Unit - 1
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.

The scope of a particular data model depends on two factors:

1. First, it must be loaded enough into the structure to reflect the definite correlation of the data with a real-world object.
2. Second, the formation should be so straightforward that one can adapt to process the data efficiently whenever necessary.

Some examples of Data Structures are Arrays, Linked Lists, Stack, Queue, Trees, etc. Data Structures are widely used in almost every aspect of Computer Science, i.e., Compiler Design, Operating Systems, Graphics, Artificial Intelligence, and many more.

Data Structures are the main part of many Computer Science Algorithms as they allow the programmers to manage the data in an effective way. It plays a crucial role in improving the performance of a program or software, as the main objective of the software is to store and retrieve the user's data as fast as possible.

Basic Terminologies related to Data Structures

Data Structures are the building blocks of any software or program. Selecting the suitable data structure for a program is an extremely challenging task for a programmer.

The following are some fundamental terminologies used whenever the data structures are involved:

1. **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.
2. **Data Items:** A Single unit of value is known as Data Item.
3. **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.
4. **Elementary Items:** Data Items that are unable to divide into sub-items are known as Elementary Items. For example, the ID of an Employee.
5. **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,

Attributes	ID	Name	Gender	Job Title
Values	1234	Stacey M. Hill	Female	Software Developer

Entities with similar attributes form an **Entity Set**. Each attribute of an entity set has a range of values, the set of all possible values that could be assigned to the specific attribute.

The term "information" is sometimes utilized for data with given attributes of meaningful or processed data.

1. **Field:** A single elementary unit of information symbolizing the Attribute of an Entity is known as Field.
2. **Record:** A collection of different data items are known as a Record. For example, if we talk about the employee entity, then its name, id, address, and job title can be grouped to form the record for the employee.
3. **File:** A collection of different Records of one entity type is known as a File. For example, if there are 100 employees, there will be 25 records in the related file containing data about each employee.

Understanding the Need for Data Structures

As applications are becoming more complex and the amount of data is increasing every day, which may lead to problems with data searching, processing speed, multiple requests handling, and many more. Data Structures support different methods to organize, manage, and store data efficiently. With the help of Data Structures, we can easily traverse the data items. Data Structures provide Efficiency, Reusability, and Abstraction.

Why should we learn Data Structures?

1. Data Structures and Algorithms are two of the key aspects of Computer Science.
2. Data Structures allow us to organize and store data, whereas Algorithms allow us to process that data meaningfully.
3. Learning Data Structures and Algorithms will help us become better Programmers.
4. We will be able to write code that is more effective and reliable.
5. We will also be able to solve problems more quickly and efficiently.

Understanding the Objectives of Data Structures

Data Structures satisfy two complementary objectives:

1. **Correctness:** Data Structures are designed to operate correctly for all kinds of inputs based on the domain of interest. In order words, correctness forms the primary objective of Data Structure, which always depends upon the problems that the Data Structure is meant to solve.
2. **Efficiency:** Data Structures also requires to be efficient. It should process the data quickly without utilizing many computer resources like memory space. In a real-time state, the efficiency of a data structure is a key factor in determining the success and failure of the process.

Understanding some Key Features of Data Structures

Some of the Significant Features of Data Structures are:

1. **Robustness:** Generally, all computer programmers aim to produce software that yields correct output for every possible input, along with efficient execution on all hardware platforms. This type of robust software must manage both valid and invalid inputs.
2. **Adaptability:** Building software applications like Web Browsers, Word Processors, and Internet Search Engine include huge software systems that require correct and efficient working or execution for many years. Moreover, software evolves due to emerging technologies or ever-changing market conditions.
3. **Reusability:** The features like Reusability and Adaptability go hand in hand. It is known that the programmer needs many resources to build any software, making it a costly enterprise. However, if the software is developed in a reusable and adaptable way, then it can be applied in most future applications. Thus, by executing quality data structures, it is possible to build reusable software, which appears to be cost-effective and timesaving.

Classification of Data Structures

A Data Structure delivers a structured set of variables related to each other in various ways. It forms the basis of a programming tool that signifies the relationship between the data elements and allows programmers to process the data efficiently.

We can classify Data Structures into two categories:

1. Primitive Data Structure
2. Non-Primitive Data Structure

The following figure shows the different classifications of Data Structures.

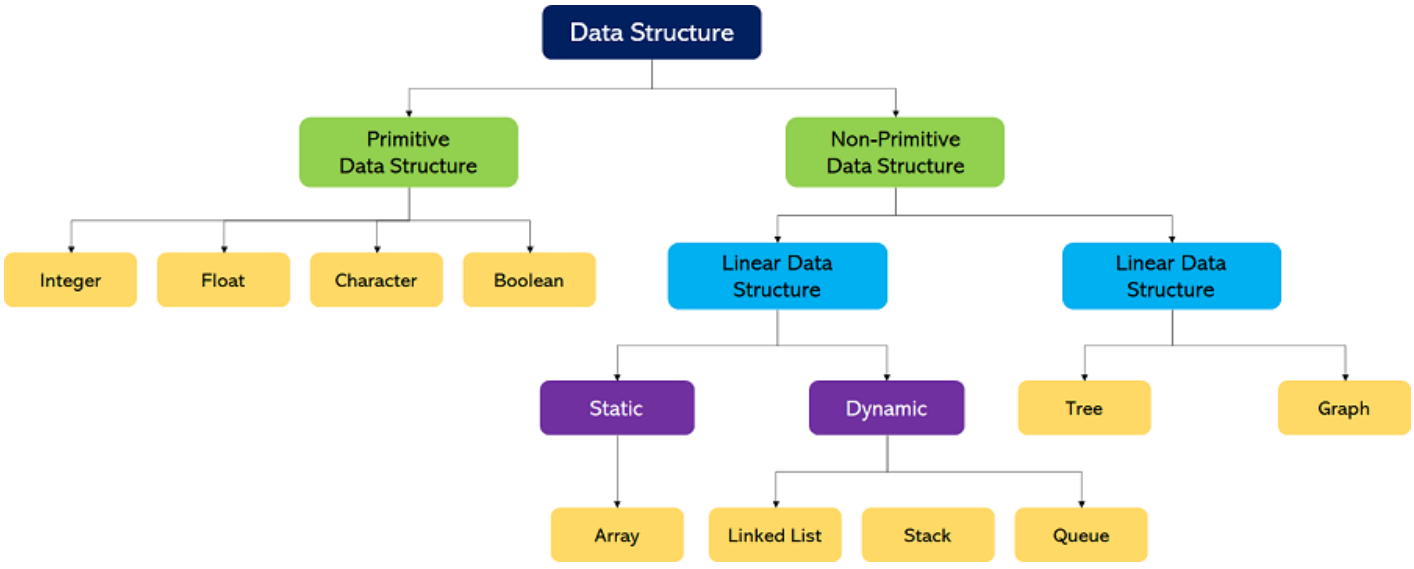


Figure : Classifications of Data Structures

Primitive Data Structures

1. **Primitive Data Structures** are the data structures consisting of the numbers and the characters that come **in-built** into programs.
2. These data structures can be manipulated or operated directly by machine-level instructions.

3. Basic data types like **Integer, Float, Character**, and **Boolean** come under the Primitive Data Structures.
4. These data types are also called **Simple data types**, as they contain characters that can't be divided further

Non-Primitive Data Structures

1. **Non-Primitive Data Structures** are those data structures derived from Primitive Data Structures.
2. These data structures can't be manipulated or operated directly by machine-level instructions.
3. The focus of these data structures is on forming a set of data elements that is either **homogeneous** (same data type) or **heterogeneous** (different data types).
4. Based on the structure and arrangement of data, we can divide these data structures into two sub-categories -
    a. Linear Data Structures
    b. Non-Linear Data Structures

Linear Data Structures

A data structure that preserves a linear connection among its data elements is known as a Linear Data Structure. The arrangement of the data is done linearly, where each element consists of the successors and predecessors except the first and the last data element. However, it is not necessarily true in the case of memory, as the arrangement may not be sequential.

Based on memory allocation, the Linear Data Structures are further classified into two types:

1. **Static Data Structures:** The data structures having a fixed size are known as Static Data Structures. The memory for these data structures is allocated at the compiler time, and their size cannot be changed by the user after being compiled; however, the data stored in them can be altered. The **Array** is the best example of the Static Data Structure as they have a fixed size, and its data can be modified later.
2. **Dynamic Data Structures:** The data structures having a dynamic size are known as Dynamic Data Structures. The memory of these data structures is allocated at the run time, and their size varies during the run time of the code. Moreover, the user can change the size as well as the data elements stored in these data structures at the run time of the code. **Linked Lists, Stacks**, and **Queues** are common examples of dynamic data structures

Types of Linear Data Structures

The following is the list of Linear Data Structures that we generally use:

1. Arrays

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.

An Array is a list of elements where each element has a unique place in the list. The data elements of the array share the same variable name; however, each carries a different index number called a subscript. We can access any data element from the list with the help of its location in the list. Thus, the key feature of the arrays to understand is that the data is stored in contiguous memory locations, making it possible for the users to traverse through the data elements of the array using their respective indexes.

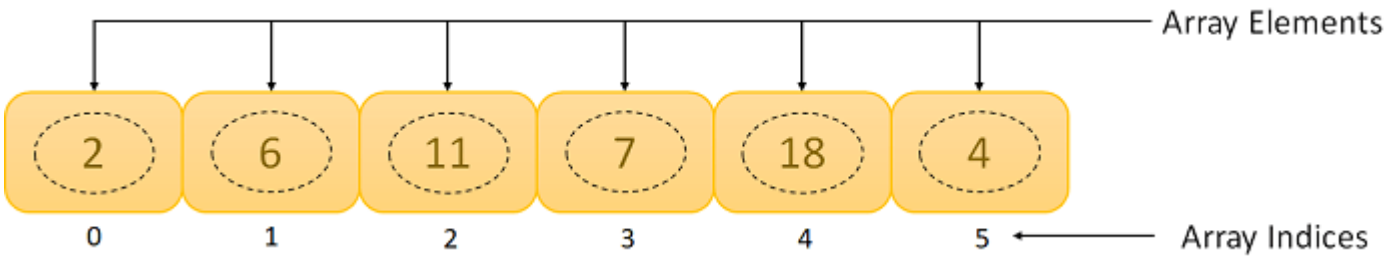


Figure . An Array

Arrays can be classified into different types:

a. **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.

b. **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.

c. **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.

Some Applications of Array:

a. We can store a list of data elements belonging to the same data type.

b. Array acts as an auxiliary storage for other data structures.

c. The array also helps store data elements of a binary tree of the fixed count.

d. Array also acts as a storage of matrices.

2. Linked Lists

A **Linked List** is another example of a linear data structure used to store a collection of data elements dynamically. Data elements in this data structure are represented by the Nodes, connected using links or pointers. Each node contains two fields, the information field consists of the actual data, and the pointer field consists of the address of the subsequent nodes in the list. The pointer of the last node of the linked list consists of a null pointer, as it points to nothing. Unlike the Arrays, the user can dynamically adjust the size of a Linked List as per the requirements.

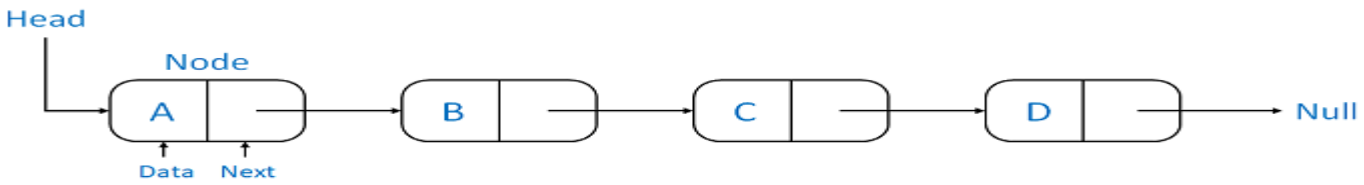


Figure . A Linked List

Linked Lists can be classified into different types:

a. **Singly Linked List:** A Singly Linked List is the most common type of Linked List. Each node has data and a pointer field containing an address to the next node.

b. **Doubly Linked List:** A Doubly Linked List consists of an information field and two pointer fields. The information field contains the data. The first pointer field contains an address of the previous node, whereas another pointer field contains a reference to the next node. Thus, we can go in both directions (backward as well as forward).

c. **Circular Linked List:** The Circular Linked List is similar to the Singly Linked List. The only key difference is that the last node contains the address of the first node, forming a circular loop in the Circular Linked List.

Some Applications of Linked Lists:

a. The Linked Lists help us implement stacks, queues, binary trees, and graphs of predefined size.

b. We can also implement Operating System's function for dynamic memory management.

c. Linked Lists also allow polynomial implementation for mathematical operations.

d. We can use Circular Linked List to implement Operating Systems or application functions that Round Robin execution of tasks.

e. Circular Linked List is also helpful in a Slide Show where a user requires to go back to the first slide after the last slide is presented.

f. Doubly Linked List is utilized to implement forward and backward buttons in a browser to move forward and backward in the opened pages of a website.

3. Stacks

A **Stack** is a Linear Data Structure that follows the **LIFO** (Last In, First Out) principle that allows operations like insertion and deletion from one end of the Stack, i.e., Top. Stacks can be implemented with the help of contiguous memory, an Array, and non-contiguous memory, a Linked List. Real-life examples of Stacks are piles of books, a deck of cards, piles of money, and many more.

Figure . A Real-life Example of Stack

The above figure represents the real-life example of a Stack where the operations are performed from one end only, like the insertion and removal of new books from the top of the Stack. It implies that the insertion and deletion in the Stack can be done only from the top of the Stack. We can access only the Stack's tops at any given time.

The primary operations in the Stack are as follows:

a. **Push:** Operation to insert a new element in the Stack is termed as Push Operation.

b. **Pop:** Operation to remove or delete elements from the Stack is termed as Pop Operation.

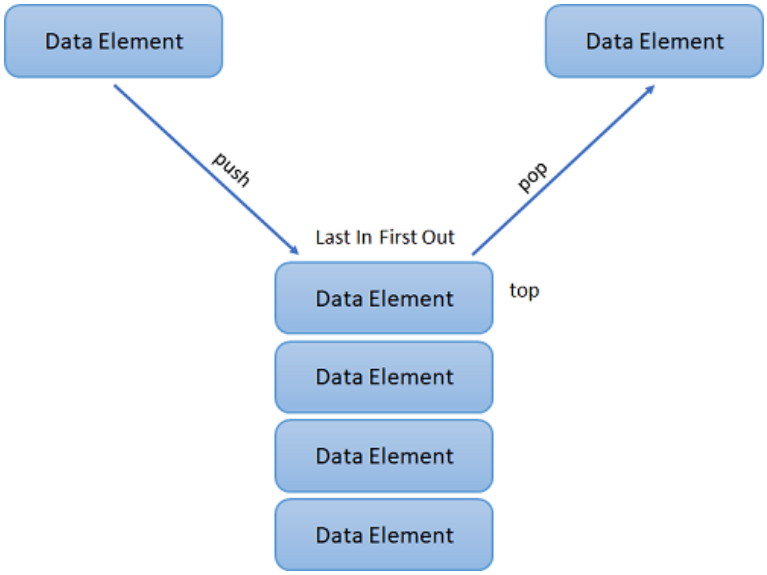


Figure . A Stack

Some Applications of Stacks:

a. The Stack is used as a Temporary Storage Structure for recursive operations.

b. Stack is also utilized as Auxiliary Storage Structure for function calls, nested operations, and deferred/postponed functions.

c. We can manage function calls using Stacks.

d. Stacks are also utilized to evaluate the arithmetic expressions in different programming languages.

e. Stacks are also helpful in converting infix expressions to postfix expressions.

f. Stacks allow us to check the expression's syntax in the programming environment.

g. We can match parenthesis using Stacks.

h. Stacks can be used to reverse a String.

i. Stacks are helpful in solving problems based on backtracking.

j. We can use Stacks in depth-first search in graph and tree traversal.

k. Stacks are also used in Operating System functions.

l. Stacks are also used in UNDO and REDO functions in an edit.

4. Queues

A **Queue** is a linear data structure similar to a Stack with some limitations on the insertion and deletion of the elements. The insertion of an element in a Queue is done at one end, and the removal is done at another or opposite end. Thus, we can conclude that the Queue data structure follows FIFO (First In, First Out) principle to manipulate the data elements. Implementation of Queues can be done using Arrays, Linked Lists, or Stacks. Some real-life examples of Queues are a line at the ticket counter, an escalator, a car wash, and many more.

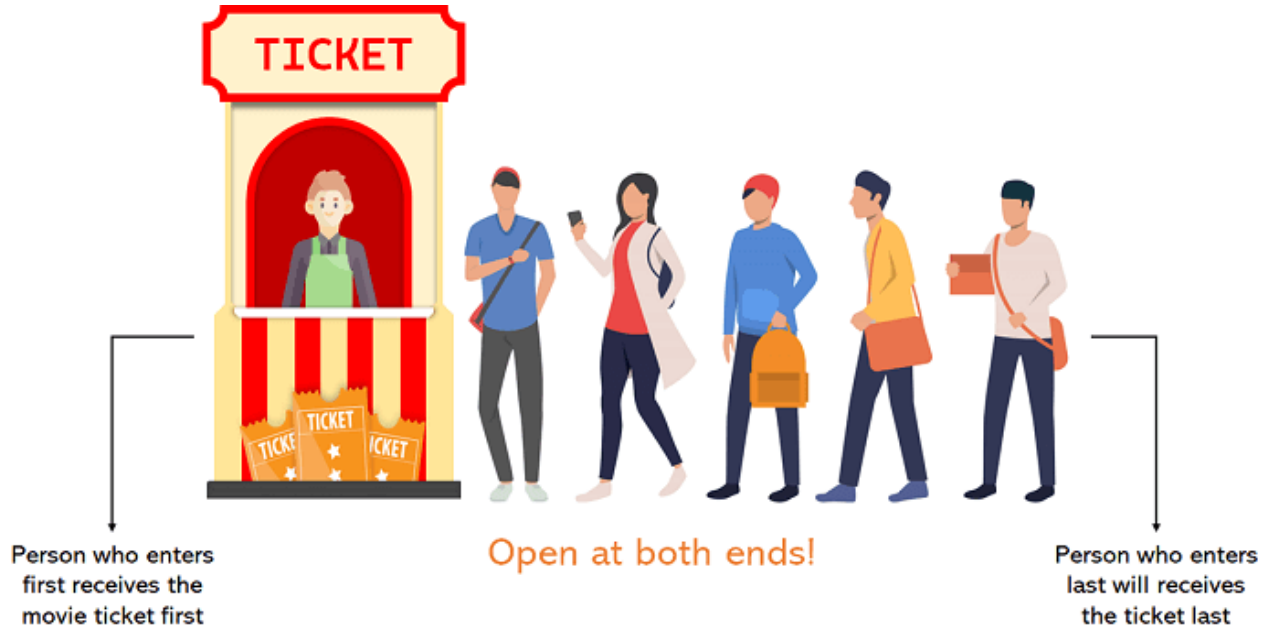


Figure . A Real-life Example of Queue

The above image is a real-life illustration of a movie ticket counter that can help us understand the Queue where the customer who comes first is always served first. The customer arriving last will undoubtedly be served last. Both ends of the Queue are open and can execute different operations. Another example is a food court line where the customer is inserted from the rear end while the customer is removed at the front end after providing the service they asked for.

The following are the primary operations of the Queue:

a. **Enqueue:** The insertion or Addition of some data elements to the Queue is called Enqueue. The element insertion is always done with the help of the rear pointer.

b. **Dequeue:** Deleting or removing data elements from the Queue is termed Dequeue. The deletion of the element is always done with the help of the front pointer.

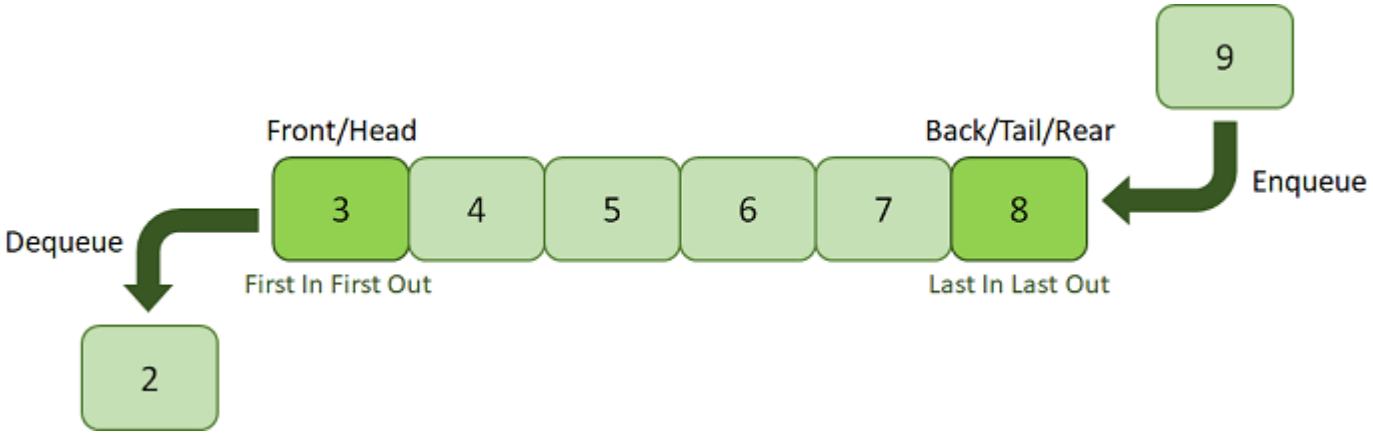


Figure 8. A Queue

Some Applications of Queues:

a. Queues are generally used in the breadth search operation in Graphs.

b. Queues are also used in Job Scheduler Operations of Operating Systems, like a keyboard buffer queue to store the keys pressed by users and a print buffer queue to store the documents printed by the printer.

c. Queues are responsible for CPU scheduling, Job scheduling, and Disk Scheduling.

d. Priority Queues are utilized in file-downloading operations in a browser.

e. Queues are also used to transfer data between peripheral devices and the CPU.

f. Queues are also responsible for handling interrupts generated by the User Applications for the CPU.

Non-Linear Data Structures

Non-Linear Data Structures are data structures where the data elements are not arranged in sequential order. Here, the insertion and removal of data are not feasible in a linear manner. There exists a hierarchical relationship between the individual data items.

Types of Non-Linear Data Structures

The following is the list of Non-Linear Data Structures that we generally use:

1. Trees

A Tree is a Non-Linear Data Structure and a hierarchy containing a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the "children").

The Tree data structure is a specialized method to arrange and collect data in the computer to be utilized more effectively. It contains a central node, structural nodes, and sub-nodes connected via edges. We can also say that the tree data structure consists of roots, branches, and leaves connected.

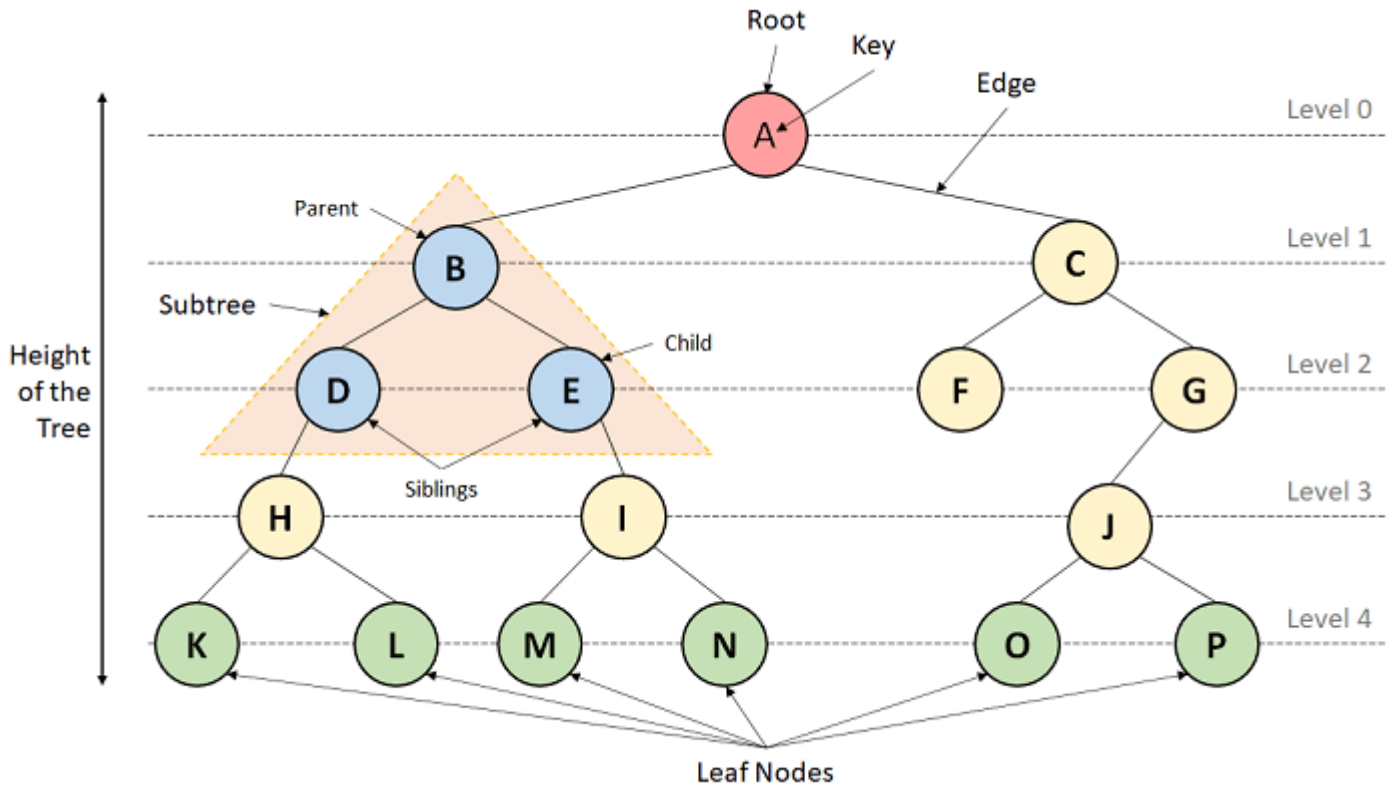


Figure . A Tree

Trees can be classified into different types:

a. **Binary Tree:** A Tree data structure where each parent node can have at most two children is termed a Binary Tree.

b. **Binary Search Tree:** A Binary Search Tree is a Tree data structure where we can easily maintain a sorted list of numbers.

c. **AVL Tree:** An AVL Tree is a self-balancing Binary Search Tree where each node maintains extra information known as a Balance Factor whose value is either -1, 0, or +1.

d. **B-Tree:** A B-Tree is a special type of self-balancing Binary Search Tree where each node consists of multiple keys and can have more than two children.

Some Applications of Trees:

a. Trees implement hierarchical structures in computer systems like directories and file systems.

b. Trees are also used to implement the navigation structure of a website.

c. We can generate code like Huffman's code using Trees.

d. Trees are also helpful in decision-making in Gaming applications.

e. Trees are responsible for implementing priority queues for priority-based OS scheduling functions.

f. Trees are also responsible for parsing expressions and statements in the compilers of different programming languages.

g. We can use Trees to store data keys for indexing for Database Management System (DBMS).

h. Spanning Trees allows us to route decisions in Computer and Communications Networks.

i. Trees are also used in the path-finding algorithm implemented in Artificial Intelligence (AI), Robotics, and Video Games Applications.

2. Graphs

A Graph is another example of a Non-Linear Data Structure comprising a finite number of nodes or vertices and the edges connecting them. The Graphs are utilized to address problems of the real world in which it denotes the problem area as a network such as social networks, circuit networks, and telephone networks. For instance, the nodes or vertices of a Graph can represent a single user in a telephone network, while the edges represent the link between them via telephone.

The Graph data structure, G is considered a mathematical structure comprised of a set of vertices, V and a set of edges, E as shown below:

$G = (V,E)$

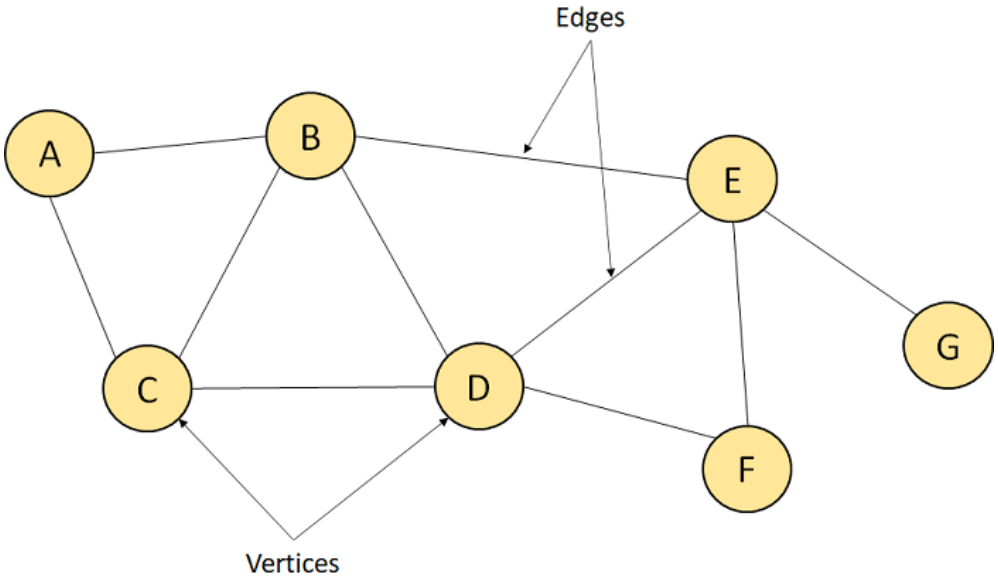


Figure . A Graph

The above figure represents a Graph having seven vertices A, B, C, D, E, F, G, and ten edges [A, B], [A, C], [B, C], [B, D], [B, E], [C, D], [D, E], [D, F], [E, F], and [E, G].

Depending upon the position of the vertices and edges, the Graphs can be classified into different types:

a. **Null Graph:** A Graph with an empty set of edges is termed a Null Graph.

b. **Trivial Graph:** A Graph having only one vertex is termed a Trivial Graph.

c. **Simple Graph:** A Graph with neither self-loops nor multiple edges is known as a Simple Graph.

d. **Multi Graph:** A Graph is said to be Multi if it consists of multiple edges but no self-loops.

e. **Pseudo Graph:** A Graph with self-loops and multiple edges is termed a Pseudo Graph.

f. **Non-Directed Graph:** A Graph consisting of non-directed edges is known as a Non-Directed Graph.

g. **Directed Graph:** A Graph consisting of the directed edges between the vertices is known as a Directed Graph.

h. **Connected Graph:** A Graph with at least a single path between every pair of vertices is termed a Connected Graph.

i. **Disconnected Graph:** A Graph where there does not exist any path between at least one pair of vertices is termed a Disconnected Graph.

j. **Regular Graph:** A Graph where all vertices have the same degree is termed a Regular Graph.

k. **Complete Graph:** A Graph in which all vertices have an edge between every pair of vertices is known as a Complete Graph.

l. **Cycle Graph:** A Graph is said to be a Cycle if it has at least three vertices and edges that form a cycle.

m. **Cyclic Graph:** A Graph is said to be Cyclic if and only if at least one cycle exists.

n. **Acyclic Graph:** A Graph having zero cycles is termed an Acyclic Graph.

o. **Finite Graph:** A Graph with a finite number of vertices and edges is known as a Finite Graph.

p. **Infinite Graph:** A Graph with an infinite number of vertices and edges is known as an Infinite Graph.

q. **Bipartite Graph:** A Graph where the vertices can be divided into independent sets A and B, and all the vertices of set A should only be connected to the vertices present in set B with some edges is termed a Bipartite Graph.

r. **Planar Graph:** A Graph is said to be a Planar if we can draw it in a single plane with two edges intersecting each other.

s. **Euler Graph:** A Graph is said to be Euler if and only if all the vertices are even degrees.

t. **Hamiltonian Graph:** A Connected Graph consisting of a Hamiltonian circuit is known as a Hamiltonian Graph.

Some Applications of Graphs:

a. Graphs help us represent routes and networks in transportation, travel, and communication applications.

b. Graphs are used to display routes in GPS.

c. Graphs also help us represent the interconnections in social networks and other network-based applications.

d. Graphs are utilized in mapping applications.

e. Graphs are responsible for the representation of user preference in e-commerce applications.

f. Graphs are also used in Utility networks in order to identify the problems posed to local or municipal corporations.

g. Graphs also help to manage the utilization and availability of resources in an organization.

h. Graphs are also used to make document link maps of the websites in order to display the connectivity between the pages through hyperlinks.

i. Graphs are also used in robotic motions and neural networks.

Basic Operations of Data Structures

In the following section, we will discuss the different types of operations that we can perform to manipulate data in every data structure:

1. **Traversal:** Traversing a data structure means accessing each data element exactly once so it can be administered. For example, traversing is required while printing the names of all the employees in a department.
2. **Search:** Search is another data structure operation which means to find the location of one or more data elements that meet certain constraints. Such a data element may or may not be present in the given set of data elements. For example, we can use the search operation to find the names of all the employees who have the experience of more than 5 years.
3. **Insertion:** Insertion means inserting or adding new data elements to the collection. For example, we can use the insertion operation to add the details of a new employee the company has recently hired.

4. **Deletion:** Deletion means to remove or delete a specific data element from the given list of data elements. For example, we can use the deleting operation to delete the name of an employee who has left the job.
5. **Sorting:** Sorting means to arrange the data elements in either Ascending or Descending order depending on the type of application. For example, we can use the sorting operation to arrange the names of employees in a department in alphabetical order or estimate the top three performers of the month by arranging the performance of the employees in descending order and extracting the details of the top three.
6. **Merge:** Merge means to combine data elements of two sorted lists in order to form a single list of sorted data elements.
7. **Create:** Create is an operation used to reserve memory for the data elements of the program. We can perform this operation using a declaration statement. The creation of data structure can take place either during the following:
    a. Compile-time
    b. Run-time
    For example, the **malloc()** function is used in C Language to create data structure.
8. **Selection:** Selection means selecting a particular data from the available data. We can select any particular data by specifying conditions inside the loop.
9. **Update:** The Update operation allows us to update or modify the data in the data structure. We can also update any particular data by specifying some conditions inside the loop, like the Selection operation.
10. **Splitting:** The Splitting operation allows us to divide data into various subparts decreasing the overall process completion time.

Understanding the Abstract Data Type

As per the **National Institute of Standards and Technology (NIST)**, a data structure is an arrangement of information, generally in the memory, for better algorithm efficiency. Data Structures include linked lists, stacks, queues, trees, and dictionaries. They could also be a theoretical entity, like the name and address of a person.

From the definition mentioned above, we can conclude that the operations in data structure include:

a. A high level of abstractions like addition or deletion of an item from a list.
b. Searching and sorting an item in a list.
c. Accessing the highest priority item in a list.

Whenever the data structure does such operations, it is known as an **Abstract Data Type (ADT)**.

We can define it as a set of data elements along with the operations on the data. The term "abstract" refers to the fact that the data and the fundamental operations defined on it are being studied independently of their implementation. It includes what we can do with the data, not how we can do it.

An ADI implementation contains a storage structure in order to store the data elements and algorithms for fundamental operation. All the data structures, like an array, linked list, queue, stack, etc., are examples of ADT.

Understanding the Advantages of using ADTs

In the real world, programs evolve as a consequence of new constraints or requirements, so modifying a program generally requires a change in one or multiple data structures. For example, suppose we want to insert a new field into an employee's record to keep track of more details about each employee. In that case, we can improve the efficiency of the program by replacing an Array with a Linked structure. In such a situation, rewriting every procedure that utilizes the modified structure is unsuitable. Hence, a better alternative is to separate a data structure from its implementation information. This is the principle behind the usage of Abstract Data Types (ADT).

Some Applications of Data Structures

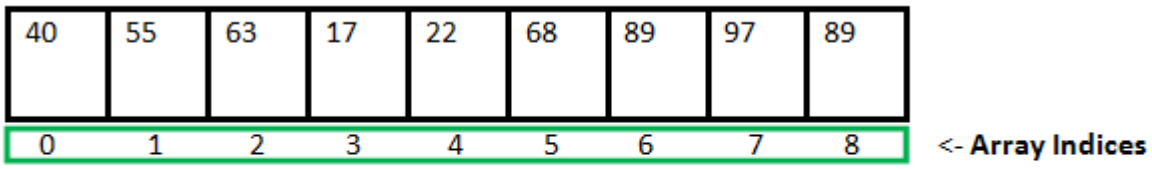
The following are some applications of Data Structures:

1. Data Structures help in the organization of data in a computer's memory.
2. Data Structures also help in representing the information in databases.
3. Data Structures allows the implementation of algorithms to search through data (For example, search engine).
4. We can use the Data Structures to implement the algorithms to manipulate data (For example, word processors).
5. We can also implement the algorithms to analyse data using Data Structures (For example, data miners).
6. Data Structures support algorithms to generate the data (For example, a random number generator).
7. Data Structures also support algorithms to compress and decompress the data (For example, a zip utility).
8. We can also use Data Structures to implement algorithms to encrypt and decrypt the data (For example, a security system).
9. With the help of Data Structures, we can build software that can manage files and directories (For example, a file manager).
10. We can also develop software that can render graphics using Data Structures. (For example, a web browser or 3D rendering software).

Apart from those, as mentioned earlier, there are many other applications of Data Structures that can help us build any desired software.

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

Since C is a structured language, it has some fixed rules for programming. One of them includes changing the size of an array. An array is a collection of items stored at contiguous memory locations.



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

As can be seen, the length (size) of the array above is 9. But what if there is a requirement to change this length (size)? For example,

- If there is a situation where only 5 elements are needed to be entered in this array. In this case, the remaining 4 indices are just wasting memory in this array. So there is a requirement to lessen the length (size) of the array from 9 to 5.
- Take another situation. In this, there is an array of 9 elements with all 9 indices filled. But there is a need to enter 3 more elements in this array. In this case, 3 indices more are required. So the length (size) of the array needs to be changed from 9 to 12.

This procedure is referred to as **Dynamic Memory Allocation in C**.
Therefore, C **Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

1. malloc()
2. calloc()
3. free()
4. realloc()

Let's look at each of them in greater detail.

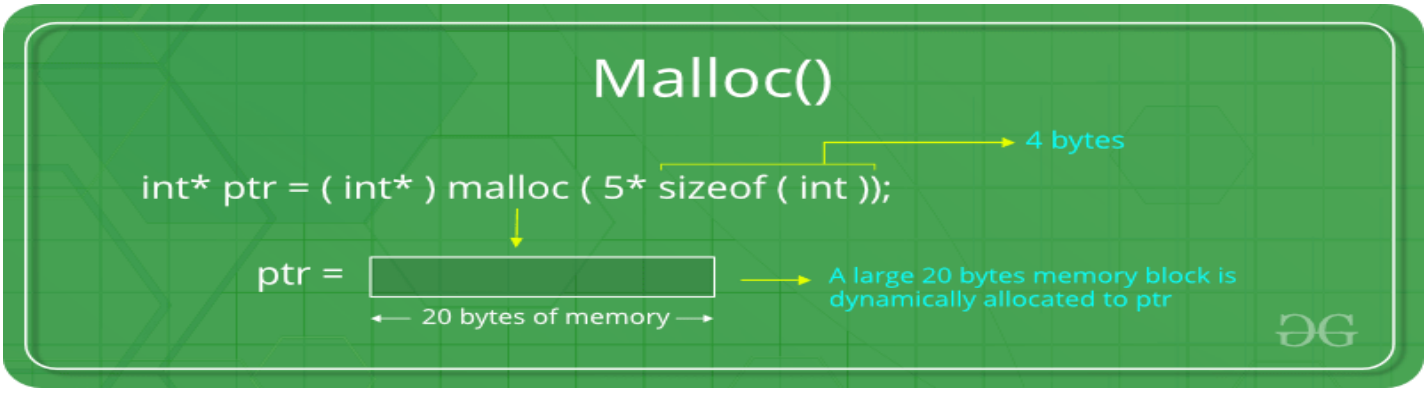
C malloc() method

The **"malloc"** or **"memory allocation"** method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't Initialize memory at execution time so that it has initialized each block with the default garbage value initially.

Syntax of malloc() in C

```
ptr = (cast-type*) malloc(byte-size)
For Example:
```

ptr = (int*) malloc(100 * sizeof(int));
Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.



If space is insufficient, allocation fails and returns a NULL pointer.

Example of malloc() in C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

Output

```
Enter number of elements: 5

Memory successfully allocated using malloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

C calloc() method

1. **“calloc”** or **“contiguous allocation”** method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
2. It initializes each block with a default value ‘0’.
3. It has two parameters or arguments as compare to malloc().

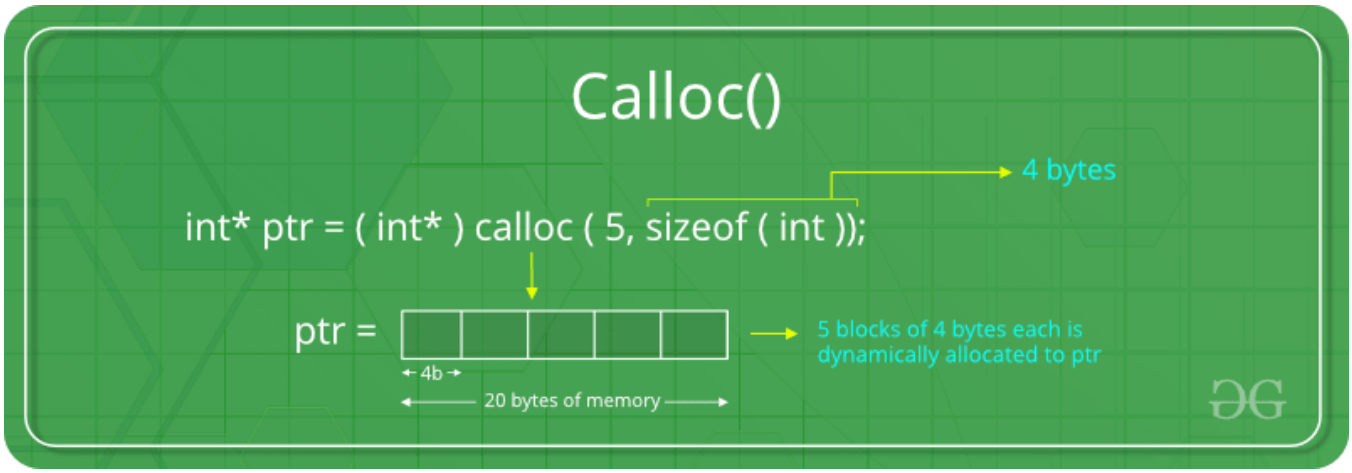
Syntax of calloc() in C

```
ptr = (cast-type*)calloc(n, element-size);
here, n is the no. of elements and element-size is the size of each element.
```

For Example:

ptr = (float*) calloc(25, sizeof(float));

This statement allocates contiguous space in memory for 25 elements each with the size of the float.



If space is insufficient, allocation fails and returns a NULL pointer.

Example of calloc() in C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
```

```
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }

    return 0;
}
```

Output

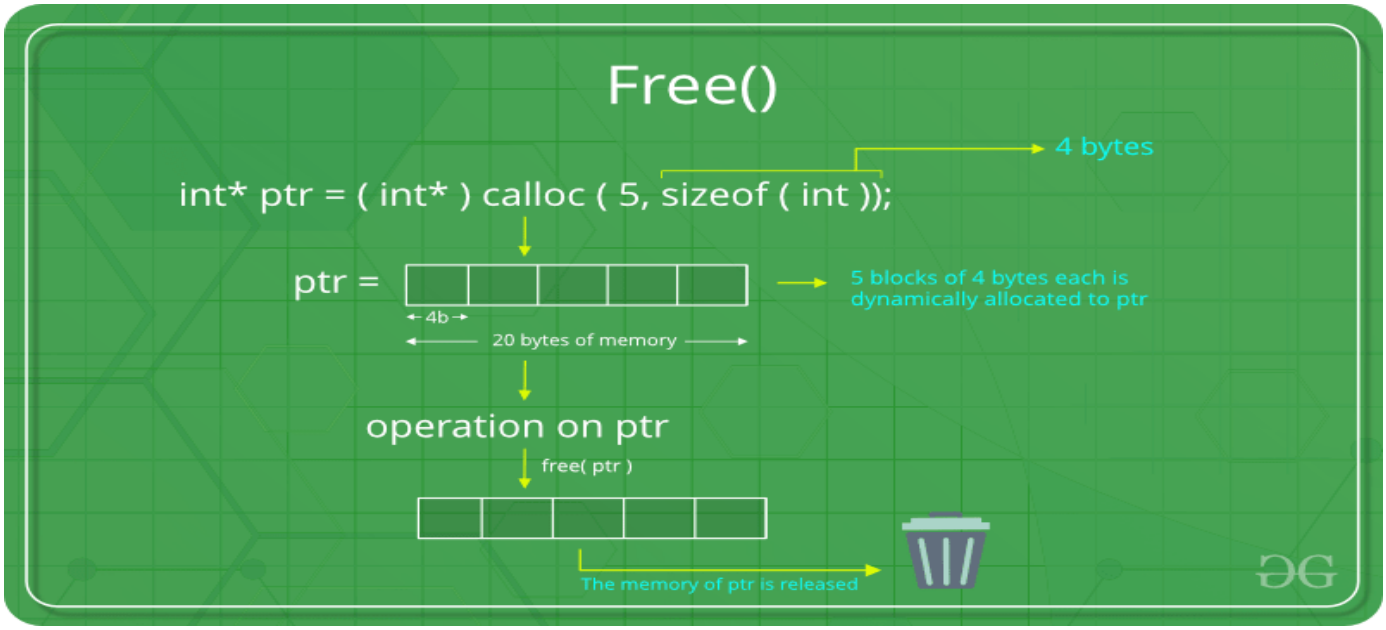
```
Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,
```

C free() method

“free” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C

```
free(ptr);
```



Example of free() in C

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{

    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Free the memory
        free(ptr);
        printf("Malloc Memory successfully freed.\n");

        // Memory has been successfully allocated
        printf("\nMemory successfully allocated using calloc.\n");

        // Free the memory
        free(ptr1);
        printf("Calloc Memory successfully freed.\n");
    }

    return 0;
}
```

Output

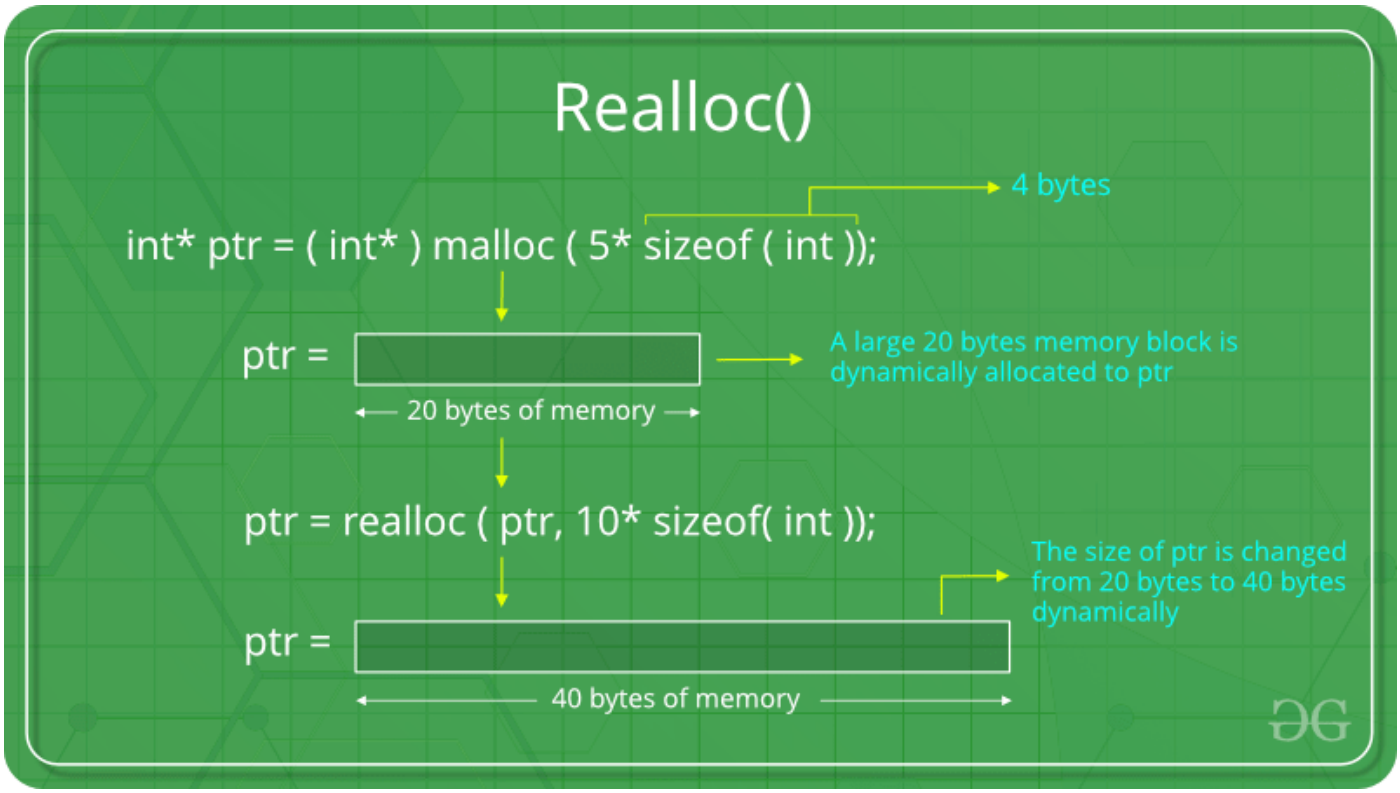
```
Enter number of elements: 5

Memory successfully allocated using malloc.

Malloc Memory successfully freed.


Memory successfully allocated using calloc.

Calloc Memory successfully freed.
```

C realloc() method

“realloc” or **“re-allocation”** method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C

```
ptr = realloc(ptr, newSize);
where ptr is reallocated with new size 'newSize'.
```



If space is insufficient, allocation fails and returns a NULL pointer.

Example of realloc() in C

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
```

```
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using calloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        // Get the new size for the array
        n = 10;
        printf("\n\nEnter the new size of the array: %d\n", n);

        // Dynamically re-allocate memory using realloc()
        ptr = (int*)realloc(ptr, n * sizeof(int));

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");

        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
}
```

Output

```
Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,


Enter the new size of the array: 10

Memory successfully re-allocated using realloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
```

Big O Notation in Data Structures

Asymptotic analysis is the study of how the algorithm's performance changes when the order of the input size changes. We employ big-notation to asymptotically confine the expansion of a running time to within constant factors above and below. The amount of time, storage, and other resources required to perform an algorithm determine its efficiency. Asymptotic notations are used to determine the efficiency. For different types of inputs, an algorithm's performance may vary. The performance will fluctuate as the input size grows larger.

When the input tends towards a certain value or a limiting value, asymptotic notations are used to represent how long an algorithm takes to execute. When the input array is already sorted, for example, the time spent by the method is linear, which is the best scenario.

However, when the input array is in reverse order, the method takes the longest (quadratic) time to sort the items, which is the worst-case scenario. It takes average time when the input array is not sorted or in reverse order. Asymptotic notations are used to represent these durations.

Big O notation classifies functions based on their growth rates: several functions with the same growth rate can be written using the same O notation. The symbol O is utilized since a function's development rate is also known as the order of the function. A large O notation description of a function generally only offers an upper constraint on the function's development rate.

It would be convenient to have a form of asymptotic notation that means "the running time grows at most this much, but it could grow more slowly." We use "big-O" notation for just such occasions.

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

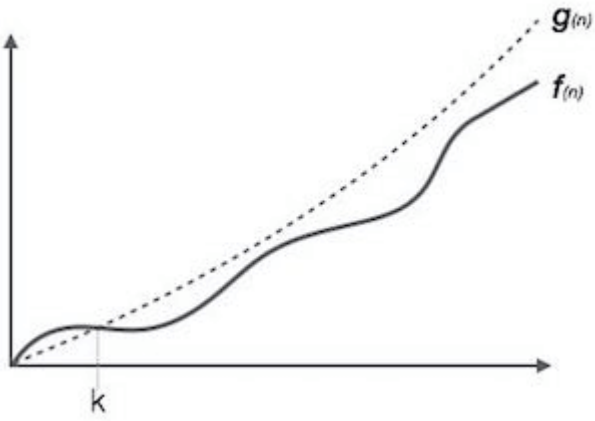
The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below:

- Big oh Notation (?)
- 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.



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:

For example:

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)≤c.g(n) for all n≥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.

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)

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.

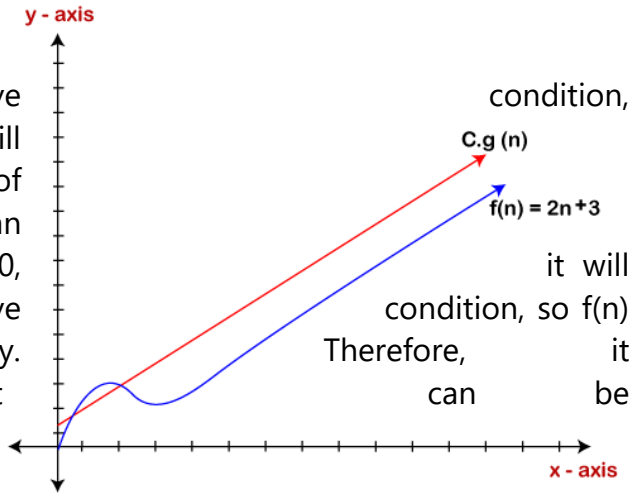
If n=2

2*2+3<=5*2

7<=10

For n=2, the above condition is true.

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). It can be represented graphically as:



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.

Advantages of Big O Notation

- When examining the efficiency of an algorithm using run-time inputs, asymptotic analysis is quite useful. Otherwise, if we do it manually with passing test cases for various inputs, performance may vary as the algorithm's input changes.
- When the algorithm is executed on multiple computers, its performance varies. As a result, we pick an algorithm whose performance does not change much as the number of inputs increases. As a result, a mathematical representation provides a clear understanding of the top and lower boundaries of an algorithm's run-time.

Examples

Now let us have a deeper look at the Big O notation of various examples:

O(1):

```
void constantTimeComplexity(int arr[])
{
   printf("First element of array = %d",arr[0]);
}
```

This function runs in O(1) time (or "constant time") relative to its input. The input array could be 1 item or 1,000 items, but this function would still just require one step.

O(n):

```
void linearTimeComplexity(int arr[], int size)
{
   for (int i = 0; i < size; i++)
   {
```

```
        printf("%d\n", arr[i]);
    }
}
```

This function runs in O(n) time (or "linear time"), where n is the number of items in the array. If the array has 10 items, we have to print 10 times. If it has 1000 items, we have to print 1000 times.

O(n^2):

```
void quadraticTimeComplexity(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
     }
}
```

Here we're nesting two loops. If our array has n items, our outer loop runs n times, and our inner loop runs n times for each iteration of the outer loop, giving us n^2 total prints. If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times. Thus this function runs in O(n^2) time (or "quadratic time").

O(2^n):

```
int fibonacci(int num)
{
    if (num <= 1) return num;
    return fibonacci(num - 2) + fibonacci(num - 1);
}
```

An example of an O(2^n) function is the recursive calculation of Fibonacci numbers. O(2^n) denotes an algorithm whose growth doubles with each addition to the input data set. The growth curve of an O(2^n) function is exponential - starting off very shallow, then rising meteorically.

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.***

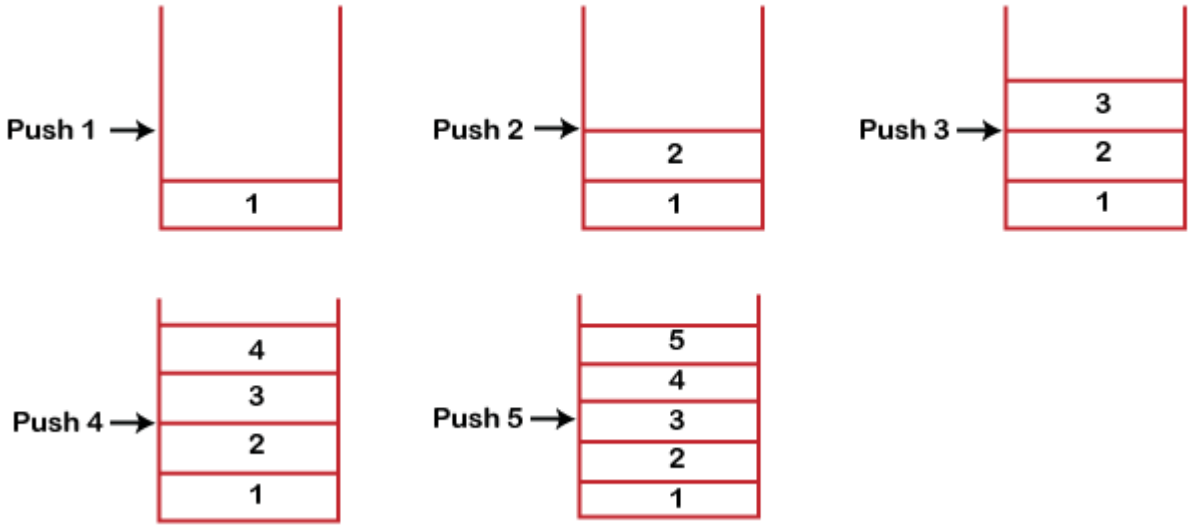
Some key points related to stack

- 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:

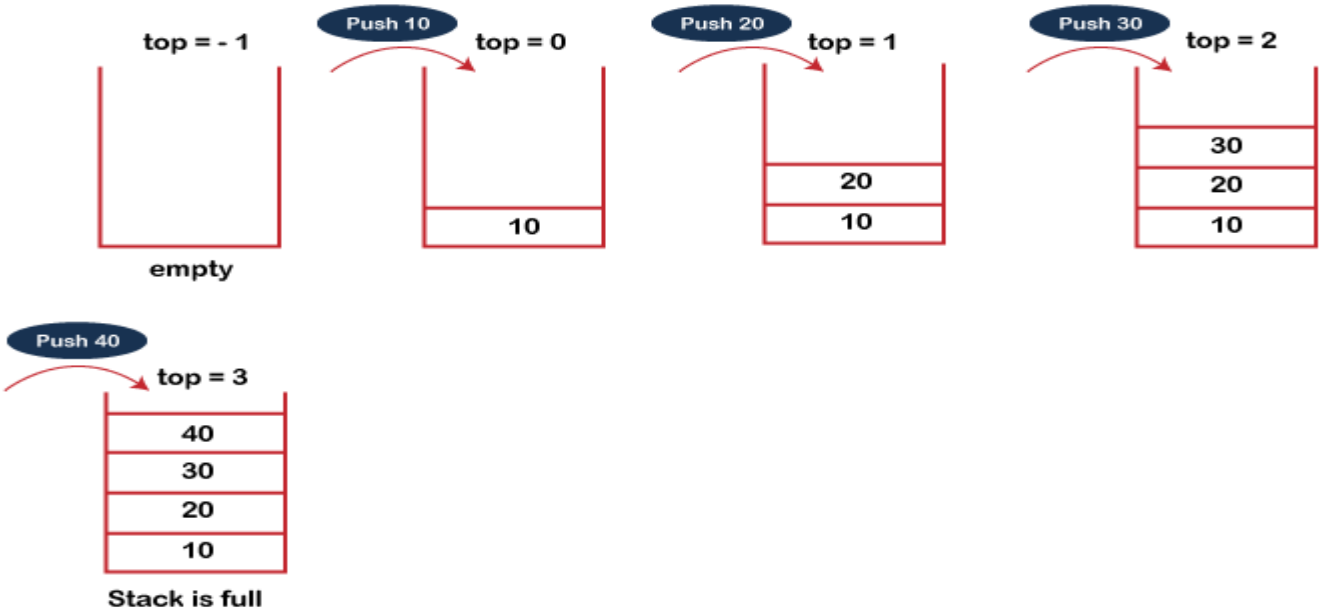
- **push():** When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.

- **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:

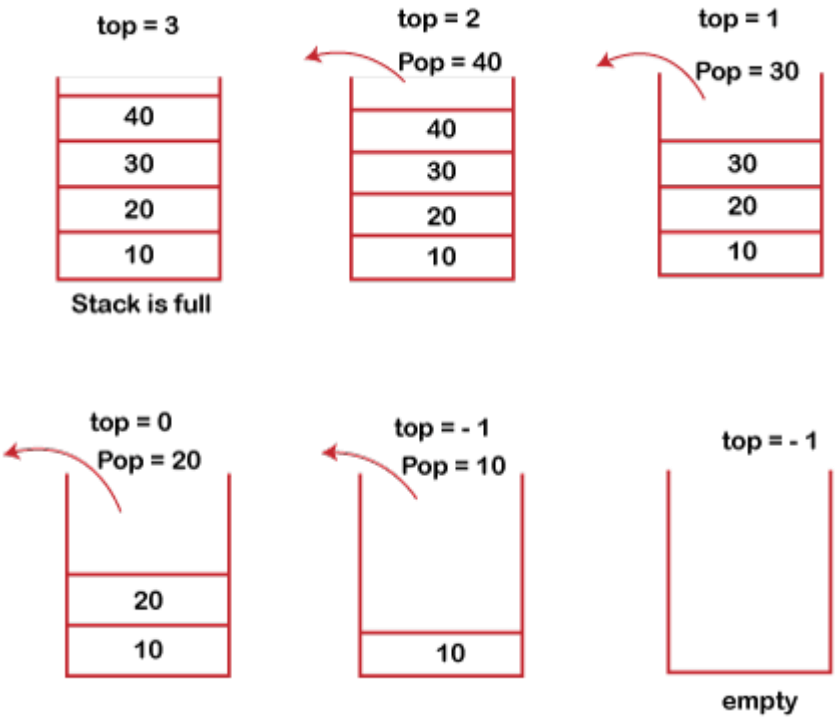
- Before inserting an element in a stack, we check whether the stack is full.
- If we try to insert the element in a stack, and the stack is full, then the ***overflow*** condition occurs.
- When we initialize a stack, we set the value of top as -1 to check that the stack is empty.
- When the new element 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**.



Applications of Stack

The following are the applications of the stack:

- **Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

```
int main()
{
  cout<<"Hello";
  cout<<"javaTpoint";
}
```

As we know, each program has *an opening* and *closing* braces; when the opening braces come, we push the braces in a stack, and when the closing braces appear, we pop the opening braces from the stack. Therefore, the net value comes out to be zero. If any symbol is left in the stack, it means that some syntax occurs in a program.

- **String reversal:** Stack is also used for reversing a string. For example, we want to reverse a "**javaTpoint**" string, so we can achieve this with the help of a stack. First, we push all the characters of the string in a stack until we reach the null character. After pushing all the characters, we start taking out the character one by one until we reach the bottom of the stack.
- **UNDO/REDO:** It can also be used for performing UNDO/REDO operations. For example, we have an editor in which we write 'a', then 'b', and then 'c'; therefore, the text written in an editor is abc. So, there are three states, a, ab, and abc, which are stored in a stack. There would be two stacks in which one stack shows UNDO state, and the other shows REDO state.

If we want to perform UNDO operation, and want to achieve 'ab' state, then we implement pop operation.

- **Recursion:** The recursion means that the function is calling itself again. To maintain the previous states, the compiler creates a system stack in which all the previous records of the function are maintained.
- **DFS(Depth First Search):** This search is implemented on a Graph, and Graph uses the stack data structure.
- **Backtracking:** Suppose we have to create a path to solve a maze problem. If we are moving in a particular path, and we realize that we come on the wrong way. In order to come at the beginning of the path to create a new path, we have to use the stack data structure.
- **Expression conversion:** Stack can also be used for expression conversion. This is one of the most important applications of stack. The list of the expression conversion is given below:
  - Infix to prefix
  - Infix to postfix
  - Prefix to infix
  - Prefix to postfix
  - Postfix to infix
- **Memory management:** The stack manages the memory. The memory is assigned in the contiguous memory blocks. The memory is known as stack memory as all the variables are assigned in a function call stack memory. The memory size assigned to the program is known to the compiler. When the function is created, all its variables are assigned in the stack memory. When the function completed its execution, all the variables assigned in the stack are released.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm:
A simple algorithm for the stack push operation using an array, along with the defined variables:

Variables:
- `stack`: An array representing the stack data structure.
- `top`: An integer representing the index of the top element in the stack.
- `MAX_SIZE`: An integer representing the maximum size of the stack.

Algorithm for Stack Push Operation:
1. Initialize Stack and Top:
  - Create an array `stack` of size `MAX_SIZE` to store the elements of the stack.
  - Initialize `top` to -1 to indicate an empty stack.

2. Check for Stack Overflow:
  - If `top` is equal to `MAX_SIZE - 1`, then the stack is full, and further push operations are not possible. Display an error message indicating stack overflow.

3. Increment Top and Push Element:
  - Increment `top` by 1 to move it to the next position.
  - Assign the new element to the position `stack[top]`.

Algorithm Steps:

1. Initialize Stack and Top:
- Let stack[MAX_SIZE]
- Let top = -1
2. Check for Stack Overflow:
- If top == MAX_SIZE - 1:
- Display "Stack Overflow" & Exit
- Else:
- Increment top by 1
3. Increment Top and Push Element:
Increment top by 1
stack[top] = element
4. Exit

This algorithm describes the process of pushing an element onto the stack using an array-based implementation. It ensures that the stack does not overflow and correctly updates the top pointer before inserting the new element.

Time Complexity : o(1)

implementation of push algorithm in C language

```
void push (int val, int n) //n is size of the stack
{
   if (top == n -1)
   printf("\n Overflow");
   else
   {
   top = top +1;
   stack[top] = val;
   }
}
```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.

The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm :

Here's a simple algorithm for the stack pop operation using an array, along with the defined variables:

Variables:
- `stack`: An array representing the stack data structure.
- `top`: An integer representing the index of the top element in the stack.
- `MAX_SIZE`: An integer representing the maximum size of the stack.

Algorithm for Stack Pop Operation:

1. Check for Stack Underflow:
   - If `top` is equal to -1, then the stack is empty, and further pop operations are not possible. Display an error message indicating stack underflow.
2. Remove and Return Top Element:
   - Store the top element of the stack in a variable `element`.
   - Decrement `top` by 1 to remove the top element from the stack.
   - Return the `element`.

Algorithm Steps:

1. Check for Stack Underflow:
- If top == -1:
- Display "Stack Underflow"
- Else:
- Store stack[top] in element
- Decrement top by 1
- Return element

This algorithm describes the process of popping an element from the stack using an array-based implementation. It ensures that the stack does not underflow and correctly updates the top pointer while removing and returning the top element.

Time Complexity: o(1)

Implementation of POP algorithm using C language

```
int pop ()
{
   if(top == -1)
   {
      printf("Underflow");
     return 0;
   }
   else
```

```
    {
        return stack[top - - ];
    }
}
```

Visiting each element of the stack (Peek operation)

Peek operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in an already empty stack.

Algorithm :

A simple algorithm for visiting each element of the stack using an array, along with the defined variables:

Variables:
- `stack`: An array representing the stack data structure.
- `top`: An integer representing the index of the top element in the stack.
- `MAX_SIZE`: An integer representing the maximum size of the stack.

Algorithm for Visiting Each Element of the Stack:

1. Initialize Stack and Top:
   - Create an array `stack` of size `MAX_SIZE` to store the elements of the stack.
   - Initialize `top` to -1 to indicate an empty stack.

2. Check if Stack is Empty:
   - If `top` is equal to -1, then the stack is empty. Display an error message indicating that the stack is empty.

3. Visit Each Element:
   - Iterate over the elements of the stack from index 0 to `top`.
   - Access each element of the stack using the index `i` and perform the desired operation (e.g., printing the element).

Algorithm Steps:

1. Initialize Stack and Top:
- Let stack[MAX_SIZE]
- Let top = -1

2. Check if Stack is Empty:
- If top == -1:
- Display "Stack is Empty"
- Else:
- Initialize i = 0

3. Visit Each Element:
- For i = 0 to top:
- Perform operation on stack[i]

This algorithm describes the process of visiting each element of the stack using an array-based implementation. It ensures that the stack is not empty before attempting to access its elements, and iterates over the stack from index 0 to the top element, performing the desired operation on each element.

How To Implement a Stack in C Programming

A stack is a *linear data structure*, a collection of items of the same type.

In a stack, the insertion and deletion of elements happen only at one endpoint. The behavior of a stack is described as “Last In, First Out” (LIFO). When an element is “pushed” onto the stack, it

becomes the first item that will be “popped” out of the stack. In order to reach the oldest entered item, you must pop all the previous items.

In this article, you will learn about the concept of stack data structure and its implementation using arrays in C.

Operations Performed on Stacks

The following are the basic operations served by stacks.

- `push`: Adds an element to the top of the stack.
- `pop`: Removes the topmost element from the stack.
- `isEmpty`: Checks whether the stack is empty.
- `isFull`: Checks whether the stack is full.
- `top`: Displays the topmost element of the stack.

Underlying Mechanics of Stacks

Initially, a pointer (`top`) is set to keep the track of the topmost item in the stack. The stack is initialized to `-1`.
Then, a check is performed to determine if the stack is empty by comparing `top` to `-1`.
As elements are added to the stack, the position of `top` is updated.
As soon as elements are popped or deleted, the topmost element is removed and the position of `top` is updated.

Implementing Stack in C

Stacks can be represented using structures, pointers, arrays, or linked lists.

This example implements stacks using arrays in C:

```
#include <stdio.h>
#include <stdlib.h>
#define SIZE 4
int top = -1, inp_array[SIZE];
void push();
void pop();
void show();

int main()
{
    int choice;

    while (1)
    {
        printf("\nPerform operations on the stack:");
        printf("\n1.Push the element\n2.Pop the element\n3.Show\n4.End");
        printf("\n\nEnter the choice: ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            push();
            break;
        case 2:
```

```
            pop();
            break;
        case 3:
            show();
            break;
        case 4:
            exit(0);

        default:
            printf("\nInvalid choice!!");
        }
    }
}

void push()
{
    int x;

    if (top == SIZE - 1)
    {
        printf("\nOverflow!!");
    }
    else
    {
        printf("\nEnter the element to be added onto the stack: ");
        scanf("%d", &x);
        top = top + 1;
        inp_array[top] = x;
    }
}

void pop()
{
    if (top == -1)
    {
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nPopped element: %d", inp_array[top]);
        top = top - 1;
    }
}

void show()
{
    if (top == -1)
    {
```

```
        printf("\nUnderflow!!");
    }
    else
    {
        printf("\nElements present in the stack: \n");
        for (int i = top; i >= 0; --i)
            printf("%d\n", inp_array[i]);
    }
}
```

This program presents the user with four options:

1. Push the element
2. Pop the element
3. Show
4. End

It waits for the user to input a number.

- If the user selects `1`, the program handles a `push()`. First, it checks to see if `top` is equivalent to `SIZE - 1`. If `true`, `"Overflow!!"` is displayed. Otherwise, the user is asked to provide the new element to add to the stack.
- If the user selects `2`, the program handles a `pop()`. First, it checks to see if `top` is equivalent to `-1`. If `true`, `"Underflow!!"` is displayed. Otherwise, the topmost element is removed and the program outputs the resulting stack.
- If the user selects `3`, the program handles a `show()`. First, it checks to see if `top` is equivalent to `-1`. If `true`, `"Underflow!!"` is displayed. Otherwise, the program outputs the resulting stack.
- If the user selects `4`, the program exits.

Execute this code to `push()` the number `"10"`onto the stack:

```
Output
Perform operations on the stack:
1.Push the element
2.Pop the element
3.Show
4.End


Enter the choice: 1


Enter the element to be inserted onto the stack: 10
```

Then `show()` the elements on the stack:

```
Output
Perform operations on the stack:
1.Push the element
2.Pop the element
3.Show
4.End


Enter the choice: 3


Elements present in the stack:
10
```

Then `pop()`:

```
Output
Perform operations on the stack:
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice: 2
Popped element: 10
```

Now, the stack is empty. Attempt to `pop()` again:

```
Output
Perform operations on the stack:
1.Push the element
2.Pop the element
3.Show
4.End

Enter the choice: 3
Underflow!!
```

Continue to experiment with this program to understand how a stack works.

Time Complexity of Stack Operations

Only a single element can be accessed at a time in stacks.

While performing `push()` and `pop()` operations on the stack, it takes `O(1)` time.

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 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)

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 -

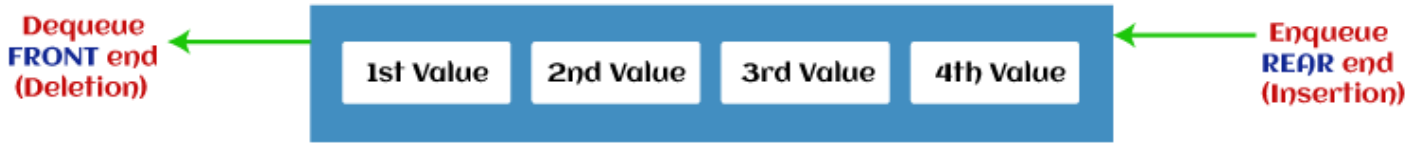
- Simple Queue or Linear Queue

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

Let's discuss each of the type of queue.

Simple Queue or Linear Queue

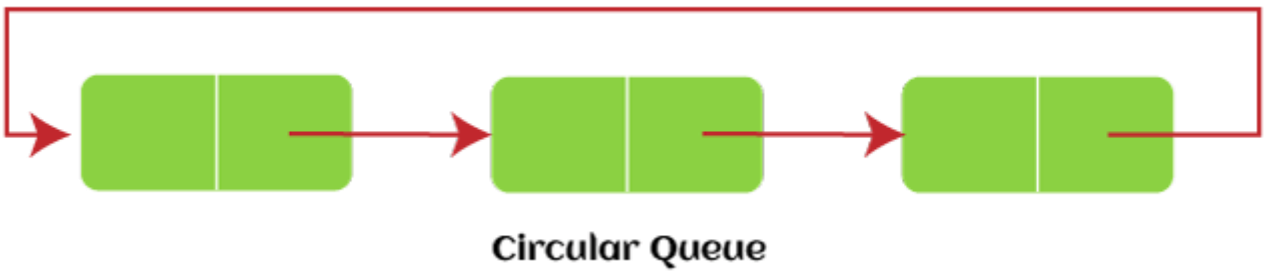
In Linear Queue, an insertion takes place from one end while the deletion occurs from another end. The end at which the insertion takes place is known as the rear end, and the end at which the deletion takes place is known as front end. It strictly follows the FIFO rule.



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

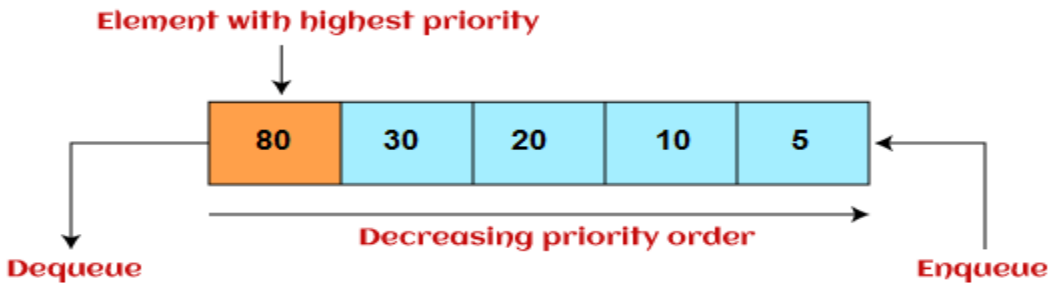
In Circular Queue, all the nodes are represented as circular. It is similar to the linear Queue except that the last element of the queue is connected to the first element. It is also known as Ring Buffer, as all the ends are connected to another end. The representation of circular queue is shown in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



Insertion in priority queue takes place based on the arrival, while deletion in the priority queue occurs based on the priority. Priority queue is mainly used to implement the CPU scheduling algorithms.

There are two types of priority queue that are discussed as follows -

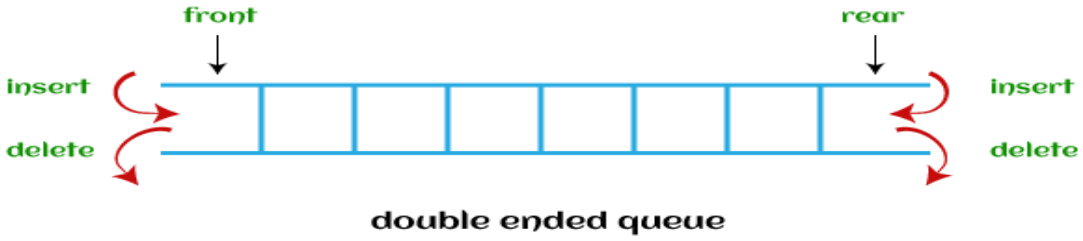
- **Ascending priority queue -** In ascending priority queue, elements can be inserted in arbitrary order, but only smallest can be deleted first. Suppose an array with elements 7, 5, and 3 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 3, 5, 7.
- **Descending priority queue -** In descending priority queue, elements can be inserted in arbitrary order, but only the largest element can be deleted first. Suppose an array with elements 7, 3, and 5 in the same order, so, insertion can be done with the same sequence, but the order of deleting the elements is 7, 5, 3.

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.

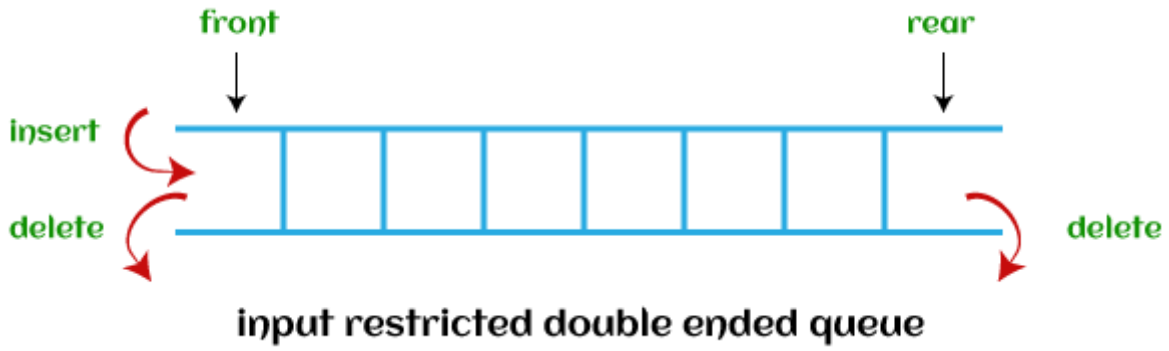
Deque can be used both as stack and queue as it allows the insertion and deletion operations on both ends. Deque can be considered as stack because stack follows the LIFO (Last In First Out) principle in which insertion and deletion both can be performed only from one end. And in deque, it is possible 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 -

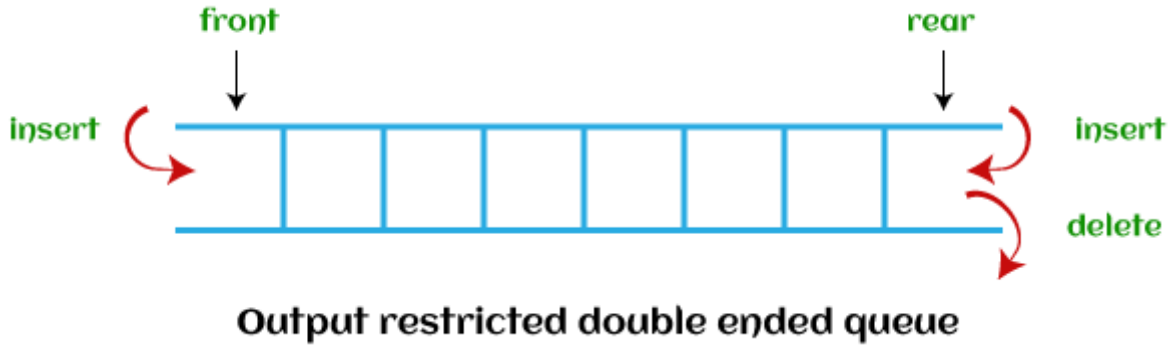


There are two types of deque that are discussed as follows -

- **Input restricted deque -** As the name implies, in input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



- **Output restricted deque -** As the name implies, in output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.



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:

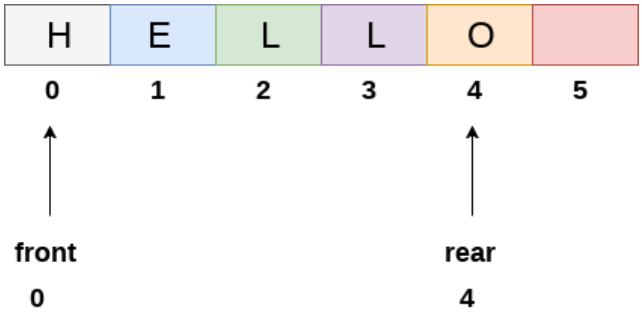
- **Implementation using array:** The sequential allocation in a Queue can be implemented using an array.

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

So, that's all about the article. Hope, the article will be helpful and informative to you.

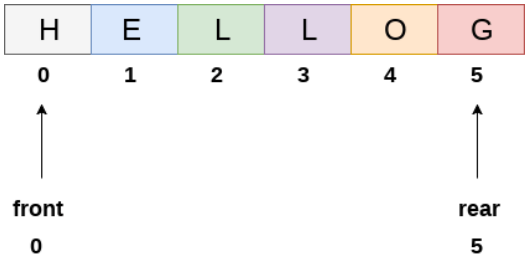
Array representation of Queue

We can easily represent queue by using linear arrays. There are two variables i.e. front and rear, that are implemented in the case of every queue. Front and rear variables point to the position from where insertions and deletions are performed in a queue. Initially, the value of front and queue is -1 which represents an empty queue. Array representation of a queue containing 5 elements along with the respective values of front and rear, is shown in the following figure.



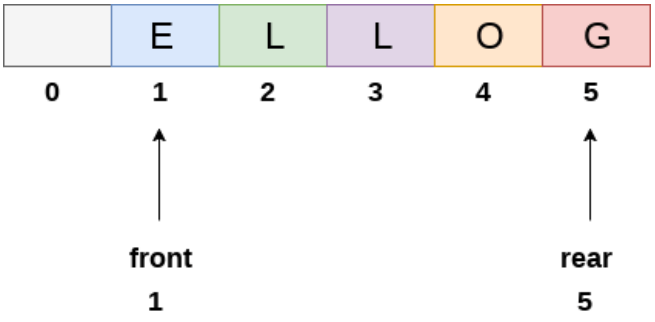
Queue

The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element into the queue shown in the above figure, the queue will look something like following. The value of rear will become 5 while the value of front remains same.



Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



Queue after deleting an element

Algorithm to insert any element in a queue

Check if the queue is already full by comparing rear to max - 1. if so, then return an overflow error.

If the item is to be inserted as the first element in the list, in that case set the value of front and rear to 0 and insert the element at the rear end.

Otherwise keep increasing the value of rear and insert each element one by one having rear as the index.

Algorithm

- **Step 1:** IF REAR = MAX - 1
  Write OVERFLOW
  Go to step
  [END OF IF]
- **Step 2:** IF FRONT = -1 and REAR = -1
  SET FRONT = REAR = 0
  ELSE
  SET REAR = REAR + 1
  [END OF IF]
- **Step 3:** Set QUEUE[REAR] = NUM
- **Step 4:** EXIT

C Function

```
void insert (int queue[], int max, int front, int rear, int item)
{
   if (rear + 1 == max)
   {
      printf("overflow");
   }
   else
   {
      if(front == -1 && rear == -1)
```

```
        {
            front = 0;
           rear = 0;
        }
        else
        {
           rear = rear + 1;
        }
        queue[rear]=item;
    }
}
```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

Algorithm

- **Step 1:** IF FRONT = -1 or FRONT > REAR
  Write UNDERFLOW
  ELSE
  SET VAL = QUEUE[FRONT]
  SET FRONT = FRONT + 1
  [END OF IF]
- **Step 2:** EXIT

C Function

```
int delete (int queue[], int max, int front, int rear)
{
   int y;
   if (front == -1 || front > rear)

   {
       printf("underflow");
   }
   else
   {
       y = queue[front];
      if(front == rear)
      {
         front = rear = -1;
          else
```

```
            front = front + 1;

        }
        return y;
    }
}
```

Menu driven program to implement queue using array

```
#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
    int choice;
    while(choice != 4)
    {
        printf("\n*************************Main Menu*****************************\n");
        printf("\n==========================================================
==============\n");
        printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");
        printf("\nEnter your choice ?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            insert();
            break;
            case 2:
            delete();
            break;
            case 3:
            display();
            break;
            case 4:
            exit(0);
            break;
            default:
```

```
            printf("\nEnter valid choice??\n");
        }
    }
}
void insert()
{
    int item;
    printf("\nEnter the element\n");
    scanf("\n%d",&item);
    if(rear == maxsize-1)
    {
        printf("\nOVERFLOW\n");
        return;
    }
    if(front == -1 && rear == -1)
    {
        front = 0;
        rear = 0;
    }
    else
    {
        rear = rear+1;
    }
    queue[rear] = item;
    printf("\nValue inserted ");

}
void delete()
{
    int item;
    if (front == -1 || front > rear)
    {
        printf("\nUNDERFLOW\n");
        return;

    }
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
```

```
            rear = -1 ;
          }
          else
          {
            front = front + 1;
          }
          printf("\nvalue deleted ");
        }


    }

    void display()
    {
      int i;
      if(rear == -1)
      {
        printf("\nEmpty queue\n");
      }
      else
      {  printf("\nprinting values .....\n");
        for(i=front;i<=rear;i++)
        {
          printf("\n%d\n",queue[i]);
        }
      }
    }
```

Output:

```
*************Main Menu**************

=============================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
123

Value inserted

*************Main Menu**************

=============================================
```

```
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?1

Enter the element
90

Value inserted

*************Main Menu**************

==================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?2

value deleted

*************Main Menu**************
==========================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?3

printing values .....

90

*************Main Menu**************

==========================================

1.insert an element
2.Delete an element
3.Display the queue
4.Exit

Enter your choice ?4
```

Drawback of array implementation

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

- **Memory wastage :** The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.

limitation of array representation of queue

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements, is shown. The value of the front variable is 5, therefore, we can not reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and can not be used in the future (for this queue).

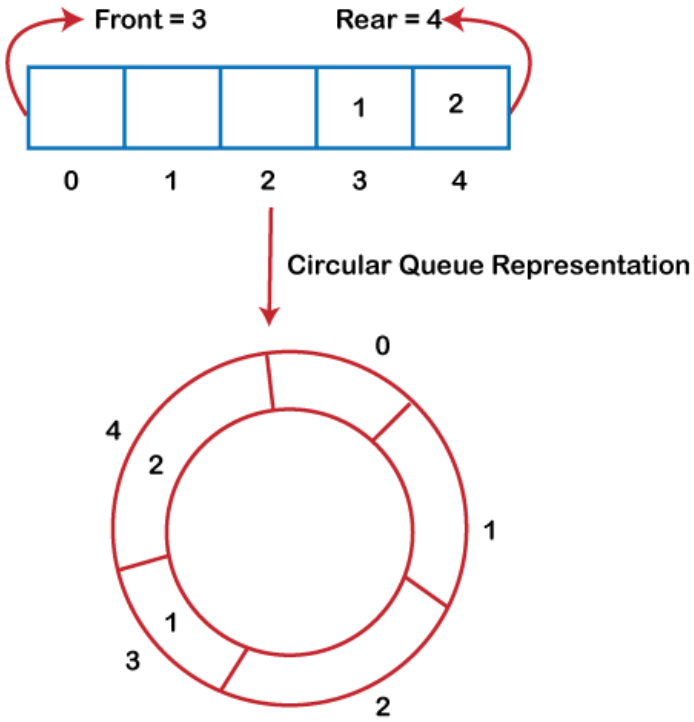
- **Deciding the array size**

On of the most common problem with array implementation is the size of the array which requires to be declared in advance. Due to the fact that, the queue can be extended at runtime depending upon the problem, the extension in the array size is a time taking process and almost impossible to be performed at runtime since a lot of reallocations take place. Due to this reason, we can declare the array large enough so that we can store queue elements as enough as possible but the main problem with this declaration is that, most of the array slots (nearly half) can never be reused. It will again lead to memory wastage.

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 0^{th} 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 != max - 1**, then rear will be incremented to **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
- **If front != 0 and rear = 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 = 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 (REAR+1)%MAX = FRONT
Write " OVERFLOW "
Goto step 4
[End OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE IF REAR = MAX - 1 and FRONT ! = 0
SET REAR = 0
ELSE
SET REAR = (REAR + 1) % MAX
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT

Dequeue Operation

The steps of dequeue operation are given below:

- First, we check whether the Queue is empty or not. If the queue is empty, we cannot perform the dequeue operation.
- When the element is deleted, the value of front gets decremented by 1.
- If there is only one element left which is to be deleted, then the front and rear are reset to -1.

Algorithm to delete an element from the circular queue

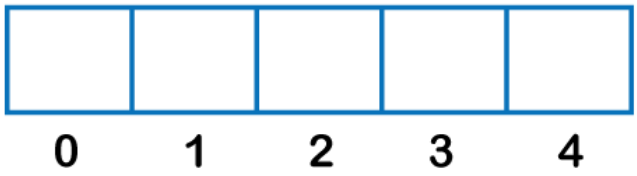
Step 1: IF FRONT = -1
Write " UNDERFLOW "
Goto Step 4
[END of IF]

Step 2: SET VAL = QUEUE[FRONT]

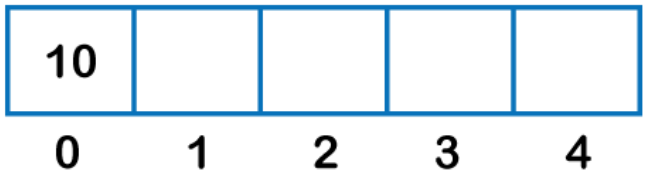
Step 3: IF FRONT = REAR
SET FRONT = REAR = -1
ELSE
IF FRONT = MAX -1
SET FRONT = 0
ELSE
SET FRONT = FRONT + 1
[END of IF]
[END OF IF]

Step 4: EXIT

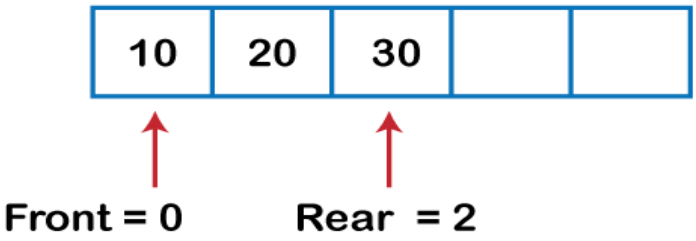
Let's understand the enqueue and dequeue operation through the diagrammatic representation.

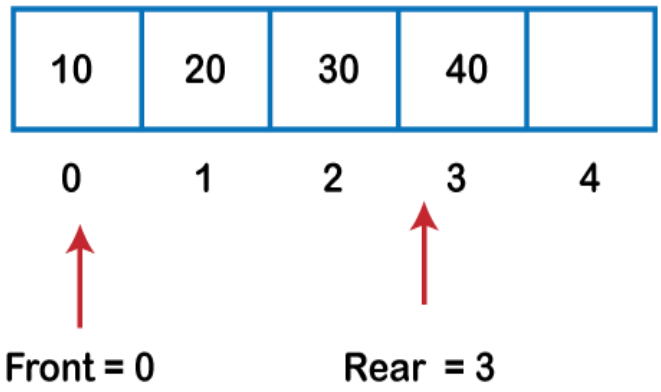


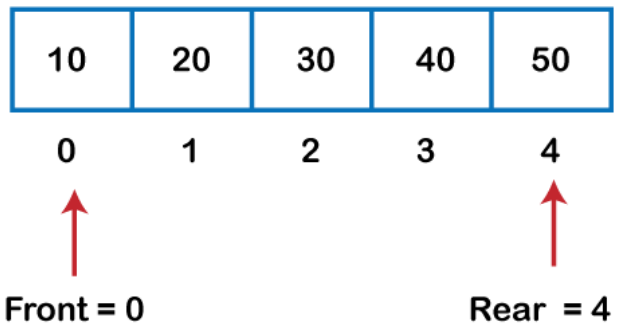
Front = -1
Rear = -1

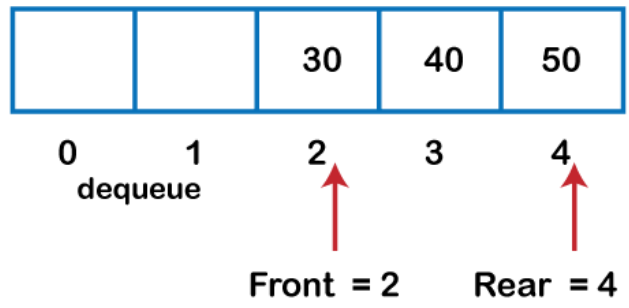


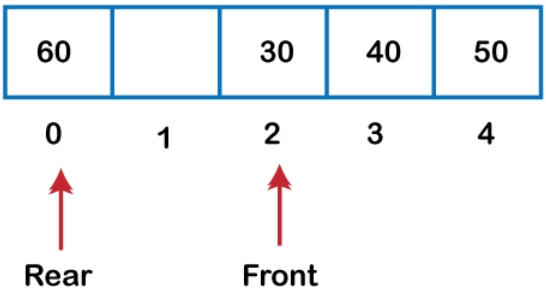
Front = 0
Rear = 0

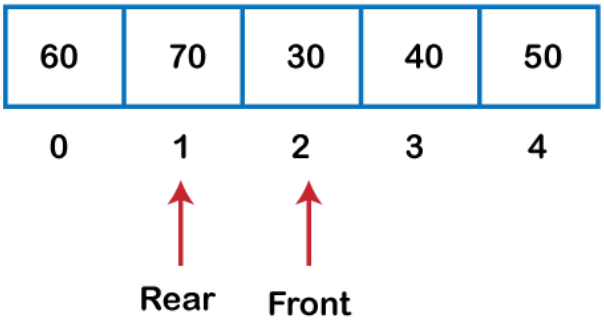

10
20
30
Front = 0
Rear = 2


10
20
30
40
0
1
2
3
4
Front = 0
Rear = 3


10
20
30
40
50
0
1
2
3
4
Front = 0
Rear = 4


30
40
50
0
1
2
3
4
dequeue
Front = 2
Rear = 4





Deque (or double-ended queue)

What is a queue?

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. Insertion in the queue is done from one end known as the **rear end** or the **tail,** 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 person enters last in the queue gets the ticket at last.

What is a Deque (or double-ended queue)

The deque stands for Double Ended Queue. Deque is a linear data structure where the insertion and deletion operations are performed from both ends. We can say that deque is a generalized version of the queue.

Though the insertion and deletion in a deque can be performed on both ends, it does not follow the FIFO rule. The representation of a deque is given as follows -

Representation of deque

Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.

input restricted double ended queue

Output restricted Queue

In output restricted queue, deletion operation can be performed at only one end, while insertion can be performed from both ends.

Output restricted double ended queue

Operations performed on deque

There are the following operations that can be applied on a deque -

- Insertion at front
- Insertion at rear
- Deletion at front
- Deletion at rear

We can also perform peek operations in the deque along with the operations listed above. Through peek operation, we can get the deque's front and rear elements of the deque. So, in addition to the above operations, following operations are also supported in deque -

- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not

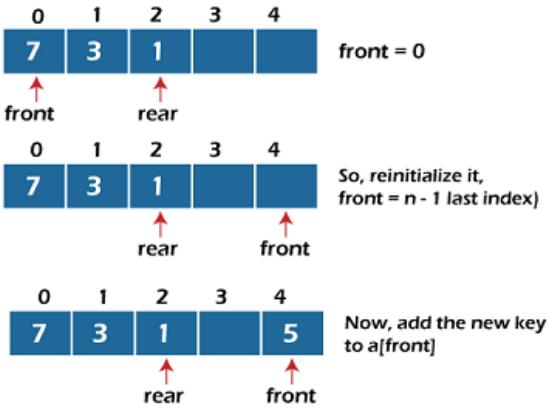
- Checks whether the deque is empty or not

Now, let's understand the operation performed on deque using an example.

Insertion at the front end

In this operation, the element is inserted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is full or not. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

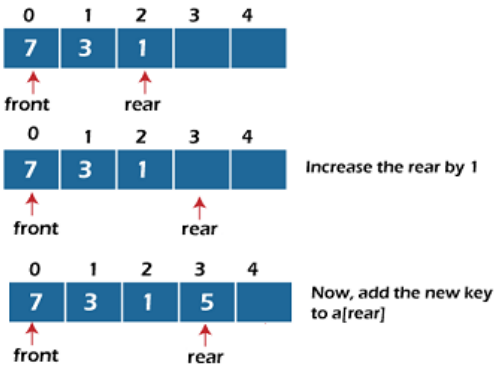
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



Insertion at the rear end

In this operation, the element is inserted from the rear end of the queue. Before implementing the operation, we first have to check again whether the queue is full or not. If the queue is not full, then the element can be inserted from the rear end by using the below conditions -

- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, increment the rear by 1. If the rear is at last index (or size - 1), then instead of increasing it by 1, we have to make it equal to 0.



Deletion at the front end

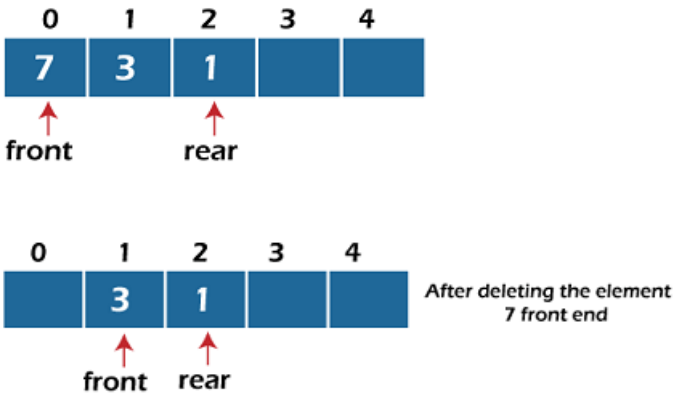
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

If the deque has only one element, set rear = -1 and front = -1.

Else if front is at end (that means front = size - 1), set front = 0.

Else increment the front by 1, (i.e., front = front + 1).



Deletion at the rear end

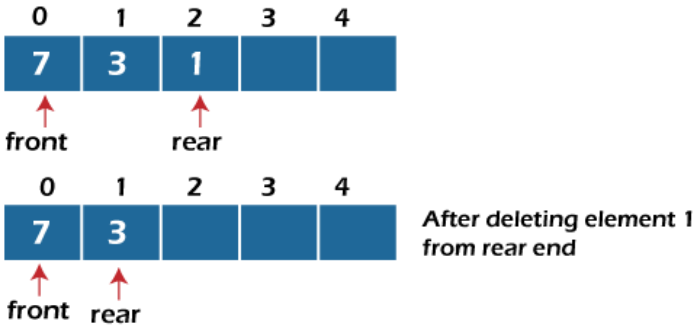
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion.

If the deque has only one element, set rear = -1 and front = -1.

If rear = 0 (rear is at front), then set rear = n - 1.

Else, decrement the rear by 1 (or, rear = rear -1).



Check empty

This operation is performed to check whether the deque is empty or not. If front = -1, it means that the deque is empty.

Check full

This operation is performed to check whether the deque is full or not. If front = rear + 1, or front = 0 and rear = n - 1 it means that the deque is full.

The time complexity of all of the above operations of the deque is O(1), i.e., constant.

Applications of deque

- Deque can be used as both stack and queue, as it supports both operations.
- Deque can be used as a palindrome checker means that if we read the string from both ends, the string would be the same.

Priority queue

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Characteristics of a Priority queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

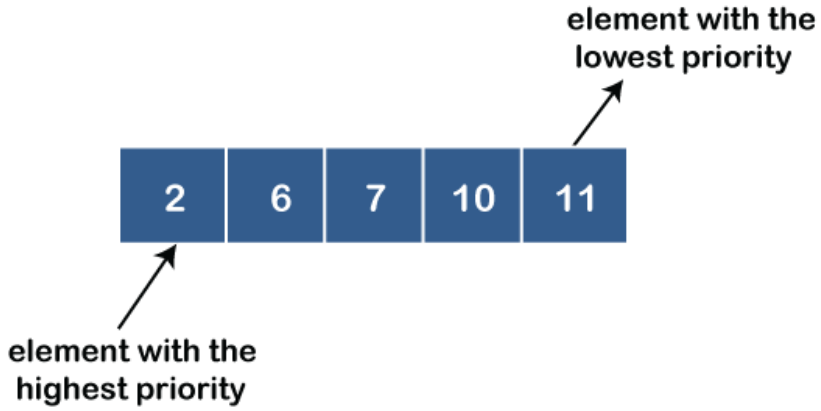
- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.

- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

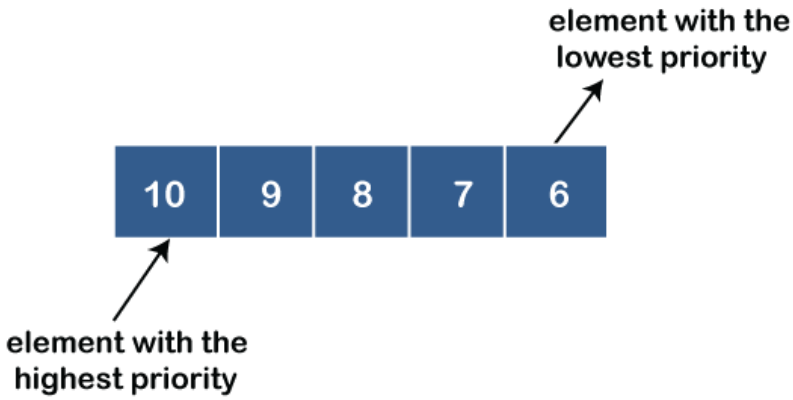
Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.

Let's create the priority queue step by step.

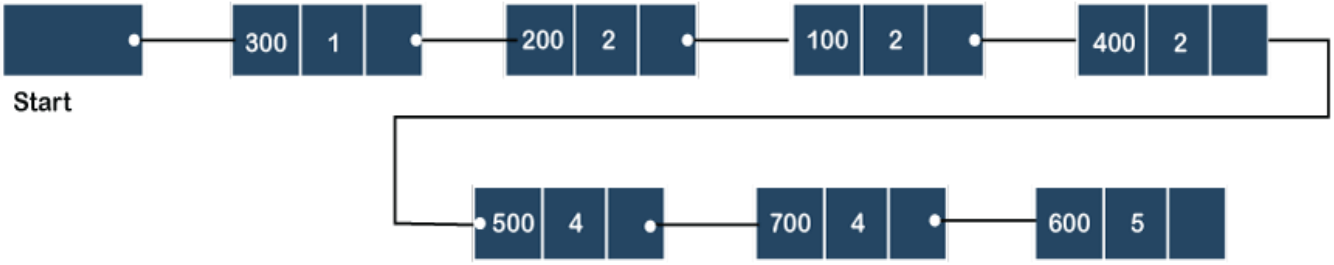
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

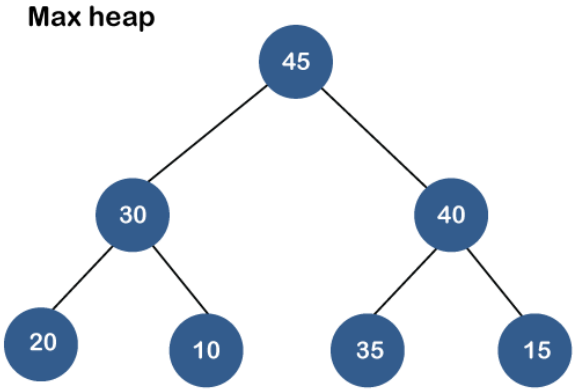
Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

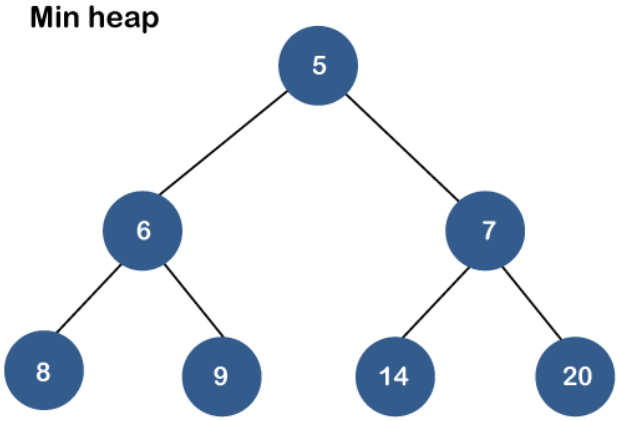
What is Heap?

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.



- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.



Both the heaps are the binary heap, as each has exactly two child nodes.

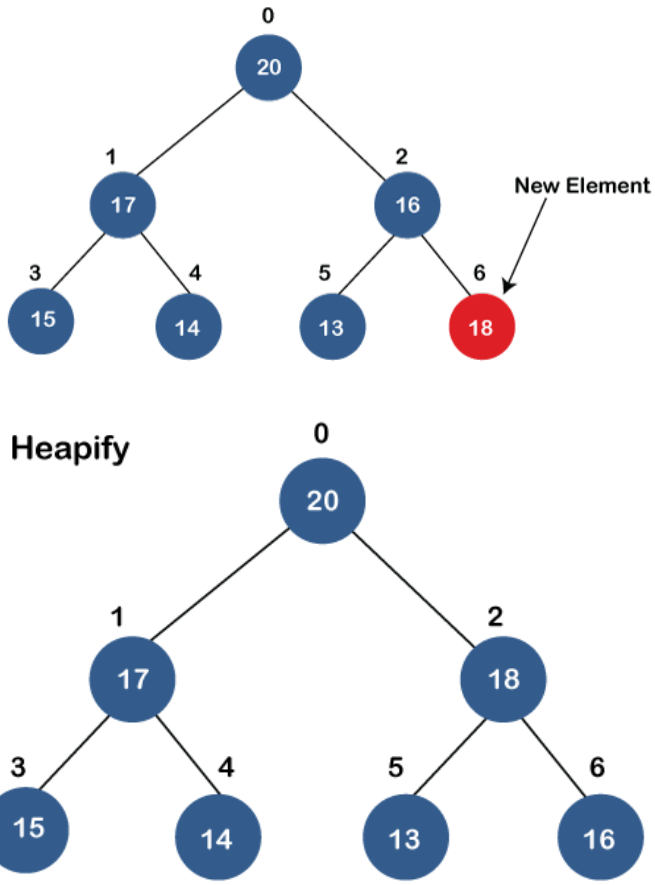
Priority Queue Operations

The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

- **Inserting the element in a priority queue (max heap)**

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.



- **Removing the minimum element from the priority queue**

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

Arithmetic Notations

Any arithmetic expression consists of operands and operators. The way we arrange our operators and operands to write the arithmetic expression is called Notation.

The following are the three different notations for writing the Arithmetic expression.

Infix Expression

An expression is said to be in infix notation if the operators in the expression are placed in between the operands on which the operator works. For example,

```
a + b * c
```

Infix expressions are easy to read, write and understand by humans, but not by computer It's costly, in terms of time and space, to process Infix expressions

Postfix expression (Reverse Polish Notation)

An expression is said to be in postfix notation if the operators in the expression are placed after the operands on which the operator works. For example,

```
abc*+
```

It's the most used notation for evaluating arithmetic expressions

Prefix expression (or Polish Notation)

An expression is said to be in prefix notation if the operators in the expression are placed before the operands on which the operator works. For example,

```
+a*bc
```

Precedence Order and Associativity of Operators

The precedence order of operators is given in the below table.

Precedence	Type	Operators	Associativity
1	Postfix	() [] -> . ++ —	Left to Right
2	Unary	+ – ! ~ ++ — (type)* & sizeof	Right to Left
3	Multiplicative	* / %	Left to Right
4	Additive	+ –	Left to Right
5	Shift	<<, >>	Left to Right
6	Relational	< <= > >=	Left to Right
7	Equality	== !=	Left to Right

8	Bitwise AND	&	Left to Right
9	Bitwise XOR	^	Left to Right
10	Bitwise OR	|	Left to Right
11	Logical AND	&&	Left to Right
12	Logical OR	||	Left to Right
13	Conditional	?:	Right to Left
14	Assignment	= += -+ *= /= %= >>= <<= &= ^= |=	Right to Left
15	Comma	,	Left to Right

we are going to see how we can convert Infix expressions to postfix expressions. For simplicity and understanding purposes, we will consider only '+', '-', '/', '*', '(' and ')'.

Infix to Postfix Conversion using Stack in C

Conversion of Infix to Postfix can be done using stack. The stack is used to reverse the order of operators. Stack stores the operator because it can not be added to the postfix expression until both of its operands are added. The precedence of the operator also matters while converting infix to postfix using stack, which we will discuss in the algorithm. Note: Parentheses are used to override the precedence of operators, and they can be nested parentheses that need to be evaluated from inner to outer.

Algorithm for Conversion of Infix to Postfix using Stack in C

Here are the steps of the algorithm to convert Infix to postfix using stack in C:

- Scan all the symbols one by one from left to right in the given Infix Expression.
- If the reading symbol is an operand, then immediately append it to the Postfix Expression.
- If the reading symbol is left parenthesis '(', then Push it onto the Stack.
- If the reading symbol is right parenthesis ')', then Pop all the contents of the stack until the respective left parenthesis is popped and append each popped symbol to Postfix Expression.
- If the reading symbol is an operator (+, –, *, /), then Push it onto the Stack. However, first, pop the operators which are already on the stack that have higher or equal precedence than the current operator and append them to the postfix. If an open parenthesis is there on top of the stack then push the operator into the stack.
- If the input is over, pop all the remaining symbols from the stack and append them to the postfix.

Infix to Postfix Conversion using Stack in C

Infix expression => a * (b + c + d)

(1) Infix: a * (b + c + d)

'a' is an operand append it to the post fix

Stack:

Postfix: a

(2) Infix: a * (b + c + d)

'*' is an operator push it into the stack after removing the higher or equal precedence operator.

Stack: *

Postfix: a

(3) Infix: a * (b + c + d)

'(' is left parenthesis push it into the stack

Stack: (*

Postfix: a

(4) Infix: a * (b + c + d)

'b' is operand append to postfix

Stack: (*

Postfix: ab

(5) Infix: a * (b + c + d)

'+' is an operator follow the step 5 of algorithm. Top of stack is open parantheses, So just push it.

Stack: + (*

Postfix: ab

(6) Infix: a * (b + c + d)

'c' is an operator follow the step 2 of algorithm and append it to the postfix.

Stack: + (*

Postfix: abc

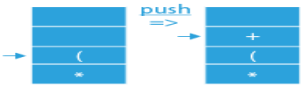
(7) Infix: a * (b + c + d)

=> '+' is an operator, so push it into stack after removing higher or equal priority operators from top of stack.

=> Current top of stack is '+', it has equal precedence with with input symbol '+'. So, pop it and append to postfix.

postfix => a b c +

=> Now the top of stack is '(', so push the input operator.



(8) Infix: a * (b + c + d)

'd' is an operand. append it to postfix.

Stack: + (*

Postfix: a b c + d

(9) Infix: a * (b + c + d)

')' is an right paranthesis, flow step n of algorithm and pop all content of stack until respective '(' is popped and append to the output.

Stack: *

Postfix: a b c + d +

(10) Infix: a * (b + c + d)

Input is over, so pop all remaining symbol and append to the postfix.

Stack:

Postfix: a b c + d + *

Final Expression

Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Let us dry run the above infix expression and find out its corresponding postfix expression.

The above string is parsed from left to right. For each token, the elements in the stack as well as the corresponding postfix expression up to that point is shown using the table below:

Element	Stack contents	Postfix Expression
K		K
+	+	
L	+	K L
-	-	K L +
M	-	K L + M
*	-*	K L + M
N	-*	K L + M N
+	+	K L + M N * -
(	+ (	K L + M N * -
O	+ (^	K L + M N * - O
^	+ (^	K L + M N * - O
P	+ (^	K L + M N * - O P
)	+	K L + M N * - O P ^
*	+ *	K L + M N* - O P ^
W	+ *	K L + M N* - O P ^ W
/	+ /	K L + M N* - O P ^ W *
U	+ /	K L + M N* - O P ^W * U
/	+ /	K L + M N* - O P ^W * U /
V	+ /	K L + M N* - O P ^ W * U / V
*	+ *	K L + M N* - O P ^W * U / V /
T	+ *	K L + M N* - O P ^W * U / V / T
+	+	K L + M N* - O P ^W * U / V / T * +
Q	+	K L + M N* - O P ^W * U / V / T * + Q
		K L + M N* - O P ^W * U / V / T * + Q+

The final postfix expression is $KL+MN*-OPW*U/V/T*+Q+$

Explanation of the above table:

At step 1, since we have an operand (K), we are appending it our postfix operation. Then we encounter an operator(+), and check that our stack is empty. We push the operator in our stack. Next, we encounter another operand (L), and append it to our postfix operation. Now after step 3, our postfix expression contains KL, and stack contains +. The next element is (-), and we check that top of stack contains +, which holds equal precedence as (-). Thus we pop out (+), and append it our expression, which now looks like KL+. Since the stack is empty, (-) goes inside the stack. We continue this till the last element of our infix expression. To understand why an element is being added to the postfix expression or the stack, or popped out from the stack, refer to the rules for conversion discussed in the previous section.

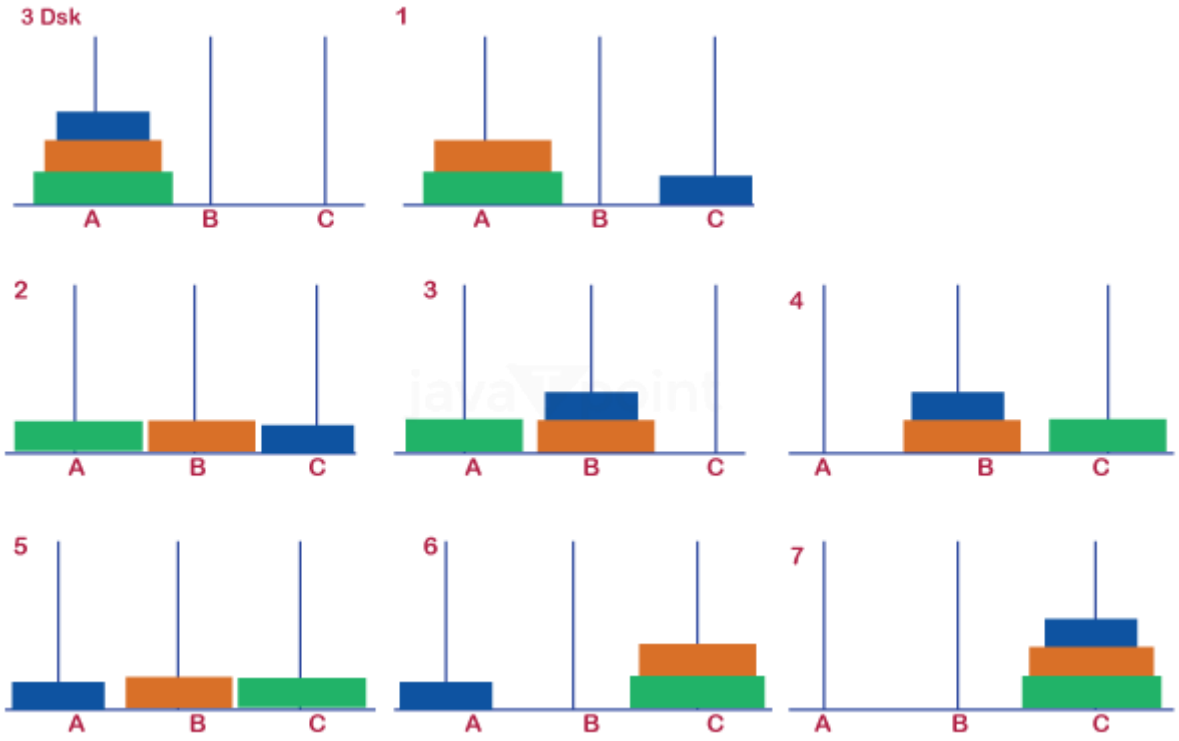
Tower of Hanoi

1. It is a classic problem where you try to move all the disks from one peg to another peg using only three pegs.

2. Initially, all of the disks are stacked on top of each other with larger disks under the smaller disks.

3. You may move the disks to any of three pegs as you attempt to relocate all of the disks, but you cannot place the larger disks over smaller disks and only one disk can be transferred at a time.

This problem can be easily solved by Divide & Conquer algorithm



In the above 7 step all the disks from peg A will be transferred to C given Condition:

1. Only one disk will be shifted at a time.
2. Smaller disk can be placed on larger disk.

Let T (n) be the total time taken to move n disks from peg A to peg C

1. Moving n-1 disks from the first peg to the second peg. This can be done in T (n-1) steps.
2. Moving larger disks from the first peg to the third peg will require first one step.

3. Recursively moving n-1 disks from the second peg to the third peg will require again T (n-1) step.

So, total time taken T (n) = T (n-1)+1+ T(n-1)

A recursive algorithm for Tower of Hanoi can be driven as follows –

1. Identify the three pegs: source (A), auxiliary (B), and destination (C).
2. Arrange the discs on the source peg in decreasing order of size, with the largest disc at the bottom.
3. Move the top (n-1) discs from the source peg to the auxiliary peg using the destination peg as a temporary peg.
4. Move the largest disc from the source peg to the destination peg.
5. Move the (n-1) discs from the auxiliary peg to the destination peg using the source peg as a temporary peg.
6. Repeat recursively steps 3-5 recursively until all discs are moved from the source peg to the destination peg.