

What are Data Structures?

A data structure is defined as a format for arranging, processing, accessing, and storing data.

Data structures are the combination of both simple and complex forms, all of which are made to organise data for a certain use.

Users find it simple to access the data they need and use it appropriately thanks to data structures.

To make you understand in simpler terms, look at the below example

If you want to store data in

One on the other - Stacks

Linear fashion - Array/ Linked List

Hierarchical Fashion - Trees

Connect Nodes – Graph

What are Data Structures in Java?

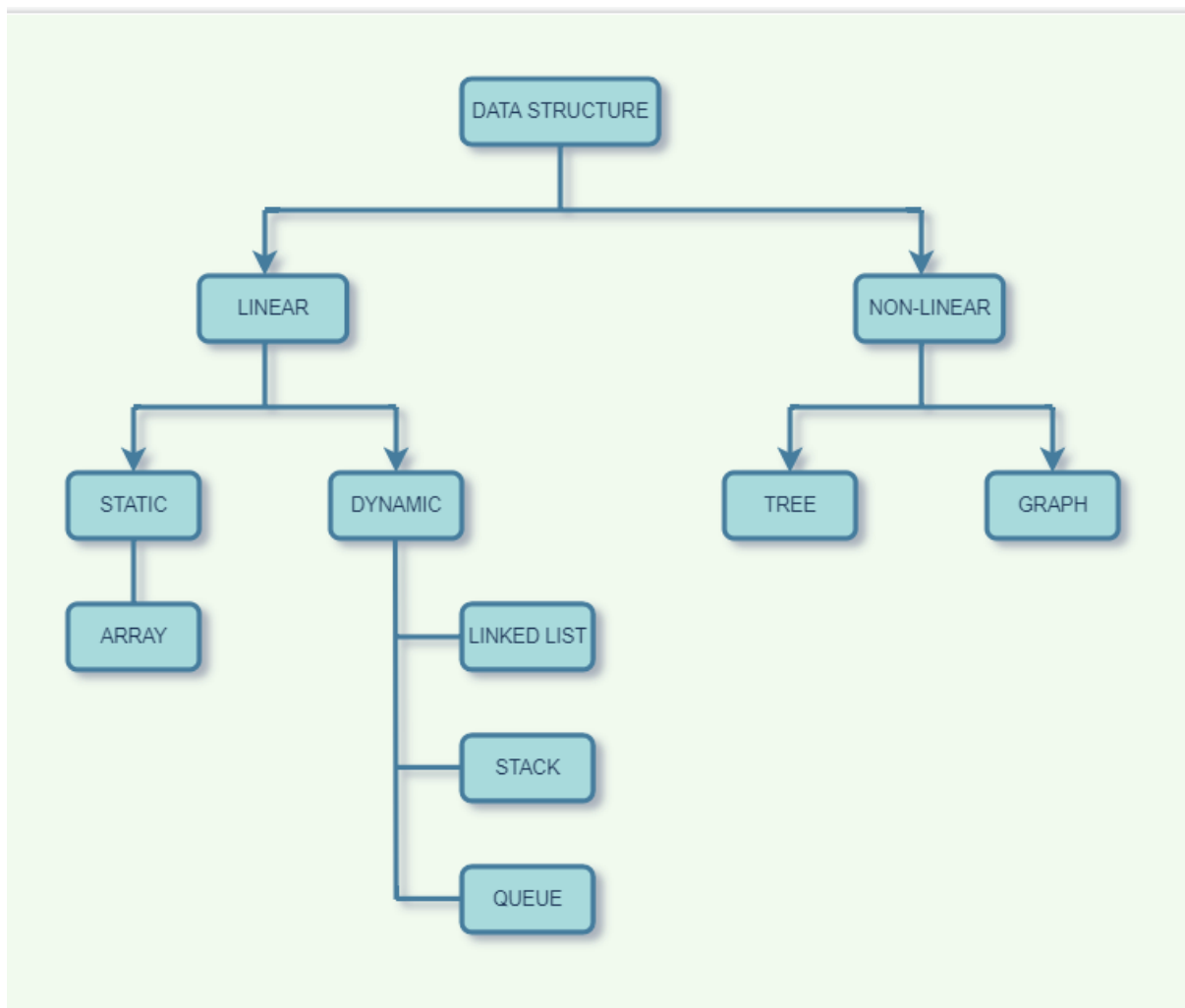
Data Structure in java is defined as the collection of data pieces that offers an effective means of storing and organising data in a computer. Linked List, Stack, Queue, and arrays are a few examples of java data structures.

Types of Data Structures in Java

Here is the list of some of the common types of data structures in Java:

- Array
- Linked List
- Stack
- Queue
- Binary Tree
- Binary Search Tree
- Heap
- Hashing
- Graph

Here is the pictorial representation of types of java data structures



Further classification of types of Data Structures

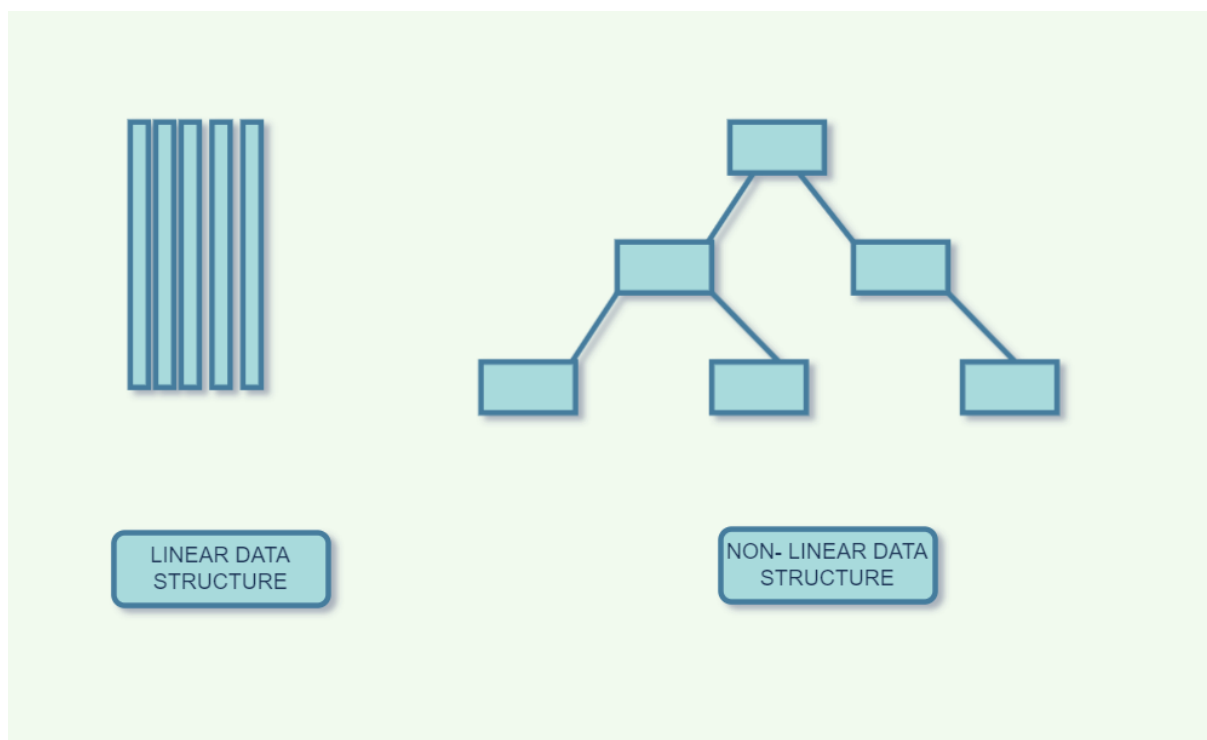
There are two types of Data Structures:-

1. Primitive Data Structures
2. Non-primitive Data Structures

Primitive data Structures are also called Primitive Data Types. byte, short, int, float, char, boolean, long, and double are primitive Data types.

Non-primitive data Structures – Non-primitive Data Structures are of two types:-

1. Linear Data Structures
2. Non-linear Data Structures



Linear Data Structures – The elements arranged in a linear fashion are called Linear Data Structures. Here, each element is connected to one other element only. Linear Data Structures are as follows:

- Arrays
 - Single dimensional Array
 - Multidimensional Array
- Stack
- Queue
- Linked List
 - Singly-linked list
 - Doubly Linked list
 - Circular Linked List

Non-Linear Data Structures – The elements arranged in a non-linear fashion are called Non-Linear Data Structures. Here, each element is connected to n-other elements. Non-Linear Data Structures are as follows:

- Trees
 - Binary Tree
 - Binary Search Tree
 - AVL Tree
 - Red-Black Tree
- Graph
- Heap
 - MaxHeap
 - MinHeap
- Hash
 - HashSet
 - HashMap

Advantages of Data Structures in java

- Efficiency
- Reusability
- Processing Speed
- Abstraction
- Data Searching

Classification of Data Structures

Data Structures can be classified as:-

- Static Data Structures are the Data structures whose size is declared and fixed at Compile Time and cannot be changed later are called Static Data structures.
- Example – Arrays
- Dynamic Data Structures are the Data Structures whose size is not fixed at compile time and can be decided at runtime depending upon requirements are called Dynamic Data structures.
- Example – Binary Search Tree

The building blocks of algorithms

An algorithm is a step by step process that describes how to solve a problem in a way that always gives a correct answer. When there are multiple algorithms for a particular problem (and there often are!), the best algorithm is typically the one that solves it the fastest.

As computer programmers, we are constantly using algorithms, whether it's an existing algorithm for a common problem, like sorting an array, or if it's a completely new algorithm unique to our program. By understanding algorithms, we can make better decisions about which existing algorithms to use and learn how to make new algorithms that are correct and efficient.

An algorithm is made up of three basic building blocks: sequencing, selection, and iteration.

Sequencing: An algorithm is a step-by-step process, and the order of those

steps are crucial to ensuring the correctness of an algorithm.

Here's an algorithm for translating a word into Pig Latin, like from "pig" to "ig-pay":

- Append "-".
- 2. Append first letter
- 3. Append "ay"
- 4. Remove first letter

Try following those steps in different orders and see what comes out. Not the same, is it?

Selection: Algorithms can use selection to determine a different set of steps to execute based on a Boolean expression.

Here's an improved algorithm for Pig Latin that handles words that starts with vowels, so that "eggs" becomes "eggs-yay" instead of the unpronounceable "ggs-eay":

- 1. Append "-"
- 2. Store first letter
- 3. If first letter is vowel:
 - a. Append "yay"
- 4. Otherwise:
 - a. Append first letter
 - b. Append "ay"
 - c. Remove first letter

Iteration: Algorithms often use repetition to execute steps a certain number of times or until a certain condition is met.

We can add iteration to the previous algorithm to translate a complete phrase, so that "peanut butter and jelly" becomes "eanut-pay utter-bay and-yay elly-jay":

1. Store list of words
2. For each word in words:
 - a. Append hyphen
 - b. If first letter is vowel:
 - i. Append "yay"
 - c. Otherwise:
 - i. Append first letter
 - ii. Append "ay"
 - iii. Remove first letter

By combining sequencing, selection, and iteration, we've successfully come up with an algorithm for Pig Latin translation.

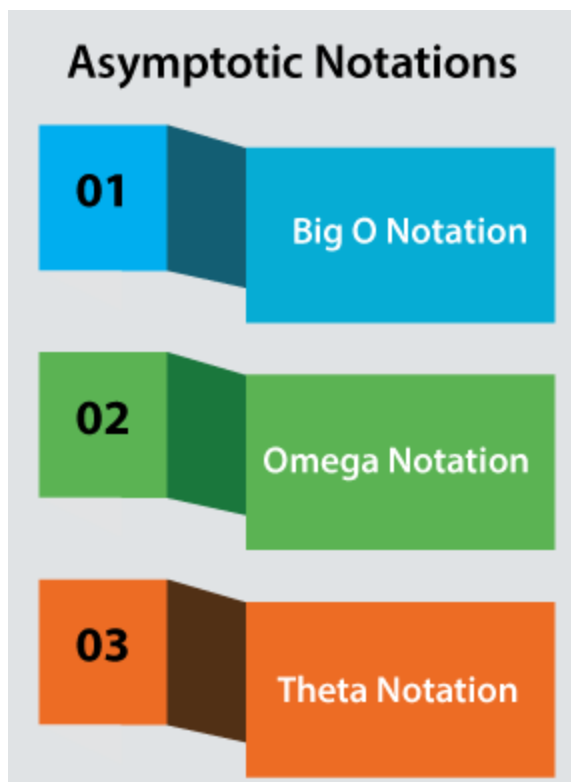
Big O Notation in Java

In Data Structure and Algorithms in Java programming, we have learned so many algorithms where we have understood different aspects and purposes of an algorithm. We have also studied the complexity of an algorithm and how to analyze and calculate an algorithm's complexity. We have found the time and space complexity of an algorithm and concluded that the algorithm that has less time and space complexity is evaluated as the best algorithm. We understood how to find the best case, worst case, and the average case of an algorithm. Thus, for analyzing all such complexities and representing them, the concept of Asymptotic Notation is used under which there are different types available for representing the complexities. One such type is Big O Notation.

In this section, we will discuss the Big O notations and briefly introduce Asymptotic notations and its types.

What are Asymptotic Notations

These are the mathematical notations that are used for the asymptotic analysis of the algorithms. The term 'asymptotic' describes an expression where a variable exists whose value tends to infinity. In short, it is a method that describes the limiting behavior of an expression. Thus, using asymptotic notations, we analyze the complexities of an algorithm and its performance. Using the asymptotic notations, we determine and show the complexities after analyzing it. Therefore, there are three types of asymptotic notations through which we can analyze the complexities of the algorithms:



- Big O Notation (O): It represents the upper bound of the runtime of an algorithm. Big O Notation's role is to calculate the longest time an algorithm can take for its execution, i.e., it is used for calculating the worst-case time complexity of an algorithm.

- Omega Notation ($\Omega(n)$): It represents the lower bound of the runtime of an algorithm. It is used for calculating the best time an algorithm can take to complete its execution, i.e., it is used for measuring the best case time complexity of an algorithm.
- Theta Notation ($\Theta(n)$): It carries the middle characteristics of both Big O and Omega notations as it represents the lower and upper bound of an algorithm.

So, these three asymptotic notations are the most used notations, but other than these, there are more common asymptotic notations also present, such as linear, logarithmic, cubic, and many more.

Big O Notation

The Big O notation is used to express the upper bound of the runtime of an algorithm and thus measure the worst-case time complexity of an algorithm. It analyses and calculates the time and amount of memory required for the execution of an algorithm for an input value.

How does Big O notation analyze the Space complexity

It is essential to determine both runtime and space complexity for an algorithm. It's because on analyzing the runtime performance of the algorithm, we get to know the execution time the algorithm is taking, and on analyzing the space complexity of the algorithm, we get to know the memory space the algorithm is occupying. Thus, for measuring the space complexity of an algorithm, it is required to compare the worst-case space complexities performance of the algorithm.

Big O Notation is one of those things that I was taught at university, but I never really grasped the concept. I knew enough to answer very basic questions on it, but that was about it. Nothing has changed since then as I have not used or heard any of my colleagues mention it since I started working. So, I thought I'd spend some time going back over it and write this post summarizing the basics of Big O Notation along with some code examples to help explain it.

So, what is Big O Notation? In simple terms:

- It is the relative representation of the complexity of an algorithm.
- It describes how an algorithm performs and scales.
- It describes the upper bound of the growth rate of a function and could be thought of the worst case scenario.

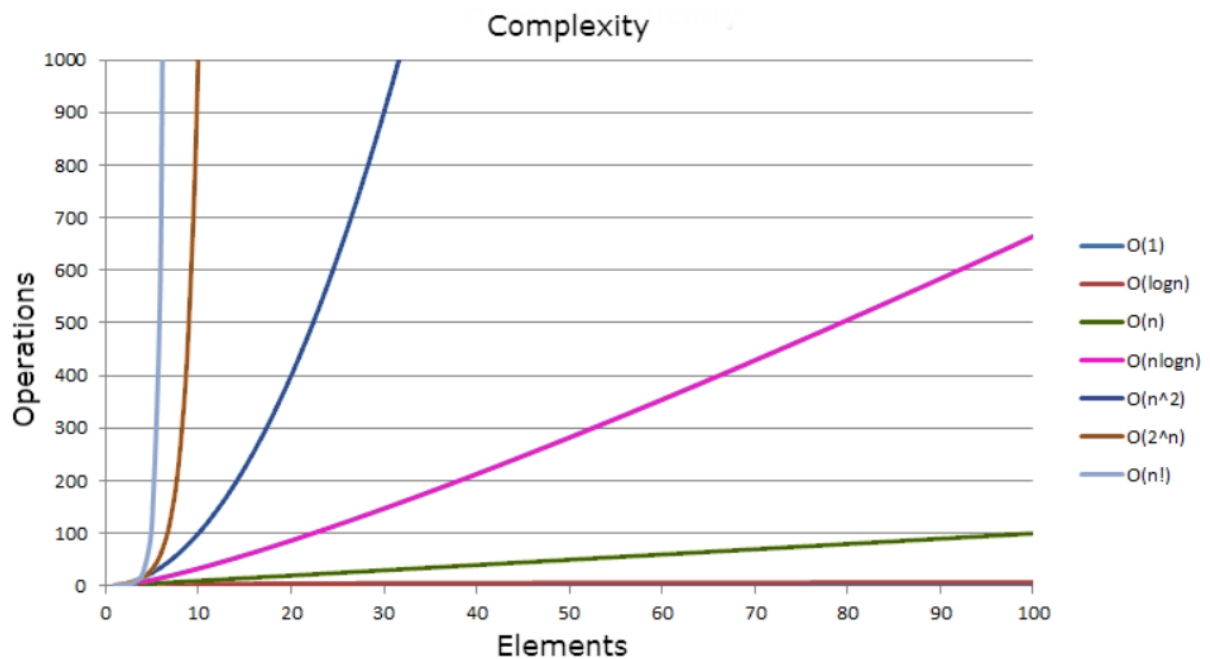
Now for a quick look at the syntax: $O(n^2)$.

n is the number of elements that the function receiving as inputs. So, this example is saying that for n inputs, its complexity is equal to n^2 .

Comparison of the Common Complexities

n	Constant $O(1)$	Logarithmic $O(\log n)$	Linear $O(n)$	Linear Logarithmic $O(n \log n)$	Quadratic $O(n^2)$	Cubic $O(n^3)$
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1,024	1	10	1,024	10,240	1,048,576	1,073,741,824

As you can see from this table, as the complexity of a function increases, the number of computations or time it takes to complete a function can rise quite significantly. Therefore, we want to keep this growth as low as possible, as performance problems might arise if the function does not scale well as inputs are increased.



Graph showing how the number of operations increases with complexity.

Some code examples should help clear things up a bit regarding how complexity affects performance. The code below is written in Java but obviously, it could be written in other languages.

$O(1)$

```
1 public boolean isFirstNumberEqualToOne(List<Integer> numbers) {
2     return numbers.get(0) == 1;
3 }
```

$O(1)$ represents a function that always takes the same time regardless of input size.

$O(n)$

```
1 public boolean containsNumber(List<Integer> numbers, int comparisonNumber) {
2     for(Integer number : numbers) {
3         if(number == comparisonNumber) {
4             return true;
5         }
6     }
7     return false;
8 }
```

$O(n)$ represents the complexity of a function that increases linearly and in direct proportion to the number of inputs. This is a good example of how Big O Notation describes the *worst case scenario* as the function could return the *true* after reading the first element or *false* after reading all n elements.

$O(n^2)$

```
1 public static boolean containsDuplicates(List<String> input) {
2     for (int outer = 0; outer < input.size(); outer++) {
3         for (int inner = 0; inner < input.size(); inner++) {
4             if (outer != inner && input.get(outer).equals(input.get(inner))) {
5                 return true;
6             }
7         }
8     }
9     return false;
10 }
```

$O(n^2)$ represents a function whose complexity is directly proportional to the square of the input size. Adding more nested iterations through the input will increase the complexity which could then represent $O(n^3)$ with 3 total iterations and $O(n^4)$ with 4 total iterations.

$O(2^n)$

```
1 public int fibonacci(int number) {
2     if (number <= 1) {
3         return number;
4     } else {
5         return fibonacci(number - 1) + fibonacci(number - 2);
6     }
7 }
```

Abstract data type in data structure

Before knowing about the abstract data type, we should know about the what is a data structure.

What is data structure?

A data structure is a technique of organizing the data so that the data can be utilized efficiently. There are two ways of viewing the data structure:

- Mathematical/ Logical/ Abstract models/ Views: The data structure is the way of organizing the data that requires some protocols or rules. These rules need to be modeled that come under the logical/abstract model.

- Implementation: The second part is the implementation part. The rules must be implemented using some programming language.

Why data structure?

The following are the advantages of using the data structure:

- These are the essential ingredients used for creating fast and powerful algorithms.
- They help us to manage and organize the data.
- Data structures make the code cleaner and easier to understand.

What is 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. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details. The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.

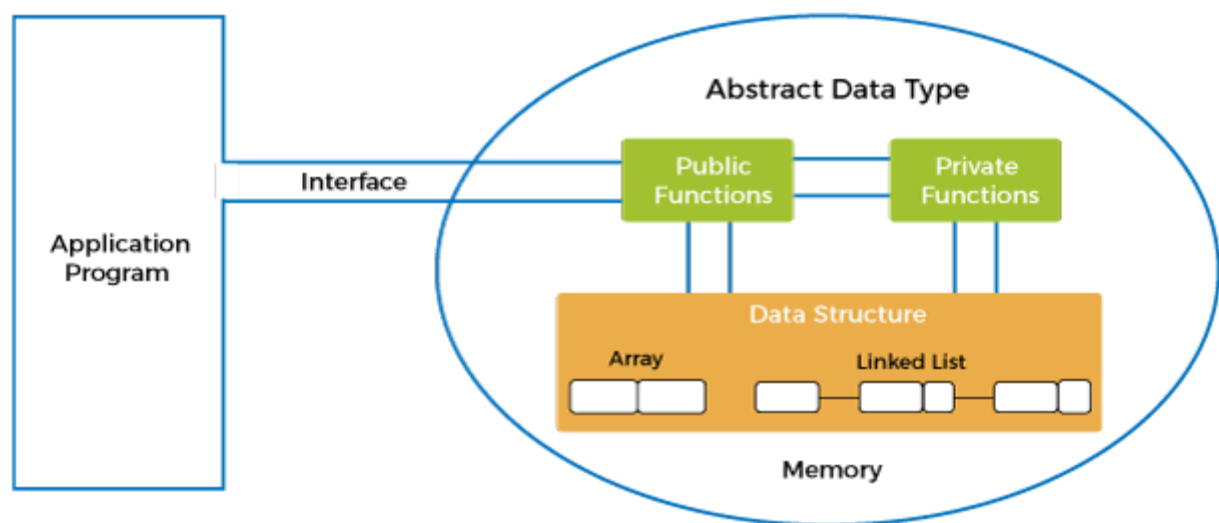
For example, a List is an abstract data type that is implemented using a dynamic array and linked list. A queue is implemented using linked list-based queue, array-based queue, and stack-based queue. A Map is implemented using Tree map, hash map, or hash table.

Abstract data type model

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.



The above figure shows the ADT model. There are two types of models in the ADT model, i.e., the public function and the private function. The ADT model also contains the data structures that we are using in a program. In this model, first encapsulation is performed, i.e., all the data is wrapped in a single unit, i.e., ADT. Then, the abstraction is performed means showing the operations that can be performed on the data structure and what are the data structures that we are using in a program.

The user of data type does not need to know how that data type is implemented, for example, we have been using Primitive values like int, float, char data types only with the knowledge that these data type can operate and be performed on without any idea of how they are implemented.

So a user only needs to know what a data type can do, but not how it will be implemented. Think of ADT as a black box which hides the inner structure and design of the data type. Now we'll define three ADTs namely List ADT, Stack ADT, Queue ADT.

1. List ADT

- The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.
- The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.
- The List ADT Functions is given below:
 - `get()` – Return an element from the list at any given position.
 - `insert()` – Insert an element at any position of the list.
 - `remove()` – Remove the first occurrence of any element from a non-empty list.
 - `removeAt()` – Remove the element at a specified location from a non-empty list.
 - `replace()` – Replace an element at any position by another element.
 - `size()` – Return the number of elements in the list.
 - `isEmpty()` – Return true if the list is empty, otherwise return false.
 - `isFull()` – Return true if the list is full, otherwise return false.

2. Stack ADT

- In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
- The program allocates memory for the *data* and *address* is passed to the stack ADT.
- The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
- The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.
- `push()` – Insert an element at one end of the stack called top.
- `pop()` – Remove and return the element at the top of the stack, if it is not empty.
- `peek()` – Return the element at the top of the stack without removing it, if the stack is not empty.
- `size()` – Return the number of elements in the stack.
- `isEmpty()` – Return true if the stack is empty, otherwise return false.

- isFull() – Return true if the stack is full, otherwise return false.

3. Queue ADT

- The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
- Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program's responsibility is to allocate memory for storing the data.
- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false.

Features of ADT:

- Abstraction: The user does not need to know the implementation of the data structure only essentials are provided.
- Better Conceptualization: ADT gives us a better conceptualization of the real world.

- Robust: The program is robust and has the ability to catch errors.

From these definitions, we can clearly see that the definitions do not specify how these ADTs will be represented and how the operations will be carried out. There can be different ways to implement an ADT, for example, the List ADT can be implemented using arrays, or singly linked list or doubly linked list. Similarly, stack ADT and Queue ADT can be implemented using arrays or linked lists.

Let's understand the abstract data type with a real-world example.

If we consider the smartphone. We look at the high specifications of the smartphone, such as:

- 4 GB RAM

- Snapdragon 2.2ghz processor
- 5 inch LCD screen
- Dual camera
- Android 8.0

The above specifications of the smartphone are the data, and we can also perform the following operations on the smartphone:

- `call()`: We can call through the smartphone.
- `text()`: We can text a message.
- `photo()`: We can click a photo.
- `video()`: We can also make a video.

The smartphone is an entity whose data or specifications and operations are given above. The abstract/logical view and operations are the abstract or logical views of a smartphone.

The implementation view can differ because the syntax of programming languages is different, but the abstract/logical view of the data structure would remain the same. Therefore, we can say that the abstract/logical view is independent of the implementation view.

Note: We know the operations that can be performed on the predefined data types such as int, float, char, etc., but we don't know the implementation details of the data types. Therefore, we can say that the abstract data type is considered as the hidden box that hides all the internal details of the data type.

Array in Data Structure

In this article, we will discuss the array in data structure. Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

In Java programming, they are the derived data types that can store the primitive type of data such as int, char, double, float, etc. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define a different variable for the marks in different subjects. Instead, we can define an array that can store the marks in each subject at the contiguous memory locations.

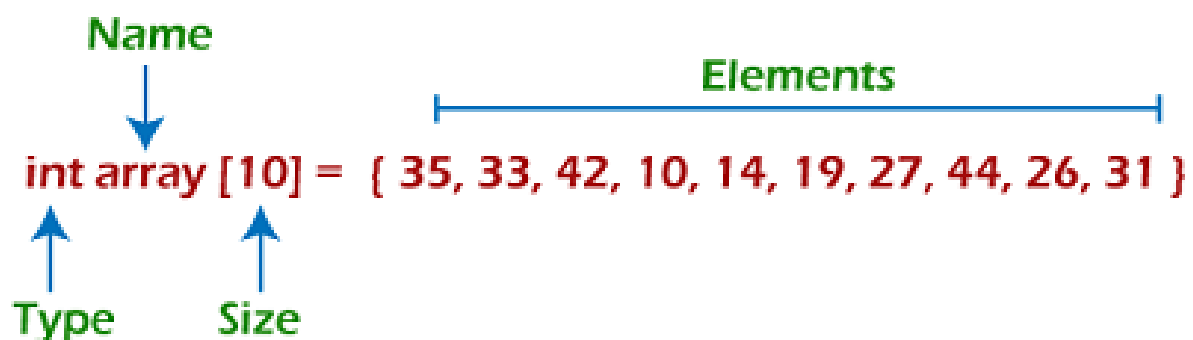
Properties of array

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

We can represent an array in various ways in different programming languages. As an illustration, let's see the declaration of array in C language -



As per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

Why are arrays required?

Arrays are useful because -

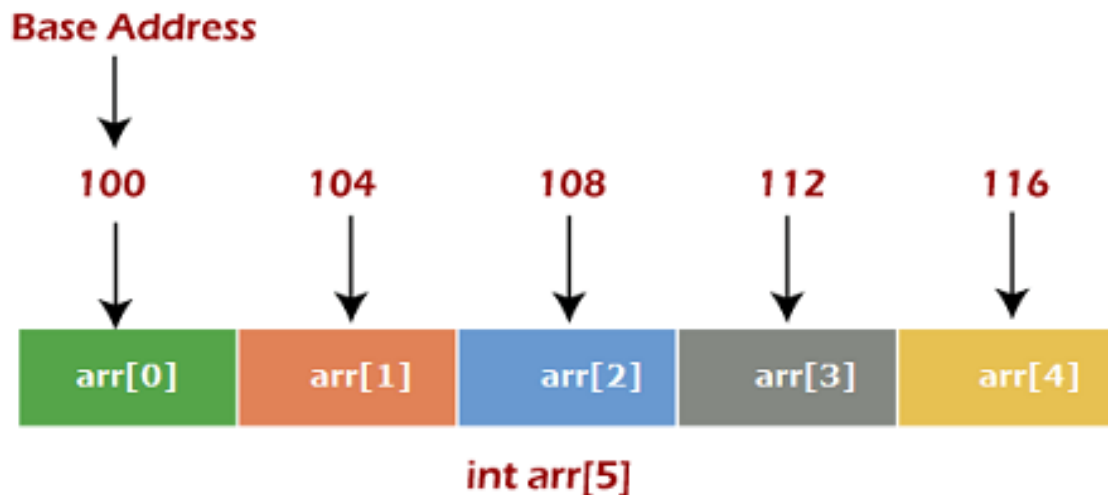
- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.
- Arrays are good for storing multiple values in a single variable -
In computer programming; most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Memory allocation of an array

As stated above, 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 -

1. 0 (zero-based indexing): The first element of the array will be `arr[0]`.
2. 1 (one-based indexing): The first element of the array will be `arr[1]`.
3. n (n - based indexing): The first element of the array can reside at any random index number.



In the above image, we have shown the memory allocation of an array `arr` of size 5. The array follows a 0-based indexing approach. The base address of the array is 100 bytes. It is the address of `arr[0]`. Here, the size of the data type used is 4 bytes; therefore, each element will take 4 bytes in the memory.

How to access an element from the array?

We required the information given below to access any random element from the array -

- Base Address of the array.
- Size of an element in bytes.
- Type of indexing, array follows.

The formula to calculate the address to access an array element -

1. Byte address of element `A[i]` = base address + size * (i - first index)

Here, size represents the memory taken by the primitive data types. As an instance, `int` takes 2 bytes, `float` takes 4 bytes of memory space in Java programming.

Basic operations

Now, let's discuss the basic operations supported in the array -

- 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 of Array operations

Time and space complexity of various array operations are described in the following table.

Time Complexity

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

Space Complexity

In array, space complexity for worst case is $O(n)$.

Advantages of Array

- Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

Disadvantages of Array

- Array is homogenous. It means that the elements with similar data type can be stored in it.
- In array, there is static memory allocation that is size of an array cannot be altered.
- There will be wastage of memory if we store less number of elements than the declared size.

Stack

Stack is abstract data type which depicts Last in first out (LIFO) behavior.

What is a 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.*

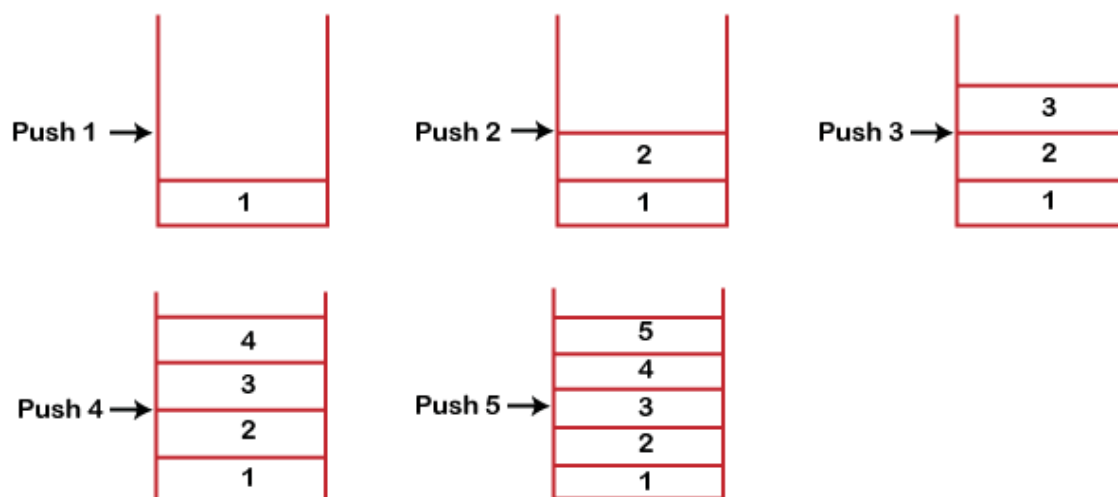
Some key points related to stack

- 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.

Working of Stack

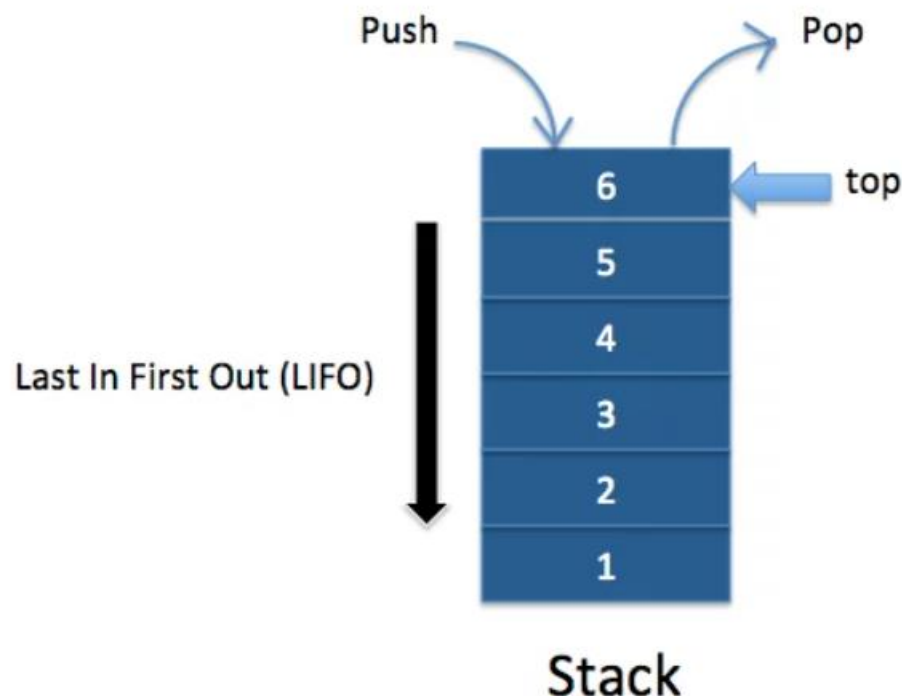
Stack works on the LIFO pattern. As we can observe in the below figure there are five memory blocks in the stack; therefore, the size of the stack is 5.

Suppose we want to store the elements in a stack and let's assume that stack is empty. We have taken the stack of size 5 as shown below in which we are pushing the elements one by one until the stack becomes full.



Since our stack is full as the size of the stack is 5. In the above cases, we can observe that it goes from the top to the bottom when we were

entering the new element in the stack. The stack gets filled up from the bottom to the top.



When we perform the delete operation on the stack, there is only one way for entry and exit as the other end is closed. It follows the LIFO pattern, which means that the value entered first will be removed last. In the above case, the value 5 is entered first, so it will be removed only after the deletion of all the other elements.

Standard Stack Operations

The following are some common operations implemented on the stack:

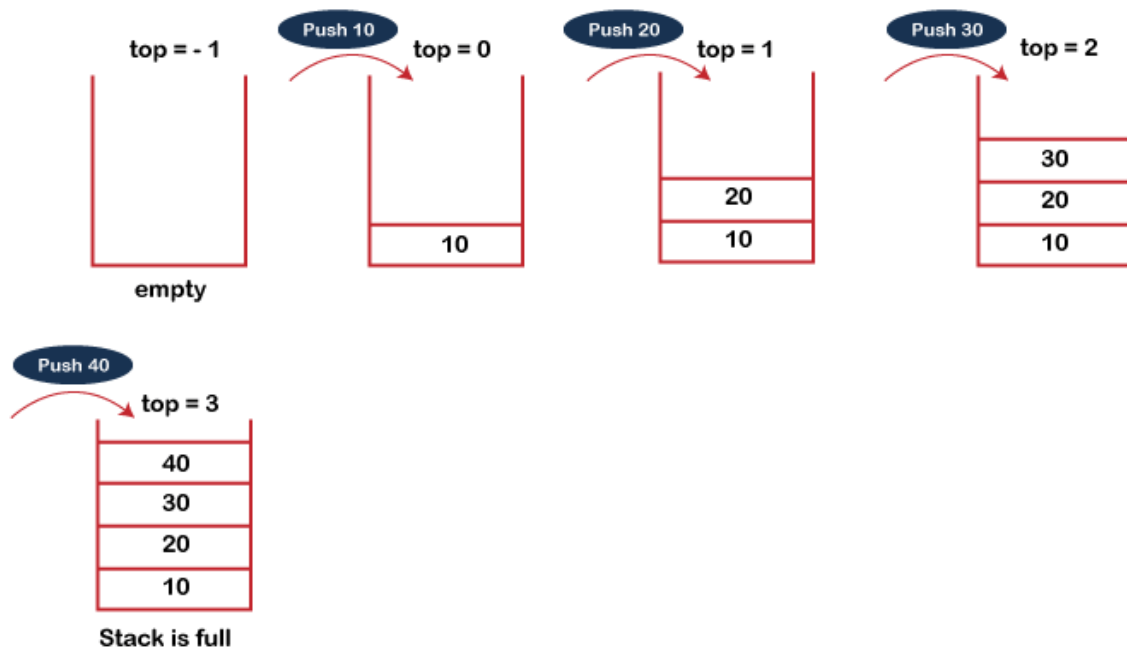
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.

- `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

The steps involved in the PUSH operation is given below:

- 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.



POP operation

The steps involved in the POP operation is given below:

- 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., $top = top - 1$.


top = 3

40
30
20
10

Stack is full

top = 2


Pop = 40



40
30
20
10

top = 1


Pop = 30



30
20
10

top = 0


Pop = 20



20
10

top = - 1

Pop = 10



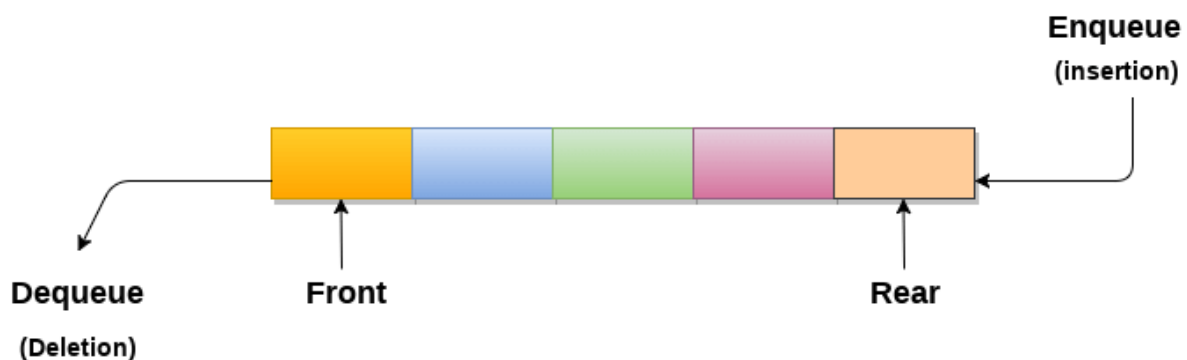
10

top = - 1

empty

Queue

1. 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.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. 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.
3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Complexity

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

Types of Queue

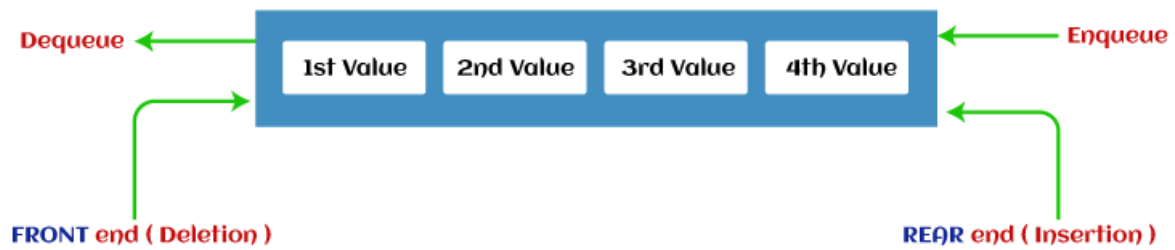
In this article, we will discuss the types of queue. But before moving towards the types, we should first discuss the brief introduction of the queue.

What is a Queue?

Queue is the data structure that is similar to the queue in the real world. A queue is a data structure in which whatever comes first will go out first, and it follows the FIFO (First-In-First-Out) policy. Queue can also be defined as the list or collection in which the insertion is done from one end known as the rear end or the tail of the queue, whereas the deletion is done from another end known as the front end or the head of the queue.

The real-world example of a queue is the ticket queue outside a cinema hall, where the person who enters first in the queue gets the ticket first, and the last person enters in the queue gets the ticket at last. Similar approach is followed in the queue in data structure.

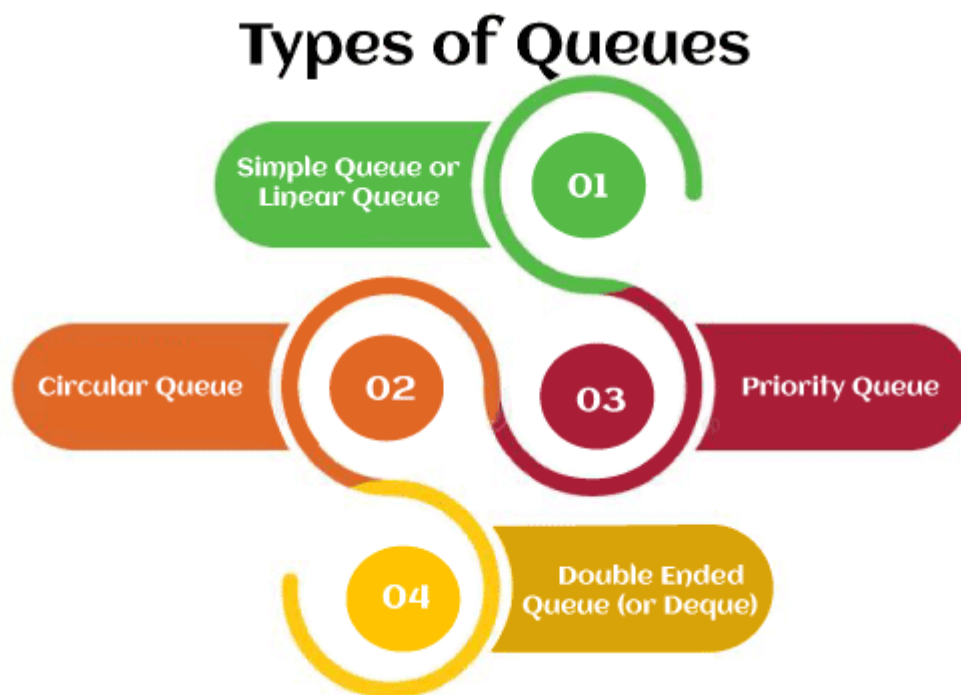
The representation of the queue is shown in the below image -



Now, let's move towards the types of queue.

Types of Queue

There are four different types of queue that are listed as follows -

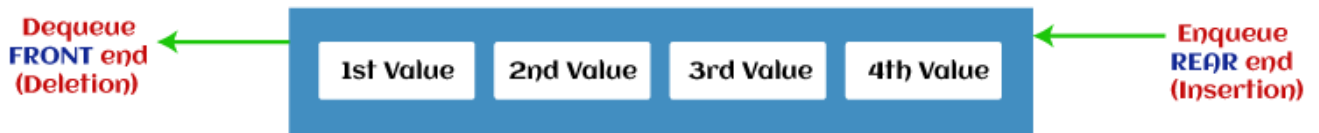


- Simple Queue or Linear Queue
- Circular Queue
- Priority Queue
- Double Ended Queue (or Deque)

Let's discuss each of the type of queue.

Simple Queue or Linear Queue

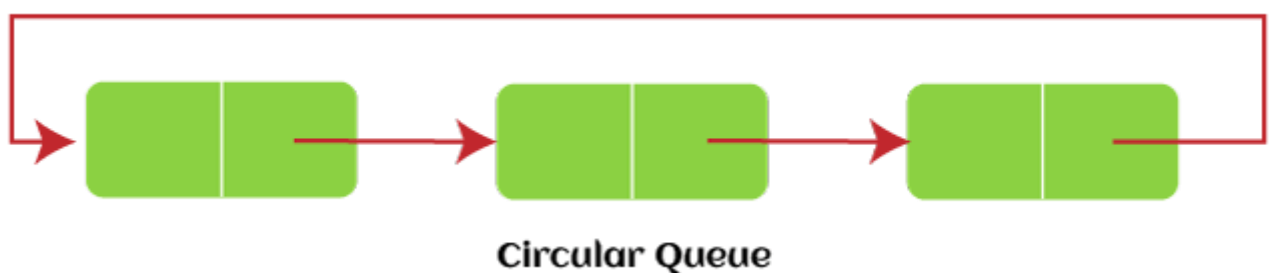
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



The major drawback of using a linear Queue is that insertion is done only from the rear end. If the first three elements are deleted from the Queue, we cannot insert more elements even though the space is available in a Linear Queue. In this case, the linear Queue shows the overflow condition as the rear is pointing to the last element of the Queue.

Circular Queue

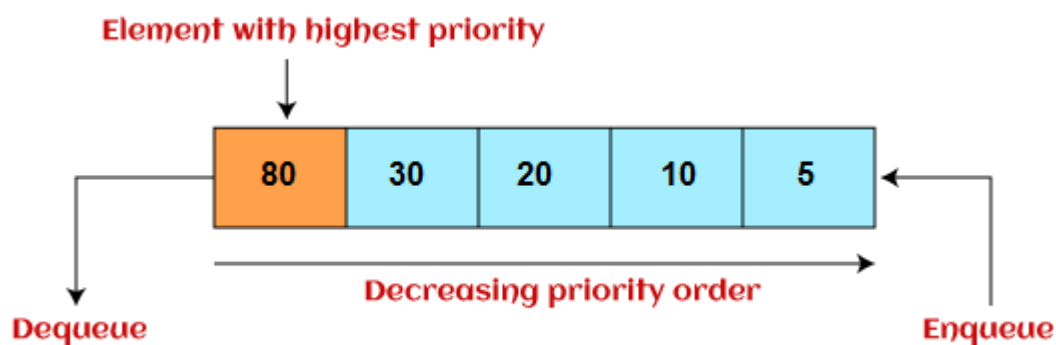
In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in the below image -



The drawback that occurs in a linear queue is overcome by using the circular queue. If the empty space is available in a circular queue, the new element can be added in an empty space by simply incrementing the value of rear. The main advantage of using the circular queue is better memory utilization.

Priority Queue

It is a special type of queue in which the elements are arranged based on the priority. It is a special type of queue data structure in which every element has a priority associated with it. Suppose some elements occur with the same priority, they will be arranged according to the FIFO principle. The representation of priority queue is shown in the below image -

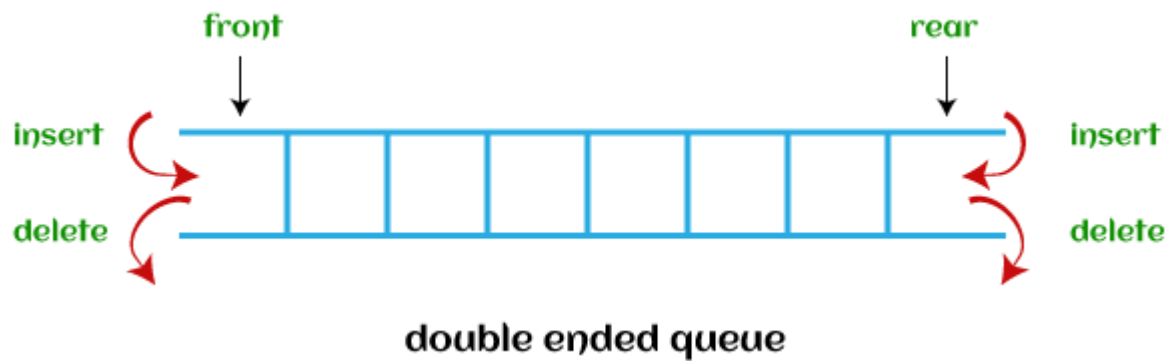


Deque (or, Double Ended Queue)

In Deque or Double Ended Queue, insertion and deletion can be done from both ends of the queue either from the front or rear. It means that we can insert and delete elements from both front and rear ends of the queue. Deque can be used as a palindrome checker means that if we read the string from both ends, then the string would be the same.

Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible to perform both insertion and deletion from one end, and Deque does not follow the FIFO principle.

The representation of the deque is shown in the below image -



Now, let's see the operations performed on the queue.

Operations performed on queue

The fundamental operations that can be performed on queue are listed as follows -

- Enqueue: The Enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- Dequeue: It performs the deletion from the front-end of the queue. It also returns the element which has been removed from the front-end. It returns an integer value.
- Peek: This is the third operation that returns the element, which is pointed by the front pointer in the queue but does not delete it.
- Queue overflow (isfull): It shows the overflow condition when the queue is completely full.
- Queue underflow (isempty): It shows the underflow condition when the Queue is empty, i.e., no elements are in the Queue.

Now, let's see the ways to implement the queue.

Ways to implement the queue

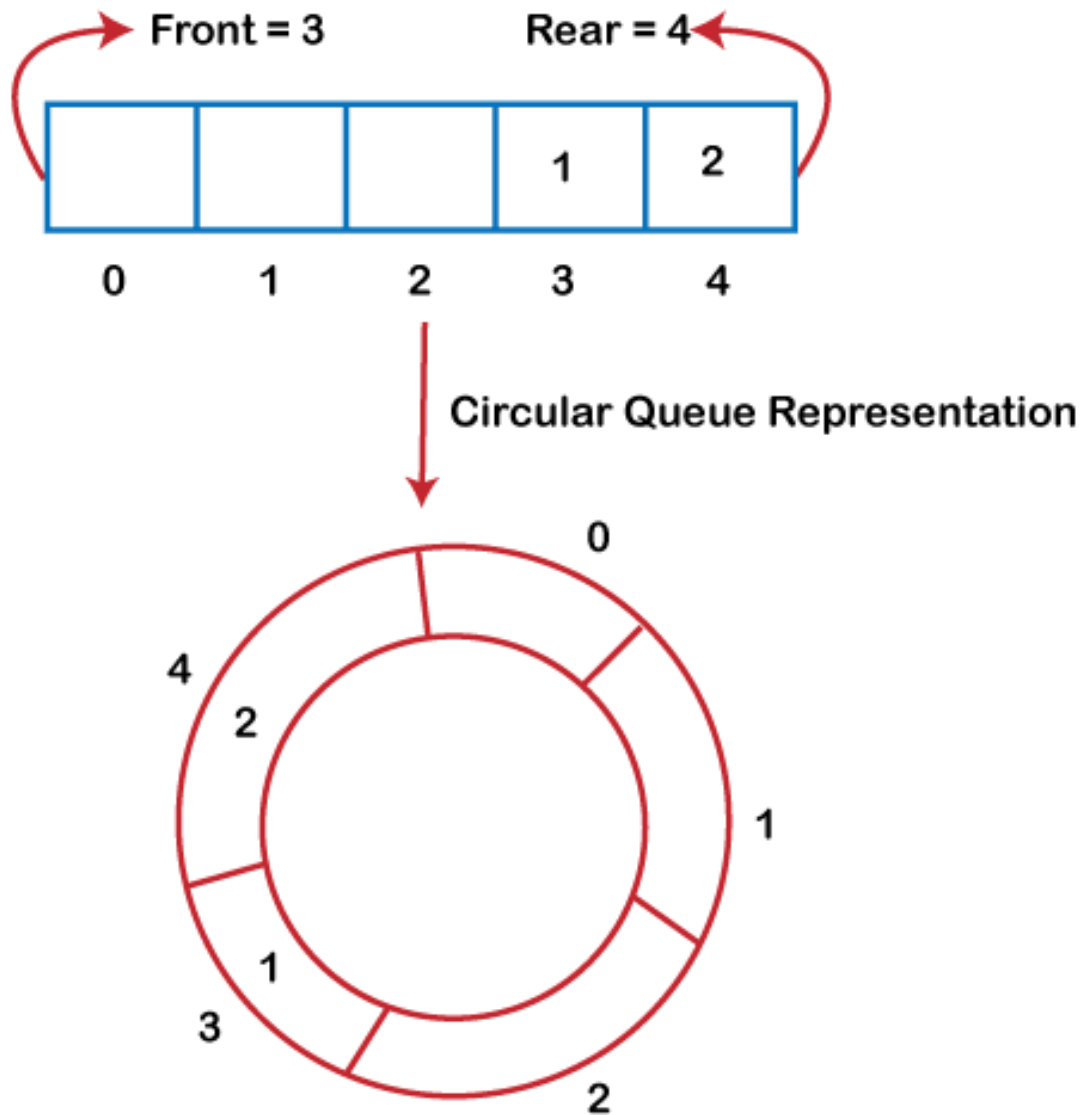
There are two ways of implementing the Queue:

- Implementation using array: The sequential allocation in a Queue can be implemented
- Implementation using Linked list: The linked list allocation in a Queue can be implemented using a linked list.

Circular Queue

Why was the concept of the circular queue introduced?

There was one limitation in the array implementation of Queue. If the rear reaches to the end position of the Queue then there might be possibility that some vacant spaces are left in the beginning which cannot be utilized. So, to overcome such limitations, the concept of the circular queue was introduced.



As we can see in the above image, the rear is at the last position of the Queue and front is pointing somewhere rather than the 0th position. In the above array, there are only two elements and other three positions are empty. The rear is at the last position of the Queue; if we try to insert the element then it will show that there are no empty spaces in the Queue. There is one solution to avoid such wastage of memory space by shifting both the elements at the left and adjust the front and rear end accordingly. It is not a practically good approach because shifting all the

elements will consume lots of time. The efficient approach to avoid the wastage of the memory is to use the circular queue data structure.

What is a Circular Queue?

A circular queue is similar to a linear queue as it is also based on the FIFO (First In First Out) principle except that the last position is connected to the first position in a circular queue that forms a circle. It is also known as a *Ring Buffer*.

Operations on Circular Queue

The following are the operations that can be performed on a circular queue:

- Front: It is used to get the front element from the Queue.
- Rear: It is used to get the rear element from the Queue.
- enQueue(value): This function is used to insert the new value in the Queue. The new element is always inserted from the rear end.
- deQueue(): This function deletes an element from the Queue. The deletion in a Queue always takes place from the front end.

Applications of Circular Queue

The circular Queue can be used in the following scenarios:

- Memory management: The circular queue provides memory management. As we have already seen that in linear queue, the memory is not managed very efficiently. But in case of a circular queue, the memory is managed efficiently by placing the elements in a location which is unused.
- CPU Scheduling: The operating system also uses the circular queue to insert the processes and then execute them.
- Traffic system: In a computer-control traffic system, traffic light is one of the best examples of the circular queue. Each light of traffic light gets ON one by one after every jinterval of time. Like red light gets ON for one minute then yellow light for one minute and then green light. After green light, the red light gets ON.

Enqueue operation

The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- When we insert a new element, the rear gets incremented, i.e., $rear = rear + 1$.

Scenarios for inserting an element

There are two scenarios in which queue is not full:

- If $rear \neq \text{max} - 1$, then rear will be incremented to $\text{mod}(\text{maxsize})$ and the new value will be inserted at the rear end of the queue.
- If $front \neq 0$ and $rear = \text{max} - 1$, it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

- When $front == 0$ & $rear = \text{max} - 1$, which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- $front == rear + 1$;

Algorithm to insert an element in a circular queue

Step 1: IF $(\text{REAR} + 1) \% \text{MAX} = \text{FRONT}$

Write " OVERFLOW "

Goto step 4

[End OF IF]

Step 2: IF $\text{FRONT} = -1$ and $\text{REAR} = -1$

SET $\text{FRONT} = \text{REAR} = 0$

ELSE IF $\text{REAR} = \text{MAX} - 1$ and $\text{FRONT} \neq 0$

SET $\text{REAR} = 0$

ELSE

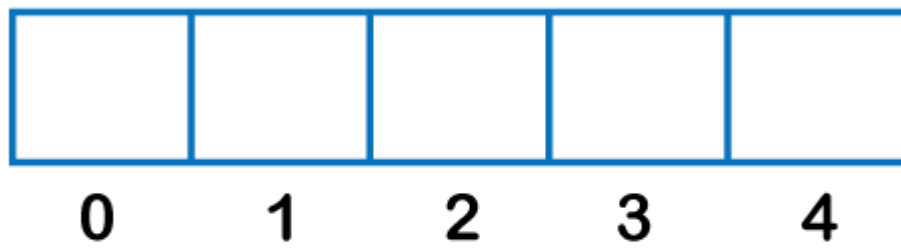
SET $\text{REAR} = (\text{REAR} + 1) \% \text{MAX}$

[END OF IF]

Step 3: SET $\text{QUEUE}[\text{REAR}] = \text{VAL}$

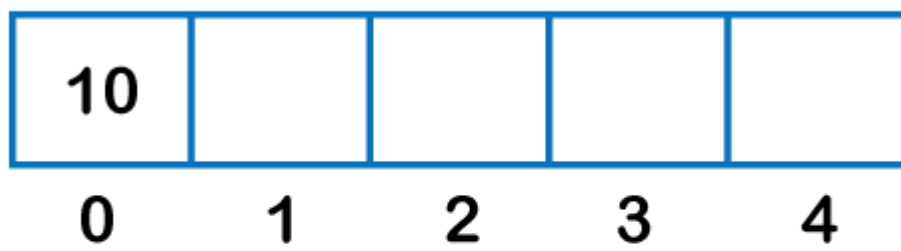
Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.



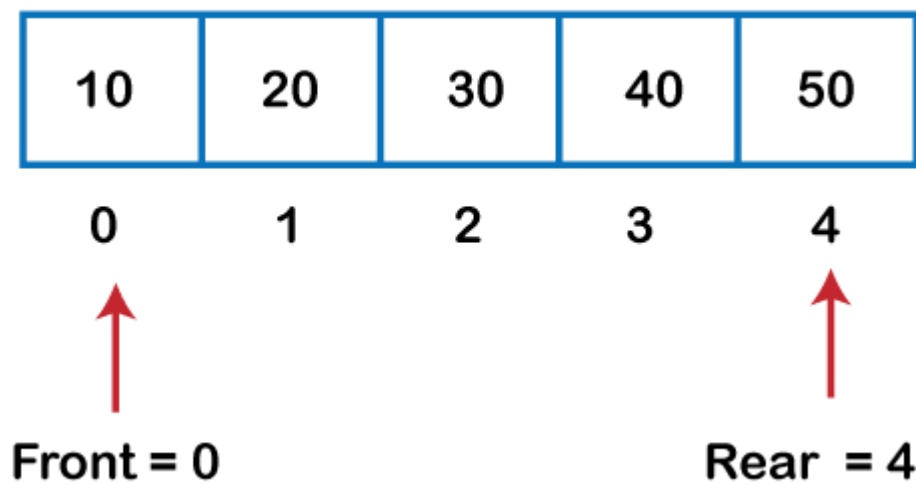
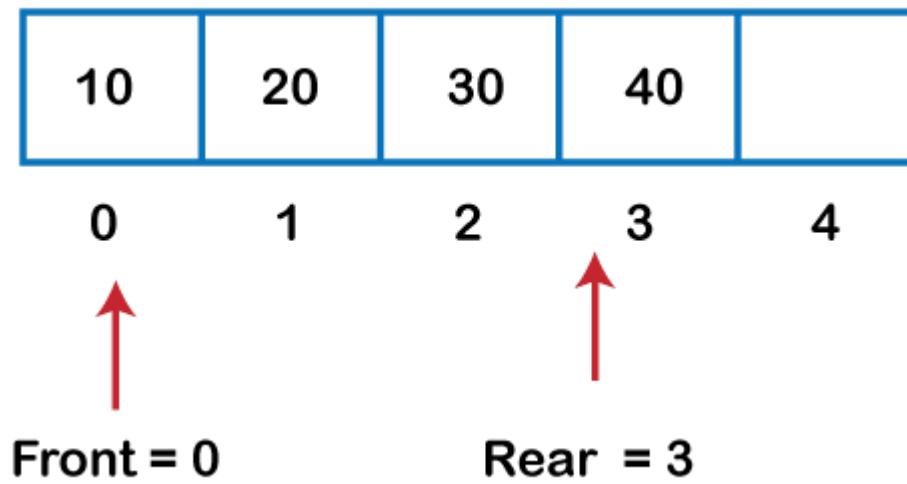
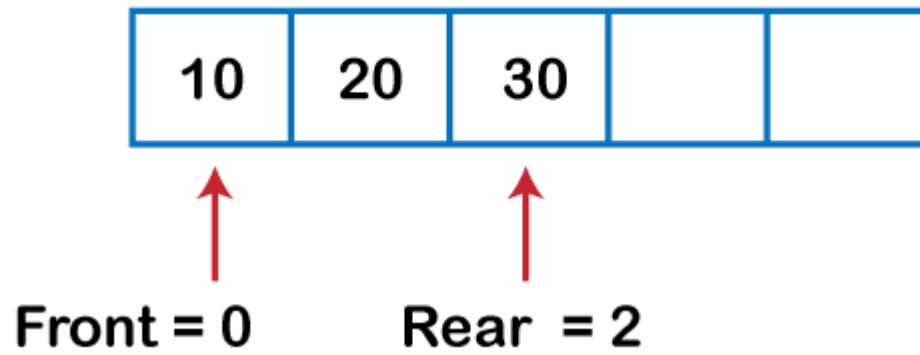
Front = -1

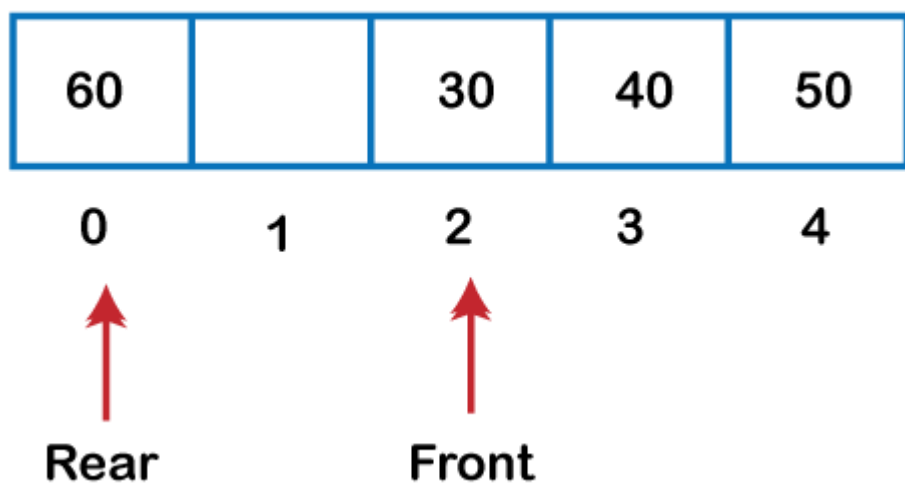
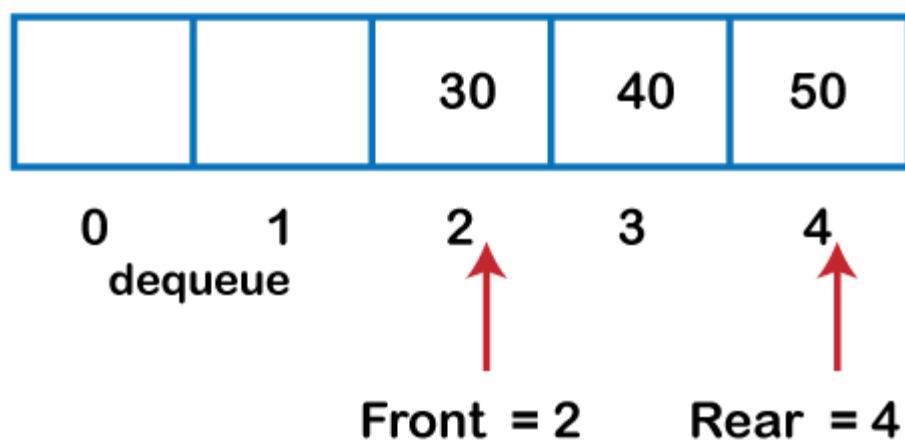
Rear = -1

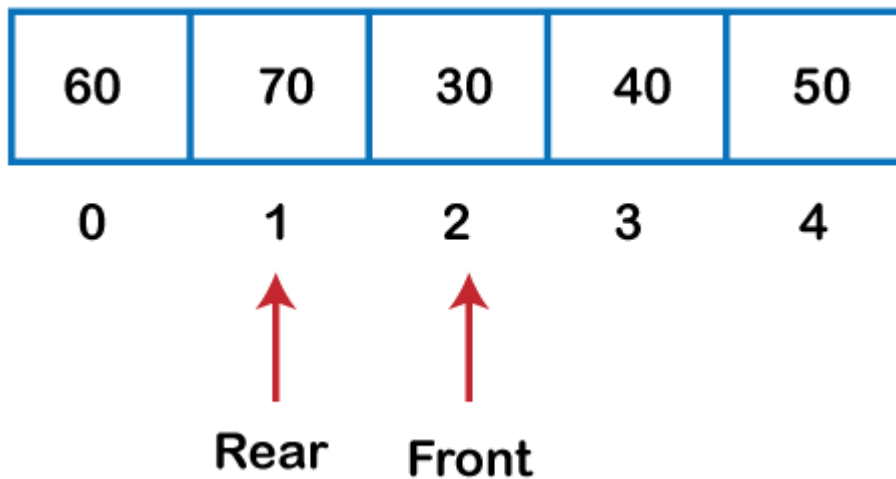


Front = 0

Rear = 0







Implementation of circular queue using linked list

As we know that linked list is a linear data structure that stores two parts, i.e., data part and the address part where address part contains the address of the next node. Here, linked list is used to implement the circular queue; therefore, the linked list follows the properties of the Queue. When we are implementing the circular queue using linked list then both the *enqueue* and *dequeue* operations take $O(1)$ time.