayMethodsExample
Original Array: [1, 4, 7, 10, 13]
Index of 7: 2
Comparison result between array1 and array2: 0
Copy of originalArray with new length: [3, 8, 4, 1, 5, 0, 0]
Copy of originalArray from index 2 to 4: [4, 1]
Sorted array using sort(): [1, 3, 4, 5, 8]
Partially sorted array from index 2 to 4: [1, 3, 4, 5, 8]
```

1.2 Classification of Data Structures:

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.

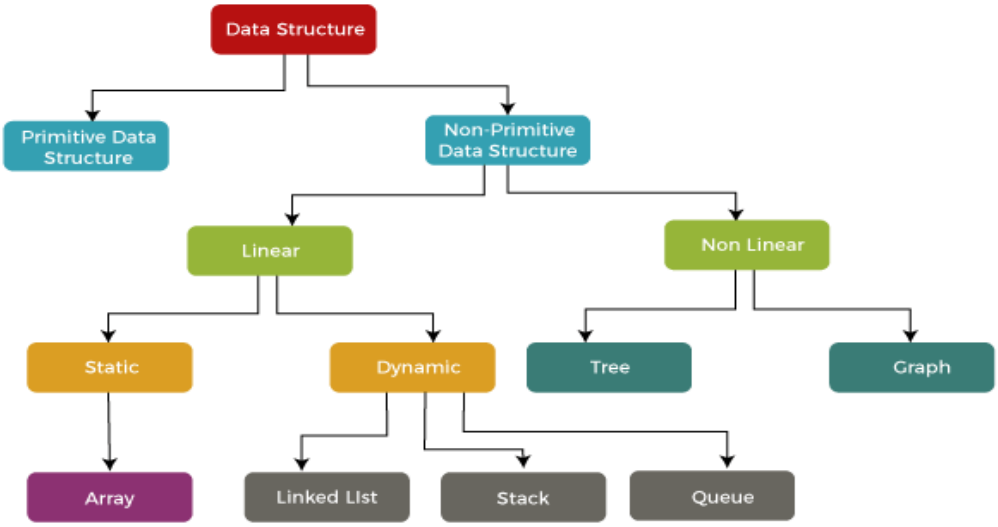
Primitive data structure:

Primitive data structure is a data structure that can hold a single value in a specific location.

The examples of primitive data structure are float, character, integer and pointer. The value to the primitive data structure is provided by the programmer.

Non-primitive data structure:

The non-primitive data structure is a kind of data structure that can hold multiple values either in a contiguous or random location. The non-primitive data structures are defined by the programmer. The non-primitive data structure is again classified into two categories, i.e., linear and non-linear data structure.



I. Linear data structure: Data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements is called a linear data structure.

Examples: Array, Stack, Queue, Linked List, etc.

1. Static data structure: Static data structure has a fixed memory size. It is easier to access the elements in a static data structure. Ex: Array.

Arrays: An array is a linear data structure and it is a collection of items stored at contiguous memory locations.

2. Dynamic data structure: In the dynamic data structure, the size is not fixed. It can be randomly updated during the runtime.

Examples: Linked List, Queue, Stack.

Linked list:

A linked list is a linear data structure in which elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:

Types of linked lists:

- Single-linked list
- Double linked list
- Circular linked list
- Doubly circular linked list

Stack:

Stack is a linear data structure that follows a particular order in which the operations are performed. The order is Last In First Out(LIFO). Entering and retrieving data is possible from only one end. The entering and retrieving of data is also called push and pop operation in a stack.

Queue:

Queue is a linear data structure that follows a particular order in which the operations are performed. The order is First In First Out(FIFO) i.e. the data item stored first will be accessed first. In this, entering and retrieving data is not done from only one end.

II. Non-linear data structure: Data structures where data elements are not placed sequentially or linearly are called non-linear data structures. In a non-linear data structure, we can't traverse all the elements in a single run only. Examples of non-linear data structures are trees and graphs.

Tree:

A tree is a non-linear and hierarchal data structure where the elements are arranged in a tree-like structure. In a tree, the topmost node is called the root node. Each node contains some data, and data can be of any type. There are different types of Tree-like

Binary Tree, Binary Search Tree, AVL Tree, B-Tree, etc.

Graph:

A graph is a non-linear data structure that consists of vertices (or nodes) and edges. It consists of a finite set of vertices and set of edges that connect a pair of nodes. The graph is used to solve the most challenging and complex programming problems.

1.3 Operations performed on the Data Structures:

There are different types of operations that can be performed for the manipulation of data in every data structure. Some operations are explained and illustrated below:

1. Traversing: Traversing a Data Structure means to visit the element stored in it. It visits data in a systematic manner. This can be done with any type of DS.

2. Searching: Searching means to find a particular element in the given data-structure. It is considered as successful when the required element is found. Searching is the operation which we can performed on data-structures like array, linked-list, tree, graph, etc.

3. Insertion: It is the operation which we apply on all the data-structures. Insertion means to add an element in the given data structure. The operation of insertion is successful when the required element is added to the required data-structure. It is unsuccessful in some cases when the size of the data structure is full and when there is no space in the data-structure to add any additional element. The insertion has the same name as an insertion in the data-structure as an array, linked-list, graph, tree. In stack, this operation is called Push. In the queue, this operation is called Enqueue.

4. Deletion: It is the operation which we apply on all the data-structures. Deletion means to delete an element in the given data structure. The operation of deletion is successful when the required element is deleted from the data structure. The deletion has the same name as a deletion in the data-structure as an array, linked-list, graph, tree, etc. In stack, this operation is called Pop. In Queue this operation is called Dequeue.

5. Sorting: The process of arranging the data elements in a data structure in a specific order (ascending or descending) by specific key values is called sorting.

1.4 Storage Structure of Linear Array:

A linear array in data structures is stored as a contiguous block of memory locations, meaning all elements are placed next to each other in memory, allowing for direct access to any element using its index number, making it a linear data structure; each element in the array occupies the same amount of space and can be easily accessed based on its position within the array.

Key points about linear array storage:

- **Contiguous memory allocation:** All elements of an array are stored in consecutive memory locations.
- **Index-based access:** Each element in the array can be accessed using a unique index number, starting from 0.
- **Homogeneous data type:** Typically, all elements in an array are of the same data type (e.g., integers, characters).

1. Memory Representation

- A linear array is stored in **sequential memory locations**.
- The base address of the array (starting memory address) determines the location of each element.
- The address of an element is calculated using an indexing formula.

2. Indexing Formula for Address Calculation

The address of an element in a linear array is determined using the formula:

Address(A[i])=Base Address+(i×Size of Data Type)\text{Address}(A[i]) = \text{Base Address} + (i \times \text{Size of Data Type})Address(A[i])=Base Address+(i×Size of Data Type)

Where:

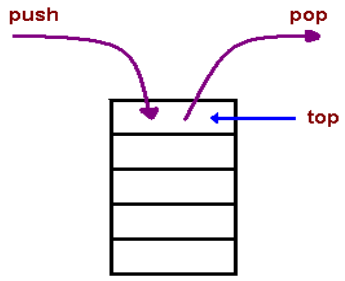
- **Base Address** = Address of the first element A[0]A[0]A[0].
- iii = Index of the element in the array.
- **Size of Data Type** = Number of bytes required for each element.

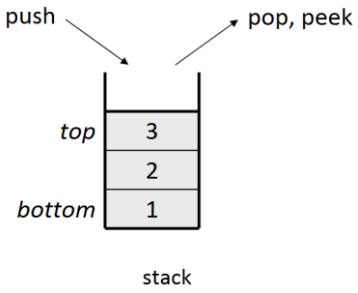
For example, in a **4-byte integer array**, if the base address is **1000**, the elements will be stored at:

- A[0]A[0]A[0] → Address = 1000
- A[1]A[1]A[1] → Address = 1000 + (1 × 4) = 1004
- A[2]A[2]A[2] → Address = 1000 + (2 × 4) = 1008
- And so on...

1.5 Stack:

- It is a linear Data Structure, that organizes the data in LIFO(Last-In-First-Out) (or) FILO(First-In-Last-Out) order.
- It is an ordered list in which insertions & deletions are done at one end called 'top' of the stack.
- The representation of stack is





- **Stack Implementation Methods:**

1. **Using Arrays** – Fixed size, efficient but may lead to overflow.
2. **Using Linked List** – Dynamic size, requires extra memory for pointers.

Applications:

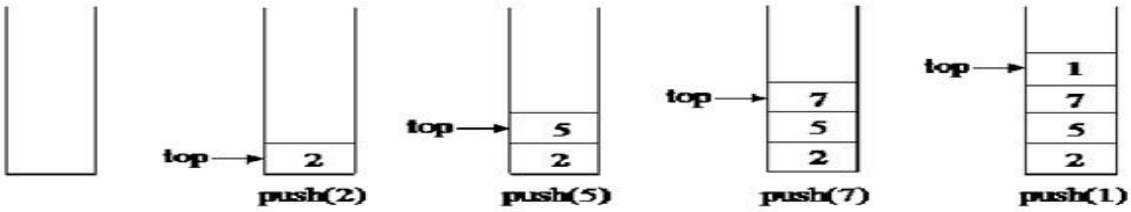
- **Function Call Management** – Keeps track of function calls (recursion).
- **Expression Evaluation** – Used in **infix to postfix conversion** and evaluation.
- **Undo/Redo Operations** – In text editors and applications.
- **Backtracking** – Used in solving mazes, puzzles, etc.
- **Browser History Navigation** – Storing visited web pages.
- **Syntax Parsing** – In compilers for checking balanced parentheses

Algorithm for Stack operations by using Arrays:

Algorithm for push():

The process of adding a new element onto a stack is called as push operation.

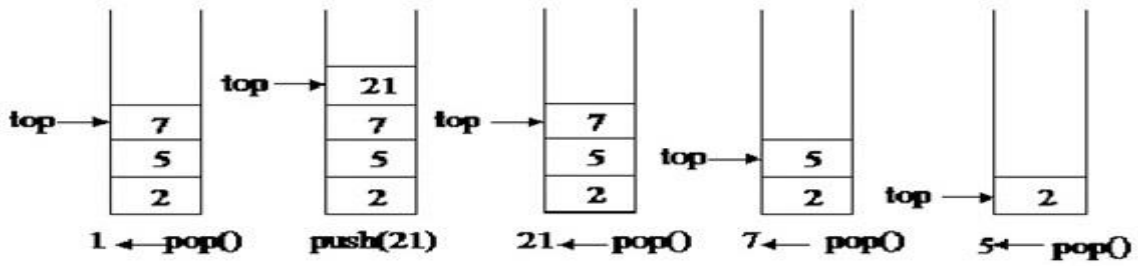
1. Check if the stack is full
   If top == size - 1, print "Stack Overflow! Cannot push val" and return.
2. Increment top by 1
   top = top + 1
3. Insert the new element at top position
   array[top] = val
4. Print confirmation message
   "Pushed val onto the stack."
5. End



Algorithm for pop():

The process of removing an element from the stack is called as pop.

1. Check if the stack is empty
   If top < 0, print "Stack Underflow! Cannot pop." and return.
2. Print the popped element
   "Popped element: array[top]"
3. Decrement top by 1
   top = top - 1
4. End



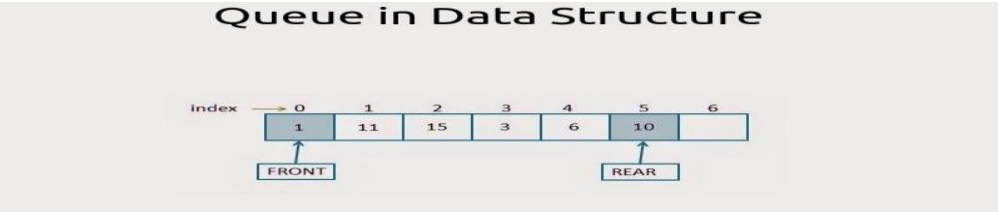
Algorithm for display:

In order to display the element in a stack, it should be traversed from 'top' of the stack to end.

1. Check if the stack is empty
   If top < 0, print "Stack is empty." and return.
2. Loop from top to 0
   Print each element in array[i]
3. End

1.6 Queue:

A **queue** is a linear data structure that follows the **First In, First Out (FIFO)** principle, meaning the first element added is the first one to be removed.



Key Characteristics of Queue:

1. **FIFO Principle** – The first element inserted is removed first.
2. **Two End Operations** – Insertions occur at the **rear**, and deletions occur at the **front**.
3. **Sequential Access** – Unlike stacks, both ends of the queue are used.

Queue Implementation Methods:

1. **Using Arrays** – Fixed size, may lead to space wastage in a simple queue.
2. **Using Linked List** – Dynamic size, requires extra memory for pointers.

Applications of Queue:

1. **CPU Scheduling** – Jobs are scheduled in order.
2. **Printer Queue** – Tasks are processed in sequence.
3. **Call Center Systems** – Customers are served in order.
4. **Traffic Management** – Vehicles at toll booths follow FIFO.
5. **Data Buffers** – Used in I/O operations, network packet handling.

Algorithm for Queue operations by using Arrays:

1. Enqueue Operation

1. If rear == size - 1, print "Queue Overflow" and return.
2. If front == -1, set front = 0.
3. Increment rear.
4. Insert the value at array[rear].
5. Print "Enqueued successfully".

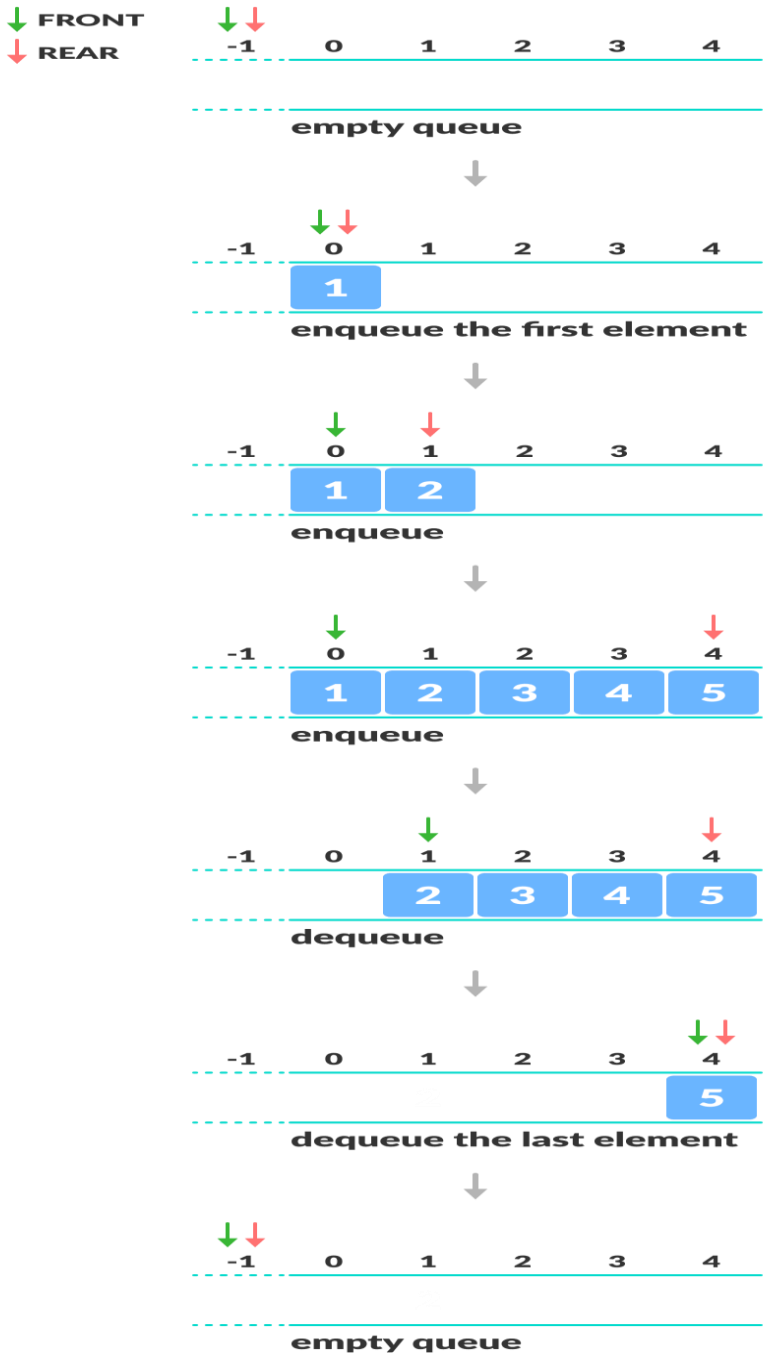
2. Dequeue Operation

1. If front == -1 or front > rear, print "Queue Underflow" and return.
2. Print "Dequeued element: array[front]".
3. Increment front.

3. Display Operation

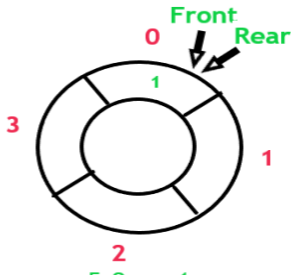
1. If front == -1 or front > rear, print "Queue is empty" and return.
2. Loop from front to rear and print array[front].

Example:



1.7 Circular Queue:

- A **Circular Queue** is a linear data structure where the last position is connected back to the first position to form a circle. This eliminates the issue of unused spaces in a simple queue.
- It is also performed based on FIFO (First In First Out) order.
- It is implemented to overcome the problem of queue (normal queue) we can insert elements until queue becomes full.
- After insert all the elements in a queue first time, now deleting two elements from the queue, and then if we try to insert new elements in the empty position then it says queue is full because rear is at last position. So we can't insert new elements in an empty space.



Algorithms for Circular Queue Operations by using Arrays:

1. Enqueue Operation:

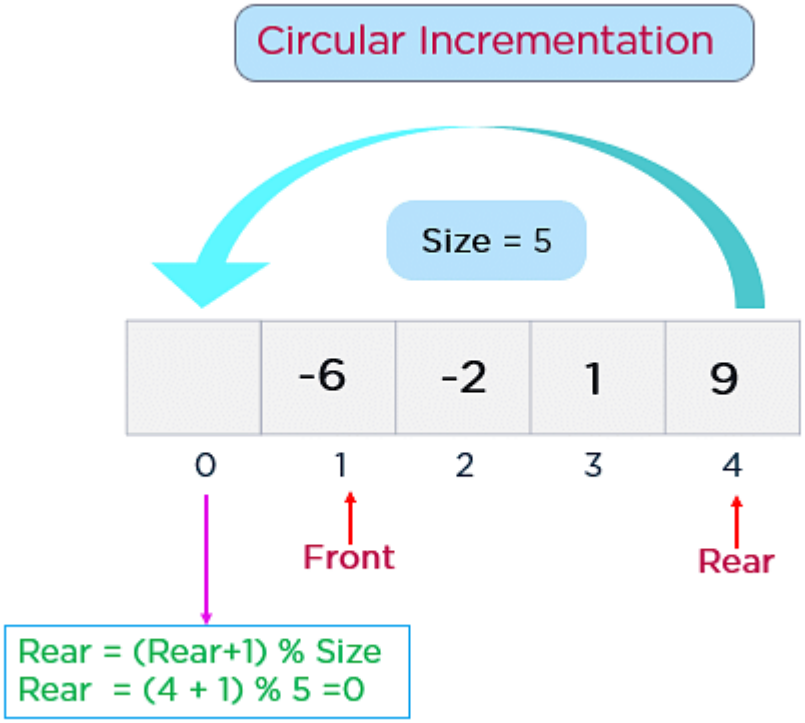
1. Check if the queue is full: (rear + 1) % size == front. If true, display "Queue Overflow" and return.
2. If the queue is empty (front == -1), set front = 0.
3. Update rear as (rear + 1) % size.
4. Insert the new element at queue[rear].
5. Print confirmation message.

2. Dequeue Operation:

1. Check if the queue is empty (front == -1). If true, display "Queue Underflow" and return.
2. Print the dequeued element queue[front].
3. If front == rear, set front = rear = -1 (queue becomes empty).
4. Otherwise, update front = (front + 1) % size.

3. Display Operation:

1. Check if the queue is empty (front == -1). If true, display "Queue is empty" and return.
2. Start from front and iterate through the queue using (i + 1) % size until i == rear.
3. Print each element during iteration.
4. Print the last element at queue[rear].



1.8 Sparce Matrix:

A sparse matrix is a matrix where most of the elements are zero. These matrices are useful in many real-world applications like graph algorithms, image processing, and scientific computing, where storing all elements explicitly would be inefficient.

Why to use Sparse Matrix instead of simple matrix ?

Storage: There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.

Computing time: Computing time can be saved by logically designing a data structure traversing only non-zero elements.

Example:

0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with triples- (Row, Column, value).

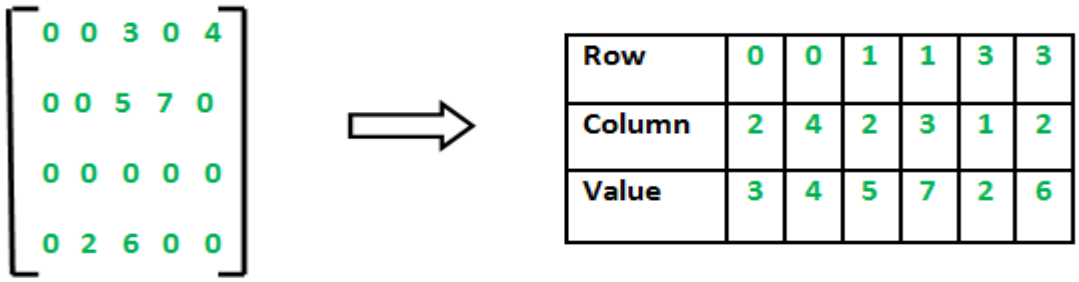
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays:

2D array is used to represent a sparse matrix in which there are three rows named as

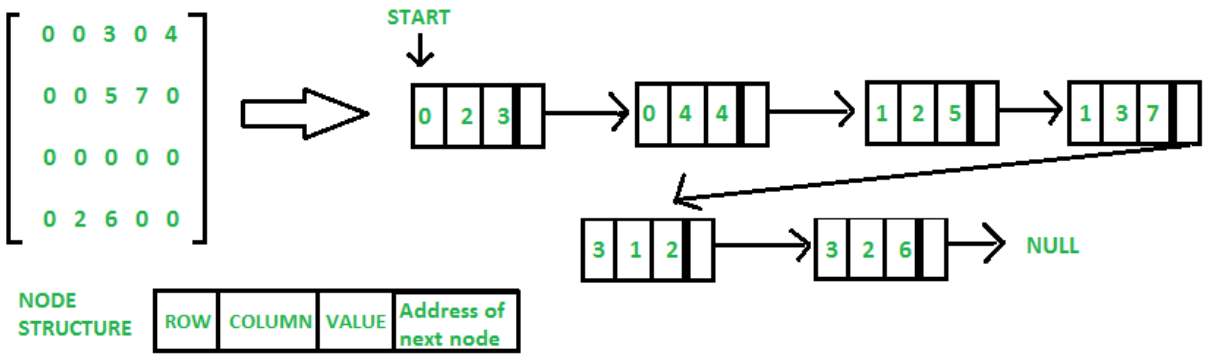
- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



Method 2: Using Linked Lists

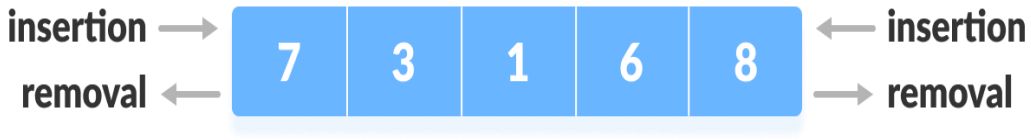
In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



1.9 Dequeue:

- A deque is a data structure that allows for the insertion and deletion of elements from both ends. It's also known as a double-ended queue.
- Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear.
- Deque supports both stack and queue operations, it can be used as both.
- A deque can function in both FIFO (First-In-First-Out) and LIFO (Last-In-First-Out) modes.



Applications of Deque:

- Applied as both stack and queue, as it supports both operations.
- Storing a web browser's history.
- Storing a software application's list of undo operations.
- Sliding Window Problems (e.g., Maximum of all subarrays of size k).
- Job Scheduling (both FIFO and LIFO are needed).
- Undo/Redo Functionality in text editors.
- Palindrome Checking (compare front and rear characters).
- Graph Algorithms (BFS and shortest path calculations).
- Expression Evaluation (used in parsers)

Types of Deque

Input Restricted Deque: In this deque, input is restricted at a single end but allows deletion at both the ends.

Output Restricted Deque: In this deque, output is restricted at a single end but allows insertion at both the ends.

Operations on Deque

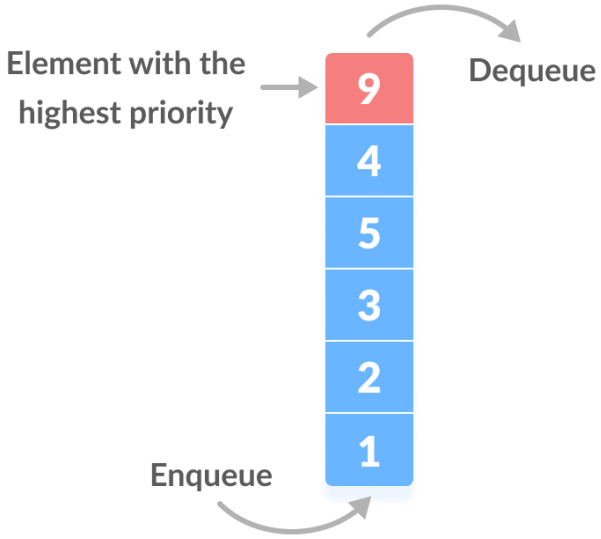
Operation	Description	Time Complexity
insertFront()	Insert element at the front	**O(1)**
insertRear()	Insert element at the rear	**O(1)**
deleteFront()	Remove element from the front	**O(1)**
deleteRear()	Remove element from the rear	**O(1)**
getFront()	Get front element	**O(1)**
getRear()	Get rear element	**O(1)**
isEmpty()	Check if deque is empty	**O(1)**
isFull()	Check if deque is full (in fixed-size implementations)	**O(1)**

1.10 Priority queue:

- A priority queue is a **special type of queue** in which each element is associated with a **priority value**. And, elements are served on the basis of their priority. That is, higher priority elements are served first.
- However, if elements with the same priority occur, they are served according to their order in the queue.
- The hospital emergency queue is an ideal real-life example of a priority queue. In this queue of patients, the patient with the most critical situation is the first in a queue, and the patient who doesn't need immediate medical attention will be the last. In this queue, the priority depends on the medical condition of the patients.

Assigning Priority Value:

Generally, the value of the element itself is considered for assigning the priority. For example,
The element with the highest value is considered the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element.
We can also set priorities according to our needs.



Difference between Priority Queue and Normal Queue:

In a queue, the first-in-first-out rule is implemented whereas, in a priority queue, the values are removed on the basis of priority. The element with the highest priority is removed first.

Implementation of Priority Queue:

Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree. Among these data structures, heap data structure provides an efficient implementation of priority queues.

Operations of a Priority Queue:

A typical priority queue supports the following operations:

1) Insertion : If the newly inserted item is of the highest priority, then it is inserted at the top. Otherwise, it is inserted in such a way that it is accessible after all higher priority items are accessed.

2) Deletion in a Priority Queue : We typically remove the highest priority item which is typically available at the top. Once we remove this item, we need mot move next priority item at the top.

3) Peek in a Priority Queue : This operation only returns the highest priority item (which is typically available at the top) and does not make any change to the priority queue.