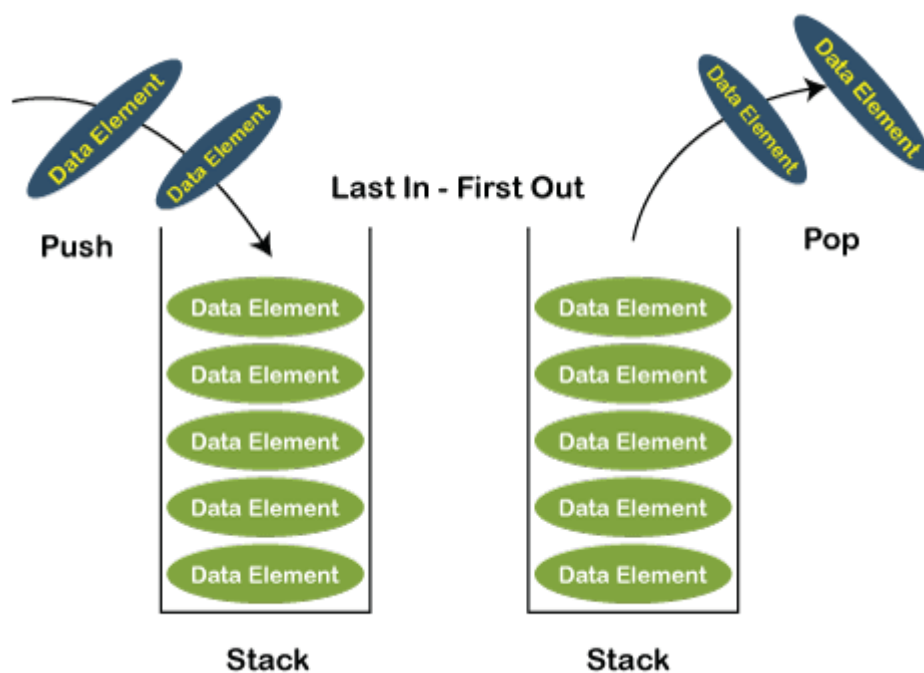


Stack vs. Queue

First, we will look at **what is stack** and **what is queue** individually, and then we will discuss the differences between stack and queue.

What is a Stack?

A Stack is a linear data structure. In case of an array, random access is possible, i.e., any element of an array can be accessed at any time, whereas in a stack, the sequential access is only possible. It is a container that follows the insertion and deletion rule. It follows the principle **LIFO (Last In First Out)** in which the insertion and deletion take place from one side known as a **top**. In stack, we can insert the elements of a similar data type, i.e., the different data type elements cannot be inserted in the same stack. The two operations are performed in LIFO, i.e., **push** and **pop** operation.



The following are the operations that can be performed on the stack:

- **push(x):** It is an operation in which the elements are inserted at the top of the stack. In the **push** function, we need to pass an element which we want to insert in a stack.

- **pop():** It is an operation in which the elements are deleted from the top of the stack. In the **pop()** function, we do not have to pass any argument.
- **peek()/top():** This function returns the value of the topmost element available in the stack. Like pop(), it returns the value of the topmost element but does not remove that element from the stack.
- **isEmpty():** If the stack is empty, then this function will return a true value or else it will return a false value.
- **isFull():** If the stack is full, then this function will return a true value or else it will return a false value.

In stack, the **top** is a pointer which is used to keep track of the last inserted element. To implement the stack, we should know the size of the stack. We need to allocate the memory to get the size of the stack. There are two ways to implement the stack:

- **Static:** The static implementation of the stack can be done with the help of arrays.
- **Dynamic:** The dynamic implementation of the stack can be done with the help of a linked list.

What is the Queue?

A Queue is a linear data structure. It is an ordered list that follows the principle FIFO (First In -First Out). A Queue is a structure that follows some restrictions on insertion and deletion. In the case of Queue, insertion is performed from one end, and that end is known as a rear end. The deletion is performed from another end, and that end is known as a front end. In Queue, the technical words for insertion and deletion are **enqueue()** and **dequeue()**, respectively whereas, in the case of the stack, the technical words for insertion and deletion are push() and pop(), respectively. Its structure contains two pointers **front pointer** and **rear pointer**, where the front pointer is a pointer that points to the element that was first added in the queue and the rear pointer that points to the element inserted last in the queue.

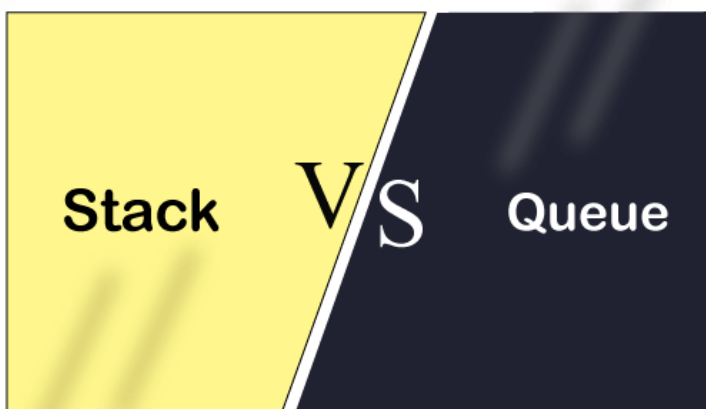


Similarities between stack and queue.

There are two similarities between the stack and queue:

- **Lineardatastructure**
Both the stack and queue are the linear data structure, which means that the elements are stored sequentially and accessed in a single run.
- **Flexibleinsize**
Both the stack and queue are flexible in size, which means they can grow and shrink according to the requirements at the run-time.

Differences between stack and queue



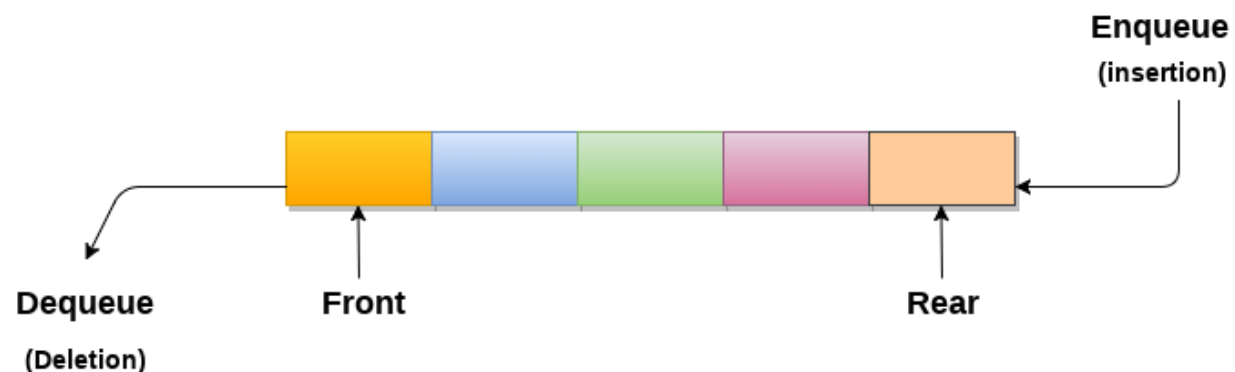
The following are the differences between the stack and queue:

Basis of Comparision	Stack	Queue
----------------------	-------	-------

Principle	It follows the principle LIFO (Last In-First Out), which implies that the element which is inserted last would be the first one to be deleted.	It follows the principle FIFO (First In -First Out), which implies that the element which is added first would be the first element to be removed from the list.
Structure	It has only one end from which both the insertion and deletion take place, and that end is known as a top.	It has two ends, i.e., front and rear end. The front end is used for the deletion while the rear end is used for the insertion.
Number of pointers used	It contains only one pointer known as a top pointer. The top pointer holds the address of the last inserted or the topmost element of the stack.	It contains two pointers front and rear pointer. The front pointer holds the address of the first element, whereas the rear pointer holds the address of the last element in a queue.
Operations performed	It performs two operations, push and pop. The push operation inserts the element in a list while the pop operation removes the element from the list.	It performs mainly two operations, enqueue and dequeue. The enqueue operation performs the insertion of the elements in a queue while the dequeue operation performs the deletion of the elements from the queue.
Examination of the empty condition	If $top == -1$, which means that the stack is empty.	If $front == -1$ or $front = rear + 1$, which means that the queue is empty.
Examination of full condition	If $top == \max - 1$, this condition implies that the stack is full.	If $rear == \max - 1$, this condition implies that the stack is full.
Variants	It does not have any types.	It is of three types like priority queue, circular queue and double ended queue.
Implementation	It has a simpler implementation.	It has a comparatively complex implementation than a stack.
Visualization	A Stack is visualized as a vertical collection.	A Queue is visualized as a horizontal collection.

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.

Types of Queues

Before understanding the types of queues, we first look at '**what is Queue**'.

What is the Queue?

A queue in the data structure can be considered similar to the queue in the real-world. A queue is a data structure in which whatever comes first will go out first. It follows the FIFO (First-In-First-Out) policy. In Queue, 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. In other words, it can be defined as a list or a collection with a constraint that the insertion can be performed at one end called as the rear end or tail of the queue and deletion is performed on another end called as the front end or the head of the queue.



Operations on Queue

There are two fundamental operations performed on a Queue:

- **Enqueue:** The enqueue operation is used to insert the element at the rear end of the queue. It returns void.
- **Dequeue:** The dequeue operation 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. The dequeue operation can also be designed to void.
- **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):** When the Queue is completely full, then it shows the overflow condition.
- **Queue underflow (isempty):** When the Queue is empty, i.e., no elements are in the Queue then it throws the underflow condition.

A Queue can be represented as a container opened from both the sides in which the element can be enqueued from one side and dequeued from another side as shown in the below figure:

What are the use cases of Queue?

Here, we will see the real-world scenarios where we can use the Queue data structure. The Queue data structure is mainly used where there is a shared resource that has to serve the multiple requests but can serve a single request at a time. In such cases, we need to use the Queue data structure for queuing up the requests. The request that arrives first in the queue will be served first. The following are the real-world scenarios in which the Queue concept is used:

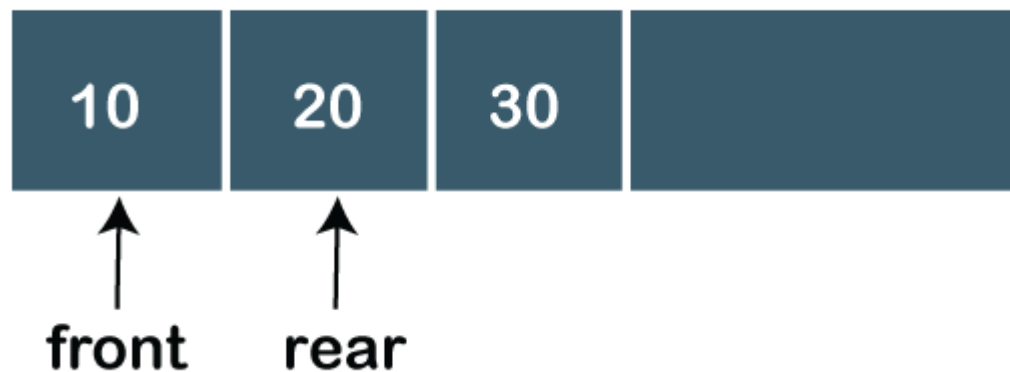
- Suppose we have a printer shared between various machines in a network, and any machine or computer in a network can send a print request to the printer. But, the printer can serve a single request at a time, i.e., a printer can print a single document at a time. When any print request comes from the network, and if the printer is busy, the printer's program will put the print request in a queue.
- . If the requests are available in the Queue, the printer takes a request from the front of the queue, and serves it.
- The processor in a computer is also used as a shared resource. There are multiple requests that the processor must execute, but the processor can serve a single request or execute a single process at a time. Therefore, the processes are kept in a Queue for execution.

Types of Queue

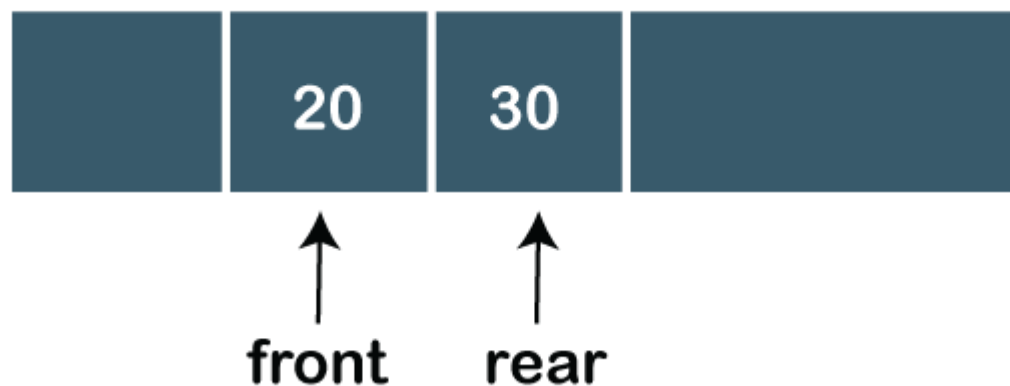
There are four types of Queues:

- **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 linear Queue can be represented, as shown in the below figure:



The above figure shows that the elements are inserted from the rear end, and if we insert more elements in a Queue, then the rear value gets incremented on every insertion. If we want to show the deletion, then it can be represented as:



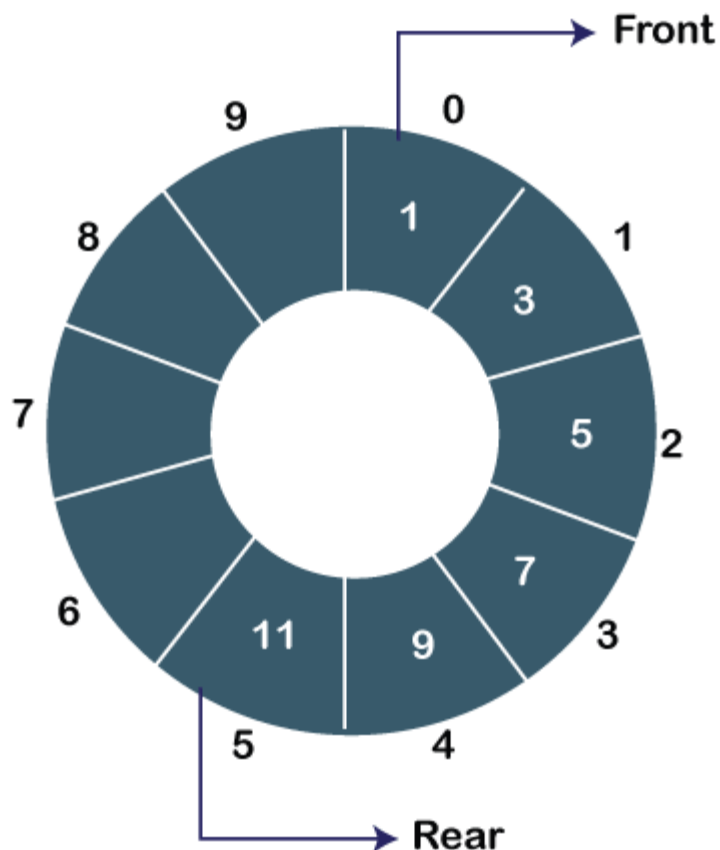
In the above figure, we can observe that the front pointer points to the next element, and the element which was previously pointed by the front pointer was deleted.

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 circular queue can be represented as:



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.

- **Priority Queue**

A priority queue is another special type of Queue data structure in which each element has some priority associated with it. Based on the priority of the element, the elements are arranged in a priority queue. If the elements occur with the same priority, then they are served according to the FIFO principle.

In priority Queue, the insertion takes place based on the arrival while the deletion occurs based on the priority. The priority Queue can be shown as:

The above figure shows that the highest priority element comes first and the elements of the same priority are arranged based on FIFO structure.

- **Deque**

Both the Linear Queue and Deque are different as the linear queue follows the FIFO principle whereas, deque does not follow the FIFO principle. In Deque, the insertion and deletion can occur from both ends.

Linear vs Circular Queue

What is a Linear Queue?

A linear queue is a linear data structure that serves the request first, which has been arrived first. It consists of data elements which are connected in a linear fashion. It has two pointers, i.e., front and rear, where the insertion takes place from the front end, and deletion occurs from the front end.



Operations on Linear Queue

There are two operations that can be performed on a linear queue:

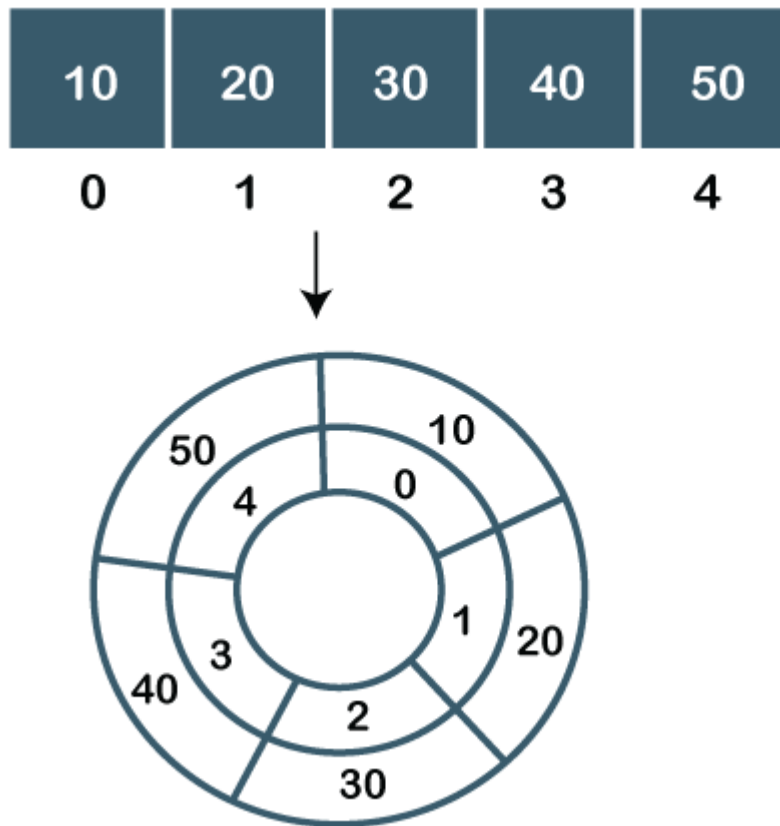
- **Enqueue:** The enqueue operation inserts the new element from the rear end.
- **Dequeue:** The dequeue operation is used to delete the existing element from the front end of the queue.

What is a Circular Queue?

As we know that in a queue, the front pointer points to the first element while the rear pointer points to the last element of the queue. The problem that arises with the linear queue is that if some empty cells occur at the beginning of the queue then we cannot insert new element at the empty space as the rear cannot be further incremented.

A circular queue is also a linear data structure like a normal queue that follows the FIFO principle but it does not end the queue; it connects the last position of the queue to the first position of the queue. If we want to insert new elements at the beginning of the queue, we can insert it using the circular queue data structure.

In the circular queue, when the rear reaches the end of the queue, then rear is reset to zero. It helps in refilling all the free spaces. The problem of managing the circular queue is overcome if the first position of the queue comes after the last position of the queue.



Conditions for the queue to be a circular queue

- Front == 0 and rear = n-1
- Front = rear + 1

If either of the above conditions is satisfied means that the queue is a circular queue.

Operations on Circular Queue

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

- **Enqueue:** It inserts an element in a queue. The given below are the scenarios that can be considered while inserting an element:
 1. If the queue is empty, then the front and rear are set to 0 to insert a new element.
 2. If queue is not empty, then the value of the rear gets incremented.
 3. If queue is not empty and rear is equal to n-1, then rear is set to 0.

- **Dequeue:** It performs a deletion operation in the Queue. The following are the points or cases that can be considered while deleting an element:
 1. If there is only one element in a queue, after the dequeue operation is performed on the queue, the queue will become empty. In this case, the front and rear values are set to -1.
 2. If the value of the front is equal to $n-1$, after the dequeue operation is performed, the value of the front variable is set to 0.
 3. If either of the above conditions is not fulfilled, then the front value is incremented.

Differences between linear Queue and Circular Queue

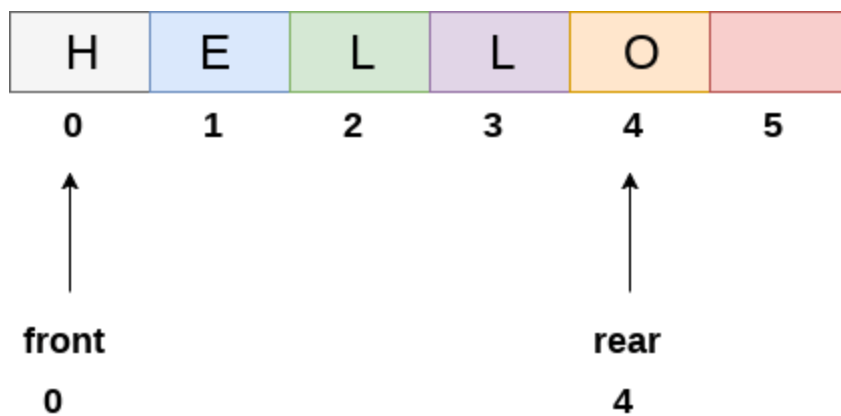


Basis of comparison	Linear Queue	Circular Queue
Meaning	The linear queue is a type of linear data structure that contains the elements in a sequential manner.	The circular queue is also a linear data structure in which the last element of the Queue is connected to the first element, thus creating a circle.
Insertion and Deletion	In linear queue, insertion is done from the rear end, and deletion is done from the front end.	In circular queue, the insertion and deletion can take place from any end.
Memory space	The memory space occupied by the linear queue is more than the circular queue.	It requires less memory as compared to linear queue.
Memory utilization	The usage of memory is inefficient.	The memory can be more efficiently utilized.
Order of	It follows the FIFO principle in order	It has no specific order for execution.

execution	to perform the tasks.	
------------------	-----------------------	--

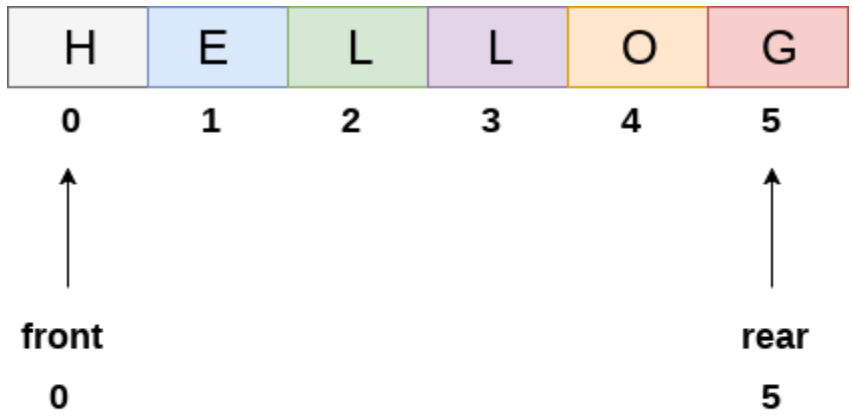
Array representation of Queue

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.



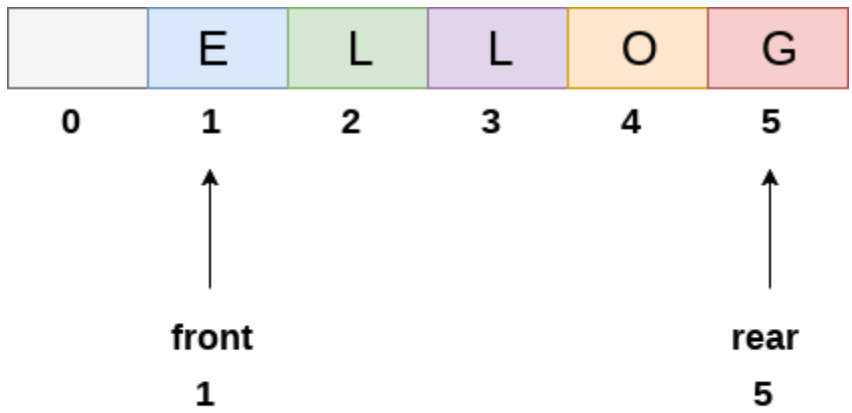
Queue

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

1. **void** insert (**int** queue[], **int** max, **int** front, **int** rear, **int** item)
2. {
3. **if** (rear + 1 == max)
4. {
5. printf("overflow");
6. }
7. **else**
8. {
9. **if**(front == -1 && rear == -1)

```

10.  {
11.    front = 0;
12.    rear = 0;
13.  }
14.  else
15.  {
16.    rear = rear + 1;
17.  }
18.  queue[rear]=item;
19. }
20. }

```

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

```

1. int delete (int queue[], int max, int front, int rear)
2. {
3.   int y;
4.   if (front == -1 || front > rear)
5.
6.   {

```

```

7.     printf("underflow");
8.   }
9.   else
10.  {
11.    y = queue[front];
12.    if(front == rear)
13.    {
14.      front = rear = -1;
15.      else
16.      front = front + 1;
17.
18.    }
19.    return y;
20.  }
21. }

```

Menu driven program to implement queue using array

```

1. #include<stdio.h>
2. #include<stdlib.h>
3. #define maxsize 5
4. void insert();
5. void delete();
6. void display();
7. int front = -1, rear = -1;
8. int queue[maxsize];
9. void main ()
10. {
11.   int choice;
12.   while(choice != 4)
13.   {
14.     printf("\n*****Main Menu*****\n");
15.     printf("\n=====
=====
=====\\n");
16.     printf("\n1.insert an element\n2.Delete an element\n3.Display the queue\n4.Exit\n");

```

```
17.    printf("\nEnter your choice ?");
18.    scanf("%d",&choice);
19.    switch(choice)
20.    {
21.        case 1:
22.            insert();
23.        break;
24.        case 2:
25.            delete();
26.        break;
27.        case 3:
28.            display();
29.        break;
30.        case 4:
31.            exit(0);
32.        break;
33.        default:
34.            printf("\nEnter valid choice??\n");
35.    }
36. }
37. }
38. void insert()
39. {
40.    int item;
41.    printf("\nEnter the element\n");
42.    scanf("\n%d",&item);
43.    if(rear == maxsize-1)
44.    {
45.        printf("\nOVERFLOW\n");
46.        return;
47.    }
48.    if(front == -1 && rear == -1)
49.    {
50.        front = 0;
```

```
51.     rear = 0;
52. }
53. else
54. {
55.     rear = rear+1;
56. }
57. queue[rear] = item;
58. printf("\nValue inserted ");
59.
60. }
61. void delete()
62. {
63.     int item;
64.     if (front == -1 || front > rear)
65.     {
66.         printf("\nUNDERFLOW\n");
67.         return;
68.
69.     }
70.     else
71.     {
72.         item = queue[front];
73.         if(front == rear)
74.         {
75.             front = -1;
76.             rear = -1 ;
77.         }
78.         else
79.         {
80.             front = front + 1;
81.         }
82.         printf("\nvalue deleted ");
83.     }
84.
```

```
85.
86. }
87.
88. void display()
89. {
90.     int i;
91.     if(rear == -1)
92.     {
93.         printf("\nEmpty queue\n");
94.     }
95.     else
96.     { printf("\nprinting values ..... \n");
97.         for(i=front;i<=rear;i++)
98.         {
99.             printf("\n%d\n",queue[i]);
100.        }
101.    }
102. }
```

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*****
=====

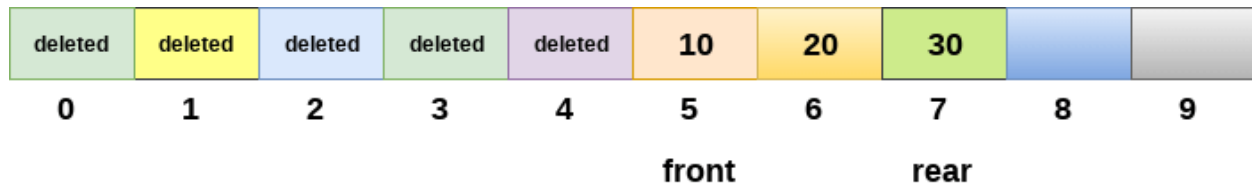
1.insert an element
2.Delete an element
3.Display the queue
4.Exit

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**

One 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.

Stacks And Stack Operations

A stack is a simple last-in, first-out (LIFO) data structure. That is, the last data element stored in a stack is the first data element retrieved from the stack. The common analogy is a stack of plates in a cafeteria: when you go through the line, you *pop* the top plate off the stack; the dish washer (stepping away from reality a bit), pushes a single clean plate on top of the stack. So a stack supports two basic operations: *push* and *pop*. Some stacks also provide additional operations: *size* (the number of data elements currently on the stack) and *peek* (look at the top element without removing it).

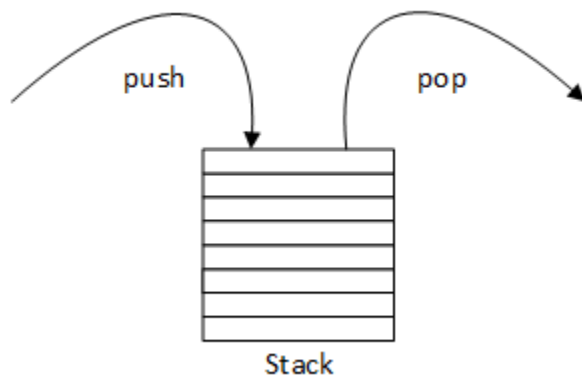


Fig-1 The primary stack operations. A new data element is stored by pushing it on the top of the stack. A data element is retrieved by popping the top element off the top of the stack and returning it.

Stacks are an important data structure in their own right and they may be implemented in several ways. We will use one of the ways of building a stack as an example of how to create and use arrays. The bottom of the stack is the first element of the array (i.e., the element at index location 0). But how do we know the location of the element at the top of the stack? Keeping track of the top element requires a second bit of information, an integer called the *stack pointer* that denotes the current top of the stack. The stack pointer is just an index into the array that represents the stack.

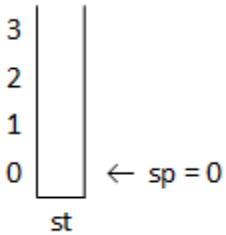


Fig2 Implementing a stack as an array. To better match the typical depiction of a stack, the array is drawn vertically rather than horizontally with the beginning of the array at the bottom and the other end left open. The array index numbers are shown on the left side of the array. Initially, the stack is empty, which is denoted by setting `sp` to 0.

We encounter two problems when we implement a stack as an array. Both the size of the stack (i.e., the maximum number of elements that the stack can hold) and the type of data that can be stored in the stack are limited by the requirement that the size and type of the array must be specified when the array is defined (i.e., when we write the code). We will return to and refine this initial implementation of a stack in subsequent chapters. Each refinement will get us closer to removing these (and other) problems but we won't see the final, clean result until we cover templates near the end of the text.

The following discussion focuses on how stacks work or behave and how that behavior can be achieved with arrays. So for now, we "solve" the two problems presented above by simply creating a stack that can only store characters implemented as a char array whose size is left ambiguous (specified as a [macro, enum, or const](#)).

```
char    st[SIZE];
int     sp = 0;
```

Stack Behavior

The various stack operations are easy to implement, but notice that push and pop use [post-increment and pre-decrement](#) respectively (this is crucial for the algorithm to work).

push

```
st[sp++] = data (sp must be < SIZE)
```

pop

```
return st[--sp] (sp must be > 0)
```

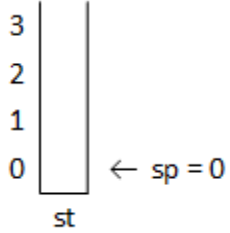
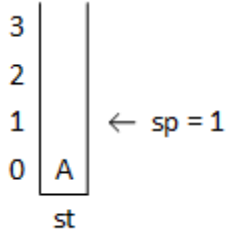
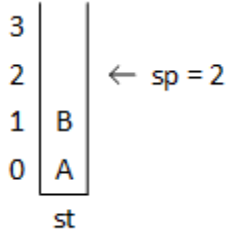
size

```
return sp;
```

peek

```
return st[sp-1]
```

Based on these operations, the snapshots shown in Figure 3 illustrate the appearance of a stack as data (characters) are stored in or pushed on to it.

Operation	Picture	Execution
Stack is empty		
push('A');		st[0] = 'A'; sp = 0 + 1;
push('B');		st[1] = 'B'; sp = 1 + 1;

<code>push('C');</code>	<pre> 3 2 C ← sp = 3 1 B 0 A st </pre>	<pre> st[2] = 'C'; sp = 2 + 1; </pre>
-------------------------	---	---------------------------------------

Fig:3 The push operation illustrated. Each call to the push function (left column) pushes a data element on to the stack. The main instruction in the push function is `st[sp++] = data`, where "data" is the function argument. The middle column abstractly illustrates how the stack (the array and the stack pointer) appear *after* each call to the push function. The right column breaks the behavior of the push function in to two steps.

Similarly, the data (characters) stored in a stack can be retrieved from or popped off of it.

Operation	Picture	Execution
<code>data = pop();</code>	<pre> 3 2 C ← sp = 2 1 B 0 A st </pre>	<pre> sp = 3 - 1; return sp[2]; </pre>
<code>data = pop();</code>	<pre> 3 2 C ← sp = 1 1 B 0 A st </pre>	<pre> sp = 2 - 1; return sp[2]; </pre>
<code>data = pop();</code>	<pre> 3 2 C ← sp = 0 1 B 0 A st </pre>	<pre> sp = 1 - 1; return sp[2]; </pre>

Fig:4 The pop operation illustrated. Each call to the pop function (middle column) removes a data element from the top of the stack and returns it. The main instruction in the pop function is `return`

`st[--sp]`. The middle column abstractly illustrates how the stack (the array and the stack pointer) appear *after* each call to the pop function. The right column breaks the behavior of the pop function in to two steps. Notice that the pop operations doesn't actually remove a character from the stack array, but the slots at and above the stack pointer are treated as empty. The next push operation will overwrite the data currently stored at the stack pointer. The stack appearing at the bottom of the table is *logically* empty.

Maintaining a stack as two discrete variables (an array and a stack pointer) is cumbersome, error-prone, and makes it difficult to support multiple stacks in a program. Fortunately, we can solve these problems (if somewhat inelegantly) if we implement a stack as a struct. But even after settling on a structure-based solution, there are still two possible paths that we can take: the first implementation is based on automatic variables while the second is based on dynamic variables. More elegant solutions based on classes and templates will follow in subsequent chapters.