Linear 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.

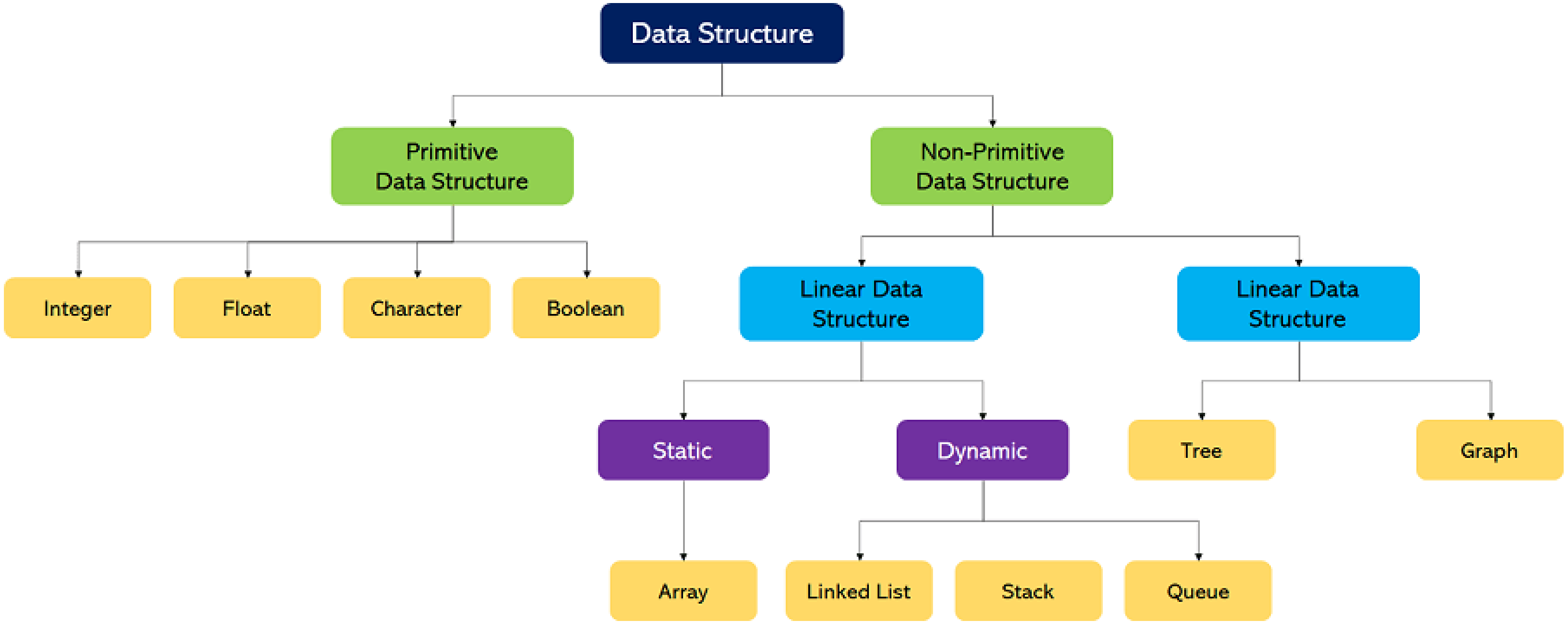
Data Structure

- Data Structure is a branch of Computer Science. The study of data structure allows us to understand the organization of data and the management of the data flow in order to increase the efficiency of any process or program.
- Data Structure is a particular way of storing and organizing data in the memory of the computer so that these data can easily be retrieved and efficiently utilized in the future when required.
- The data can be managed in various ways, like the logical or mathematical model for a specific organization of data is known as a data structure.

Data Structure: Terminologies

- Data: We can define data as an elementary value or a collection of values. For example, the Employee's name and ID are the data related to the Employee.
- Data Items: A Single unit of value is known as Data Item.
- Group Items: Data Items that have subordinate data items are known as Group Items. For example, an employee's name can have a first, middle, and last name.
- Elementary Items: Data Items that are unable to divide into sub-items are known as Elementary Items. For example, the ID of an Employee.
- Entity and Attribute: A class of certain objects is represented by an Entity. It consists of different Attributes. Each Attribute symbolizes the specific property of that Entity. For example,

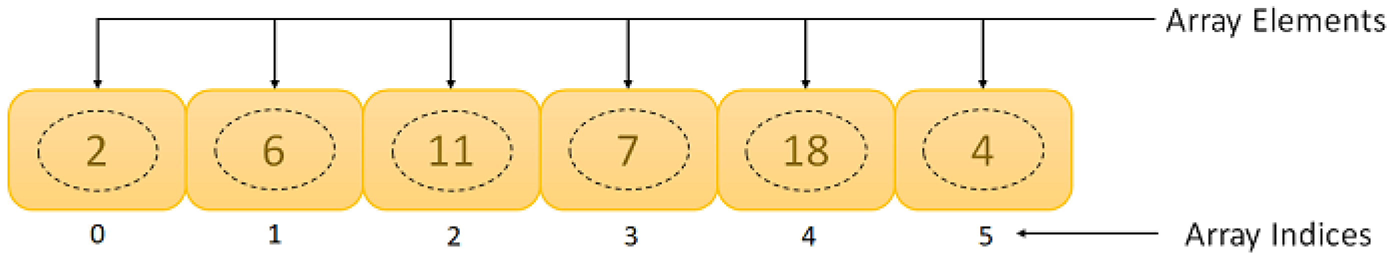
Data Structure



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.

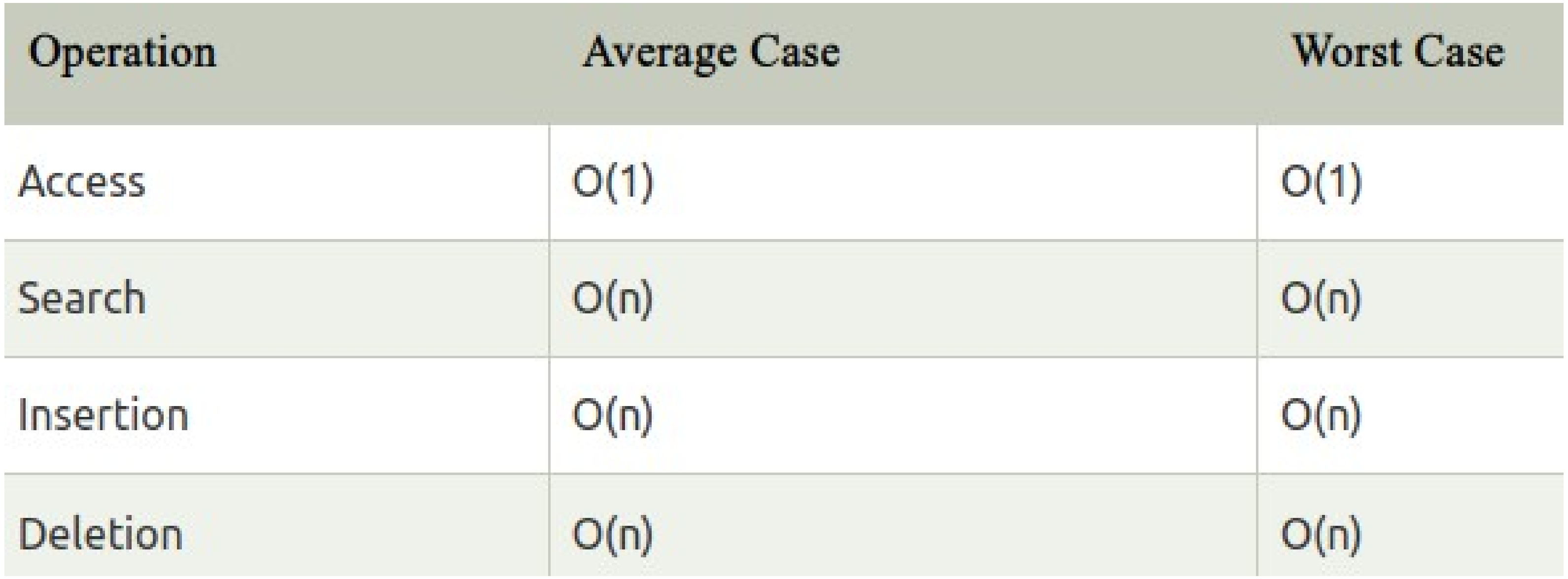
Array: Memory Allocation

- 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

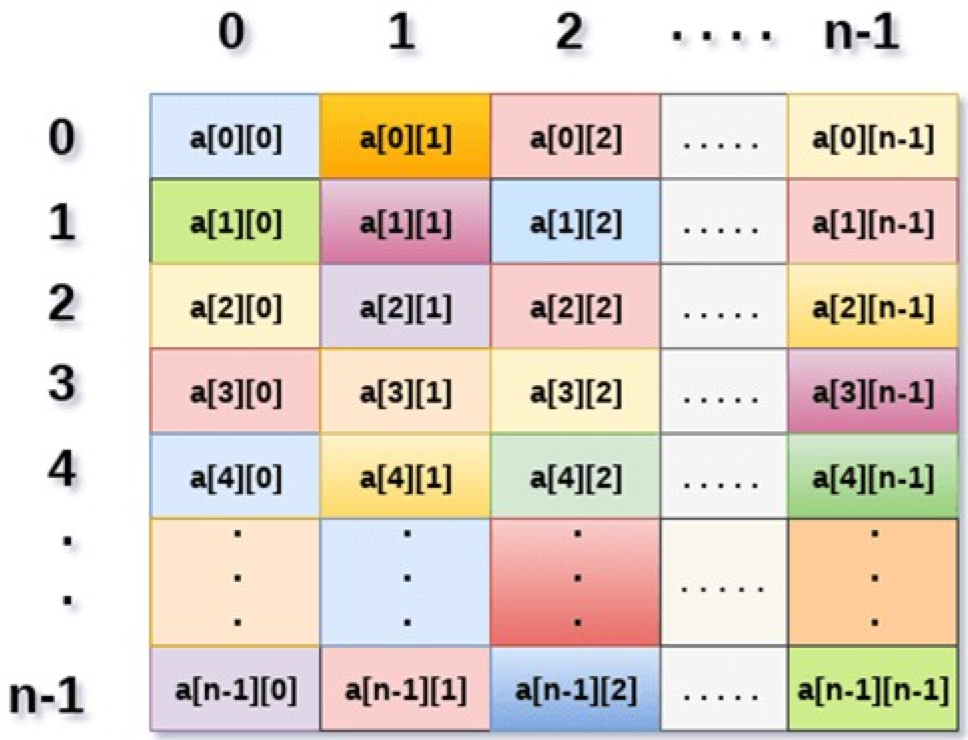


Operation	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

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;
    }
}
```

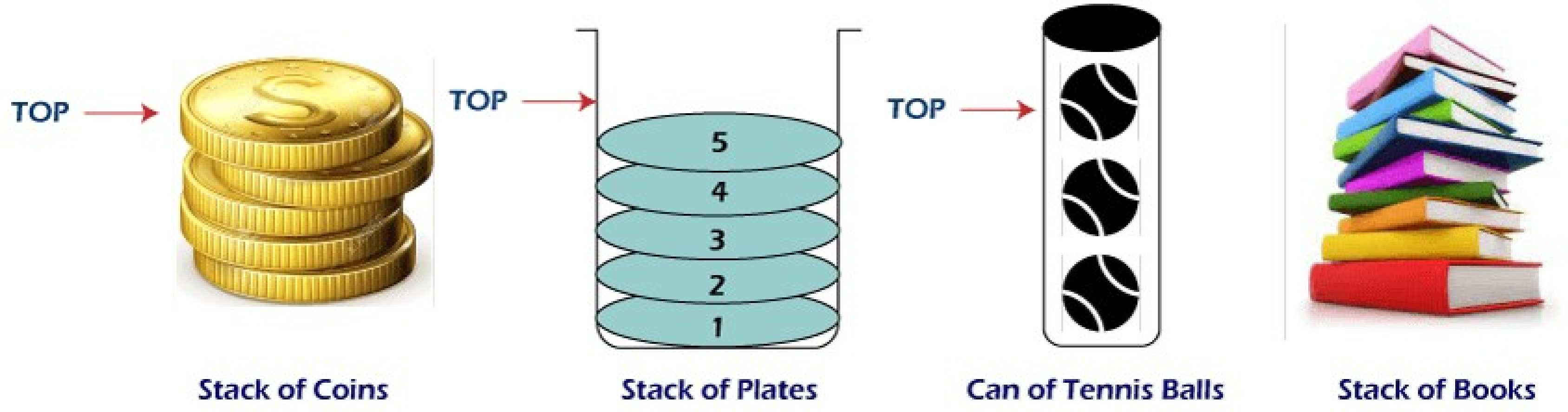
Stack

- 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.

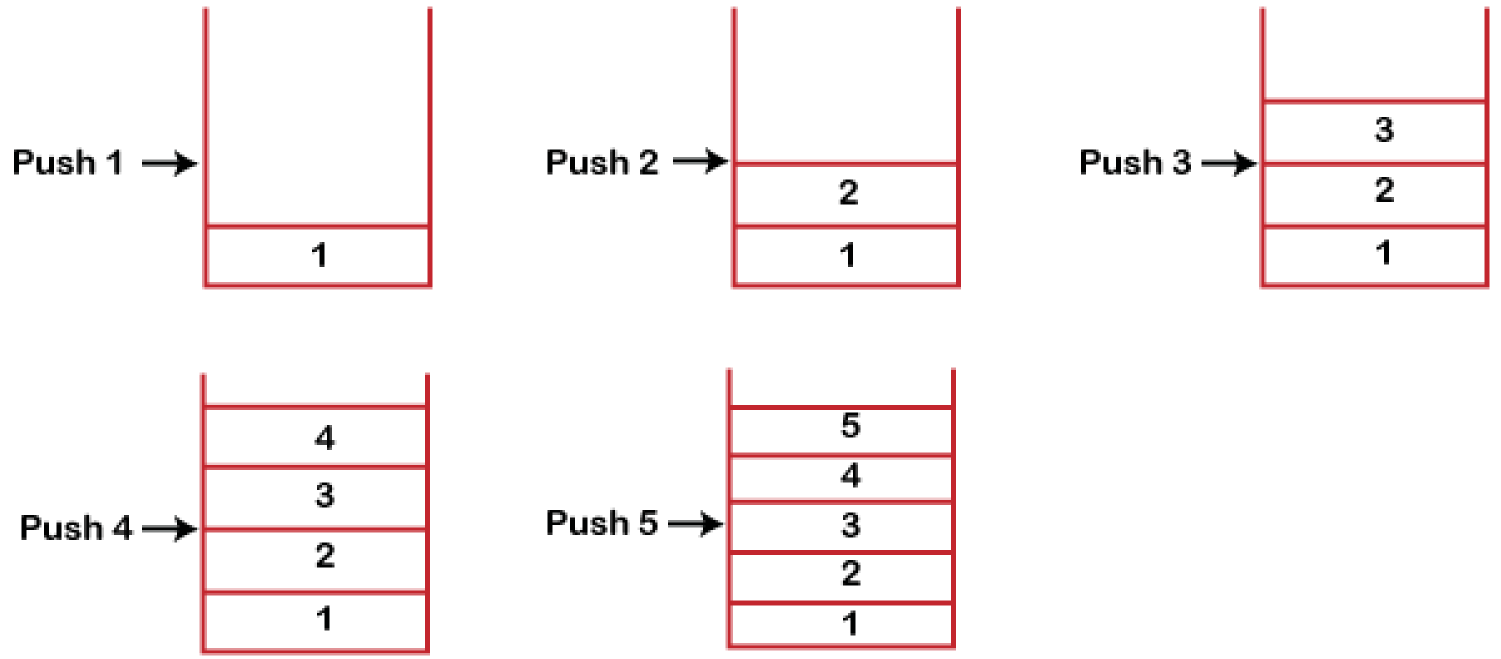
Stack

- 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



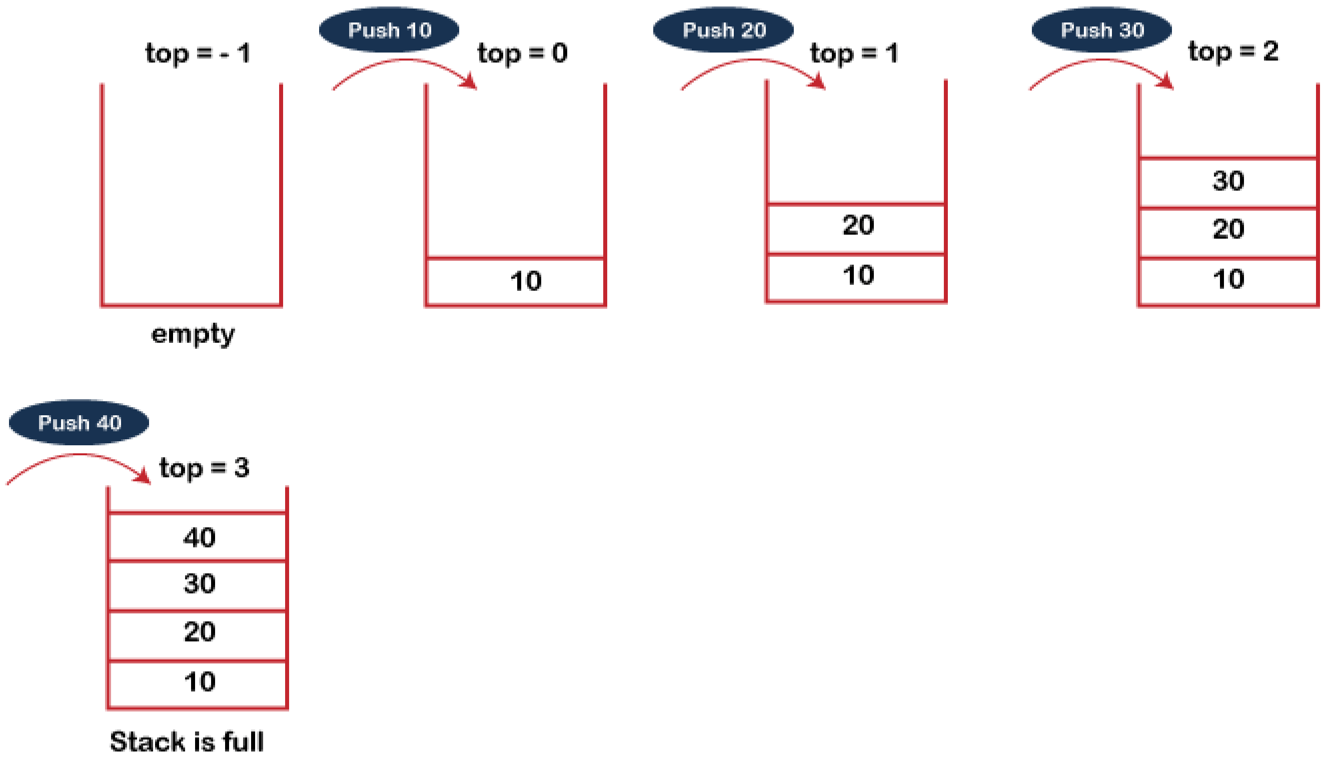
Stack Operations

- 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.

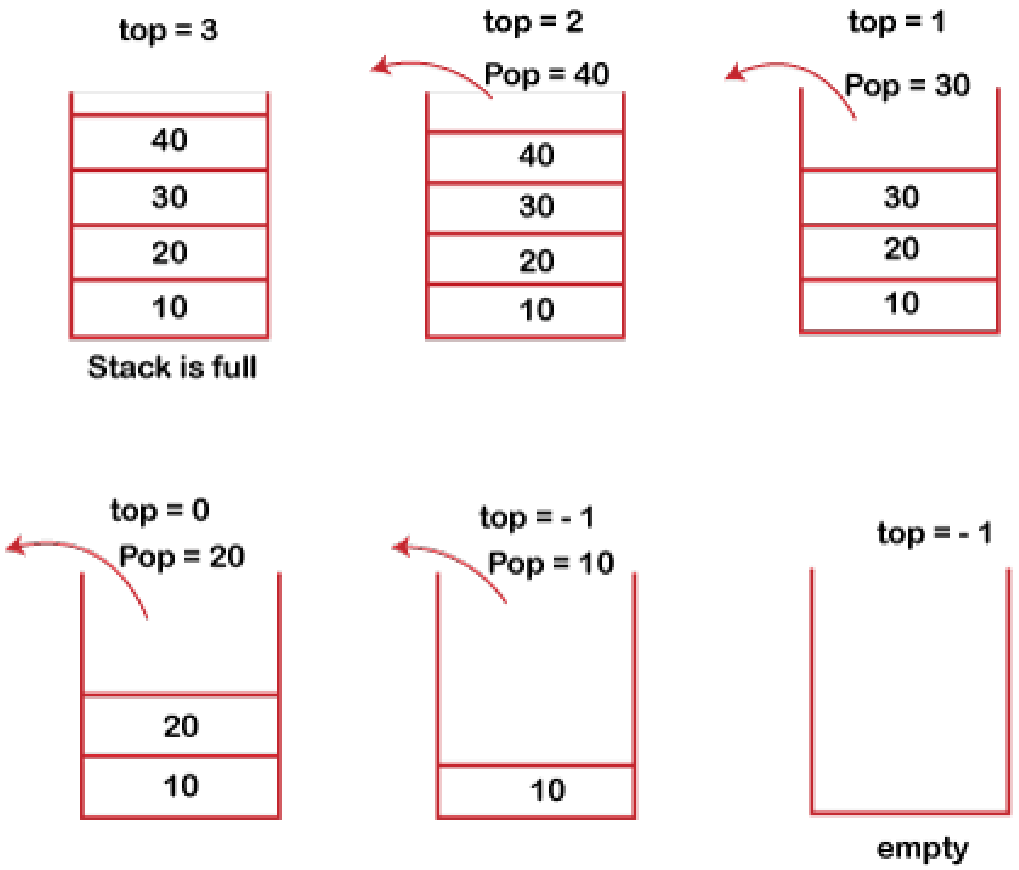
Push Operation



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.

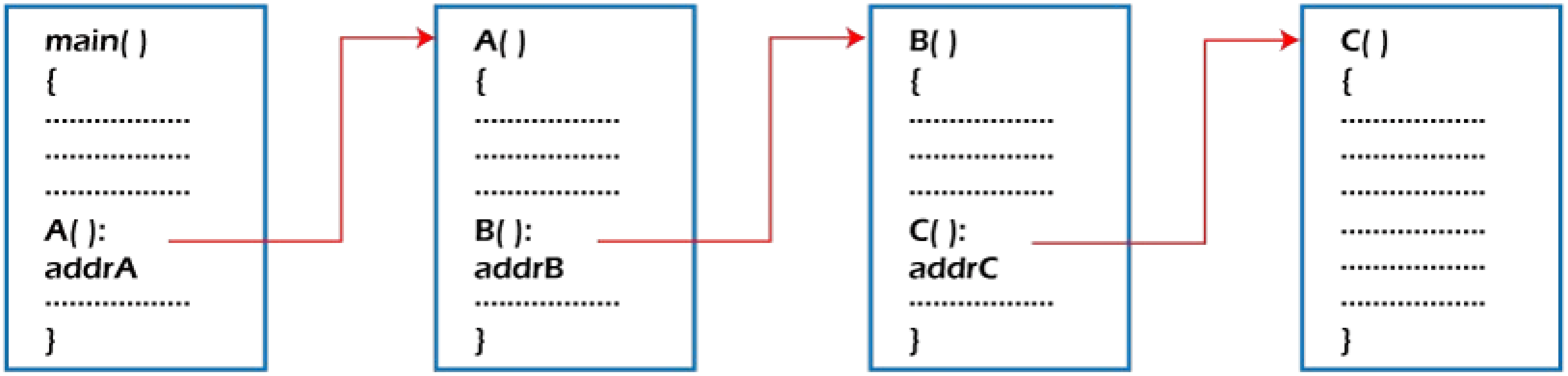
Pop operation



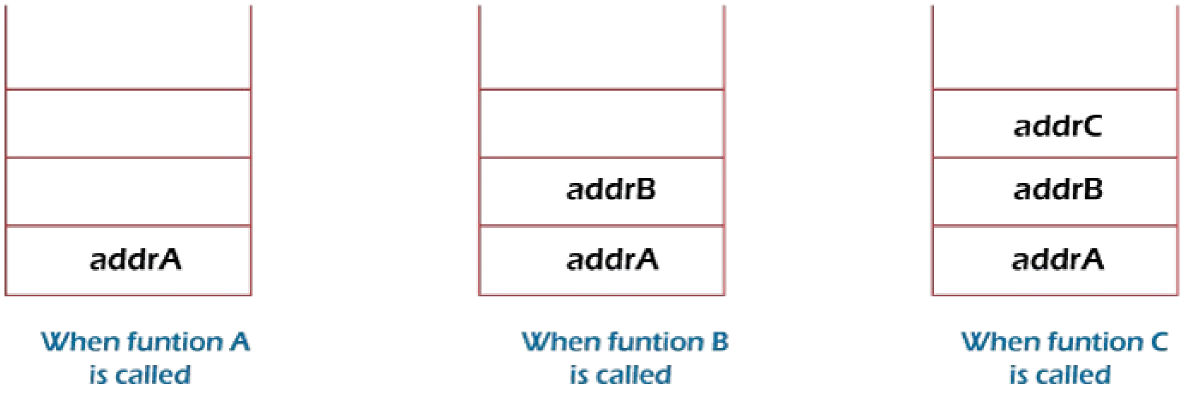
Stack Applications

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

Function Call

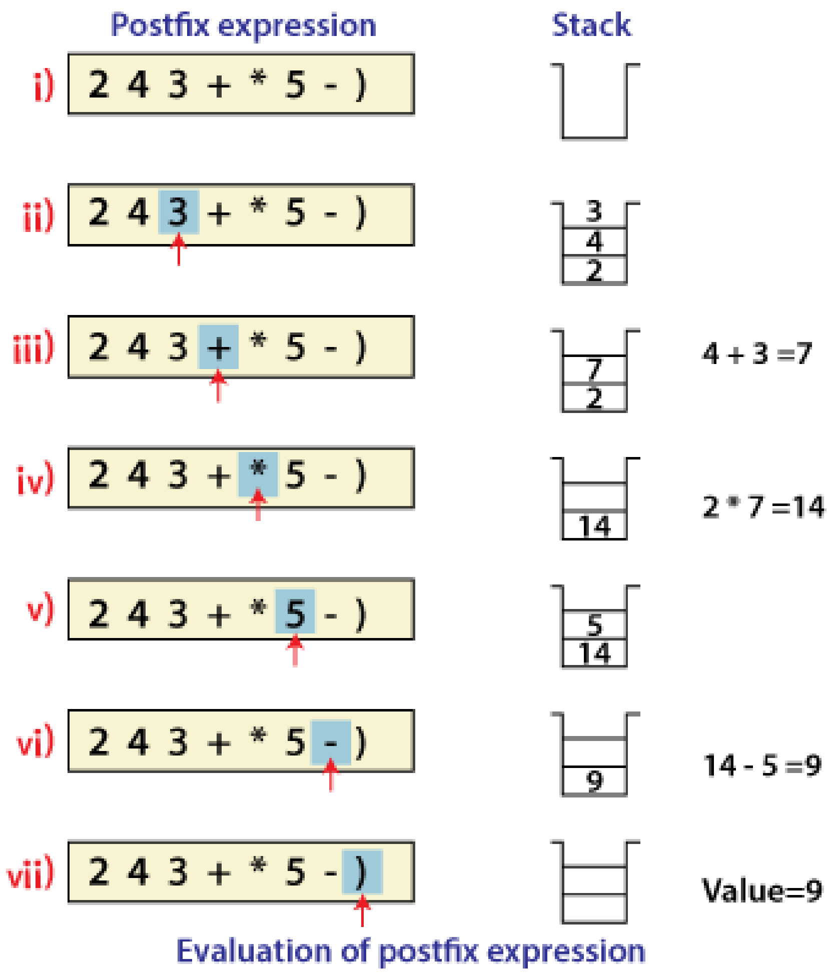


Function call



Different states of stack

Expression Evaluation



Evaluation of postfix expression

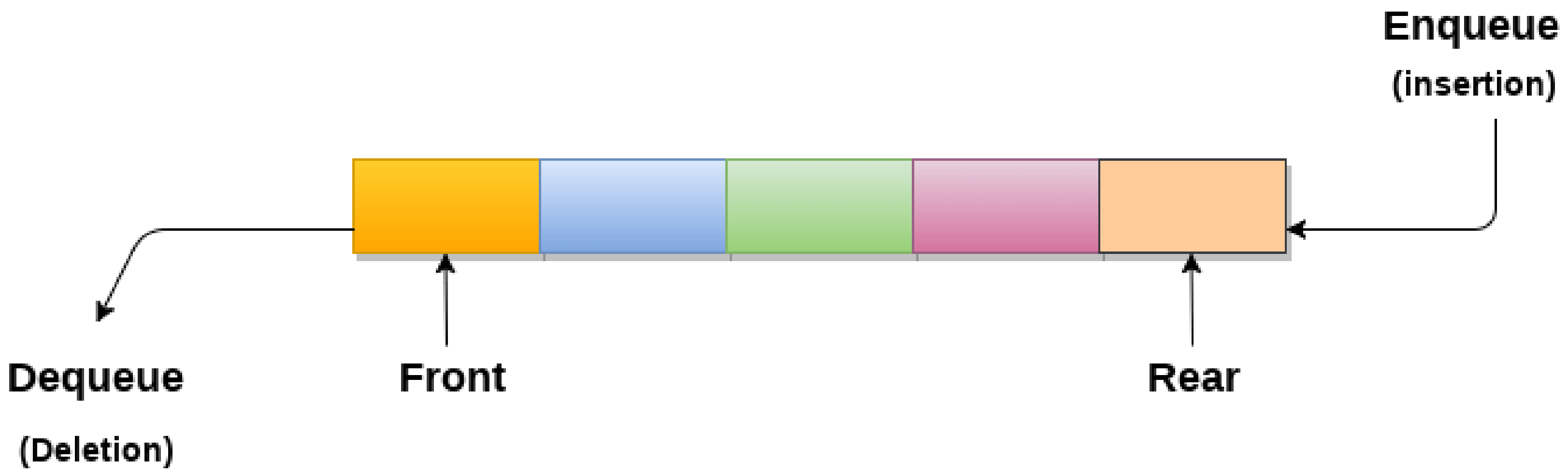
Stack

- 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.

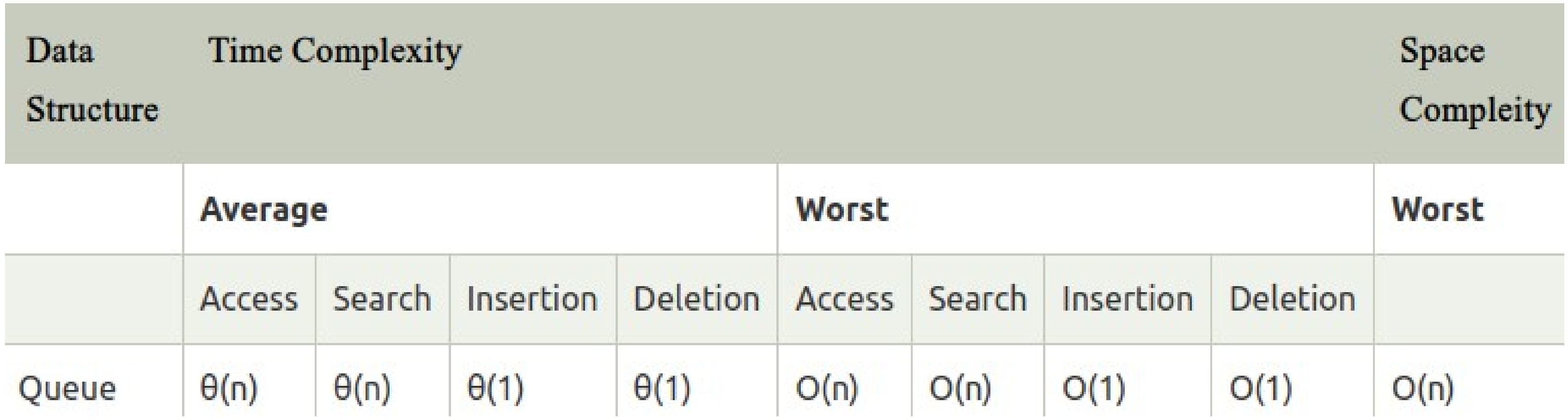
Queue



Queue

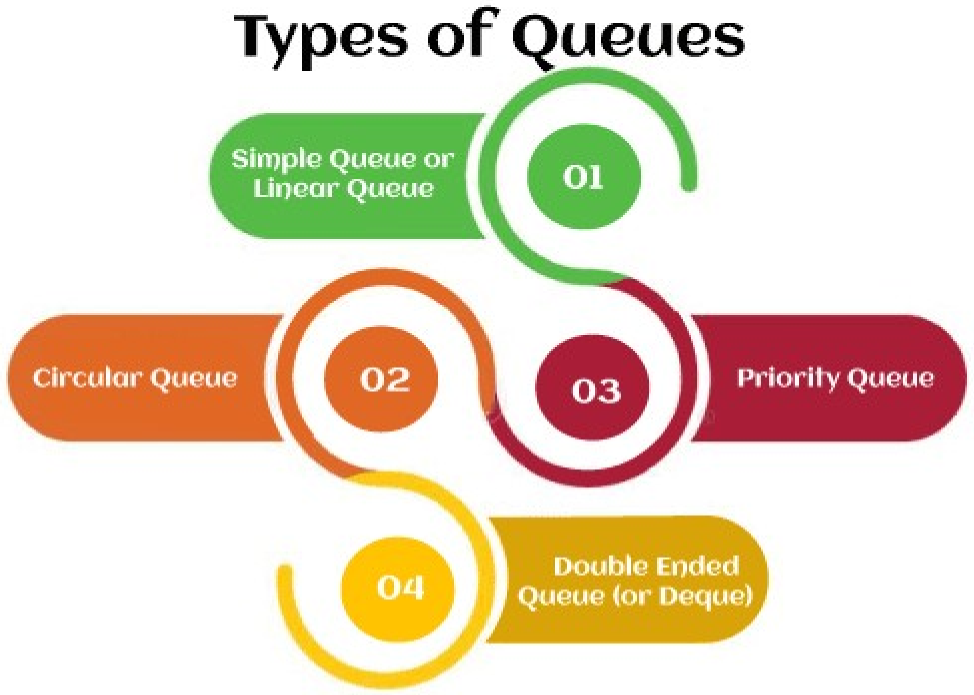
- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.
- Queues are used as buffers in most of the 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

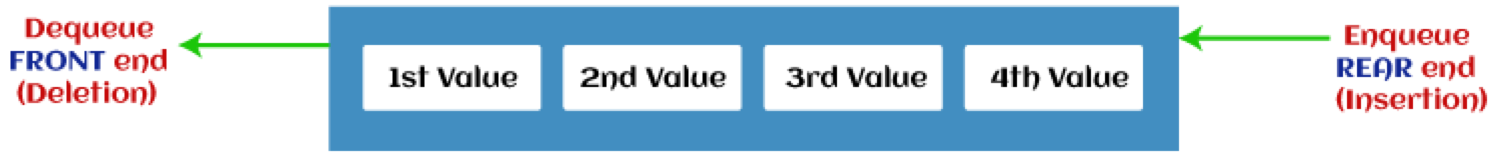


Data Structure	Time Complexity								Space Compleity
	Average				**Worst**				**Worst**
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

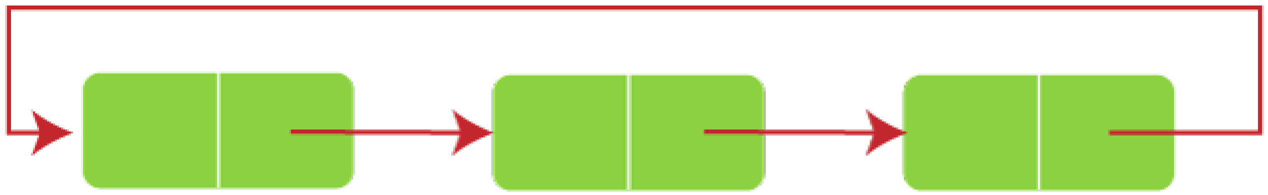
Queue: Types


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

Simple Queue

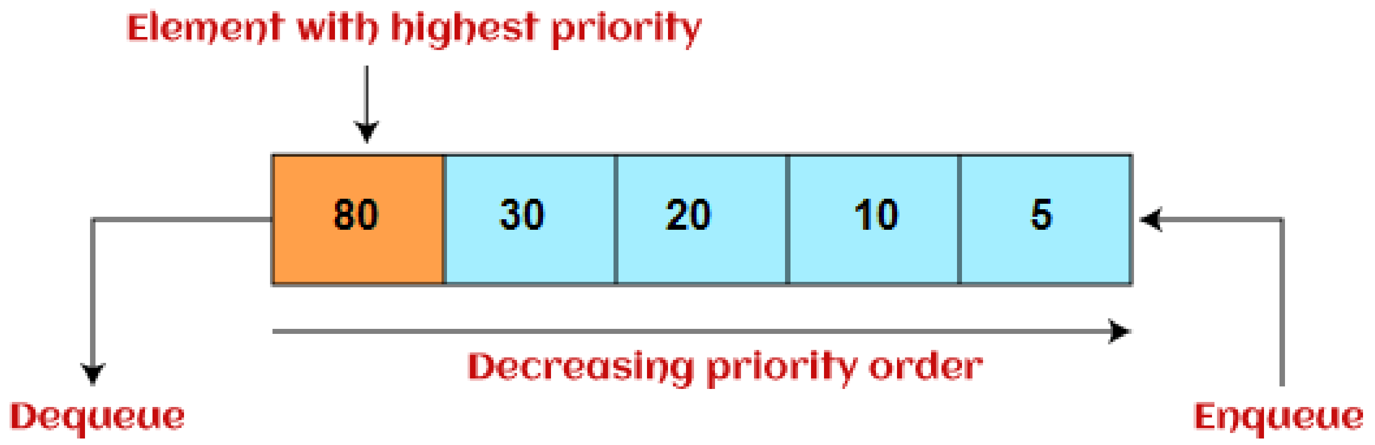


Circular Queue

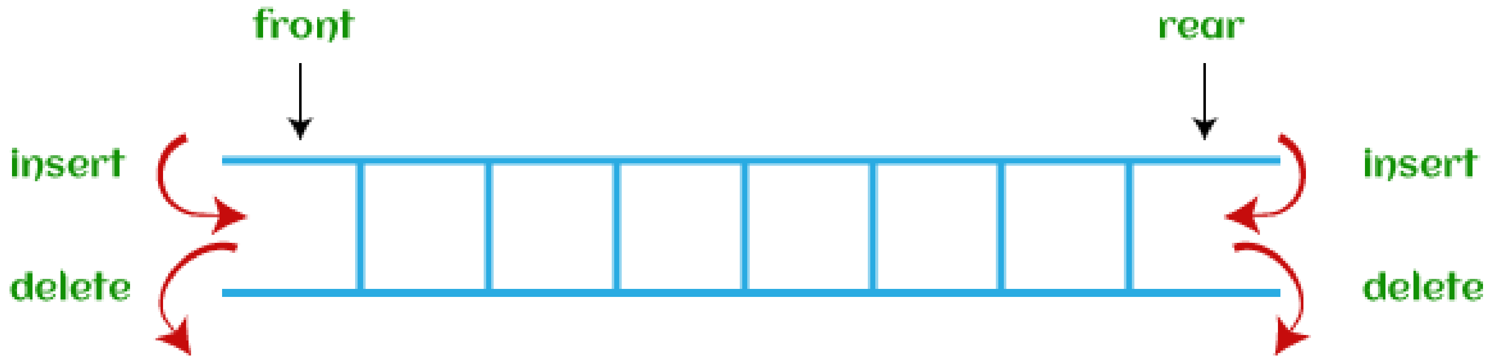


Circular Queue

Priority Queue



DeQueue



double ended queue

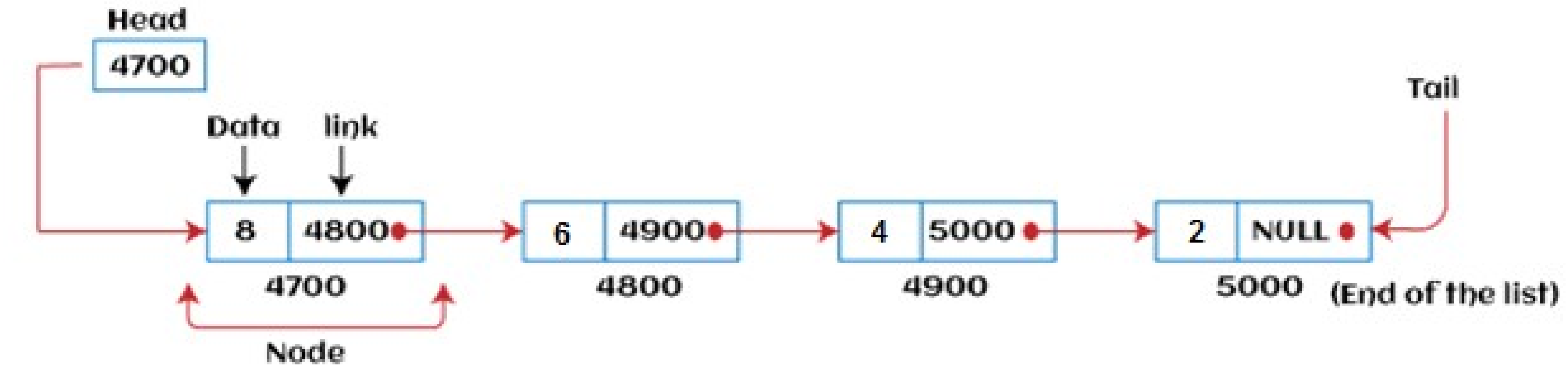
Queue

- Example.

Linked List

- Linked list is a linear data structure that includes a series of connected nodes.
- Linked list can be defined as the nodes that are randomly stored in the memory.
- A node in the linked list contains two parts, i.e., first is the data part and second is the address part.
- The last node of the list contains a pointer to the null.
- After array, linked list is the second most used data structure.
- In a linked list, every link contains a connection to another link.

Linked List


Head
4700
Data
link
8
4800
6
4900
4
5000
2
NULL
Tail
4700
4800
4900
5000
(End of the list)
Node

Linked List: Why?

- Linked list is a data structure that overcomes the limitations of arrays. Let's first see some of the limitations of arrays -
  - The size of the array must be known in advance before using it in the program.
  - Increasing the size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.
  - All the elements in the array need to be contiguously stored in the memory. Inserting an element in the array needs shifting of all its predecessors.

Linked List

- Linked list is useful because -
  - It allocates the memory dynamically. All the nodes of the linked list are non-contiguously stored in the memory and linked together with the help of pointers.
  - In linked list, size is no longer a problem since we do not need to define its size at the time of declaration.
  - List grows as per the program's demand and limited to the available memory space.

Linked List

- It is simple to declare an array, as it is of single type, while the declaration of linked list is a bit more typical than array. Linked list contains two parts, and both are of different types, i.e., one is the simple variable, while another is the pointer variable.
- We can declare the linked list by using the user-defined data type structure.
- The declaration of linked list is given as follows -

```
struct node
{
    int data;
    struct node *next;
}
```

Linked List: Advantages

- Dynamic data structure - The size of the linked list may vary according to the requirements. Linked list does not have a fixed size.
- Insertion and deletion - Unlike arrays, insertion, and deletion in linked list is easier. Array elements are stored in the consecutive location, whereas the elements in the linked list are stored at a random location. To insert or delete an element in an array, we have to shift the elements for creating the space. Whereas, in linked list, instead of shifting, we just have to update the address of the pointer of the node.

Linked List: Advantages

- Memory efficient - The size of a linked list can grow or shrink according to the requirements, so memory consumption in linked list is efficient.
- Implementation - We can implement both stacks and queues using linked list.

Linked List: Disadvantages

- Memory usage - In linked list, node occupies more memory than array. Each node of the linked list occupies two types of variables, i.e., one is a simple variable, and another one is the pointer variable.
- Traversal - Traversal is not easy in the linked list. If we have to access an element in the linked list, we cannot access it randomly, while in case of array we can randomly access it by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.
- Reverse traversing - Backtracking or reverse traversing is difficult in a linked list. In a doubly-linked list, it is easier but requires more memory to store the back pointer.

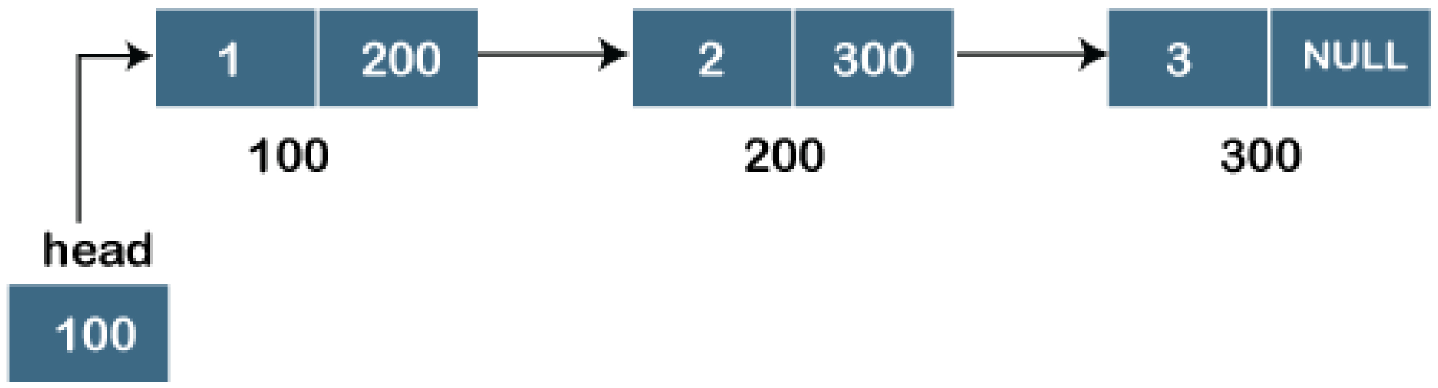
Linked List: Types

- Singly Linked list
- Doubly Linked list
- Circular Linked list
- Doubly Circular Linked list

Singly Linked List

- It is the commonly used linked list in programs. If we are talking about the linked list, it means it is a singly linked list.
- The singly linked list is a data structure that contains two parts, i.e., one is the data part, and the other one is the address part, which contains the address of the next or the successor node.
- The address part in a node is also known as a pointer.

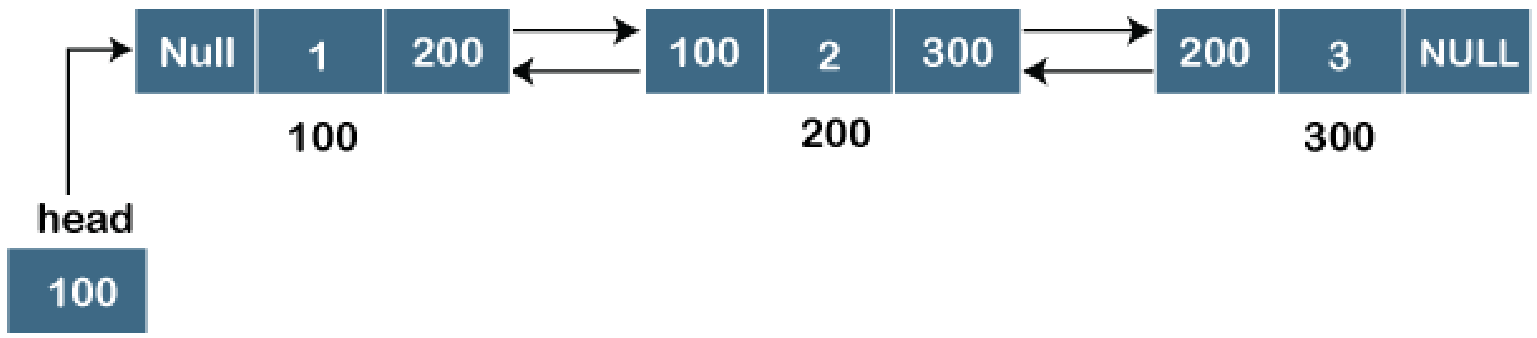
Singly Linked List



Doubly Linked List

- As the name suggests, the doubly linked list contains two pointers.
- We can define the doubly linked list as a linear data structure with three parts: the data part and the other two address part.
- In other words, a doubly linked list is a list that has three parts in a single node, includes one data part, a pointer to its previous node, and a pointer to the next node.

Doubly Linked List



Doubly Linked List

- the node in a doubly-linked list has two address parts; one part stores the address of the next while the other part of the node stores the previous node's address.
- The initial node in the doubly linked list has the NULL value in the address part, which provides the address of the previous node.
- Representation of the node in a doubly linked list

```
struct node
{
     int data;
     struct node *next;
    struct node *prev;
}
```

Circular Linked List

- A circular linked list is a variation of a singly linked list. The only difference between the singly linked list and a circular linked list is that the last node does not point to any node in a singly linked list, so its link part contains a NULL value.
- On the other hand, the circular linked list is a list in which the last node connects to the first node, so the link part of the last node holds the first node's address.
- The circular linked list has no starting and ending node.

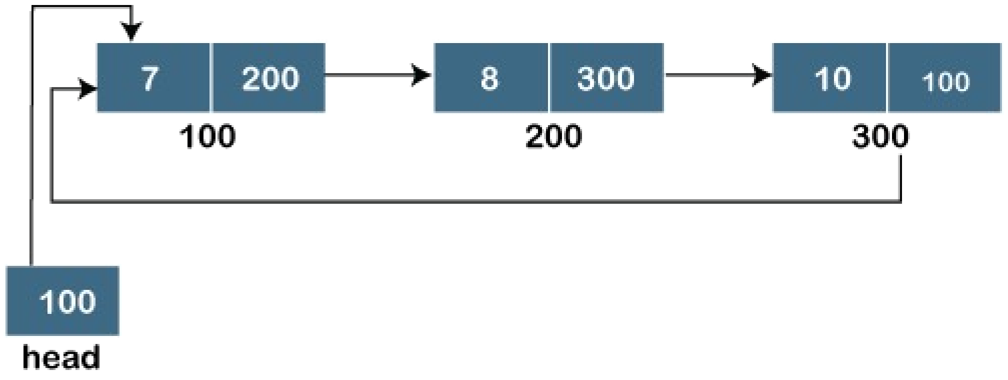
Circular Linked List

- We can traverse in any direction, i.e., either backward or forward. The diagrammatic representation of the circular linked list is shown below:

```
struct node
{
        int data;
        struct node *next;
}
```

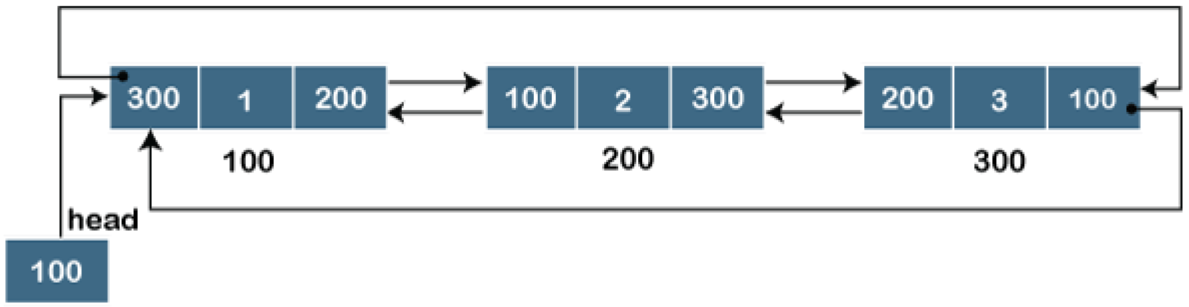
Circular Linked List

- A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node.
- The representation of the circular linked list will be similar to the singly linked list, as shown below:



Circular Doubly Linked List

- The doubly circular linked list has the features of both the circular linked list and doubly linked list.



Circular Doubly Linked List

- The representation of the doubly circular linked list in which the last node is attached to the first node and thus creates a circle.
- It is a doubly linked list also because each node holds the address of the previous node also.
- The main difference between the doubly linked list and doubly circular linked list is that the doubly circular linked list does not contain the NULL value in the previous field of the node.
- As the doubly circular linked contains three parts, i.e., two address parts and one data part so its representation is similar to the doubly linked list.

Circular Doubly Linked List

```
struct node
{
        int data;
        struct node *next;
        struct node *prev;
}
```

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