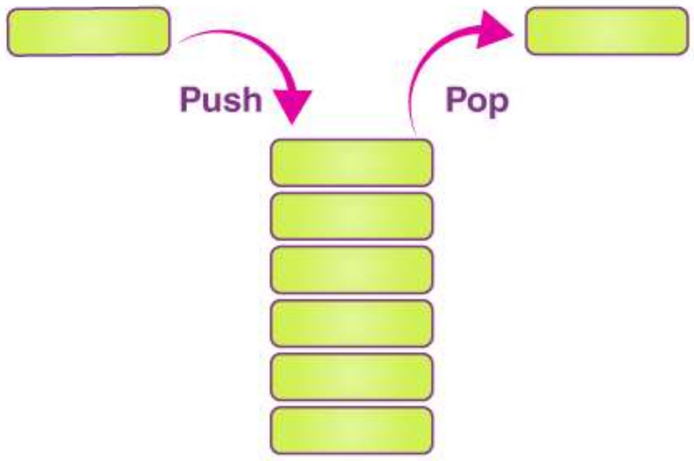
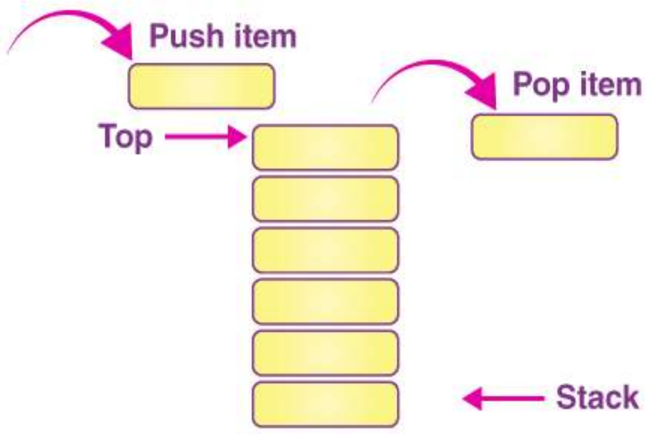
What is stack?

A Stack is a linear data structure that holds a linear, ordered sequence of elements. It is an abstract data type. A Stack works on the LIFO process (Last In First Out), i.e., the element that was inserted last will be removed first. To implement the Stack, it is required to maintain a pointer to the top of the Stack, which is the last element to be inserted because we can access the elements only on the top of the Stack.



A stack is a linear data structure which uses the same principle, i.e., the elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

Operation on Stack



1. PUSH: PUSH operation implies the insertion of a new element into a Stack. A new element is always inserted from the topmost position of the Stack; thus, we always need to check if the top is empty or not, i.e., TOP=Max-1 if this condition goes false, it means the Stack is full, and no more elements can be inserted, and even if we try to insert the element, a Stack overflow message will be displayed.

Algorithm:

```
Step-1: If TOP = Max-1
Print "Overflow"
Goto Step 4
Step-2: Set TOP= TOP + 1
Step-3: Set Stack[TOP]= ELEMENT
Step-4: END
```

2. POP: POP means to delete an element from the Stack. Before deleting an element, make sure to check if the Stack Top is NULL, i.e., TOP=NULL. If this condition goes true, it means the Stack is empty, and no deletion operation can be performed, and even if we try to delete, then the Stack underflow message will be generated.

Algorithm:

```
Step-1: If TOP= NULL
    Print “Underflow”
    Goto Step 4
Step-2: Set VAL= Stack[TOP]
Step-3: Set TOP= TOP-1
Step-4: END
```

3. PEEK: When we need to return the value of the topmost element of the Stack without deleting it from the Stack, the Peek operation is used. This operation first checks if the Stack is empty, i.e., TOP = NULL; if it is so, then an appropriate message will display, else the value will return.

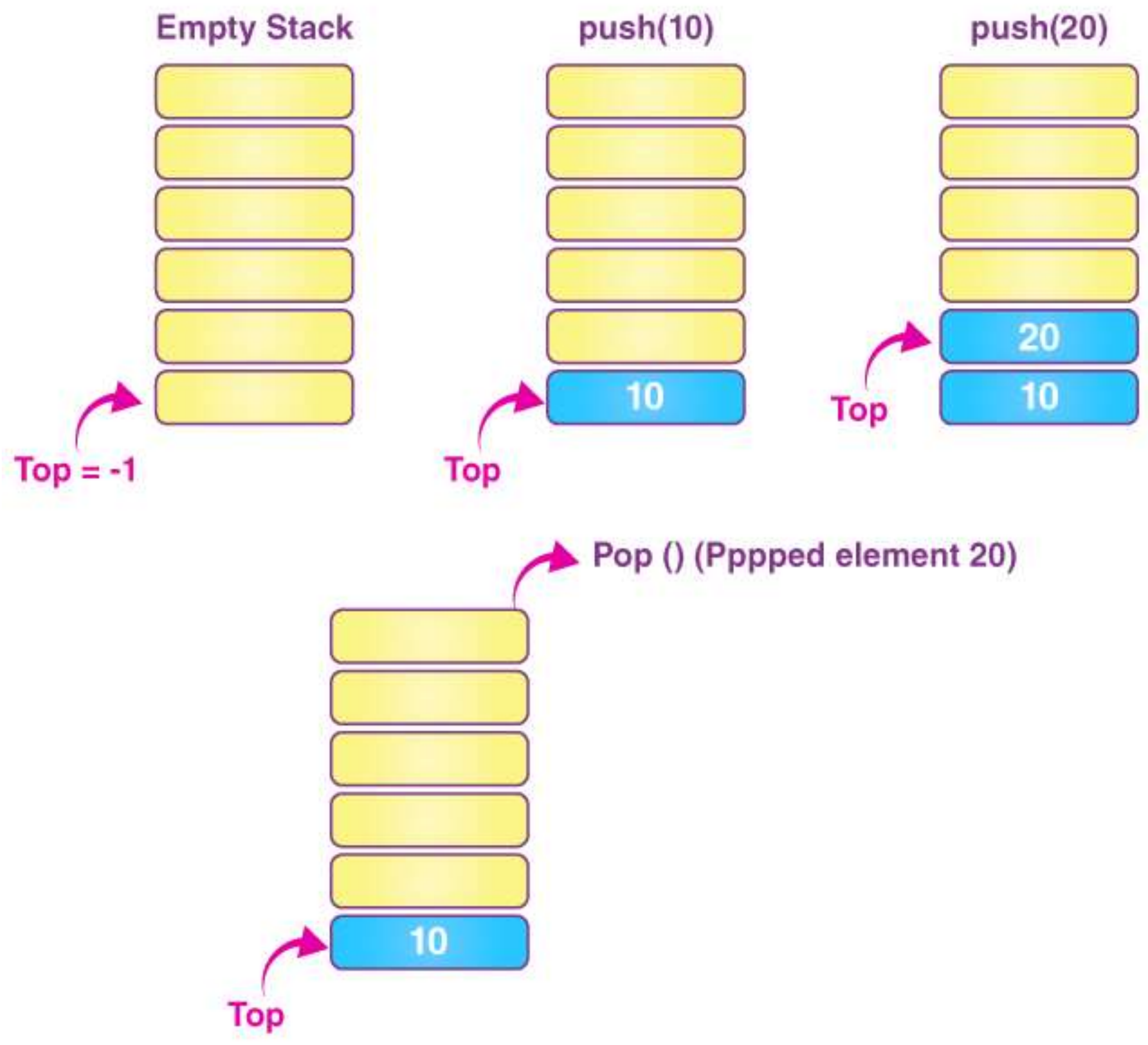
Algorithm:

```
Step-1: If TOP = NULL
    PRINT “Stack is Empty”
    Goto Step 3
Step-2: Return Stack[TOP]
Step-3: END
```

- **IsEmpty:** Checks if a stack is empty or not. Returns true if the given stack is empty; otherwise, it returns false.
- **IsStackFull:** Checks if a stack is full or not. Returns true if the given stack is full; otherwise, it returns false.

Representation of the Stack

A Stack can be a fixed specific size, or it can be dynamic, i.e., the Stack size can be changed dynamically. It can be represented by means of Pointer, Array, Structure, and Linked List.



Application of the Stack

In this section we will discuss typical problems where stacks can be easily applied for a simple and efficient solution. The topics that will be discussed in this section include the following:

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

1. A Stack can be used for evaluating expressions consisting of operands and operators.
2. Stacks can be used for Backtracking, i.e., to check parenthesis matching in an expression.
3. It can also be used to convert one form of expression to another form.
4. It can be used for systematic Memory Management.

Advantages of Stack

1. A Stack helps to manage the data in the 'Last in First out' method.
2. When the variable is not used outside the function in any program, the Stack can be used.
3. It allows you to control and handle memory allocation and de-allocation.
4. It helps to automatically clean up the objects.

Disadvantages of Stack

1. It is difficult in Stack to create many objects as it increases the risk of the Stack overflow.

2. It has very limited memory.
3. In Stack, random access is not possible.

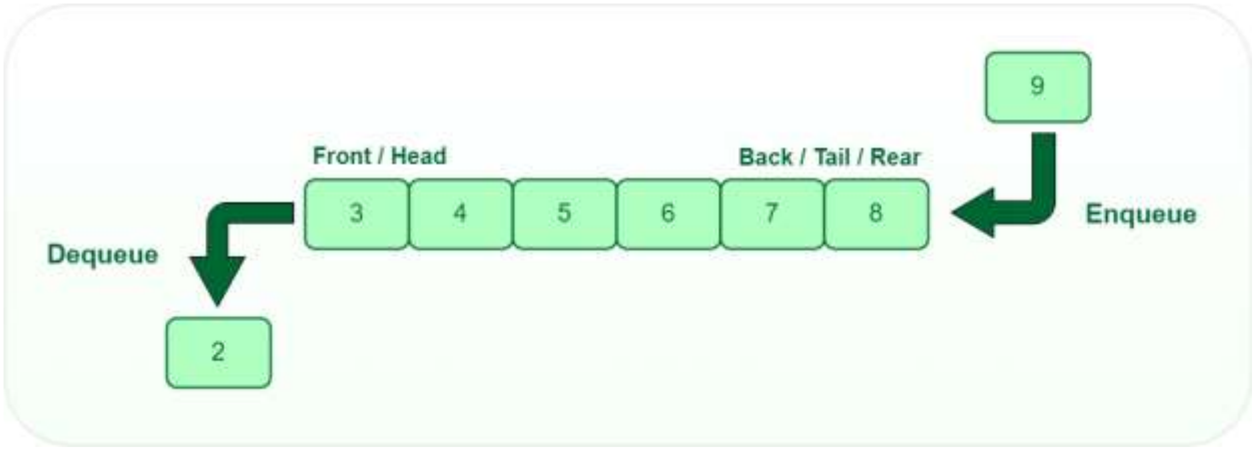
```c
#include <stdio.h>
int stack[100],i,j,choice=0,n,top=-1;
void push();
void pop();
void show();
void main ()
{

    printf("Enter the number of elements in the stack ");
    scanf("%d",&n);
    printf("*********Stack operations using array*********");

printf("\n----------------------------------------------\n");
    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            {
                push();
                break;
            }
            case 2:
            {
                pop();
                break;
            }
            case 3:
            {
                show();
                break;
            }
            case 4:
            {
                printf("Exiting....");
                break;
            }
            default:
            {
                printf("Please Enter valid choice ");
            }
        };
    }
}

void push ()
{
    int val;
    if (top == n )
    printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
```

```c
{
    if(top == -1)
    printf("Underflow");
    else
    top = top -1;
}
void show()
{
    for (i=top;i>=0;i--)
    {
        printf("%d\n",stack[i]);
    }
    if(top == -1)
    {
        printf("Stack is empty");
    }
}
```

What is Queue?

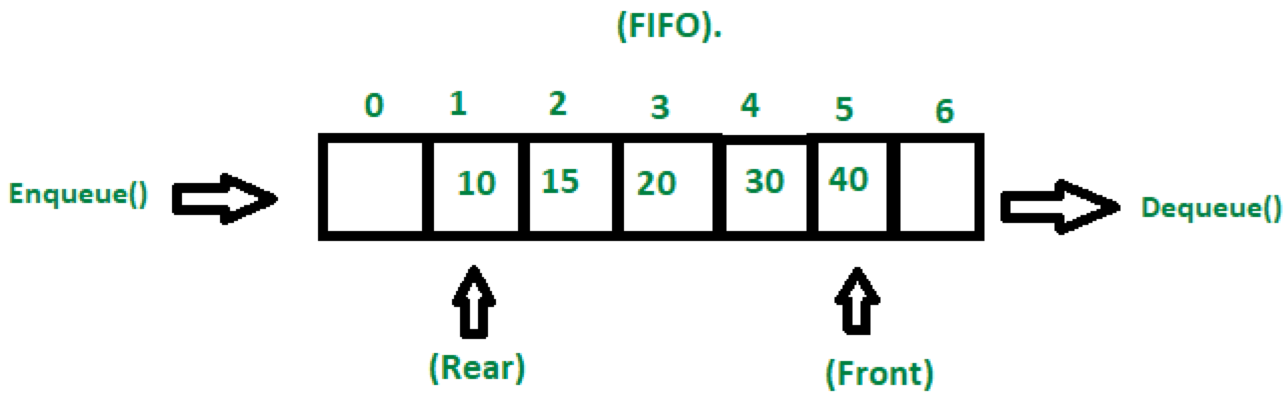
A queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

We define a queue to be a list in which all additions to the list are made at one end, and all deletions from the list are made at the other end. The element which is first pushed into the order, the operation is first performed on that.



FIFO Principle of Queue:

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First come first serve).
- Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue(sometimes, **head** of the queue), similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue. See the below figure.



Characteristics of Queue:

- Queue can handle multiple data.
- We can access both ends.
- They are fast and flexible.

Queue Representation:

Like stacks, Queues can also be represented in an array: In this representation, the Queue is implemented using the array. Variables used in this case are

- **Queue:** the name of the array storing queue elements.
- **Front:** the index where the first element is stored in the array representing the queue.
- **Rear**: the index where the last element is stored in an array representing the queue.

APPLICATIONS OF QUEUES •

Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.

• Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.

• Queues are used as buffers on MP3 players and portable CD players, iPod playlist.

- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.
- Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, by a mouse click, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a FIFO queue is the appropriate data structure.

TYPES OF QUEUES

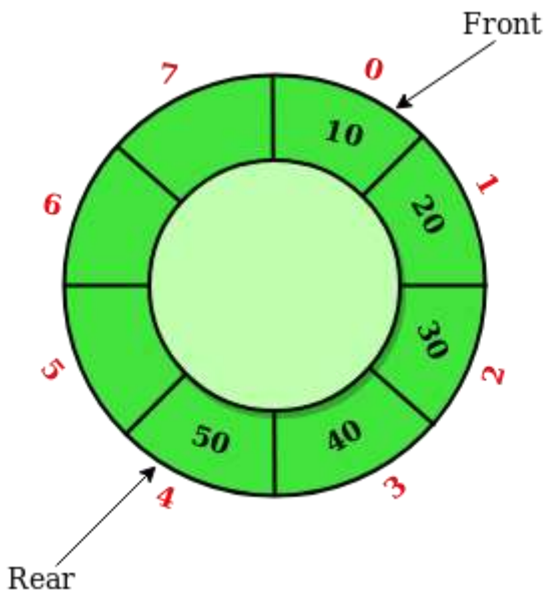
A queue data structure can be classified into the following types:

1. Circular Queue
2. De-queue
3. Priority Queue
4. linear Queue

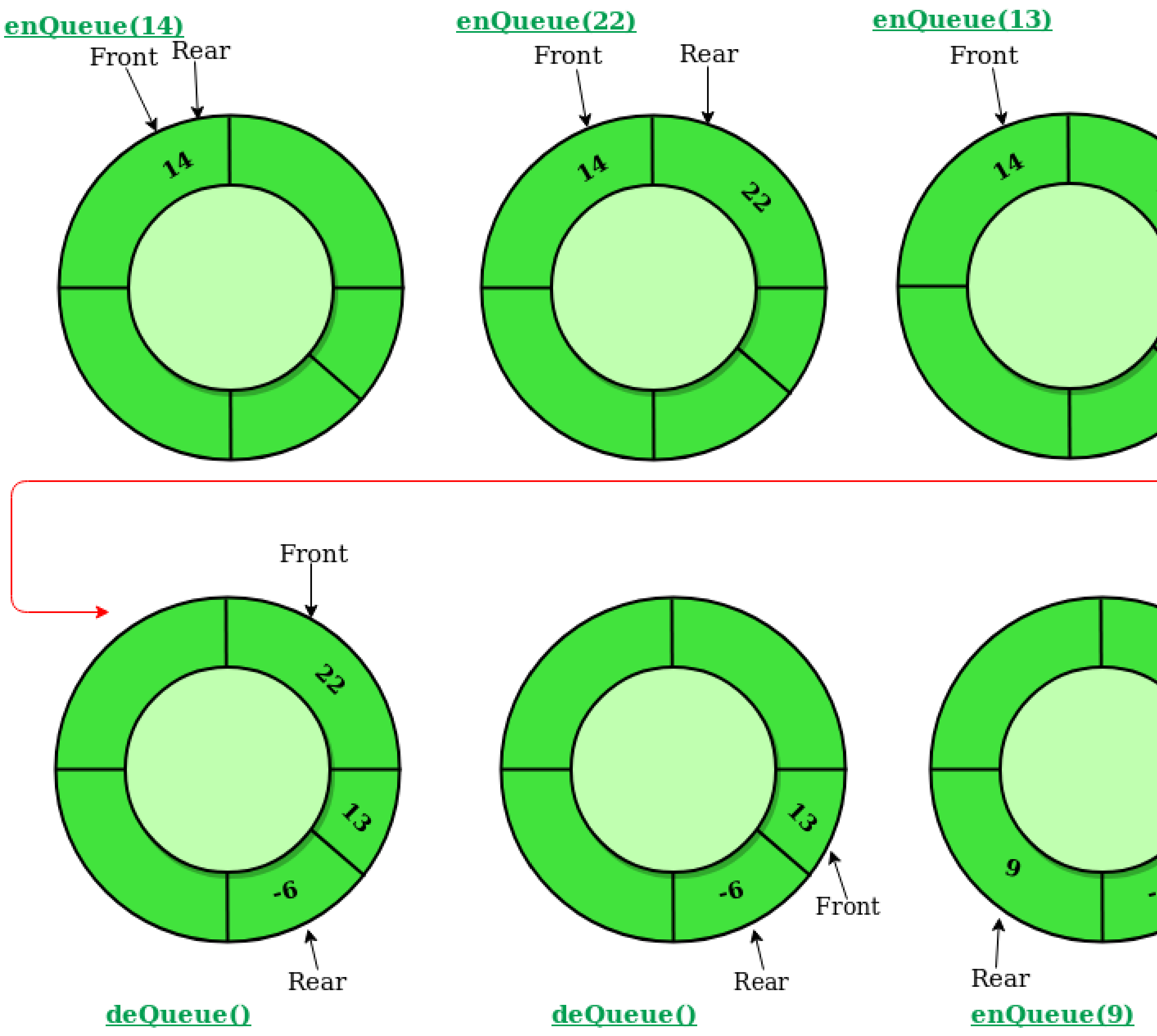
What is a Circular Queue?

A Circular Queue is a special version of queue where the last element of the queue is connected to the first element of the queue forming a circle.

The operations are performed based on FIFO (First In First Out) principle. 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.



Operations on Circular Queue:

- **Front:** Get the front item from queue.
- **Rear:** Get the last item from queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at Rear position.

1. Check whether queue is Full – Check ((rear == SIZE-1 && front == 0) || (rear == front-1)).

2. If it is full then display Queue is full. If queue is not full then, check if (rear == SIZE – 1 && front != 0) if it is true then set rear=0 and insert element.

- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from front position.
  1. Check whether queue is Empty means check (front==-1).
  2. If it is empty then display Queue is empty. If queue is not empty then step 3
  3. Check if (front==rear) if it is true then set front=rear= -1 else check if (front==size-1), if it is true then set front=0 and return the element.

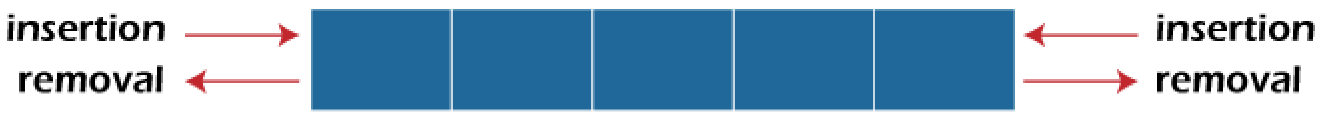
Applications:

1. **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
2. **Traffic system:** In computer controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
3. **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.

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

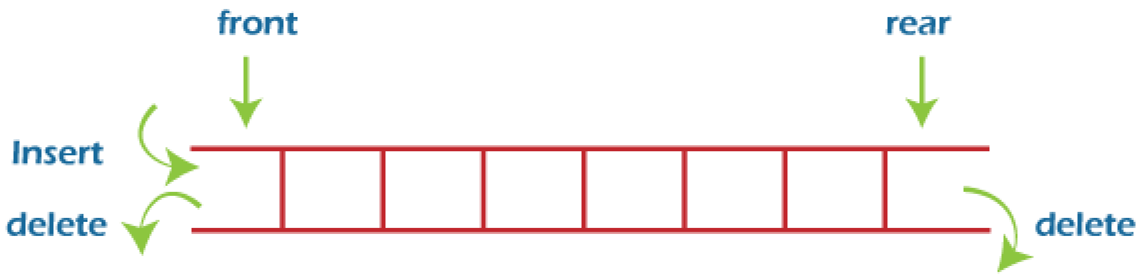
Types of deque

There are two types of deque -

- Input restricted queue
- Output restricted queue

Input restricted Queue

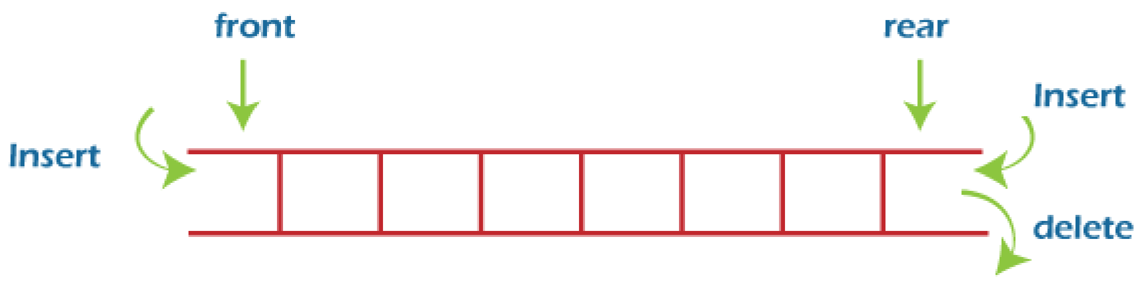
In input restricted queue, insertion operation can be performed at only one end, while deletion can be performed from both ends.



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 -

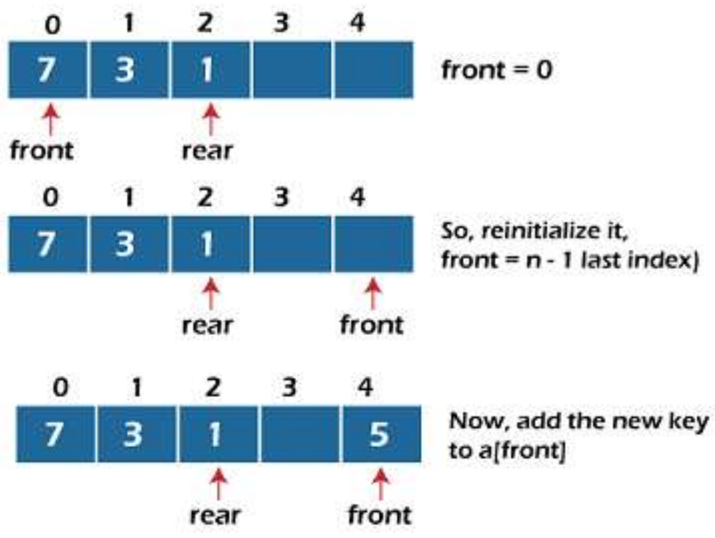
- Get the front item from the deque
- Get the rear item from the deque
- Check whether the deque is full or not
- Checks whether the deque is empty or not

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 -

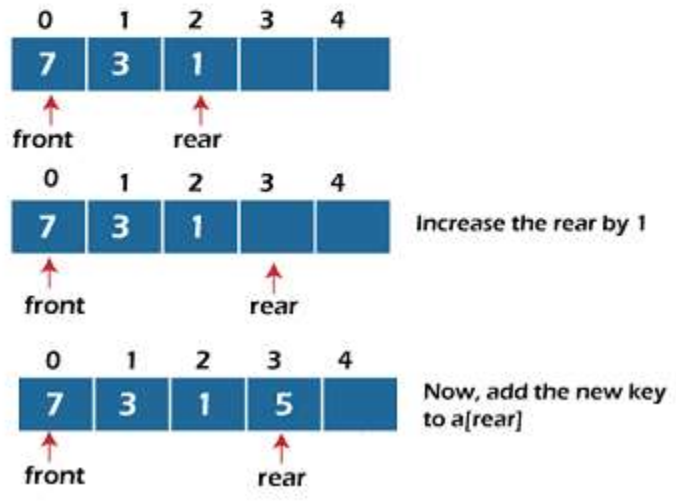
- If the queue is empty, both rear and front are initialized with 0. Now, both will point to the first element.
- Otherwise, check the position of the front if the front is less than 1 (front < 1), then reinitialize it by **front = n - 1**, i.e., the last index of the array.



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

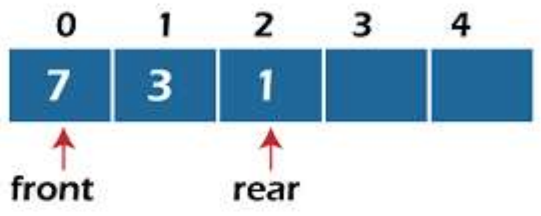
In this operation, the element is deleted from the front end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

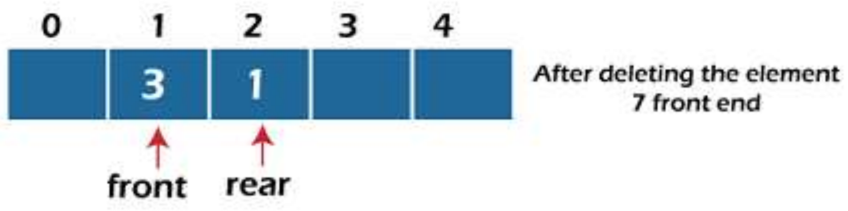
If the queue is empty, i.e., front = -1, it is the underflow condition, and we cannot perform the deletion. If the queue is not full, then the element can be inserted from the front end by using the below conditions -

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

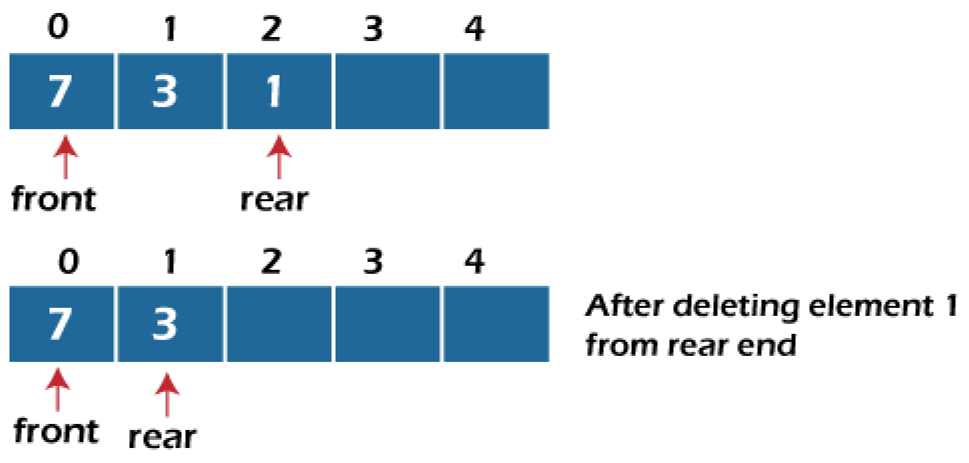
In this operation, the element is deleted from the rear end of the queue. Before implementing the operation, we first have to check whether the queue is empty or not.

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.

What is a 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

How is Priority assigned to the elements in a Priority Queue?

In a priority queue, generally, the value of an element is considered for assigning the priority.

For example, the element with the highest value is assigned the highest priority and the element with the lowest value is assigned the lowest priority. The reverse case can also be used i.e., the element with the lowest value can be assigned the highest priority. Also, the priority can be assigned according to our needs.

Operations of a Priority Queue:

A typical priority queue supports the following operations:

1) Insertion in a Priority Queue

When a new element is inserted in a priority queue, it moves to the empty slot from top to bottom and left to right. However, if the element is not in the correct place then it will be compared with the parent node. If the element is not in the correct order, the elements are swapped. The swapping process continues until all the elements are placed in the correct position.

2) Deletion in a Priority Queue

As you know that in a max heap, the maximum element is the root node. And it will remove the element which has maximum priority first. Thus, you remove the root node from the queue. This removal creates an empty slot, which will be further filled with new insertion. Then, it compares the newly inserted element with all the elements inside the queue to maintain the heap invariant.

3) Peek in a Priority Queue

This operation helps to return the maximum element from Max Heap or the minimum element from Min Heap without deleting the node from the priority queue.

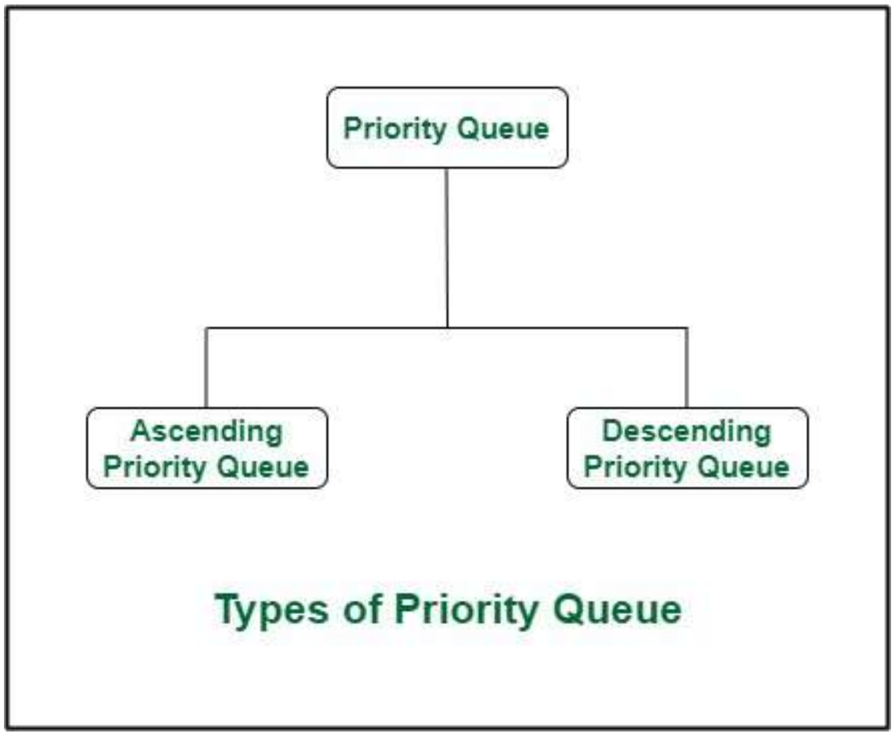
Types of Priority Queue:

1.)Ascending Order Priority Queue

As the name suggests, in ascending order priority queue, the element with a lower priority value is given a higher priority in the priority list. For example, if we have the following elements in a priority queue arranged in ascending order like 4,6,8,9,10. Here, 4 is the smallest number, therefore, it will get the highest priority in a priority queue.

2) Descending order Priority Queue

The root node is the maximum element in a max heap, as you may know. It will also remove the element with the highest priority first. As a result, the root node is removed from the queue. This deletion leaves an empty space, which will be filled with fresh insertions in the future. The heap invariant is then maintained by comparing the newly inserted element to all other entries in the queue.



Types of Priority Queues

Difference between Priority Queue and Normal Queue?

There is no priority attached to elements in a queue, the rule of first-in-first-out(FIFO) is implemented whereas, in a priority queue, the elements have a priority. The elements with higher priority are served first.

How to Implement Priority Queue?

Priority queue can be implemented using the following data structures:

- Arrays
- Linked list
- Heap data structure

- Binary search tree

Let's discuss all these in detail.

1) Implement Priority Queue Using Array:

A simple implementation is to use an array of the following structure.

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

- **enqueue():** This function is used to insert new data into the queue.
- **dequeue():** This function removes the element with the highest priority from the queue.
- **peek()/top():** This function is used to get the highest priority element in the queue without removing it from the queue.

.