# UNIT-2

## Stacks

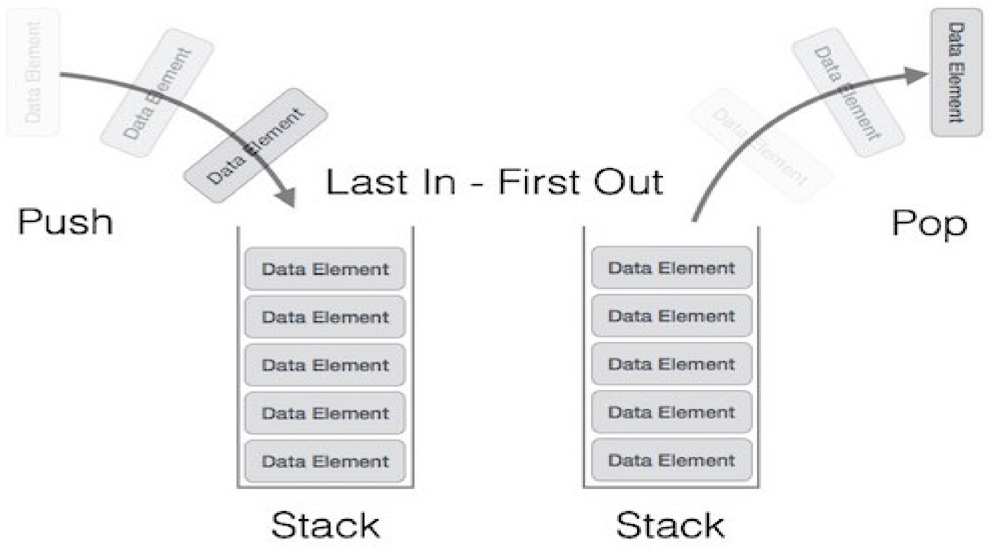
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations −



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations −

**push()** − Pushing (storing) an element on the stack.

**pop()** − Removing (accessing) an element from the stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −

**isFull()** − check if stack is full.

**isEmpty()** − check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the stack without actually removing it.

**stack functions**

isfull()

Algorithm of isfull() function −

```
Function isfull()
Begin
  if top = MAXSIZE
    return -1
  else
    return 0
  endif
end
```

isempty()

Algorithm of isempty() function −

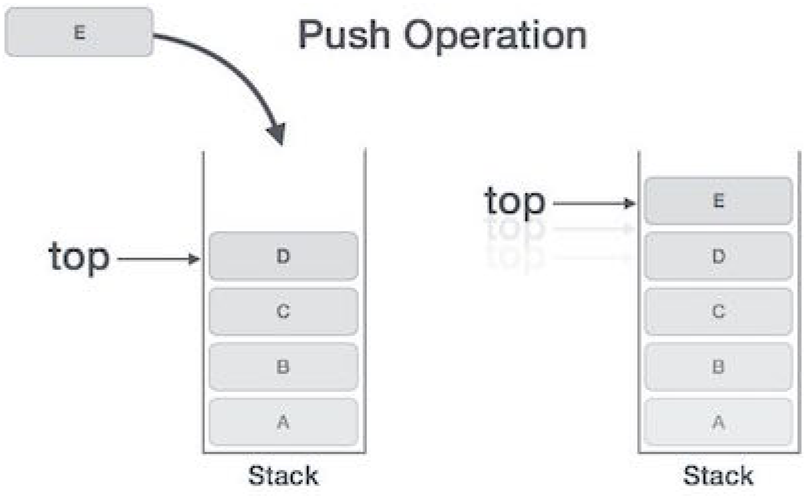
```
Function isempty()
Begin
  if top=-1 then
    return -1
  else
    return 0
  endif
```

end

**Push Operation**

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, Print(“Stack is full”), exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

```
Function push(stack, top, data)

Begin

IF  top=MAXSIZE THEN

    Print(“Stack is full”);

    return

else

  begin
```

```
   top ← top + 1
   stack[top] ← data
  end
end
```

## Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value.

A Pop operation may involve the following steps

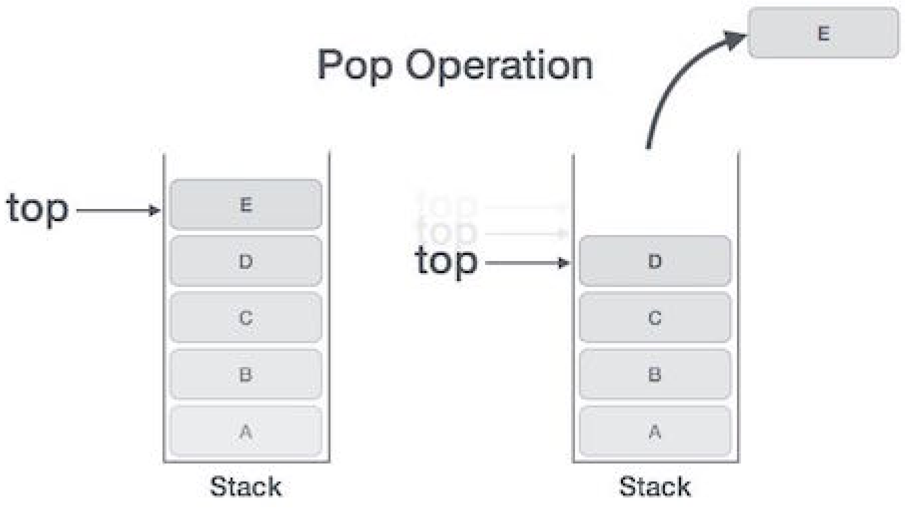
Step 1 − Checks if the stack is empty.

Step 2 − If the stack is empty, produces an error and exit.

Step 3 − If the stack is not empty, accesses the data element at which top is

pointing.

Step 4 − Decreases the value of top by 1.



A simple algorithm for Pop operation can be derived as follows −

```
Function pop(stack, top)
Begin
   If top=-1 then
     Print("Stack is empty")
     return
  else
    data ← stack[top]
    top ← top – 1
return data
end
```

## Applications of Stacks

Stacks are useful for any application requiring LIFO storage. There are many, many of these.

1. parsing context-free languages
2. evaluating arithmetic expressions
3. function call management
4. traversing trees and graphs (such as depth first traversals)
5. recursion

Function call management

| Void main() | Void f1() | Void f2() | Void f3() | Void f4() |
|---|---|---|---|---|
| { | { | { | { | { |
| ------- | -------- | --------- | --------- | -------- |
| -------- | --------- | --------- | --------- | --------- |
| Call f1() |  | Call f3() |  |  |
| -------- | --------- | -------- |  |  |
| -------- | Call f2() |  | ---------- | -------- |
|  | ------- | -------- | Call f4() | -------- |
| Call f3() | --------- | } | ---------- | } |
| ------- | } |  | ---------- |  |
| } |  |  | } |  |

**Infix notation, Prefix and Postfix Notations**

Infix Notation : Operators are written in-between their operands.

Eg. A * ( B + C ) / D            reverse=D/)C+B(*A

Prefix Notation(Reverse Polish Notation):. Operators are written before their operands

Eg. / * A + B C D

Postfix Notation(Polish Notation):. Operators are written after their operands

Eg. A B C + * D /

**Infix to Postfix Conversion (Algorithm)**

```
Create an empty stack to hold operators
        Newinfix=reverse(infix)
        for each token ‘tok’ in the input infix string {
        switch (tok) {
        case operand:
                append tok to right end of postfix;
                break;
        case '(' :
                stack.push(tok);
                break;
        case ')':
                while ( stack.top() != '(’ )
                {
                append  stack.top() to right end of postfix;
                stack.pop();
                }
                break;
        case operator:
                while ( !stack.empty() && stack.top() != '('
                                && precedence(tok) <= precedence(stack.top()) )
                 {
                append stack.top() to right end of postfix;
                stack.pop();
                }
                stack.push(operator);
                break;
        }
}
while ( !stack.empty() )
 {
append stack.top() to right end of postfix;
stack.pop();
}
```

**Example:**

Input string: A*(B+C/D)+(E+F*G/H)

| Steps | Infix String | Operator Stack | Postfix String |
|---|---|---|---|
| 1 | A | | A |
| 2 | A* | * | A |
| 3 | A*( | *( | A |
| 4 | A*(B | *( | AB |
| 5 | A*(B+ | *(+ | AB |
| 6 | A*(B+C | *(+ | ABC |
| 7 | A*(B+C/ | *(+/ | ABC |
| 8 | A*(B+C/D | *(+/ | ABCD |
| 9 | A*(B+C/D) | * | ABCD/+ |
| 10 | A*(B+C/D)+ | + | ABCD/+* |
| 11 | A*(B+C/D)+( | +( | ABCD/+* |
| 12 | A*(B+C/D)+(E | +( | ABCD/+*E |
| 13 | A*(B+C/D)+(E+ | +(+ | ABCD/+*E |
| 14 | A*(B+C/D)+(E+F | +(+ | ABCD/+*EF |
| 15 | A*(B+C/D)+(E+F* | +(+* | ABCD/+*EF |
| 16 | A*(B+C/D)+(E+F*G | +(+* | ABCD/+*EFG |
| 17 | A*(B+C/D)+(E+F*G/ | +(+/ | ABCD/+*EFG* |
| 18 | A*(B+C/D)+(E+F*G/H | +(+/ | ABCD/+*EFG*H |
| 19 | A*(B+C/D)+(E+F*G/H) | + | ABCD/+*EFG*H/+ |
| 20 | A*(B+C/D)+(E+F*G/H) | | ABCD/+*EFG*H/++ |

**Postfix Expression Evaluation using Stack Data Structure**

**Algorithm**

**//**A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

```
1. Read the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is an operand, then push it on to the Stack (opdst)
3. If the reading symbol is operator (+ , - , * , / etc.,), then
   Perform TWO pop operations
           Operand2=pop(opdst)
           Operand1=pop(opdst)
4. Perform reading symbol operation using operand1 and operand2 and push result
   back on to the Stack.
5. If the postfix Expression is not end of the string then Goto step-1
6. Perform a pop operation and display the popped value as final result.
7. End
```

Example:

Postfix String : ABCD/+*EFG*H/++

| Steps | Input string (Postfix String) | Operand Stack |
|---|---|---|
| 1 | A | A |
| 2 | AB | AB |
| 3 | ABC | ABC |
| 4 | ABCD | ABCD |
| 5 | ABCD/ | AB(C/D) |
| 6 | ABCD/+ | A(B+(C/D)) |
| 7 | ABCD/+* | (A*(B+(C/D))) |
| 8 | ABCD/+*E | (A*(B+(C/D)))E |
| 9 | ABCD/+*EF | (A*(B+(C/D)))EF |
| 10 | ABCD/+*EFG | (A*(B+(C/D)))EFG |
| 11 | ABCD/+*EFG* | (A*(B+(C/D)))E(F*G) |
| 12 | ABCD/+*EFG*H | (A*(B+(C/D)))E(F*G)H |
| 13 | ABCD/+*EFG*H/ | (A*(B+(C/D)))E((F*G)/H) |
| 14 | ABCD/+*EFG*H/+ | (A*(B+(C/D)))(E+((F*G)/H)) |
| 15 | ABCD/+*EFG*H/++ | (A*(B+(C/D)))+(E+((F*G)/H)) |

So, the expression evaluated result = pop(opdst)

## Implementing multiple Stacks in a Single Array

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | T[0] | | | | | T[1] | | | | T[2] | |
| 11 | 20 | | | 45 | 23 | 34 | | 67 | 90 | 55 | |
| a[0] | a[1] | a[2] | a[3] | a[4] | a[5] | a[6] | a[7] | a[8] | a[9] | a[10] | a[11] |
| B[0] | | | | B[1] | | | | B[2] | | | |

To implement multiple stacks in a single array, one approach is to divide the array in k slots of size n/k each, and fix the slots for different stacks, we can use arr[0] to arr[(n/k)-1] for first stack, and arr[n/k] to arr[(2n/k)-1] for stack2 and so on where arr[] is the array of size n.

In the above multiple stack, B[0] represents the first element of the stack 0, B[1] for stack1 and so on. Also T[0] represent the top of the stack 0, T[1] for stack 1 and T[2] for stack 2.

Although this method is easy to understand, but the problem with this method is inefficient use of array space.

A stack push operation may result in stack overflow even if there is space available in arr[].

```
Push(A, i, x)  // A –array for storing the elements of the stacks
               // i- ith stack,  x- new element to be inserted in to the ith stack.

{

 If T(i)=B(i+1) then

    print<<" Stack is full";
else
    T(i)=T(i)+1
    A[T(i)]=x
Endif
}
POP(A,i)
  {
   If T(i)<B(i) then
   Print("Stack is empty")
Else
     x = A[t(i)]
    T(i)=T(i)-1
}
```

## Queues

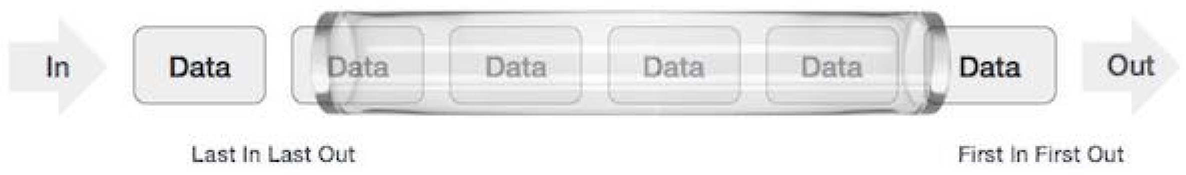
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.

LAST IN
LAST OUT
ONE WAY
FIRST IN
FIRST OUT

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

## Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure −



Queue

As in stacks, a queue can also be implemented using Arrays, Linked-lists.

## Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues −

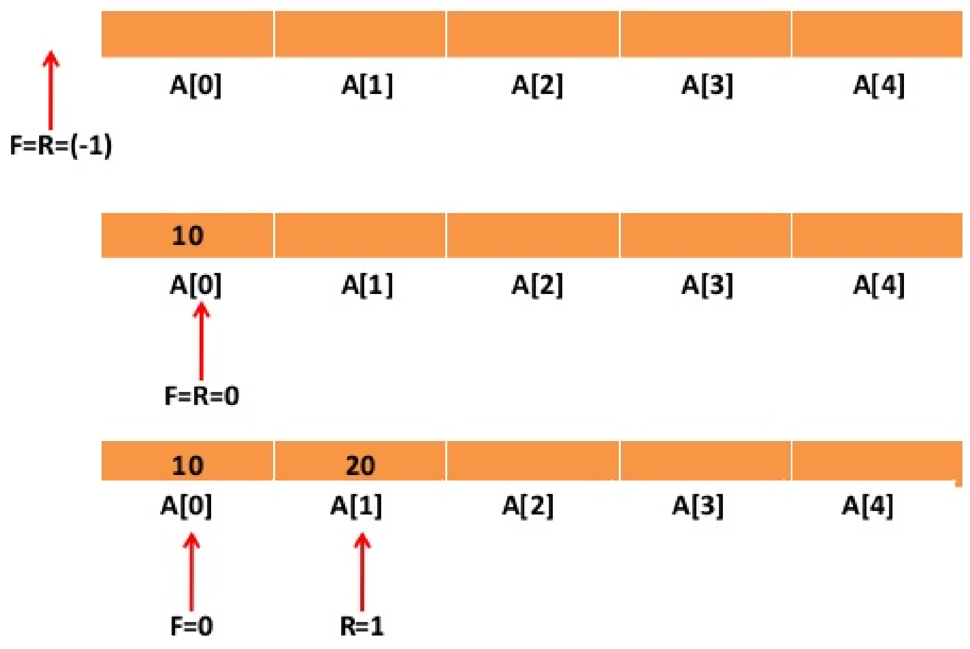
- **enqueue()** − add (store) an item to the queue.
- **dequeue()** − remove (access) an item from the queue.

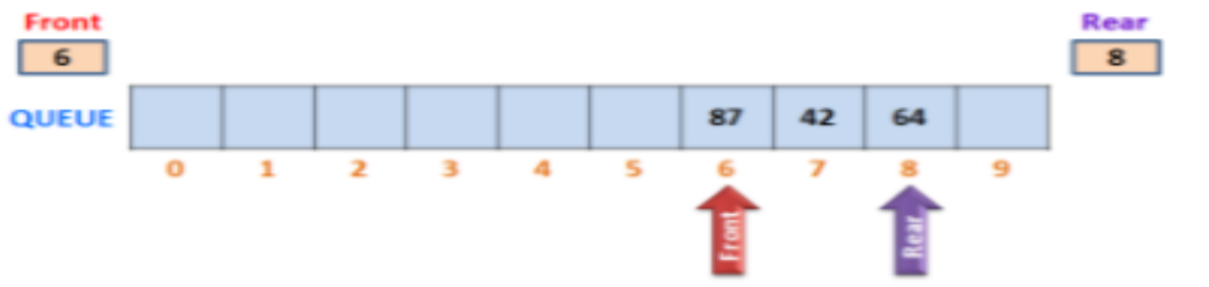
Few more functions are required to make the above-mentioned queue operation efficient. These are −

**isfull()** − Checks if the queue is full.

**isempty()** − Checks if the queue is empty.

Example of Insertion in Queue





In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue −

### isfull()

As we are using single dimension array to implement queue, we just check for the rear pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a circular linked-list, the algorithm will differ. Algorithm of isfull() function −

**Algorithm**

```
 procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

### isempty()

Algorithm of isempty() function −

**Algorithm**

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN or -1, it tells that the queue is not yet initialized, hence empty.

## Enqueue Operation

Queues maintain two data pointers, **front** and **rear**.

The following steps should be taken to enqueue (insert) data into a queue −

- **Step 1** − Check if the queue is full.
- **Step 2** − If the queue is full, produce overflow error and exit.
- **Step 3** − If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** − Add data element to the queue location, where the rear is pointing.
- **Step 5** − return success.

Implementation of enqueue() in C++ programming language −

```
Function enqueue(int data)
  If isfull())
      return 0;
  else
    rear = rear + 1;
    queue[rear] = data;
  endif
  end
```

## Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where **front** is pointing and remove the data after access. The following steps are taken to perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return

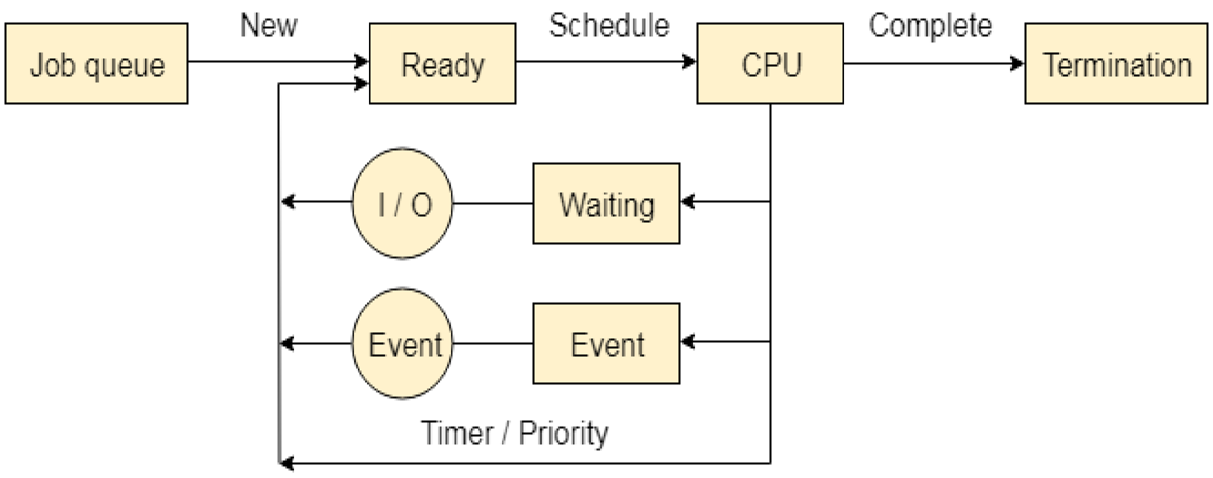
Implementation of dequeue() in C++ programming language −

```
Function dequeue()
 {
  if(isempty())
      return 0;
  else
   data = queue[front];
   front = front + 1;
   return data;
 }
```

## Applications of Queue

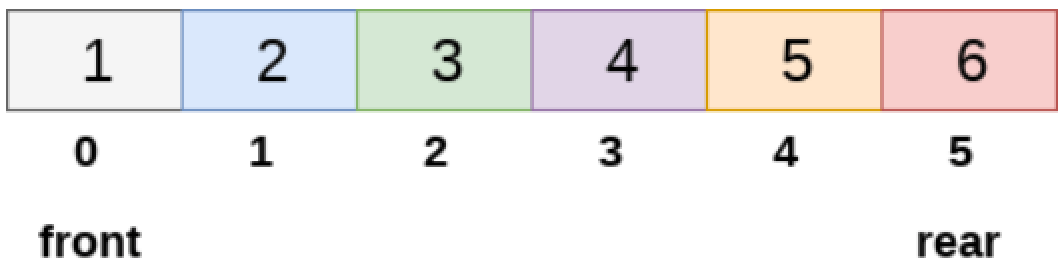
### Process Queues

The Operating system manages various types of queues for each of the process states. The PCB related to the process is also stored in the queue of the same state. If the Process is moved from one state to another state then its PCB is also unlinked from the corresponding queue and added to the other state queue in which the transition is made.

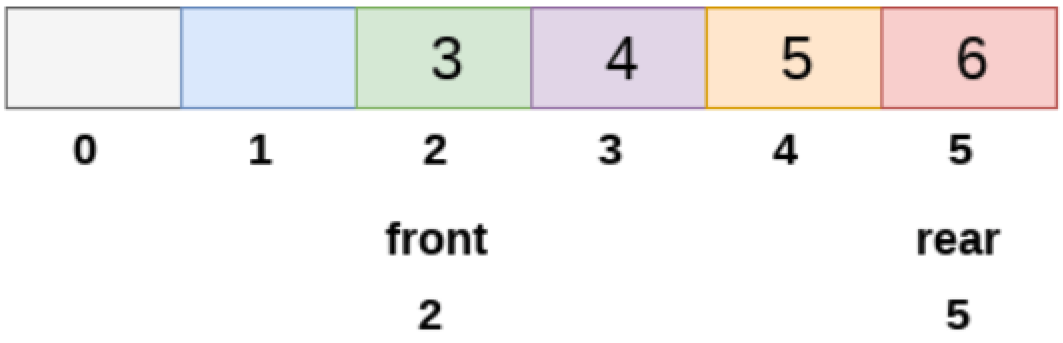


## Circular Queues

Consider the queue shown in the following figures

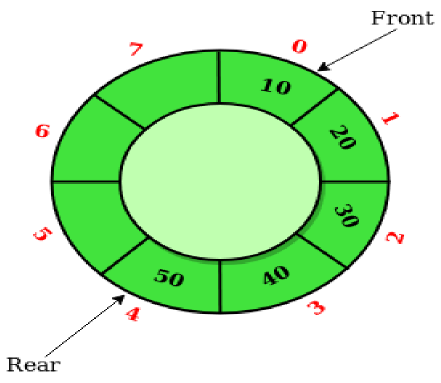


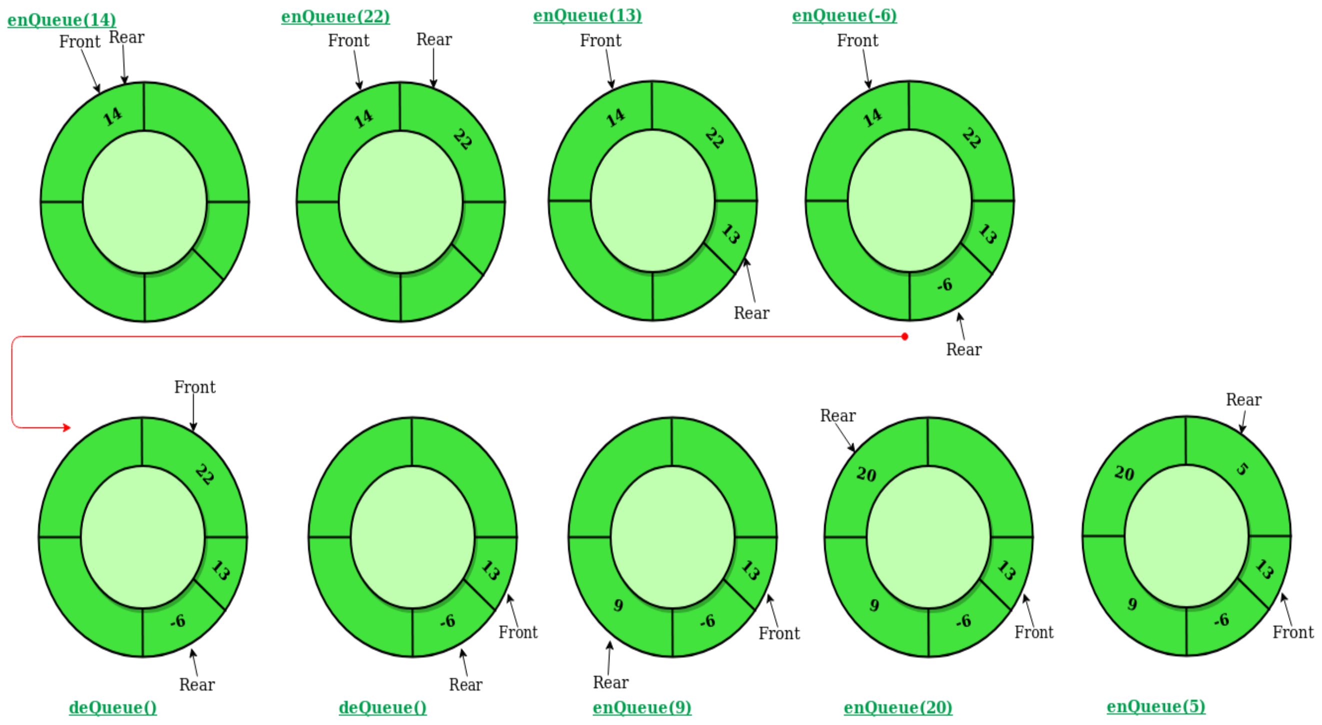
After deleting two elements the queue becomes



At this when we try to insert an element in to the queue the system will display the message that the queue is full but actually that queue is not full. This is the major limitation of linear queue.

We can override this problem by using the following type queue named **circular queue**





Algorithm to insert an element in circular queue

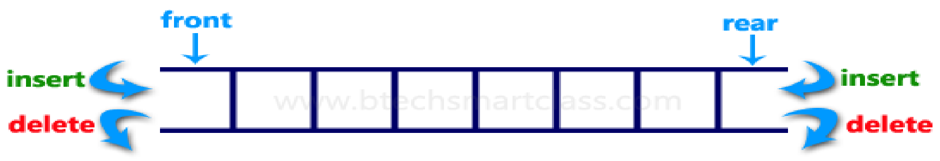
```
Enqueue(Queue,front,rear,value)
BEGIN
IF (rear+1)%MAXSIZE = front THEN
        Print " Queue is full "
        Goto step 4
ELSE
      IF front = -1
       set front = 0
 rear=((rear+1)%MAXSIZE)
 queue[rear]=value
ENDIF
END
```

Algorithm for deleting an element from a circular queue

```
Dequeue(Queue,front,rear)
BEGIN
IF (front=-1)THEN
     Print"Queue is empty"
ELSE
     Value=queue[front]
IF(front=rear)
   Set front=-1
   Set rear=-1
ELSE
    front=(front+1)%MAXSIZE
ENDIF
END
```

## Double Ended Queue Data structure(DEQUE)

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.



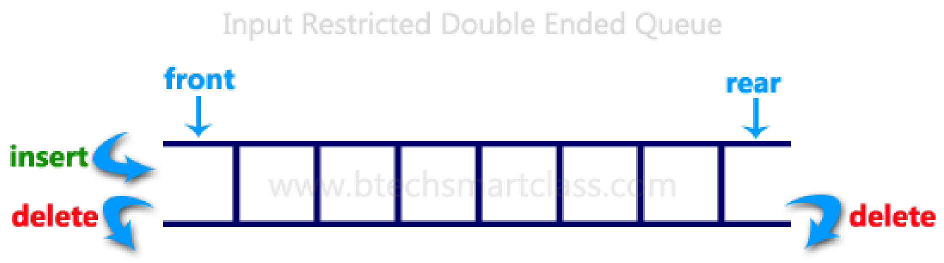
Double Ended Queue can be represented in TWO ways, those are as follows...

Input Restricted Double Ended Queue

Output Restricted Double Ended Queue

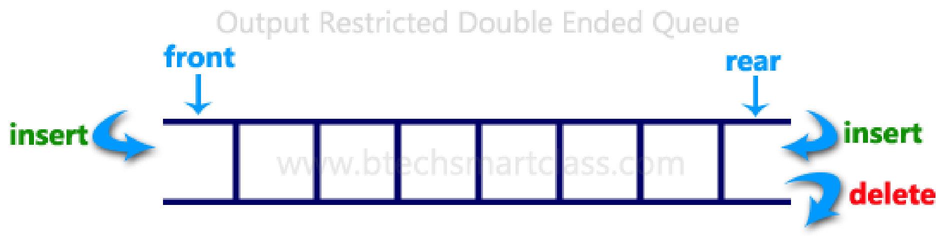
**Input Restricted Double Ended Queue**

In input restricted double-ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the ends.



**Output Restricted Double Ended Queue**

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



**Operations on Deque:**

Mainly the following four basic operations are performed on queue:

insertFront(): Adds an item at the front of Deque.

insertLast(): Adds an item at the rear of Deque.

deleteFront(): Deletes an item from front of Deque.

deleteLast(): Deletes an item from rear of Deque.

In addition to above operations, following operations are also supported

getFront(): Gets the front item from queue.

getRear(): Gets the last item from queue.

isEmpty(): Checks whether Deque is empty or not.

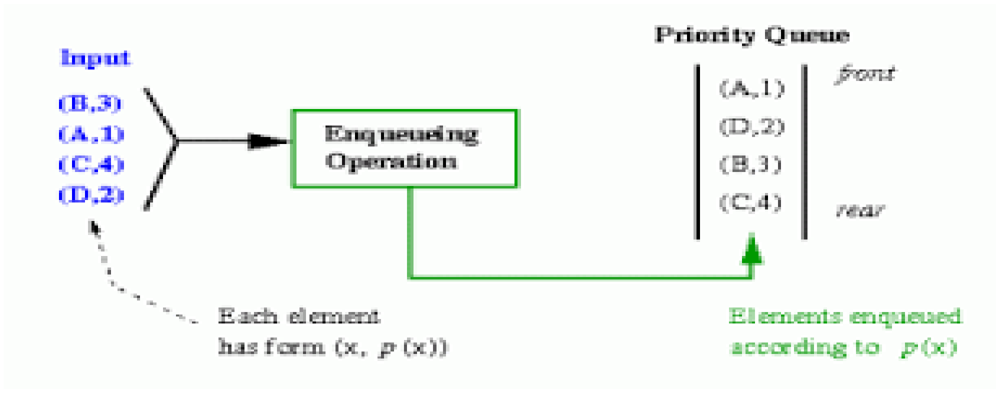
isFull(): Checks whether Deque is full or not.

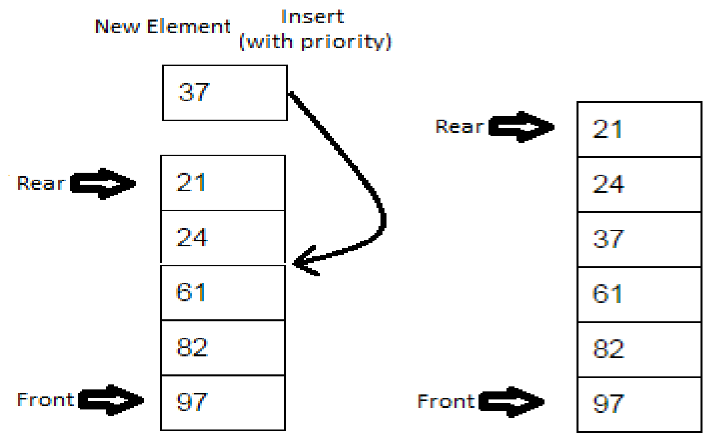
**Priority Queues**

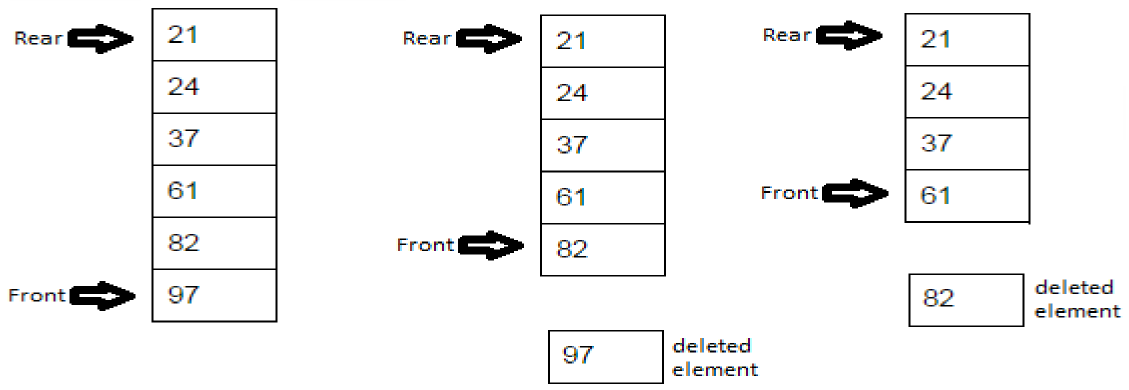
In computer science, a priority queue is an abstract data type similar to regular queue or stack data structure in which each element additionally has a "priority" associated with it. In a priority queue, an element with high priority is served before an element with low priority.

The following are the basic 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.


Input
(B,3)
(A,1)
(C,4)
(D,2)
Enqueuing Operation
Priority Queue
(A,1)
(D,2)
(B,3)
(C,4)
front
rear
Each element has form (x, p(x))
Elements enqueued according to p(x)


New Element
Insert (with priority)
37
Rear
21
24
61
82
Front
97
Rear
21
24
37
61
82
Front
97


Rear
21
24
37
61
82
Front
97
Rear
21
24
37
61
Front
82
97
deleted element
Rear
21
24
37
Front
61
82
deleted element