# Algorithms and Data Structures

Tushar B. Kute,
http://tusharkute.com

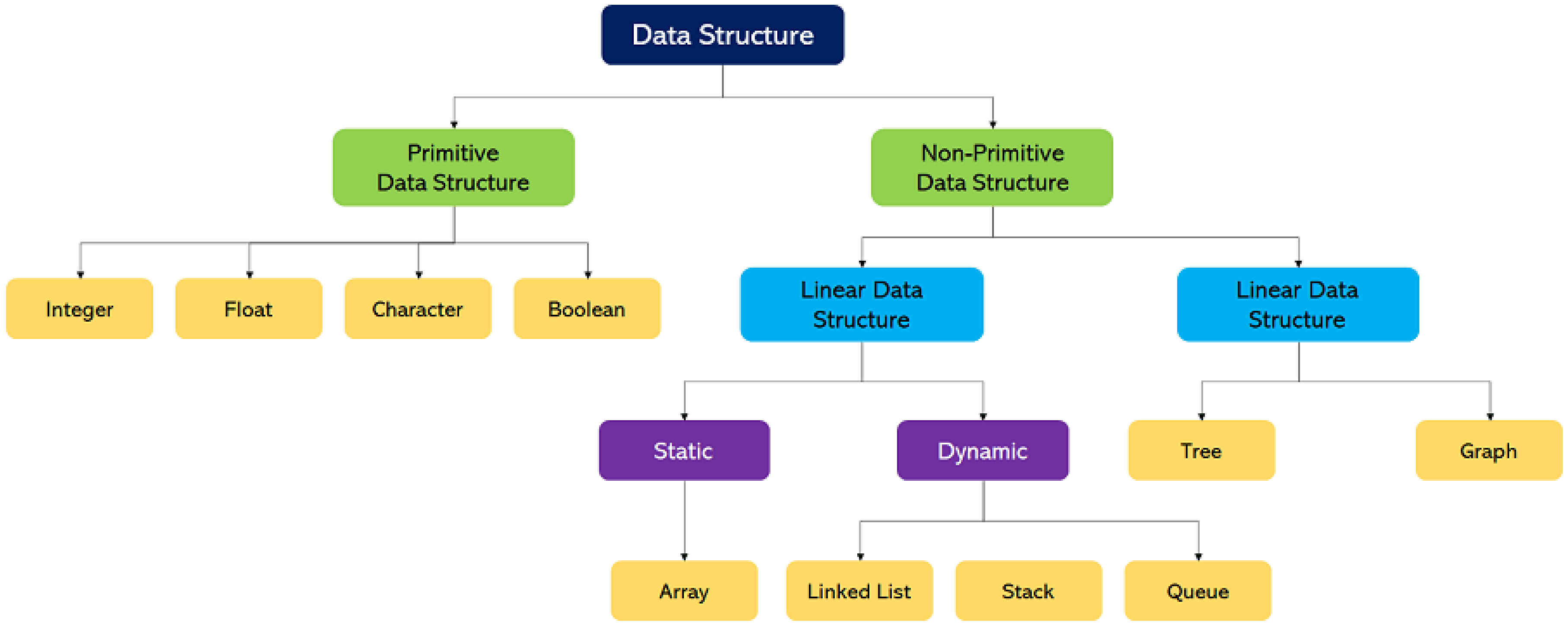
## Data Structure

- The data structure name indicates itself that organizing the data in memory.
- There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language.
- Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner.
- This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory.

## Data Structure

- Primitive Data structure
  - The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.
- Non-Primitive Data structure
  - The non-primitive data structure is divided into two types:
    - Linear data structure
    - Non-linear data structure

# Data Structure



## Algorithm

- An algorithm is a process or a set of rules required to perform calculations or some other problem-solving operations especially by a computer.
- The formal definition of an algorithm is that it contains the finite set of instructions which are being carried in a specific order to perform the specific task.
- It is not the complete program or code; it is just a solution (logic) of a problem, which can be represented either as an informal description using a Flowchart or Pseudocode.

## Algorithm : Characteristics

- Input: An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- Output: We will get 1 or more output at the end of an algorithm.
- Unambiguity: An algorithm should be unambiguous which means that the instructions in an algorithm should be clear and simple.
- Finiteness: An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- Effectiveness: An algorithm should be effective as each instruction in an algorithm affects the overall process.
- Language independent: An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

# Algorithm : Dataflow

- Problem: A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions. The set of instructions is known as an algorithm.
- Algorithm: An algorithm will be designed for a problem which is a step by step procedure.
- Input: After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- Processing unit: The input will be given to the processing unit, and the processing unit will produce the desired output.
- Output: The output is the outcome or the result of the program.

## Algorithm : Categories

- The major categories of algorithms are given below:
  - Sort: Algorithm developed for sorting the items in a certain order.
  - Search: Algorithm developed for searching the items inside a data structure.
  - Delete: Algorithm developed for deleting the existing element from the data structure.
  - Insert: Algorithm developed for inserting an item inside a data structure.
  - Update: Algorithm developed for updating the existing element inside a data structure.

## Algorithm : Analysis

- Priori Analysis:
  - Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.
- Posterior Analysis:
  - Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

## Time Complexity

- The time complexity of an algorithm is the amount of time required to complete the execution.
- The time complexity of an algorithm is denoted by the big O notation. Here, big O notation is the asymptotic notation to represent the time complexity.
- The time complexity is mainly calculated by counting the number of steps to finish the execution.

## Space Complexity

- Space complexity: An algorithm's space complexity is the amount of space required to solve a problem and produce an output.
- Similar to the time complexity, space complexity is also expressed in big O notation.

## Types of Algorithms

- The following are the types of algorithm:
  - Search Algorithm
  - Sort Algorithm

## Search Algorithms

- On each day, we search for something in our day to day life. Similarly, with the case of computer, huge data is stored in a computer that whenever the user asks for any data then the computer searches for that data in the memory and provides that data to the user.
- There are mainly two techniques available to search the data in an array:
  - Linear search
  - Binary search

## Linear Search

- Linear search is a very simple algorithm that starts searching for an element or a value from the beginning of an array until the required element is not found.
- It compares the element to be searched with all the elements in an array, if the match is found, then it returns the index of the element else it returns -1.
- This algorithm can be implemented on the unsorted list.

# Binary Search

- A Binary algorithm is the simplest algorithm that searches the element very quickly. It is used to search the element from the sorted list.
- The elements must be stored in sequential order or the sorted manner to implement the binary algorithm. Binary search cannot be implemented if the elements are stored in a random manner.
- It is used to find the middle element of the list.

## Sorting Algorithms

- Sorting algorithms are used to rearrange the elements in an array or a given data structure either in an ascending or descending order. The comparison operator decides the new order of the elements.
- Why do we need a sorting algorithm?
  - An efficient sorting algorithm is required for optimizing the efficiency of other algorithms like binary search algorithm as a binary search algorithm requires an array to be sorted in a particular order, mainly in ascending order.
  - It produces information in a sorted order, which is a human-readable format.
  - Searching a particular element in a sorted list is faster than the unsorted list.

## Asymptotic Analysis

- As we know that data structure is a way of organizing the data efficiently and that efficiency is measured either in terms of time or space.
- So, the ideal data structure is a structure that occupies the least possible time to perform all its operation and the memory space.
- Our focus would be on finding the time complexity rather than space complexity, and by finding the time complexity, we can decide which data structure is the best for an algorithm.

## How to calculate complexity?

- How to calculate f(n)?
  - Calculating the value of f(n) for smaller programs is easy but for bigger programs, it's not that easy. We can compare the data structures by comparing their f(n) values.
  - We can compare the data structures by comparing their f(n) values.
  - We will find the growth rate of f(n) because there might be a possibility that one data structure for a smaller input size is better than the other one but not for the larger sizes.

## How to calculate complexity?

- Let's look at a simple example.

  $f(n) = 5n^2 + 6n + 12$

  where n is the number of instructions executed, and it depends on the size of the input.

- When n=1

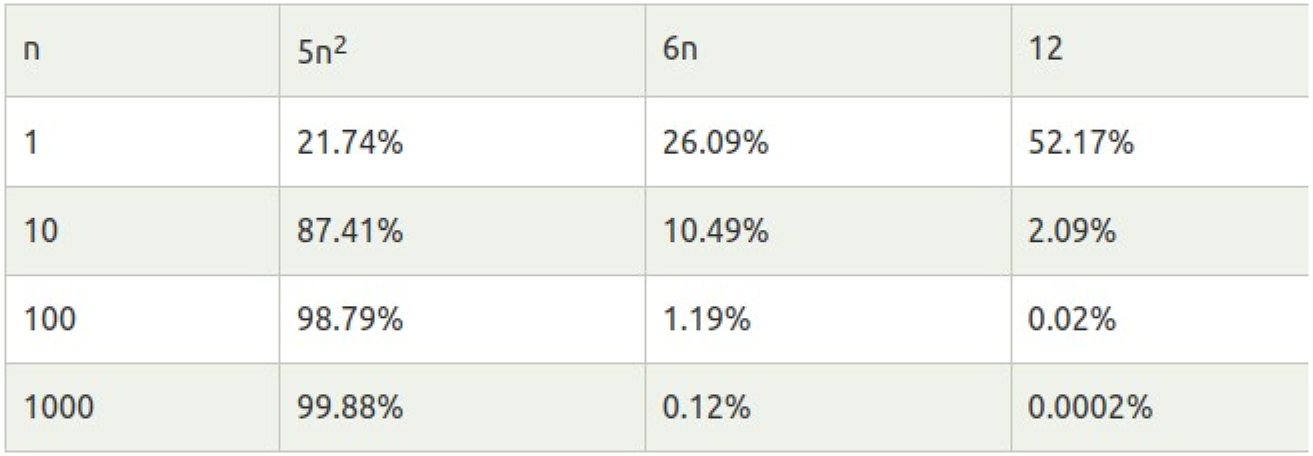
  % of running time due to $5n^2 = \frac{5}{5+6+12} * 100 = 21.74\%$

  % of running time due to $6n = \frac{5}{5+6+12} * 100 = 26.09\%$

  % of running time due to $12 = \frac{5}{5+6+12} * 100 = 52.17\%$

## How to calculate complexity?

- From the above calculation, it is observed that most of the time is taken by 12.
- But, we have to find the growth rate of f(n), we cannot say that the maximum amount of time is taken by 12. Let's assume the different values of n to find the growth rate of f(n).



| n | $5n^2$ | 6n | 12 |
|---|---|---|---|
| 1 | 21.74% | 26.09% | 52.17% |
| 10 | 87.41% | 10.49% | 2.09% |
| 100 | 98.79% | 1.19% | 0.02% |
| 1000 | 99.88% | 0.12% | 0.0002% |

# How to calculate complexity?

- Example: Running time of one operation is x(n) and for another operation, it is calculated as f(n2). It refers to running time will increase linearly with an increase in 'n' for the first operation, and running time will increase exponentially for the second operation. Similarly, the running time of both operations will be the same if n is significantly small.
- Usually, the time required by an algorithm comes under three types:
- Worst case: It defines the input for which the algorithm takes a huge time.
- Average case: It takes average time for the program execution.
- Best case: It defines the input for which the algorithm takes the lowest time

## Asymptotic Notations

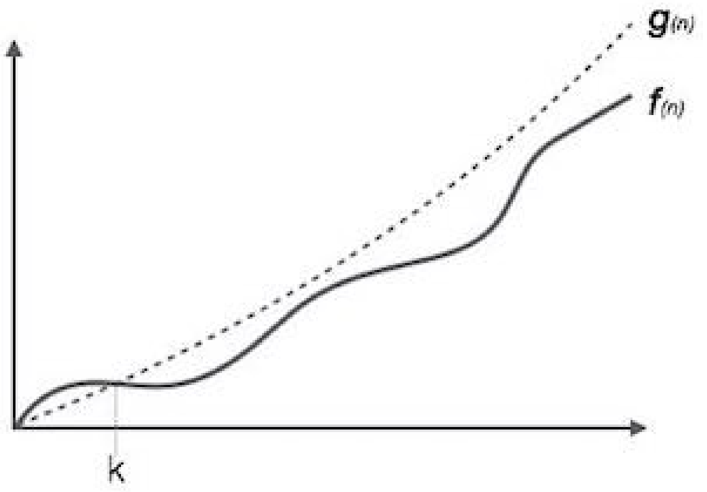
- The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:
  - Big oh Notation (O)
  - Omega Notation (Ω)
  - Theta Notation (θ)

## Big oh Notation (O)

- Big O notation is an asymptotic notation that measures the performance of an algorithm by simply providing the order of growth of the function.
- This notation provides an upper bound on a function which ensures that the function never grows faster than the upper bound.
- So, it gives the least upper bound on a function so that the function never grows faster than this upper bound.

## Big oh Notation (O)

- It is the formal way to express the upper boundary of an algorithm running time. It measures the worst case of time complexity or the algorithm's longest amount of time to complete its operation. It is represented as shown below:



## Big oh Notation (O)

- If f(n) and g(n) are the two functions defined for positive integers,
- then f(n) = O(g(n)) as f(n) is big oh of g(n) or f(n) is on the order of g(n)) if there exists constants c and no such that:

$$f(n) \leq c.g(n) \text{ for all } n \geq no$$

- This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n).
- In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.

## Big oh Notation (O)

- Let's understand through examples
- Example 1: f(n)=2n+3 , g(n)=n

  Now, we have to find Is f(n)=O(g(n))?

- To check f(n)=O(g(n)), it must satisfy the given condition:

$$f(n)<=c.g(n)$$

## Big oh Notation (O)

- First, we will replace f(n) by 2n+3 and g(n) by n.

  2n+3 <= c.n
- Let's assume c=5, n=1 then

  2*1+3<=5*1

  5<=5
- For n=1, the above condition is true.

  If n=2

  2*2+3<=5*2

  7<=10

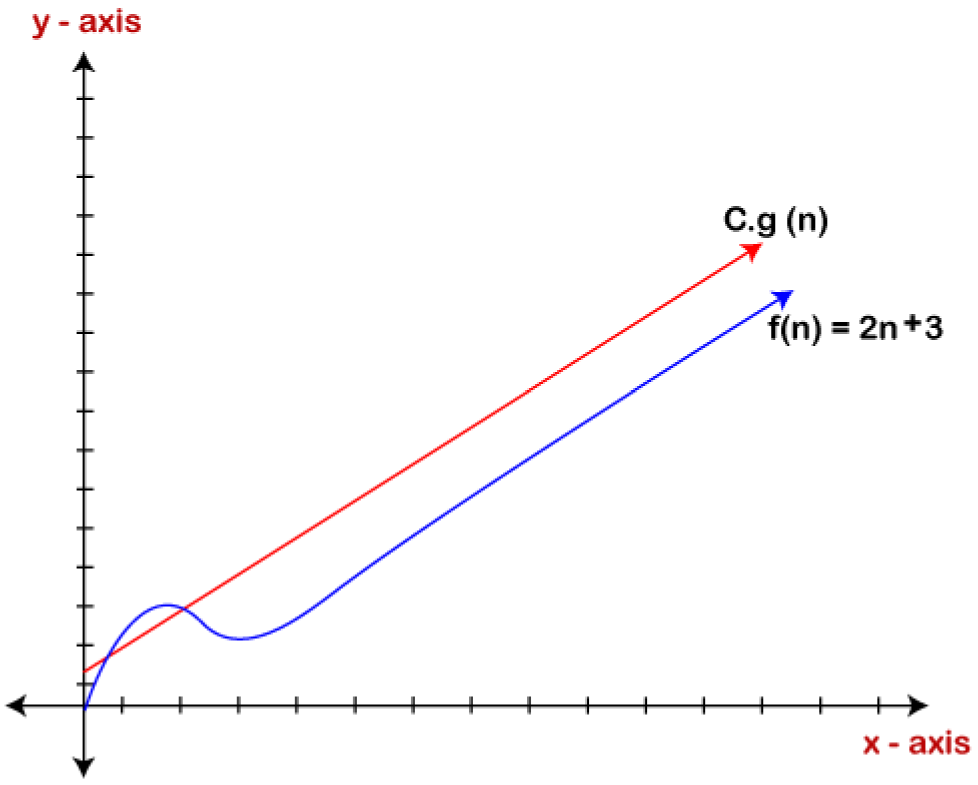
  For n=2, the above condition is true.

## Big oh Notation (O)

- We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n.
- We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n.
- As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly. Therefore, it concludes that c.g(n) is the upper bound of the f(n).

## Big oh Notation (O)

- It can be represented graphically as:



## Big oh Notation (O)

- The idea of using big o notation is to give an upper bound of a particular function, and eventually it leads to give a worst-time complexity.
- It provides an assurance that a particular function does not behave suddenly as a quadratic or a cubic fashion, it just behaves in a linear manner in a worst-case.

## Common Asymptotic Notations

| | | |
|---|---|---|
| constant | - | ?(1) |
| linear | - | ?(n) |
| logarithmic | - | ?(log n) |
| n log n | - | ?(n log n) |
| exponential | - | 2?(n) |
| cubic | - | ?(n3) |
| polynomial | - | n?(1) |
| quadratic | - | ?(n2) |

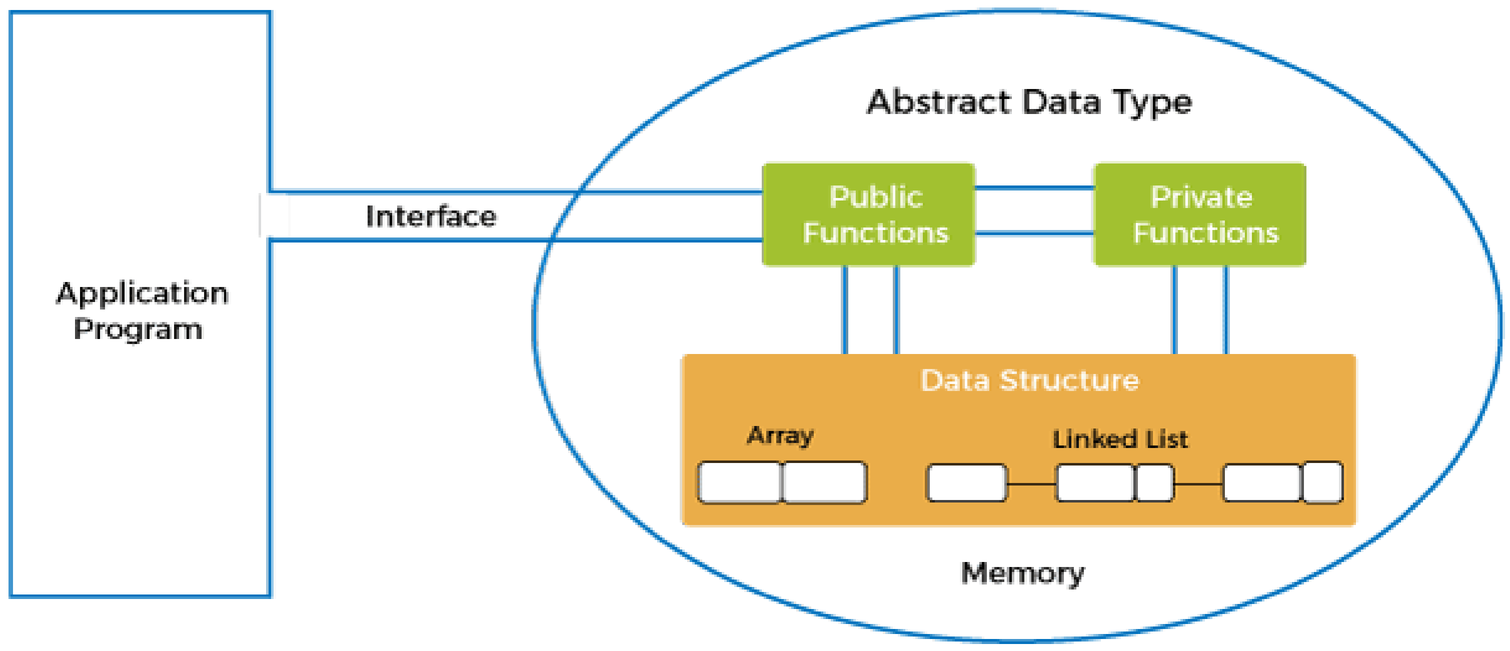
## Abstract Data Type

- An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.
- In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details.

## Abstract Data Type

- Before knowing about the abstract data type model, we should know about abstraction and encapsulation.
- Abstraction: It is a technique of hiding the internal details from the user and only showing the necessary details to the user.
- Encapsulation: It is a technique of combining the data and the member function in a single unit is known as encapsulation.

# Abstract Data Type



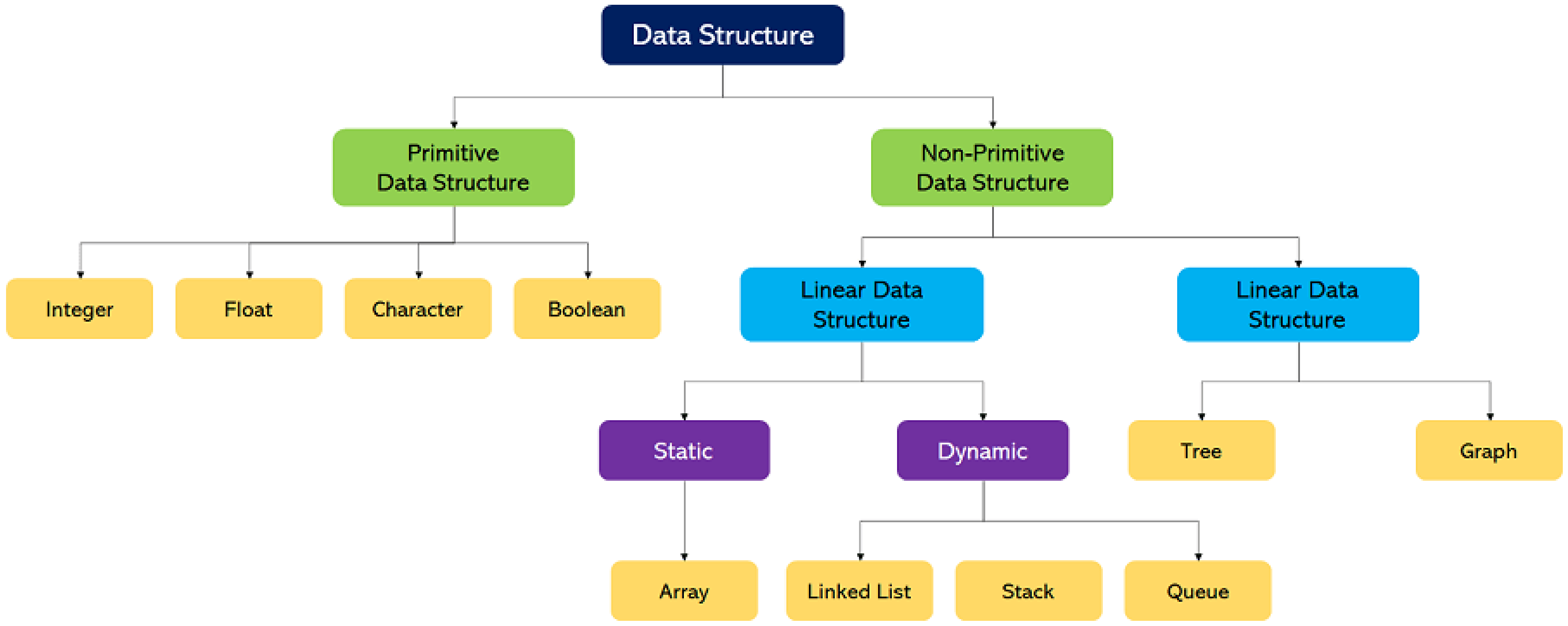
## Abstract Data Type

- Some common examples of ADTs include:
  - List: A list is a collection of ordered elements. Common operations on lists include adding and removing elements, accessing elements by index, and searching for elements.
  - Set: A set is a collection of unordered, unique elements. Common operations on sets include adding and removing elements, checking if an element is in a set, and finding the union and intersection of two sets.

# Abstract Data Type

- Stack: A stack is a data structure that follows the last-in-first-out (LIFO) principle. Elements are added to and removed from the top of the stack. Common operations on stacks include pushing and popping elements, and checking if the stack is empty.
- Queue: A queue is a data structure that follows the first-in-first-out (FIFO) principle. Elements are added to the back of the queue and removed from the front of the queue. Common operations on queues include enqueuing and dequeuing elements, and checking if the queue is empty.

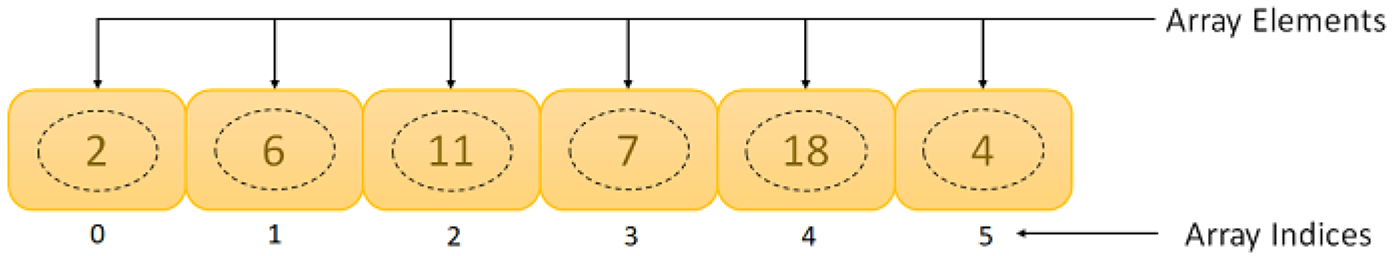
# Data Structure



## Array

- An Array is a data structure used to collect multiple data elements of the same data type into one variable.
- Instead of storing multiple values of the same data types in separate variable names, we could store all of them together into one variable.
- This statement doesn't imply that we will have to unite all the values of the same data type in any program into one array of that data type.
- But there will often be times when some specific variables of the same data types are all related to one another in a way appropriate for an array.

## Array



## Array: Types

- One-Dimensional Array: An Array with only one row of data elements is known as a One-Dimensional Array. It is stored in ascending storage location.
- Two-Dimensional Array: An Array consisting of multiple rows and columns of data elements is called a Two-Dimensional Array. It is also known as a Matrix.
- Multidimensional Array: We can define Multidimensional Array as an Array of Arrays. Multidimensional Arrays are not bounded to two indices or two dimensions as they can include as many indices are per the need.

## Array: Types

- We can store a list of data elements belonging to the same data type.
- Array acts as an auxiliary storage for other data structures.
- The array also helps store data elements of a binary tree of the fixed count.
- Array also acts as a storage of matrices.

## Array: Why?

- Arrays are useful because -
  - Sorting and searching a value in an array is easier.
  - Arrays are best to process multiple values quickly and easily.
  - Arrays are good for storing multiple values in a single variable - In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables.

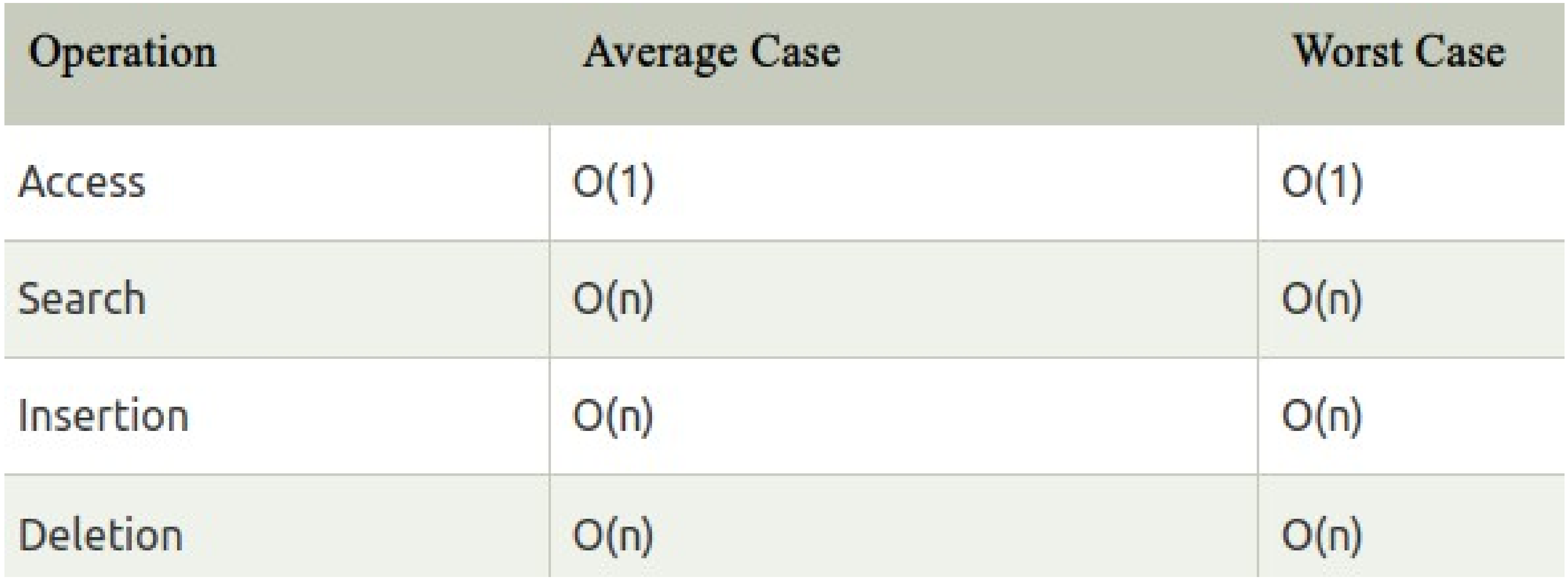
## Array: Memory Allocation

- All the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of the first element in the main memory. Each element of the array is represented by proper indexing.
- We can define the indexing of an array in the below ways -
  - 0 (zero-based indexing): The first element of the array will be arr[0].
  - 1 (one-based indexing): The first element of the array will be arr[1].
  - n (n - based indexing): The first element of the array can reside at any random index number.

## Basic Operations

- Traversal - This operation is used to print the elements of the array.
- Insertion - It is used to add an element at a particular index.
- Deletion - It is used to delete an element from a particular index.
- Search - It is used to search an element using the given index or by the value.
- Update - It updates an element at a particular index.

## Complexity

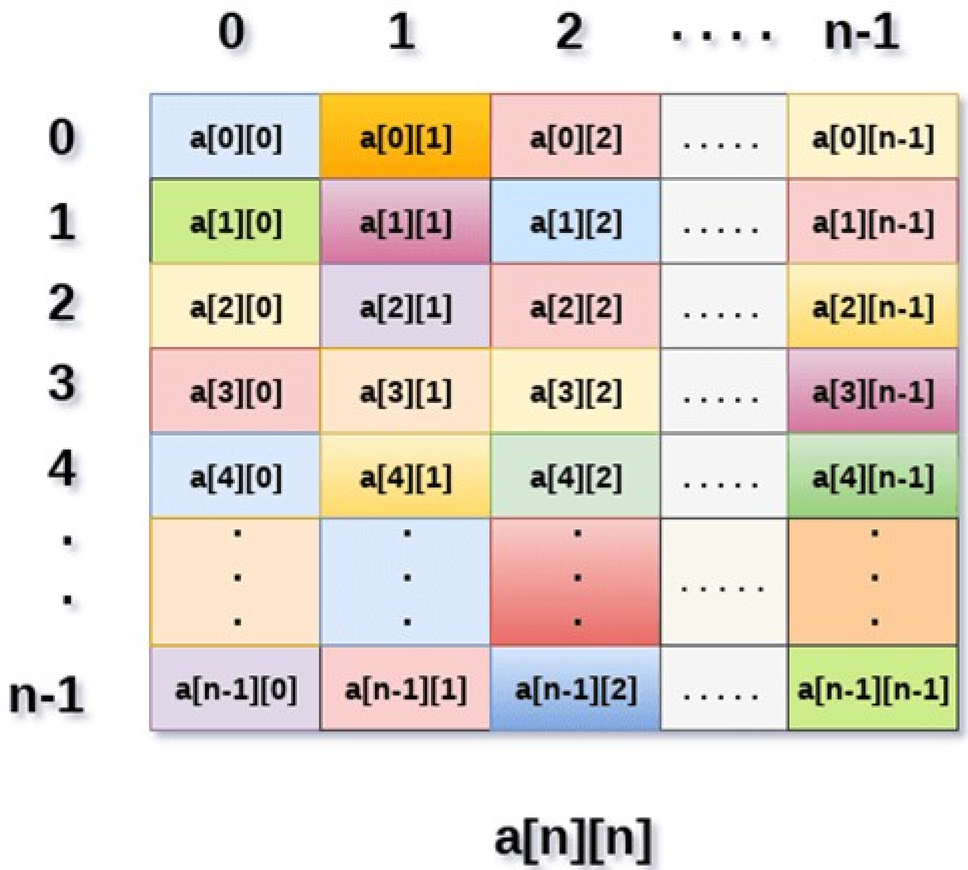


| Operation | Average Case | Worst Case |
|---|---|---|
| Access | O(1) | O(1) |
| Search | O(n) | O(n) |
| Insertion | O(n) | O(n) |
| Deletion | O(n) | O(n) |

## 2D Array

- 2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.
- However, 2D arrays are created to implement a relational database look alike data structure. It provides ease of holding bulk of data at once which can be passed to any number of functions wherever required.

## 2D Array



| | 0 | 1 | 2 | . . . . | n-1 |
|---|---|---|---|---|---|
| 0 | a[0][0] | a[0][1] | a[0][2] | . . . . . | a[0][n-1] |
| 1 | a[1][0] | a[1][1] | a[1][2] | . . . . . | a[1][n-1] |
| 2 | a[2][0] | a[2][1] | a[2][2] | . . . . . | a[2][n-1] |
| 3 | a[3][0] | a[3][1] | a[3][2] | . . . . . | a[3][n-1] |
| 4 | a[4][0] | a[4][1] | a[4][2] | . . . . . | a[4][n-1] |
| ⋮ | ⋮ | ⋮ | ⋮ | . . . . . | ⋮ |
| n-1 | a[n-1][0] | a[n-1][1] | a[n-1][2] | . . . . . | a[n-1][n-1] |

a[n][n]

## 2D Array

- We can assign each cell of a 2D array to 0 by using the following code:

```
for ( int i=0; i<n ;i++)
{
    for (int j=0; j<n; j++)
    {
        a[i][j] = 0;
    }
}
```

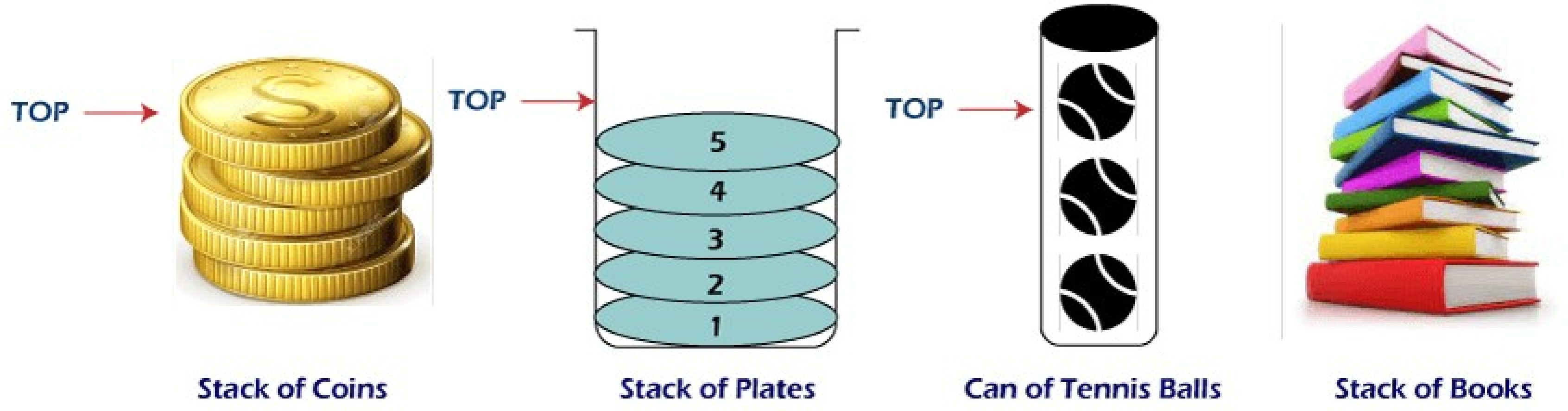
## Stack

- A Stack is a linear data structure that follows the LIFO (Last-In-First-Out) principle. Stack has one end, whereas the Queue has two ends (front and rear).
- It contains only one pointer top pointer pointing to the topmost element of the stack. Whenever an element is added in the stack, it is added on the top of the stack, and the element can be deleted only from the stack.
- In other words, a stack can be defined as a container in which insertion and deletion can be done from the one end known as the top of the stack.

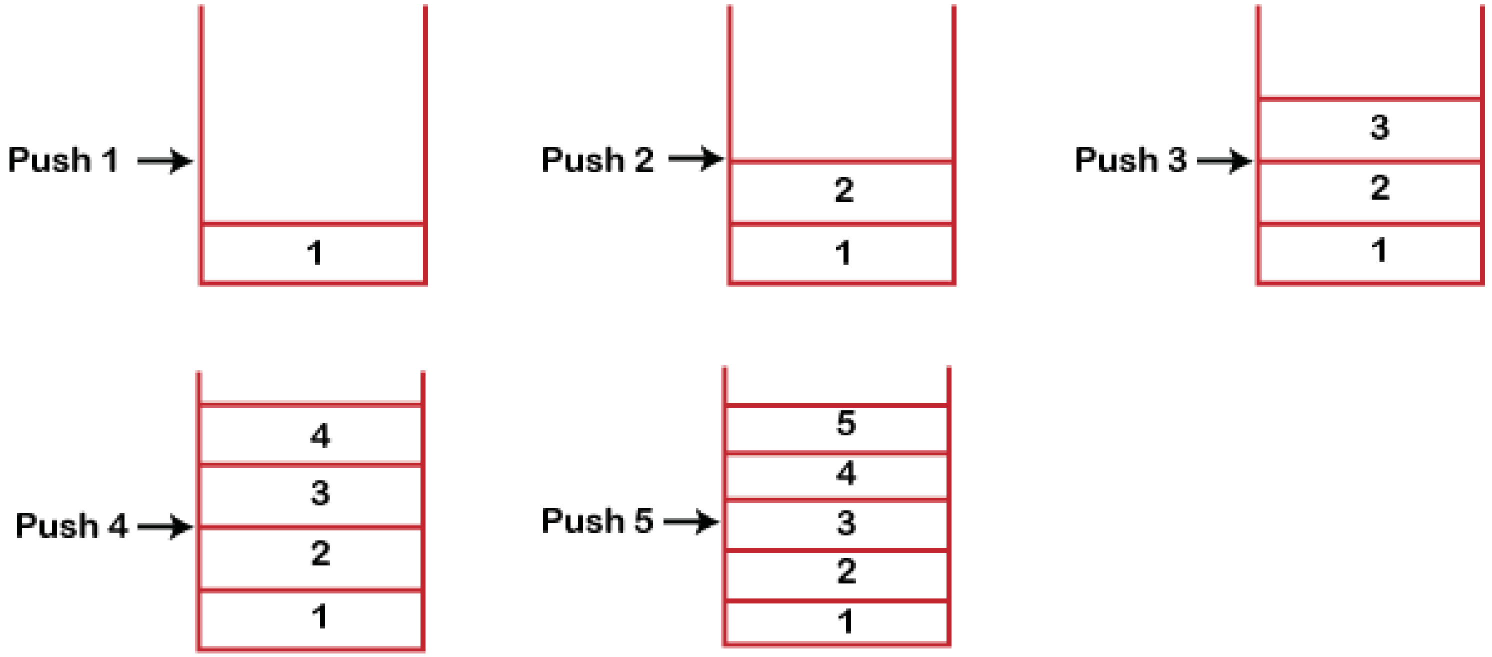
## Stack

- It is called as stack because it behaves like a real-world stack, piles of books, etc.
- A Stack is an abstract data type with a pre-defined capacity, which means that it can store the elements of a limited size.
- It is a data structure that follows some order to insert and delete the elements, and that order can be LIFO or FILO.

## Stack


TOP
TOP
5
4
3
2
1
TOP
Stack of Coins
Stack of Plates
Can of Tennis Balls
Stack of Books

## Stack



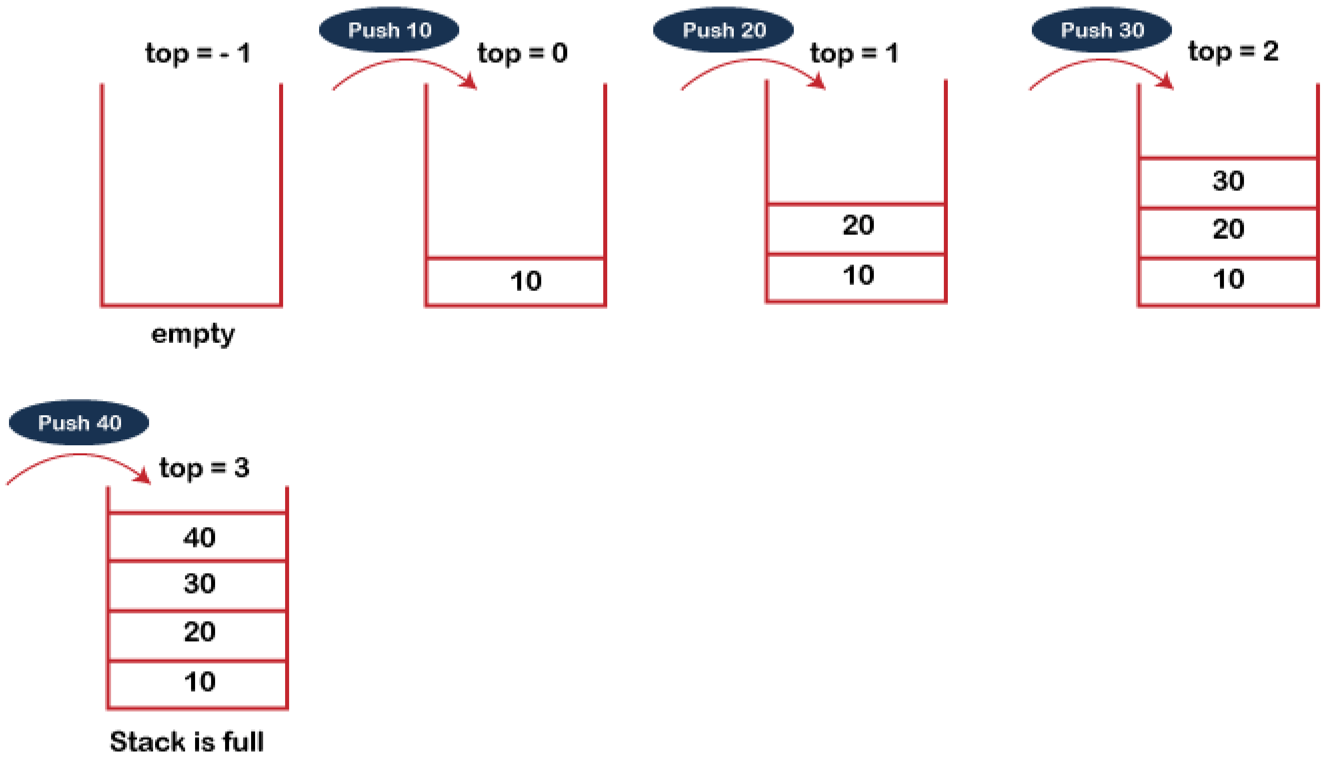
## Stack Operations

- push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.
- pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.
- isEmpty(): It determines whether the stack is empty or not.
- isFull(): It determines whether the stack is full or not.'
- peek(): It returns the element at the given position.
- count(): It returns the total number of elements available in a stack.
- change(): It changes the element at the given position.
- display(): It prints all the elements available in the stack.

## Push Operation

- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the overflow condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element is pushed in a stack, first, the value of the top gets incremented, i.e., top=top+1, and the element will be placed at the new position of the top.
- The elements will be inserted until we reach the max size of the stack.

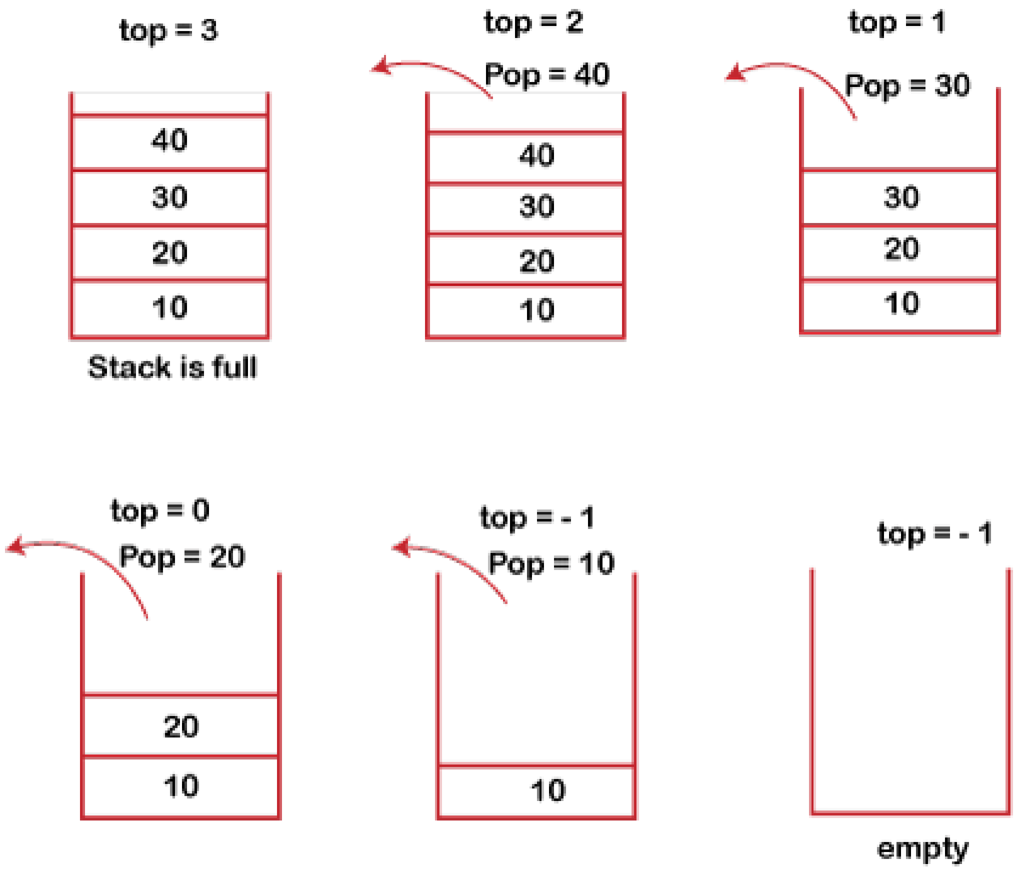
# Push Operation



## Pop operation

- Before deleting the element from the stack, we check whether the stack is empty.
- If we try to delete the element from the empty stack, then the underflow condition occurs.
- If the stack is not empty, we first access the element which is pointed by the top
- Once the pop operation is performed, the top is decremented by 1, i.e., top=top-1.

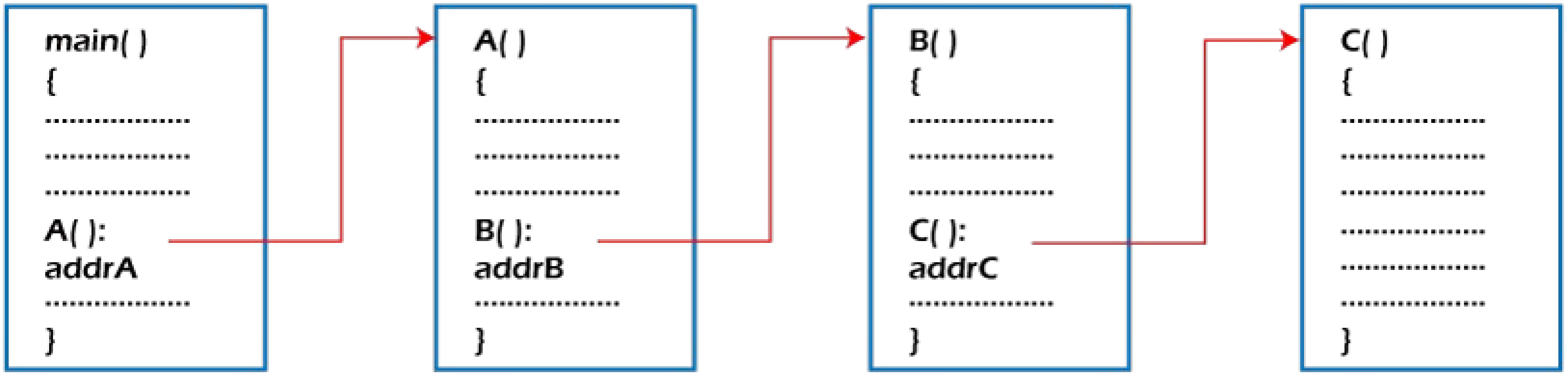
## Pop operation



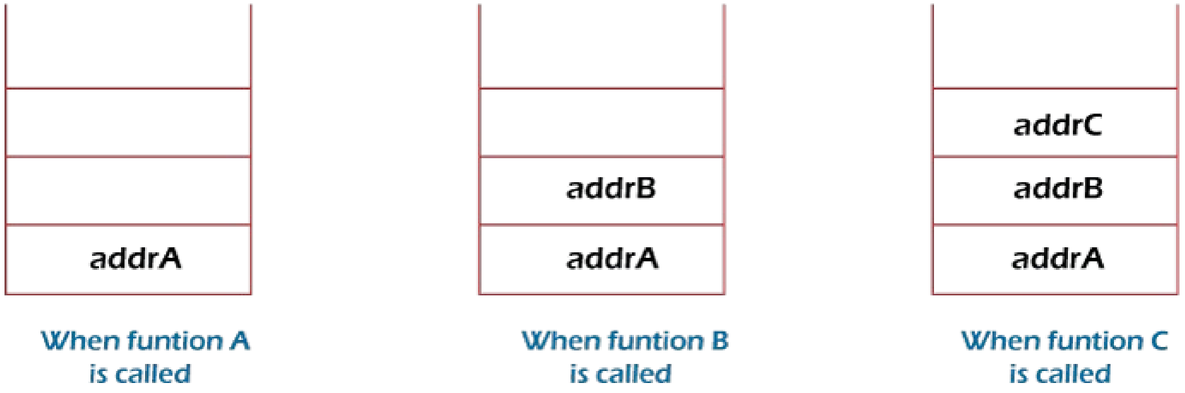
## Stack Applications

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Calls

## Function Call

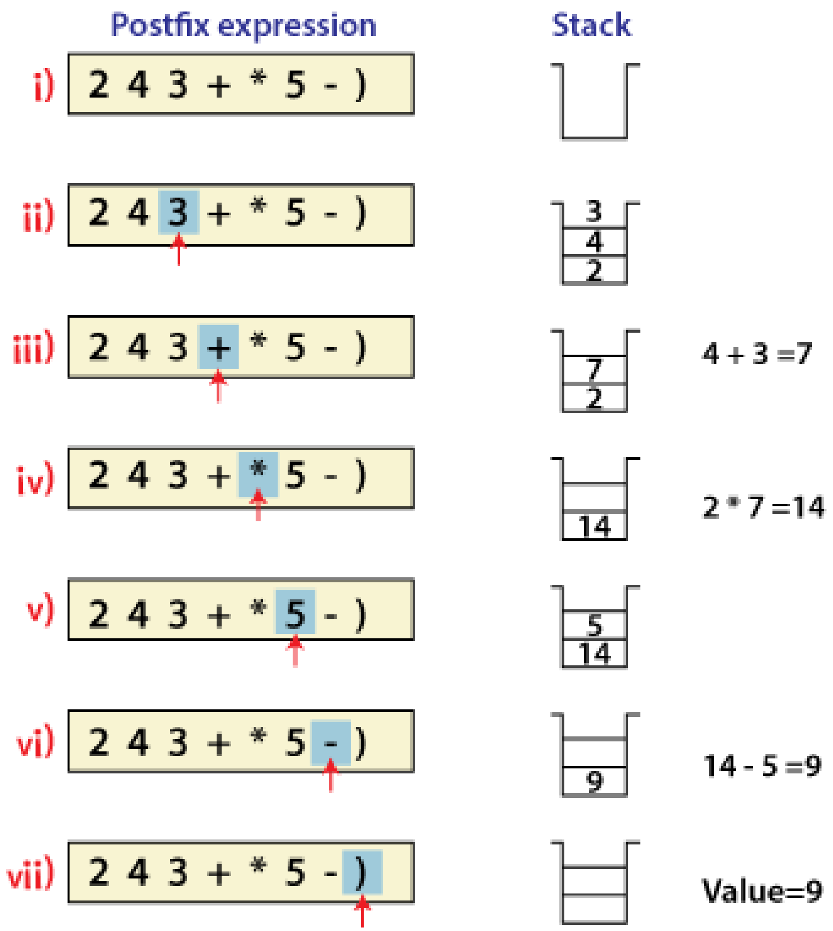


Function call



Different states of stack

# Expression Evaluation



Evaluation of postfix expression

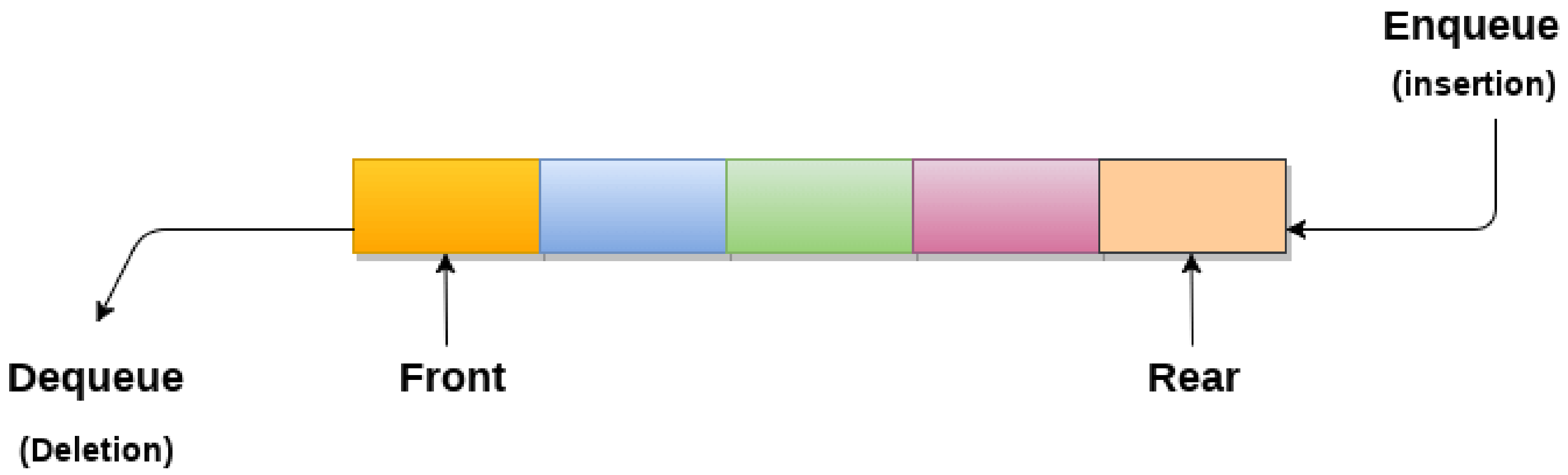
## Stack

- Example.

## Queue

- A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.
- Queue is referred to be as First In First Out list.
- For example, people waiting in line for a rail ticket form a queue.

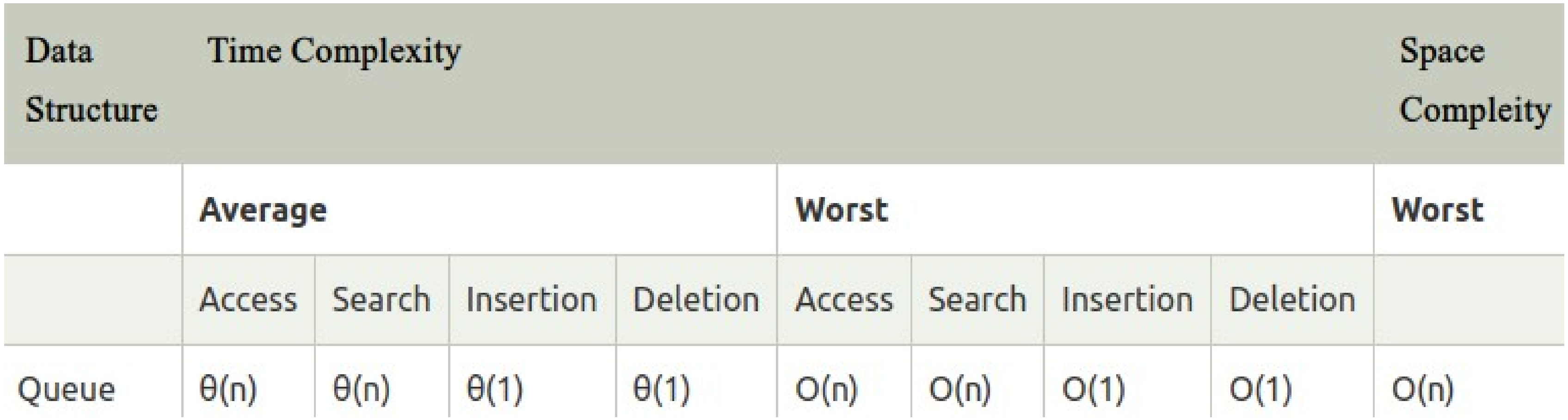
## Queue



## Queue

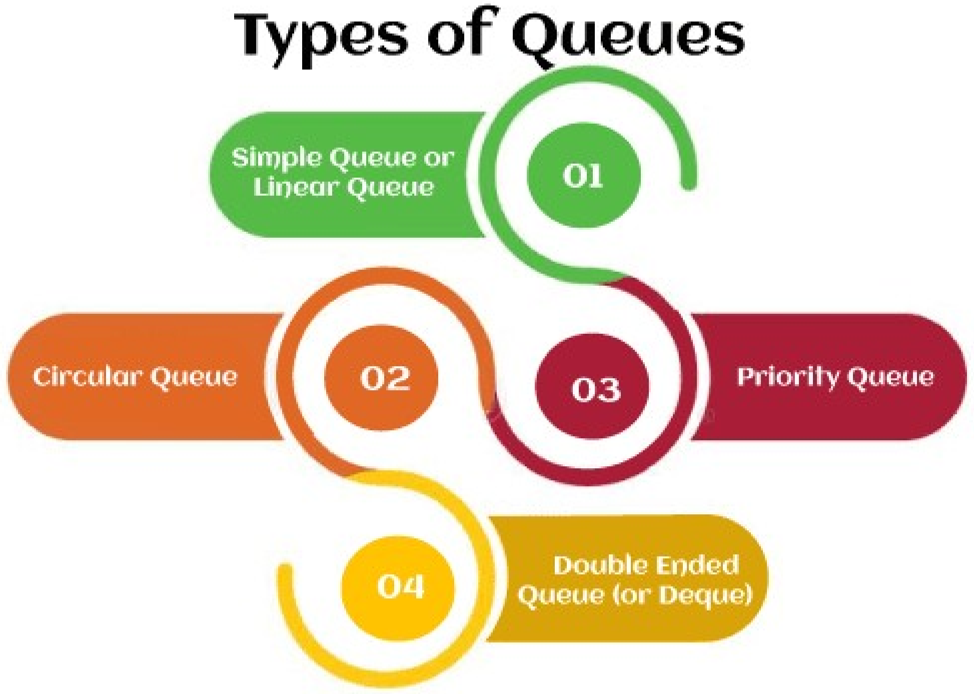
- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
- Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
- Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
- Queues are used in operating systems for handling interrupts.

# Queue

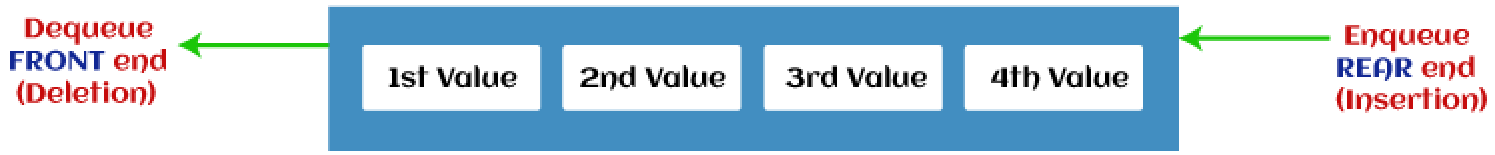


| Data Structure | Time Complexity | | | | | | | | Space Compleity |
|---|---|---|---|---|---|---|---|---|---|
| | **Average** | | | | **Worst** | | | | **Worst** |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion | |
| Queue | θ(n) | θ(n) | θ(1) | θ(1) | O(n) | O(n) | O(1) | O(1) | O(n) |

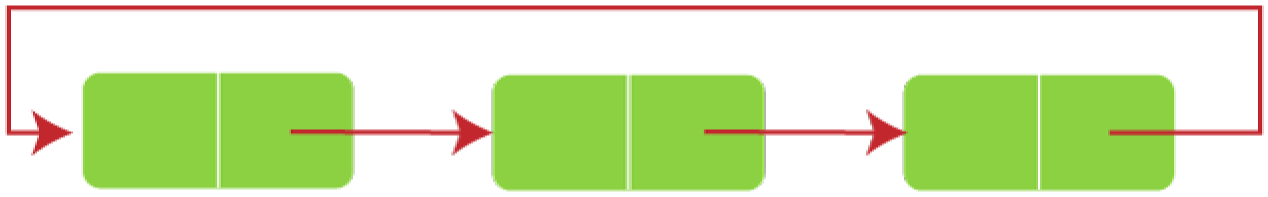
## Queue: Types


Types of Queues
01
Simple Queue or Linear Queue
02
Circular Queue
03
Priority Queue
04
Double Ended Queue (or Deque)

## Simple Queue

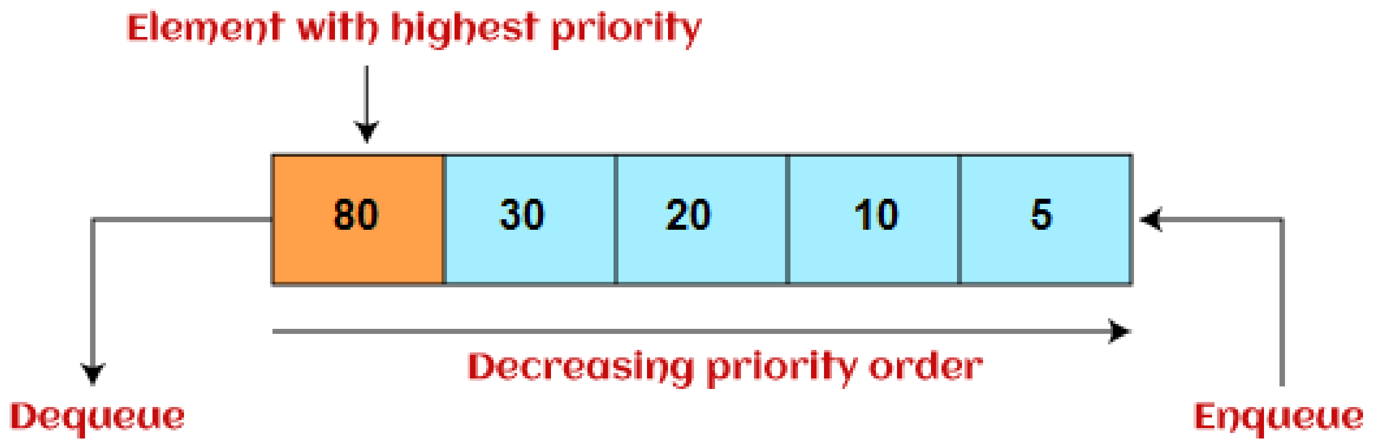


## Circular Queue

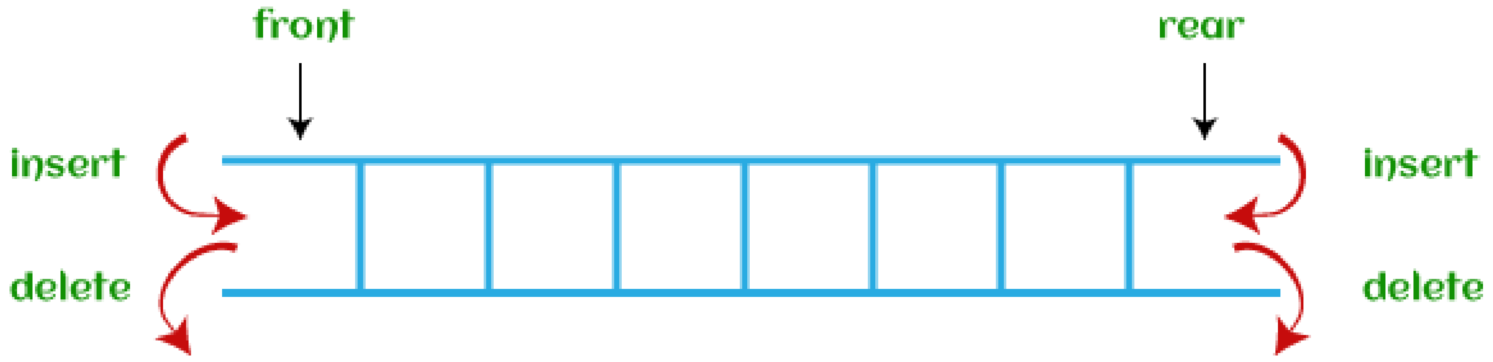


Circular Queue

# Priority Queue



## DeQueue



double ended queue

## Queue

- Example.

# Thank you

@mitu_skillologies @mITuSkillologies @mitu_group @mitu-skillologies @MITUSkillologies

kaggle

@mituskillologies

**Web Resources**
https://mitu.co.in
http://tusharkute.com

@mituskillologies

**contact@mitu.co.in**

**tushar@tusharkute.com**