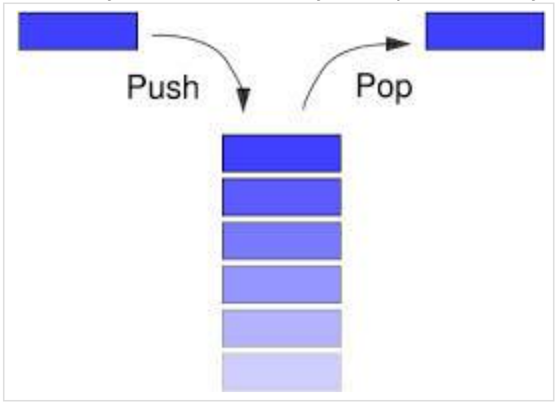
STACKS AND QUEUES

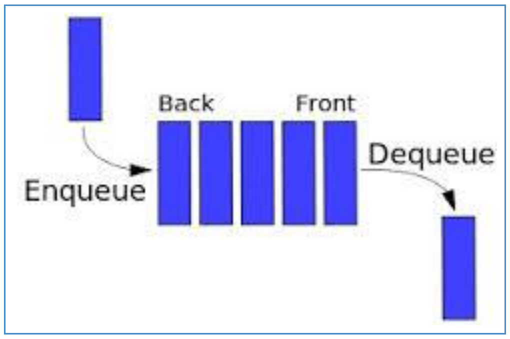
Stack:

In the pushdown stacks only two operations are allowed: push the item into the stack, and pop the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. push adds an item to the top of the stack, pop removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.



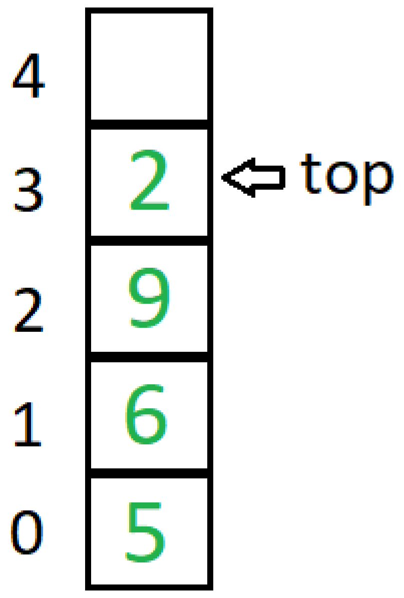
Queue:

An excellent example of a queue is a line of students in the food court of the UC. New additions to a line made to the back of the queue, while removal (or serving) happens in the front. In the queue only two operations are allowed enqueue and dequeue. Enqueue means to insert an item into the back of the queue, dequeue means removing the front item. The picture demonstrates the FIFO access. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



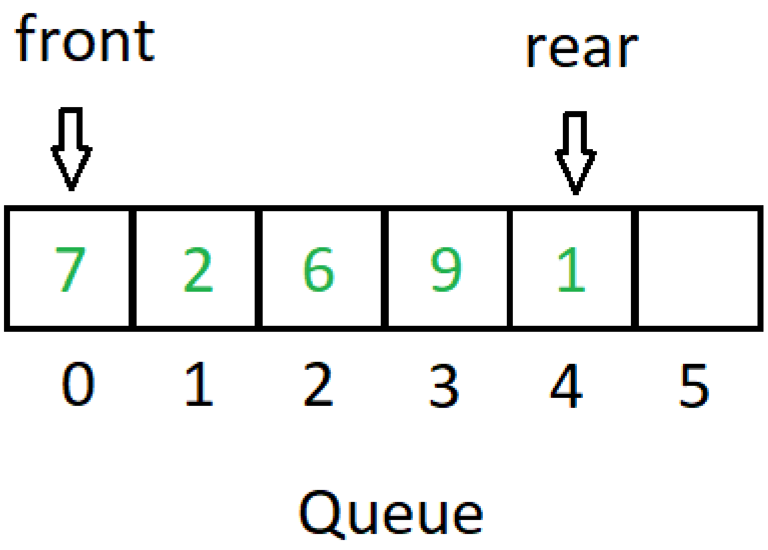
Difference between Stack and Queue Data Structures

Stack:- A stack is a linear data structure in which elements can be inserted and deleted only from one side of the list, called the **top**. A stack follows the **LIFO** (Last In First Out) principle, i.e., the element inserted at the last is the first element to come out. The insertion of an element into stack is called **push** operation, and deletion of an element from the stack is called **pop** operation. In stack we always keep track of the last element present in the list with a pointer called **top**.
The diagrammatic representation of stack is given below:



Stack

Queue:- A queue is a linear data structure in which elements can be inserted only from one side of the list called **rear**, and the elements can be deleted only from the other side called the **front**. The queue data structure follows the **FIFO** (First In First Out) principle, i.e. the element inserted at first in the list, is the first element to be removed from the list. The insertion of an element in a queue is called an **enqueue** operation and the deletion of an element is called a **dequeue** operation. In queue we always maintain two pointers, one pointing to the element which was inserted at the first and still present in the list with the **front** pointer and the second pointer pointing to the element inserted at the last with the **rear** pointer.
The diagrammatic representation of queue is given below:



Difference between Stack and Queue Data Structures

STACKS	QUEUES
Stacks are based on the LIFO principle, i.e., the element inserted at the last, is the first element to come out of the list.	Queues are based on the FIFO principle, i.e., the element inserted at the first, is the first element to come out of the list.
Insertion and deletion in stacks takes place only from one end of the list called the top.	Insertion and deletion in queues takes place from the opposite ends of the list. The insertion takes place at the rear of the list and the deletion takes place from the front of the list.
Insert operation is called push operation.	Insert operation is called enqueue operation.
Delete operation is called pop operation.	Delete operation is called dequeue operation.

STACKS	QUEUES
In stacks we maintain only one pointer to access the list, called the top, which always points to the last element present in the list.	In queues we maintain two pointers to access the list. The front pointer always points to the first element inserted in the list and is still present, and the rear pointer always points to the last inserted element.

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:

```
1. begin
2.     if top = n then stack full
3.     top = top + 1
4.     stack (top) : = item;
5. end
```

Time Complexity : o(1)

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 :

```
1. begin
2.     if top = 0 then stack empty;
3.     item := stack(top);
4.     top = top - 1;
5. end;
```

Time Complexity : o(1)

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 :

PEEK (STACK, TOP)

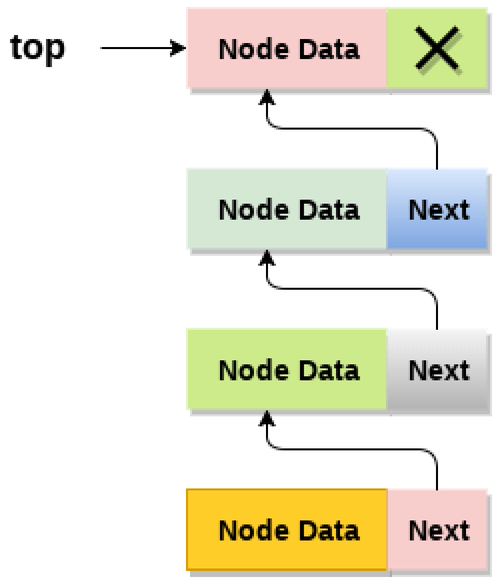
```
1. Begin
2.     if top = -1 then stack empty
3.     item = stack[top]
4.     return item
5. End
```

Time complexity: o(n)

Linked list implementation of stack

Instead of using array, we can also use linked list to implement stack. Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.

In linked list implementation of stack, the nodes are maintained non-contiguously in the memory. Each node contains a pointer to its immediate successor node in the stack. Stack is said to be overflown if the space left in the memory heap is not enough to create a node.



Stack

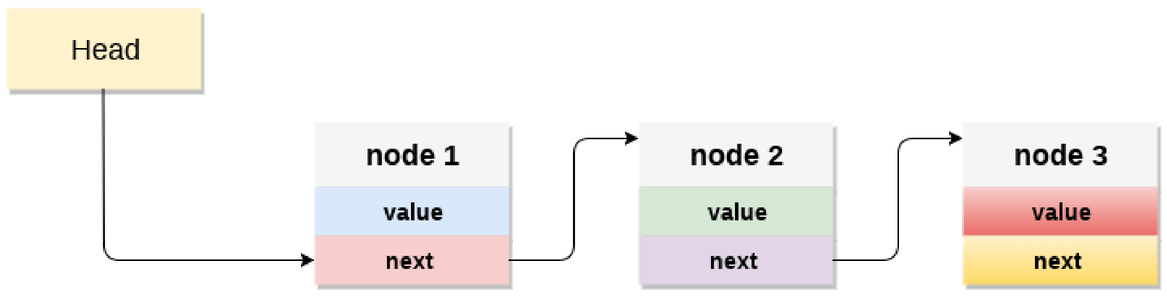
The top most node in the stack always contains null in its address field. Lets discuss the way in which, each operation is performed in linked list implementation of stack.

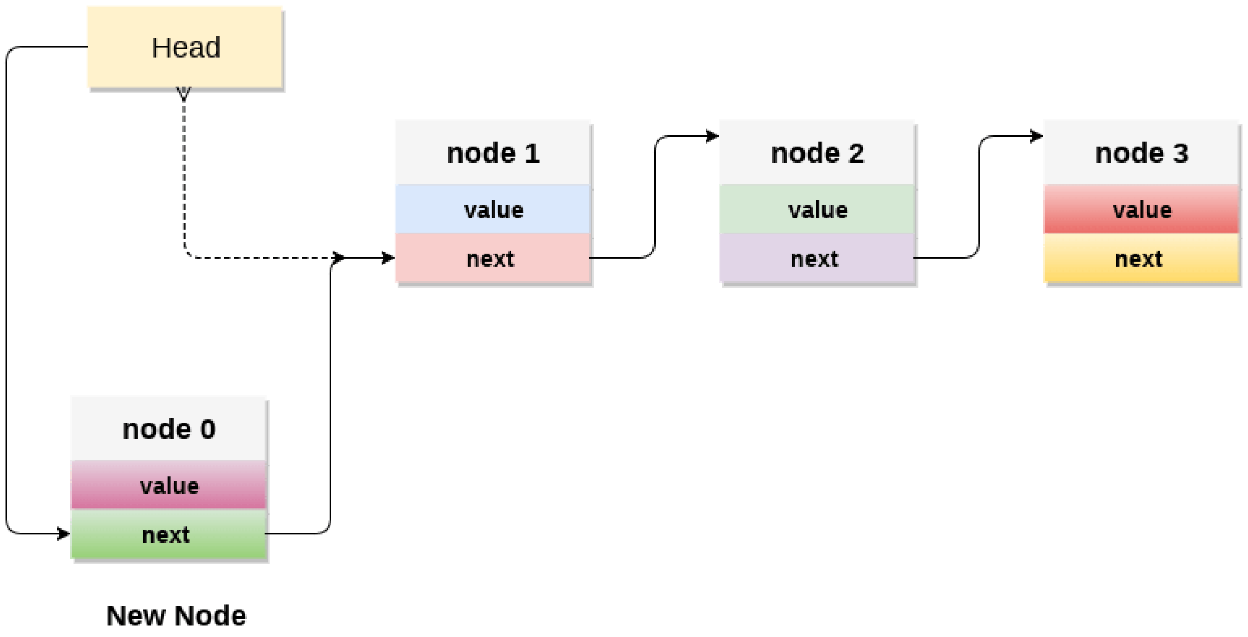
Adding a node to the stack (Push operation)

Adding a node to the stack is referred to as **push** operation. Pushing an element to a stack in linked list implementation is different from that of an array implementation. In order to push an element onto the stack, the following steps are involved.

1. Create a node first and allocate memory to it.
2. If the list is empty then the item is to be pushed as the start node of the list. This includes assigning value to the data part of the node and assign null to the address part of the node.
3. If there are some nodes in the list already, then we have to add the new element in the beginning of the list (to not violate the property of the stack). For this purpose, assign the address of the starting element to the address field of the new node and make the new node, the starting node of the list.

   Time Complexity : o(1)





Deleting a node from the stack (POP operation)

Deleting a node from the top of stack is referred to as **pop** operation. Deleting a node from the linked list implementation of stack is different from that in the array implementation. In order to pop an element from the stack, we need to follow the following steps :

1. **Check for the underflow condition:** The underflow condition occurs when we try to pop from an already empty stack. The stack will be empty if the head pointer of the list points to null.
2. **Adjust the head pointer accordingly:** In stack, the elements are popped only from one end, therefore, the value stored in the head pointer must be deleted and the node must be freed. The next node of the head node now becomes the head node.

Time Complexity : o(n)

Display the nodes (Traversing)

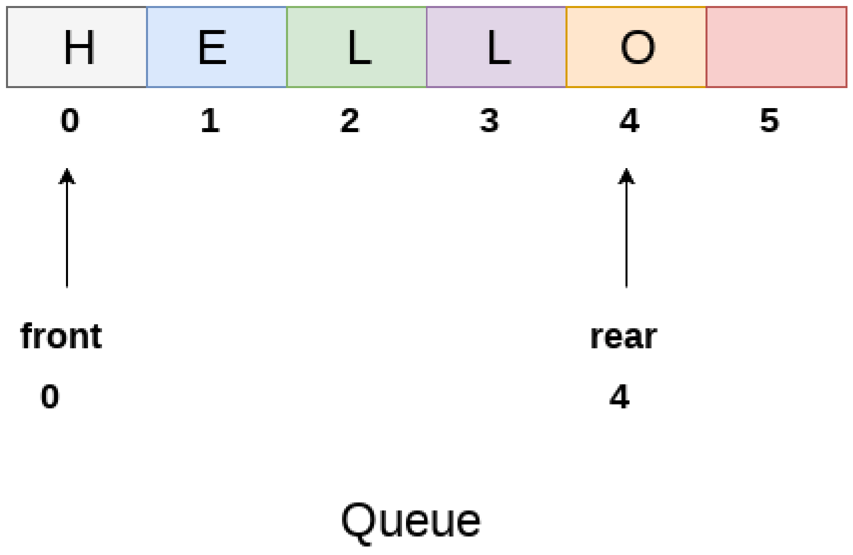
Displaying all the nodes of a stack needs traversing all the nodes of the linked list organized in the form of stack. For this purpose, we need to follow the following steps.

1. Copy the head pointer into a temporary pointer.
2. Move the temporary pointer through all the nodes of the list and print the value field attached to every node.

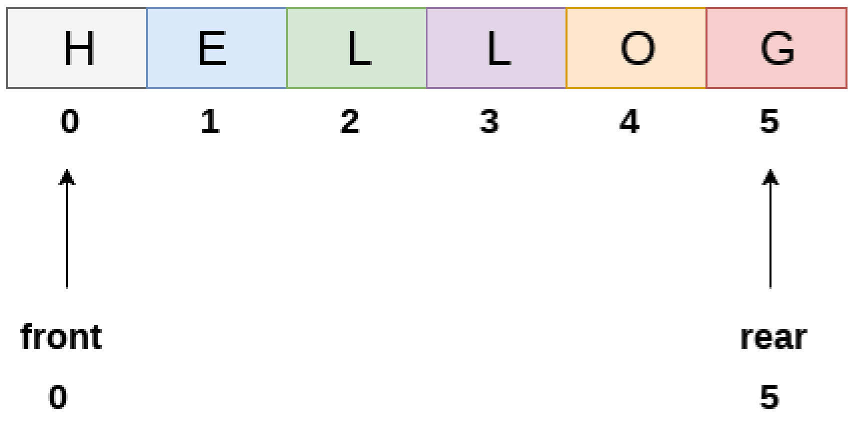
Time Complexity : o(n)

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.

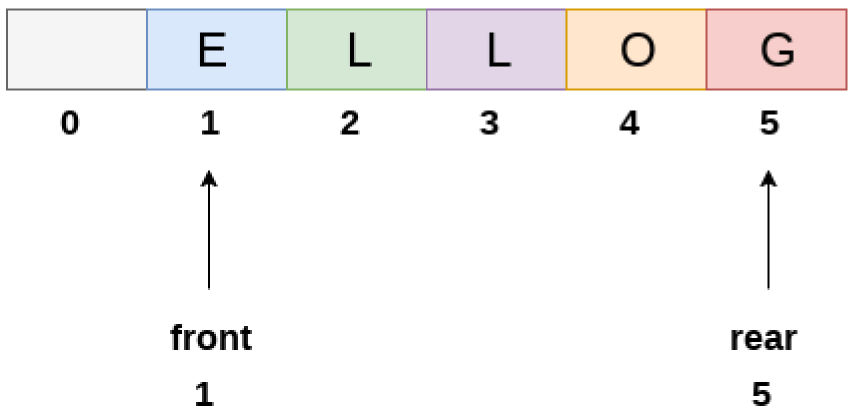


The above figure shows the queue of characters forming the English word **"HELLO"**. Since, No deletion is performed in the queue till now, therefore the value of front remains -1 . However, the value of rear increases by one every time an insertion is performed in the queue. After inserting an element 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

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.

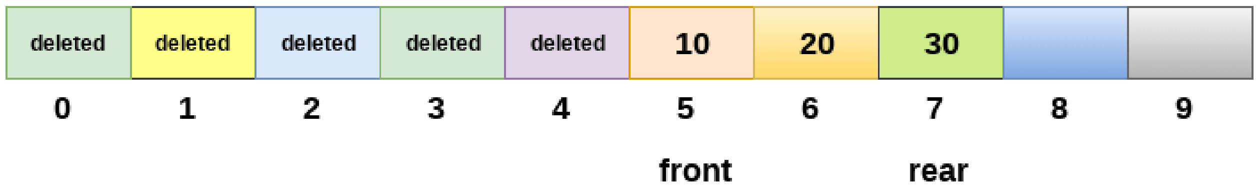
Algorithm

- **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

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.

Linked List implementation of Queue

Due to the drawbacks discussed in the previous section of this tutorial, the array implementation can not be used for the large scale applications where the queues are implemented. One of the alternative of array implementation is linked list implementation of queue.

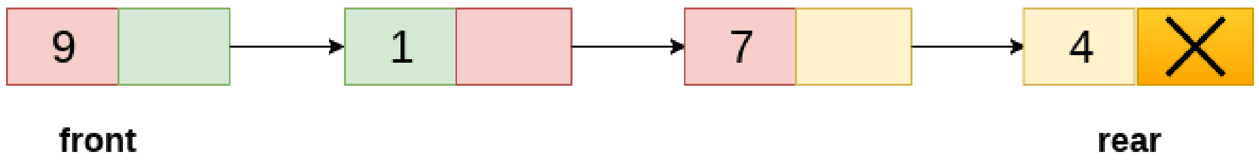
The storage requirement of linked representation of a queue with n elements is o(n) while the time requirement for operations is o(1).

In a linked queue, each node of the queue consists of two parts i.e. data part and the link part. Each element of the queue points to its immediate next element in the memory.

In the linked queue, there are two pointers maintained in the memory i.e. front pointer and rear pointer. The front pointer contains the address of the starting element of the queue while the rear pointer contains the address of the last element of the queue.

Insertion and deletions are performed at rear and front end respectively. If front and rear both are NULL, it indicates that the queue is empty.

The linked representation of queue is shown in the following figure.



Linked Queue

Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

Insert operation

The insert operation append the queue by adding an element to the end of the queue. The new element will be the last element of the queue.

Firstly, allocate the memory for the new node ptr by using the following statement.

```
Ptr = (struct node *) malloc (sizeof(struct node));
```

There can be the two scenario of inserting this new node ptr into the linked queue.

In the first scenario, we insert element into an empty queue. In this case, the condition **front = NULL** becomes true. Now, the new element will be added as the only element of the queue and the next pointer of front and rear pointer both, will point to NULL.

```
ptr -> data = item;
        if(front == NULL)
        {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
        }
```

In the second case, the queue contains more than one element. The condition front = NULL becomes false. In this scenario, we need to update the end pointer rear so that the next pointer of rear will point to the new node ptr. Since, this is a linked queue, hence we also need to make the rear pointer point to the newly added node **ptr**. We also need to make the next pointer of rear point to NULL.

```
rear -> next = ptr;
            rear = ptr;
```

3. rear->next = NULL;

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

Algorithm

```
○ Step 1: Allocate the space for the new node PTR
○ Step 2: SET PTR -> DATA = VAL
○ Step 3: IF FRONT = NULL
      SET FRONT = REAR = PTR
      SET FRONT -> NEXT = REAR -> NEXT = NULL
      ELSE
      SET REAR -> NEXT = PTR
      SET REAR = PTR
      SET REAR -> NEXT = NULL
      [END OF IF]
○ Step 4: END
```

Deletion

Deletion operation removes the element that is first inserted among all the queue elements. Firstly, we need to check either the list is empty or not. The condition front == NULL becomes true if the list is empty, in this case , we simply write underflow on the console and make exit.

Otherwise, we will delete the element that is pointed by the pointer front. For this purpose, copy the node pointed by the front pointer into the pointer ptr. Now, shift the front pointer, point to its next node and free the node pointed by the node ptr. This is done by using the following statements.

```
1. ptr = front;
2.      front = front -> next;
3.      free(ptr);
```

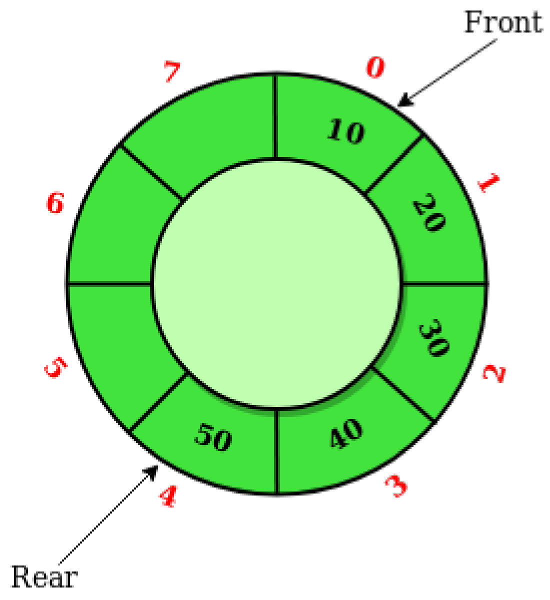
The algorithm and C function is given as follows.

Algorithm

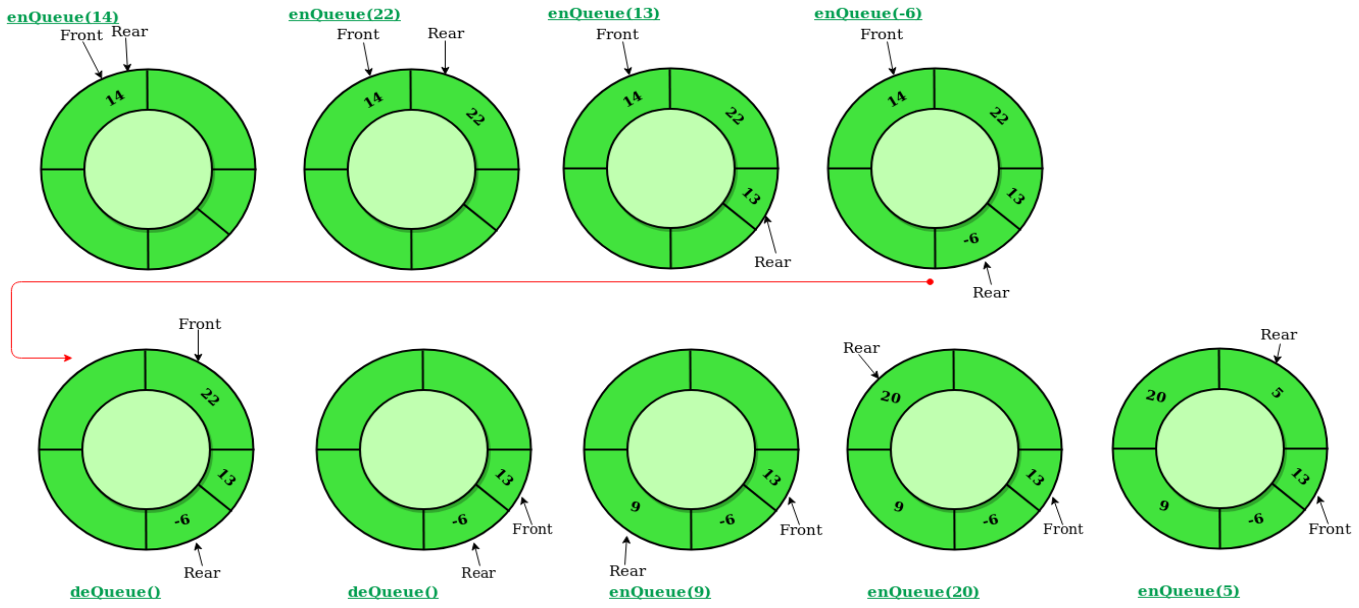
```
○ Step 1: IF FRONT = NULL
      Write " Underflow "
      Go to Step 5
      [END OF IF]
○ Step 2: SET PTR = FRONT
○ Step 3: SET FRONT = FRONT -> NEXT
○ Step 4: FREE PTR
○ Step 5: END
```

CIRCULAR QUEUES

Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **‘Ring Buffer’**.

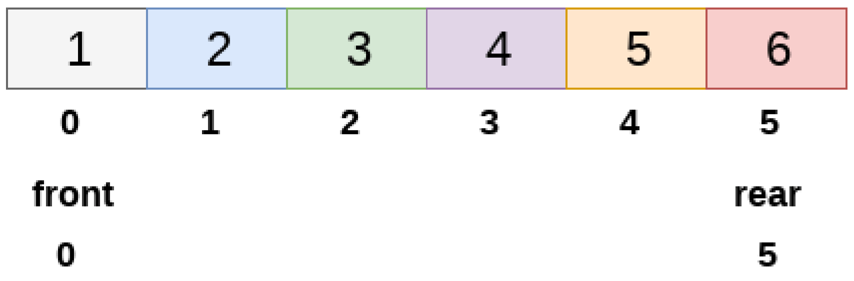


In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



- ➢ Deletions and insertions can only be performed at front and rear end respectively, as far as linear queue is concerned.

Consider the queue shown in the following figure.

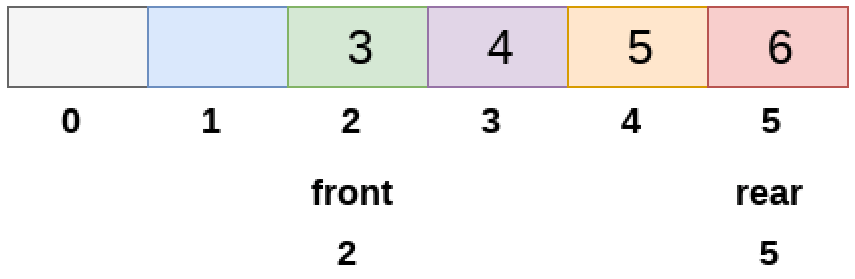


Queue

The Queue shown in above figure is completely filled and there can't be inserted any more element due to the condition **rear == max - 1 becomes true**.

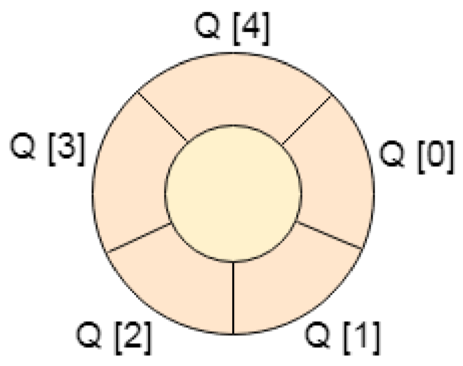
However, if we delete 2 elements at the front end of the queue, we still can not insert any element since the condition **rear = max -1 still holds**.

This is the main problem with the linear queue, although we have space available in the array, but we can not insert any more element in the queue. This is simply the memory wastage and we need to overcome this problem.



Queue after deletion of first 2 elements

One of the solution of this problem is circular queue. In the circular queue, the first index comes right after the last index. You can think of a circular queue as shown in the following figure.



Circular queue will be full when **front = -1** and **rear = max-1**. Implementation of circular queue is similar to that of a linear queue. Only the logic part that is implemented in the case of insertion and deletion is different from that in a linear queue.

Complexity

Time Complexity

Front	O(1)
Rear	O(1)
enQueue()	O(1)
deQueue()	O(1)

Insertion in Circular queue

There are three scenario of inserting an element in a queue.

1. **If (rear + 1)%maxsize = front**, the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.
2. **If rear != max - 1**, then rear will be incremented to the **mod(maxsize)** and the new value will be inserted at the rear end of the queue.
3. **If front != 0 and rear = max - 1**, then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

Algorithm to insert an element in 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
```

Algorithm to delete an element from a circular queue

To delete an element from the circular queue, we must check for the three following conditions.

1. If front = -1, then there are no elements in the queue and therefore this will be the case of an underflow condition.
2. If there is only one element in the queue, in this case, the condition rear = front holds and therefore, both are set to -1 and the queue is deleted completely.
3. If front = max -1 then, the value is deleted from the front end the value of front is set to 0.
4. Otherwise, the value of front is incremented by 1 and then delete the element at the front end.

Algorithm

```
○ 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
```

Priority Queue

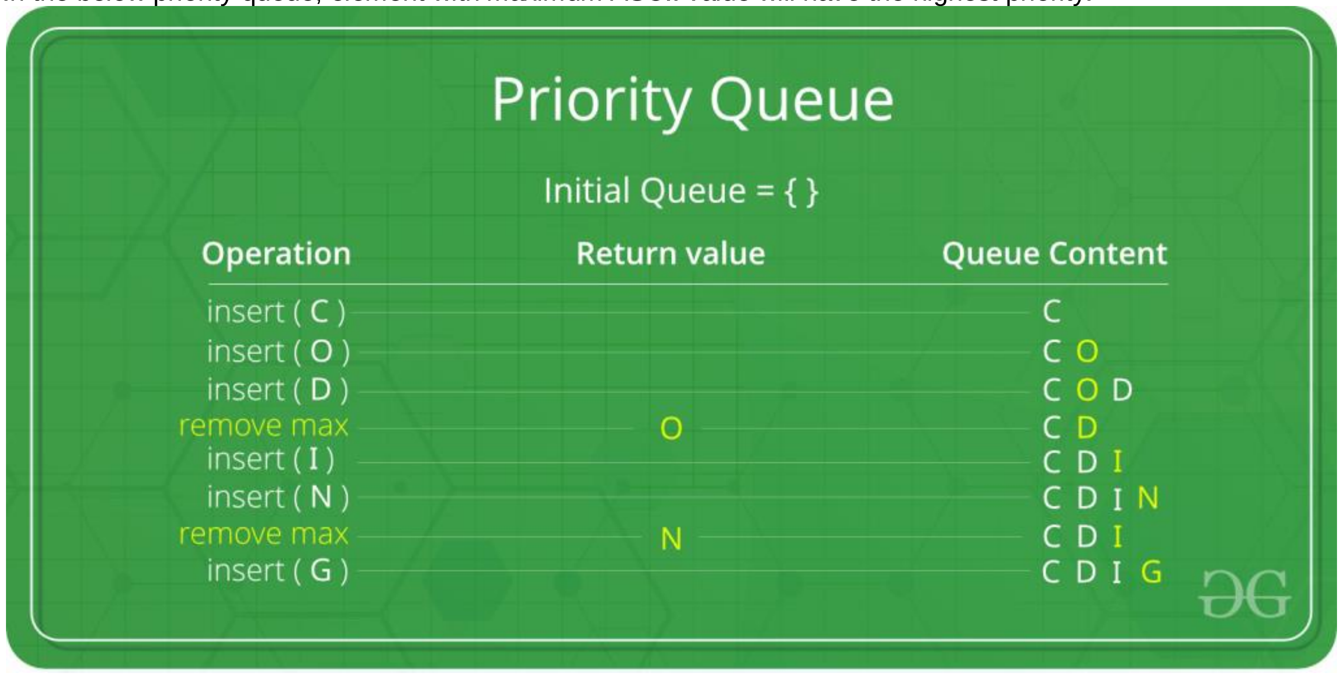
a **priority queue** is an abstract data type which is like a regular queue or stack data structure, but where additionally each element has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority. In some implementations, if two elements have the same priority, they are served according to the order in which they were enqueued, while in other implementations, ordering of elements with the same priority is undefined.

OR

Priority Queue is an extension of queue with following properties.

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

In the below priority queue, element with maximum ASCII value will have the highest priority.



Priority Queue

Initial Queue = { }

Operation	Return value	Queue Content
insert (C)		C
insert (O)		C O
insert (D)		C O D
remove max	O	C D
insert (I)		C D I
insert (N)		C D I N
remove max	N	C D I
insert (G)		C D I G

A typical priority queue supports following operations.
insert(item, priority): Inserts an item with given priority.
getHighestPriority(): Returns the highest priority item.
deleteHighestPriority(): Removes the highest priority item.

How to implement priority queue?
Using Array: A simple implementation is to use array of following structure.

```
struct item {
   int item;
   int priority;
}
```

insert() operation can be implemented by adding an item at end of array in O(1) time.

getHighestPriority() operation can be implemented by linearly searching the highest priority item in array. This operation takes O(n) time.

deleteHighestPriority() operation can be implemented by first linearly searching an item, then removing the item by moving all subsequent items one position back.

We can also use Linked List, time complexity of all operations with linked list remains same as array. The advantage with linked list is deleteHighestPriority() can be more efficient as we don't have to move items.