

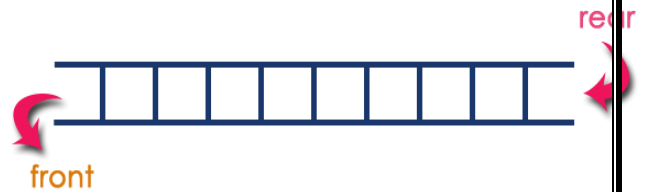
Unit – III

Syllabus:

- **Queues:** Introduction to Queues, Representation of Queues-using Arrays and using Linked list, Implementation of Queues-using Arrays and using Linked list, Application of Queues-Circular Queues, Deques, Priority Queues, Multiple Queues.
- **Stacks:** Introduction to Stacks, Array Representation of Stacks, Operations on Stacks, Linked list Representation of Stacks, Operations on Linked Stack, Applications-Reversing list, Factorial Calculation, Infix to Postfix Conversion, Evaluating Postfix Expressions.

QUEUE:

- Queue is a linear data structure in which elements can be inserted from one end called **rear** and deleted from other end called **front**.
- The deletion or insertion of elements can take place only at the front or rear end called dequeue and enqueue respectively. The first element that gets added into the queue is the first one to get removed from the queue. Hence the queue is referred to as First-In-First-Out list (FIFO).



Operations performed on Queue:

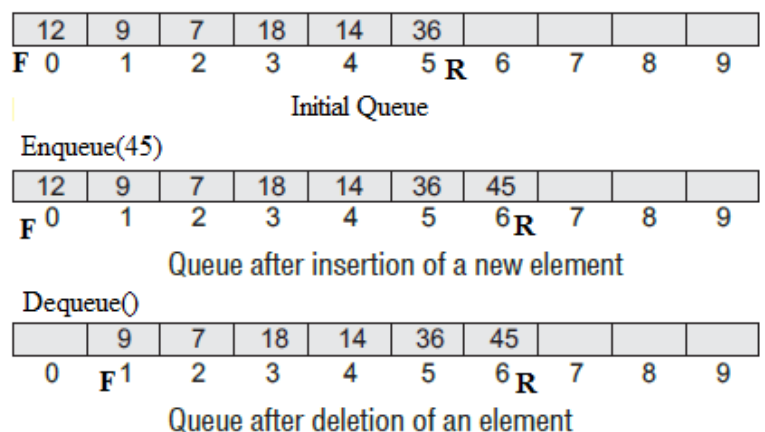
There are two possible operations performed on a queue. They are

- ✓ enqueue: Allows inserting an element at the rear of the queue.
- ✓ dequeue: Allows removing an element from the front of the queue.

REPRESENTATION OF QUEUEs:

ARRAYs: Queues can be easily represented using linear arrays. Every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively. The array representation of a queue is shown

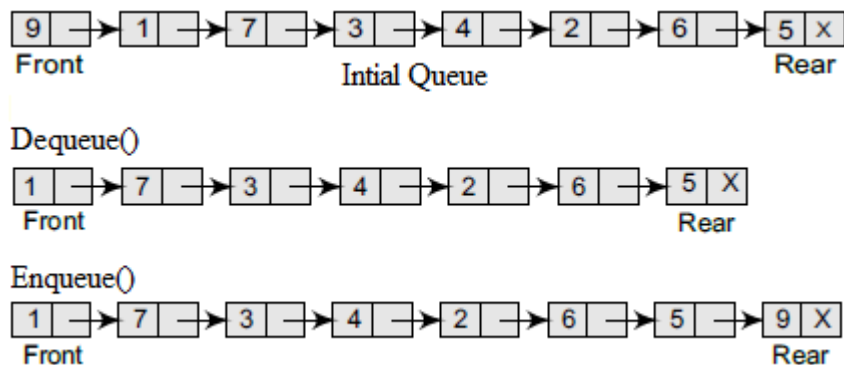
Drawback: The array must be declared to have some fixed size. If we allocate space



for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted.

LINKED LISTS:

- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element.
- The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end.
- If FRONT = REAR = NULL, then it indicates that the queue is empty.



IMPLEMENTATION OF QUEUES:

Using Arrays:

Algorithm for ENQUEUE operation

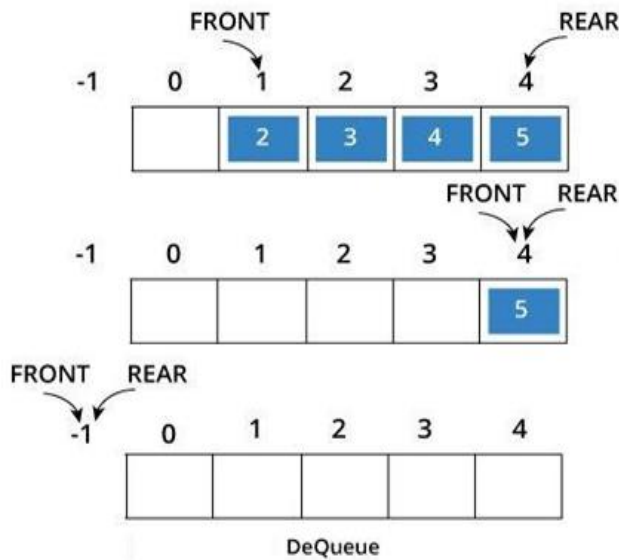
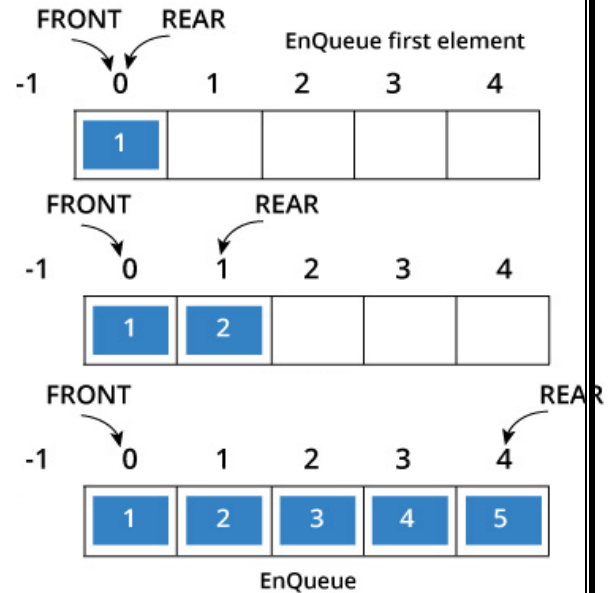
1. Check whether queue is FULL. (**rear** >= **SIZE-1**)
2. If it is **FULL**, then display an error message "Queue is FULL!!! Insertion is not possible!!!" and terminate the function.
3. If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear] = value**.

Algorithm for DEQUEUE operation

1. Check whether queue is EMPTY. (**front** == **-1**)
2. If it is **EMPTY**, then display "Queue is EMPTY!!! Deletion is not possible!!!" and terminate the function.
3. If it is **NOT EMPTY**, then display **queue[front]** as deleted element, increment the front value by one (**front ++**). If we are deleting last element both front and rear are equal (**front == rear**), then set both front and rear to '-1' (**front = rear = -1**).

Implementation:

- Let us consider a queue, which can hold maximum of five elements.
- Initially the queue is empty. An element can be added to the queue only at the rear end of the queue.
- Before adding an element in the queue, it is checked whether queue is full. If the queue is full, then addition cannot take place. Otherwise, the element is added to the end of the list at the



rear

end. If we are inserting first element into the queue then change front to 0 (Zero).

- Now, delete an element 1. The element deleted is the element at the front of the queue. So the status of the queue is:

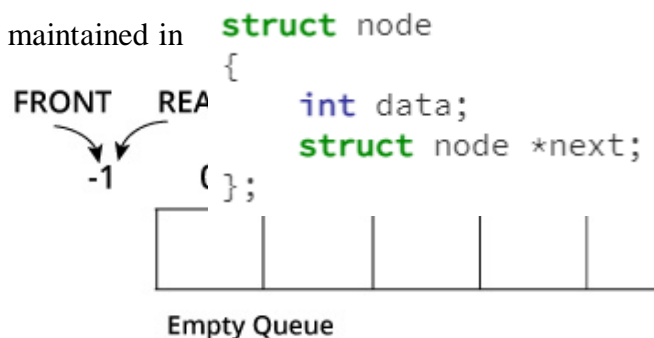
- When the last element delete 5. The element deleted at the front of the queue. So the status of the queue is empty. So change the values of front and rear to -1 (**front=rear= -1**)

The dequeue operation deletes the element from the front of the queue. Before deleting an element, it is checked if the queue is empty. If not the element pointed by front is deleted from the queue and front is now made to point to the next element in the queue.

- **Drawback:** If we implement the queue using an array, we need to specify the array size at the beginning (at compile time). We can't change the size of an array at runtime. So, the queue will only work for a fixed number of elements.

Using Linked List:

- In a linked queue, each node of the queue consists of two parts i.e. data part and the next 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. Initially

```
struct node *front = NULL, *rear = NULL;
```

Operation on Linked Queue: There are two basic operations which can be implemented on the linked queues. The operations are Enqueue and Dequeue.

Enqueue function: Enqueue function will add the element at the end of the linked list.

1. Declare a new node and allocate memory for it.
2. If front == NULL, make both front and rear points to the new node.
3. Otherwise, add the new node in rear->next (end of the list) and make the new node as the rear node. i.e. rear = new node

Dequeue function: Dequeue function will remove the first element from the queue.

1. Check whether the queue is empty or not
2. If it is the empty queue (front == NULL), We can't dequeue the element.
3. Otherwise, Make the front node points to the next node. i.e front = front->next;
if front pointer becomes NULL, set the rear pointer also NULL.

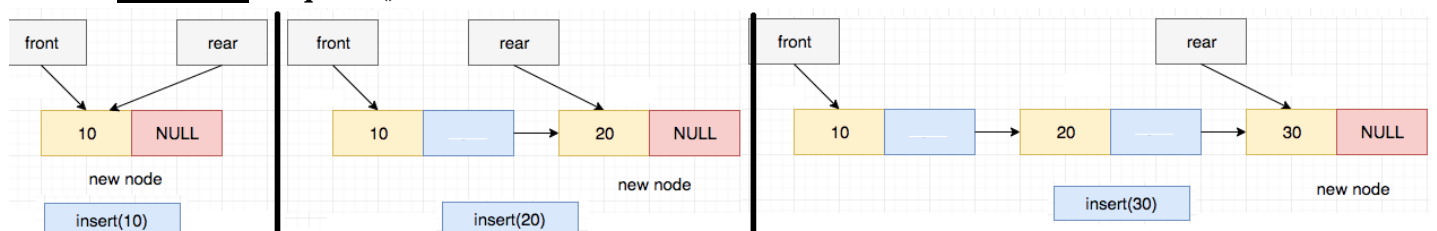
Free the front node's memory.

```
void dequeue()
{
    struct node *ptr;
    if(front == NULL)
        printf("Queue is Empty");
    else
    {
        ptr = front;
        front = front->next;
        free(ptr);
        if(front == NULL)
            rear = NULL;
    }
}
```

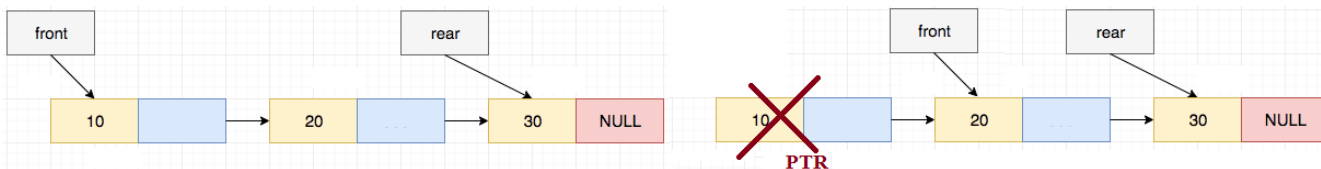
```
void enqueue(int value)
{
    struct node *newNode = malloc(sizeof(struct node));
    newNode->data = value;
    newNode->next = NULL;

    if(front == NULL && rear == NULL)
        front = rear = newNode;
    else
    {
        rear->next = newNode;
        rear = newNode;
    }
}
```

Example: Enqueue()



Dequeue()



TYPES OF QUEUES:

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

1. Circular Queue
2. Deque
3. Priority Queue
4. Multiple Queue

CIRCULAR QUEUES:

- In a Linear queue, once the queue is completely full, it's not possible to insert any more elements. When we **dequeue** any element to remove it from the queue, we are actually moving the **front** of the queue forward, but **rear** is still pointing to the last element of the queue, we cannot insert new elements.
- **Circular Queue** is also a linear data structure, which follows the principle of **FIFO**(First In First Out), but instead of ending the queue at the last position, it again starts from the first position after the last, hence making the queue behave like a circular data structure.

Operations on Circular Queue: The following are the operations that can be performed

- **enQueue(value):** This function is used to insert the new value in the Queue. The new element is always inserted from the **rear end**.
- **deQueue():** This function deletes an element from the Queue. The deletion in a Queue always takes place from the **front end**.

Enqueue operation: The steps of enqueue operation are given below:

- First, we will check whether the Queue is full or not.
- Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.
- From 2nd element onwards, When we insert a new element, the rear gets incremented, i.e., $rear=rear+1$.

Queue is not full:

- **If $rear \neq \max - 1$** , then rear will be incremented and the new value will be inserted at the rear end of the queue.
- **If $front \neq 0$ and $rear = \max - 1$** , it means that queue is not full, then set the value of rear to 0 and insert the new element there.

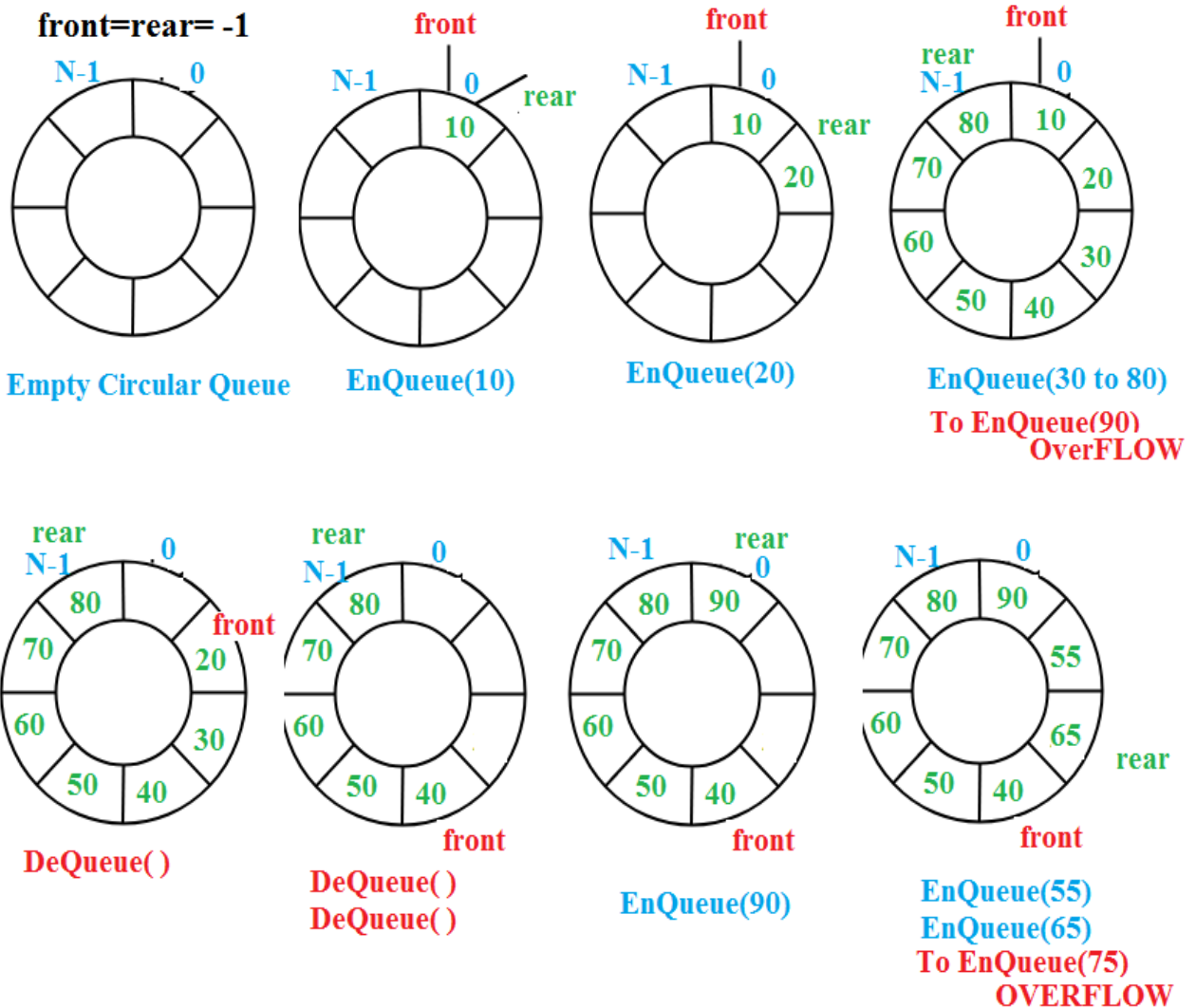
Queue is full:

- When **$front == 0$ && $rear = \max - 1$** , which means that front is at the first position of the Queue and rear is at the last position of the Queue.
- $front == rear + 1$;

Dequeue Operation: The steps of dequeue operation are given below:

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

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



Applications of Queue:

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used to transfer data asynchronously between two processes
3. Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
4. Queues are used in Playlist for jukebox to add songs to the end, play from the front.
5. Queues are used in operating system for handling interrupts. The interrupts are handled in the same order as they arrive i.e First come first served.

DEQUE:

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can be performed from either from the front or rear. Thus, it does not follow FIFO rule (First In First Out).



Types of Deque:

- 1. Input Restricted Deque:** In this deque, input is restricted at a single end but allows deletion at both the ends.
- 2. Output Restricted Deque:** In this deque, output is restricted at a single end but allows insertion at both the ends.



Operations on a Deque

- Initially take an array (deque) of size **n**. and Set two pointers at the first position and set **front = -1** and **rear = -1**.
- 1. Insert at the Front:** This operation adds an element at the front.
 - Check the position of front, If $\text{front} < 1$, we can't add elements in the front end. Otherwise decrement the front and at front location we can insert the element.
 - 2. Insert at the Rear:** This operation adds an element to the rear.
 - Check if the array is full. Then the queue is overflow. Otherwise, reinitialize $\text{rear} = 0$ & $\text{front} = 0$ for the first insertion, Else, increase rear by 1. and at rear location we can insert the element.
 - 3. Delete from the Front:** The operation deletes an element from the front.
 - Check If the deque is empty (i.e. $\text{front} = -1$), deletion cannot be performed (underflow condition). If the deque has only one element (i.e. $\text{front} = \text{rear}$), set $\text{front} = -1$ and $\text{rear} = -1$. Else, $\text{front} = \text{front} + 1$.
 - 4. Delete from the Rear:** This operation deletes an element from the rear.
 - If the deque is empty (i.e. $\text{front} = -1$), deletion cannot be performed (underflow condition). If the deque has only one element (i.e. $\text{front} = \text{rear}$), set $\text{front} = -1$ and $\text{rear} = -1$. Else, $\text{rear} = \text{rear} - 1$.

Priority Queue:-

- A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed.
- The general rules of processing the elements of a priority queue are
 - An element with higher priority is processed before an element with a lower priority.
 - Two elements with the same priority are processed on a first-come-first-served (FCFS) basis.

Array Representation of a Priority Queue:

- When arrays are used to implement a priority queue, then a separate queue for each priority number is maintained. Each of these queues will be implemented using circular arrays or circular queues. Every individual queue will have its own FRONT and REAR pointers.
- We use a two-dimensional array for this purpose where each queue will be allocated the same amount of space.
- FRONT[P] and REAR[P] contain the front and rear values of row P, where P is the priority number.

		SIZE of CQ					FRONT	REAR
		0	1	2	3	4		
p r i o r i t y	1	A					2	2
	2	F					4	4
	3	B		C	D		1	3
	4	E					0	0
	5	G		H	I	J	1	4
	6	M	K			L	3	0

Insertion:

- To insert a new element with priority P in the priority queue, add the element at the rear end of row P, where P is the row number as well as the priority number of that element.
- For example, if we have to insert an element X with priority number 2, then the priority queue will be given as shown in Fig.

ENQUEUE		SIZE of CQ					FRONT	REAR
		0	1	2	3	4		
p r i o r i t y	1	A					2	3
	2	X	F				4	0
	3	B		C	D		1	3
	4	E					0	0
	5	Z	G	H	I	J	1	0
	6	M	K			L	3	0

DEQUEUE		SIZE of CQ					FRONT	REAR	
		0	1	2	3	4			
p r i o r i t y	1	Y					3	3	A
	2	X	F				0	0	F
	3	C		D		2	3	B	
	4	E					0	0	
	5	Z	G	H	I	J	1	0	
	6	M	K			L	4	0	K

Deletion:

- To delete an element, we find the first nonempty queue and then process the front element of the first non-empty queue.
- In our priority queue, the first non-empty queue is the one with priority number 6 and the front element is K, so K will be deleted and processed first.

Multiple Queues:-

- When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.
- In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory. So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size.
- An array Queue[n] is used to represent two queues, Queue A and Queue B. The value of n is such that the combined size of both the queues will never exceed n. While operating on these queues, it is important to note one thing—queue A will grow from left to right, whereas queue B will grow from right to left at the same time.

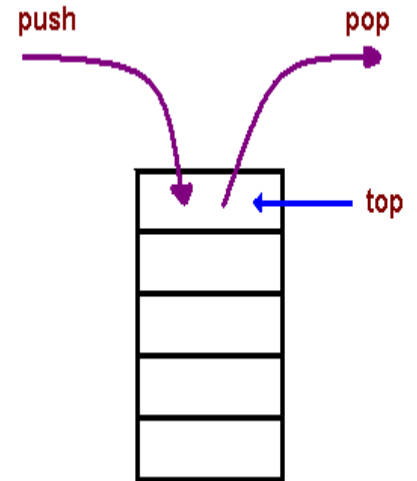
Example:

	QA	10	20	30	40		35	25	15	5	QB	
EMPTY	fA=rA = -1	fA			rA		rB			fB	fB=rB= SIZE	10
First Insertion	fA=rA = 0										fB=rB= SIZE-1	

- In the above example the array consists two queues like QA and QB. For QA there are pointers like fA(front of QA) and rA(rear of A). similarly for QB are fB & rB.
- Initially for QA, the pointer values of fA=rA= -1. For QB, the pointer values are fB=rB=SIZE. Because initially QA and QB are empty.
- For the first insertion in QA, the values of fA=rA=0. Similarly for QB, the values are fB=rB=SIZE-1.
- From the second insertion onwards we can increment only the rear pointer rA for QA and decrement the rear rB for QB.
- Delete the elements from queue only at front end. In QA, the elements can delete from fA, if you delete the element then increment fA. In QB, the elements can delete from fB, if you delete the element then decrement fB.
- When the condition $rA=rB-1$ or $rA+1=rB$ meets then the entire queue is full. If you try to insert the element in either of queues it says that QUEUE is OVERFLOW.

Stack:-

- Stack is a linear data structure in which insertion and deletion can perform at the same end called **top** of stack.
- When an item is added to a stack, the operation is called push, and when an item is removed from the stack the operation is called pop.
- Stack is also called as Last-In-First-Out (LIFO) list which means that the last element that is inserted will be the first element to be removed from the stack.
- When a stack is completely full, it is said to be Stack is **Overflow** and if stack is completely empty, it is said to be Stack is **Underflow**.



REPRESENTATION & IMPLEMENTATION STACK:

Array Representation of Stacks:

- Every stack has a variable called TOP associated with it, which is used to pointing the topmost element of the stack. It is this position where the element will be inserted to or deleted from.
- There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.
- If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX-1, then the stack is full.

A	AB	ABC	ABCD	ABCDE						
0	1	2	3	TOP = 4	5	6	7	8	9	

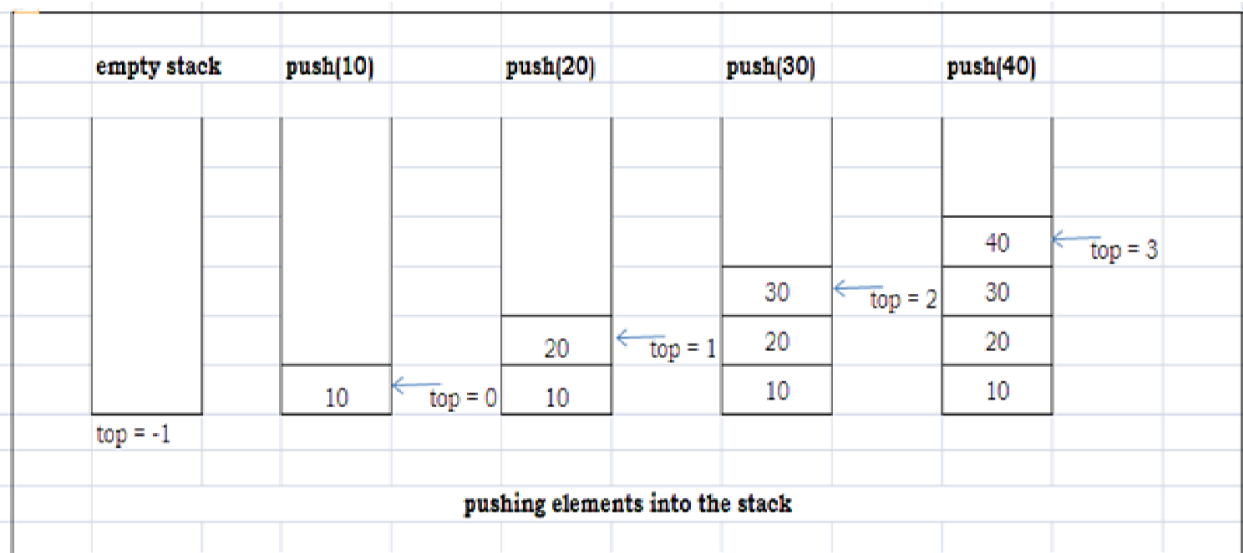
Array Implementation of Stack:

The basic operations performed in a Stack:

1. Push(x) - add element x at the top of the stack
2. Pop() - remove top element from the stack
3. peek() - get top element of the stack without removing it

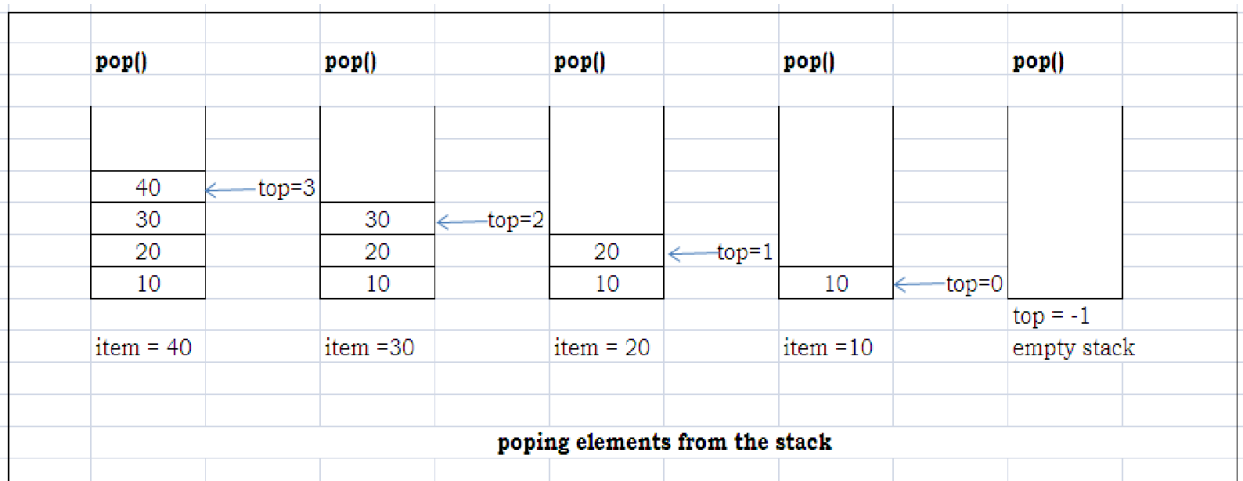
Algorithm for PUSH operation:

1. Check if the stack is **full** or not.
2. If the stack is full, then print error of overflow and exit the program.
3. If the stack is not full, then increment the top and add the element at top location.



Algorithm for POP operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top and decrement the top.



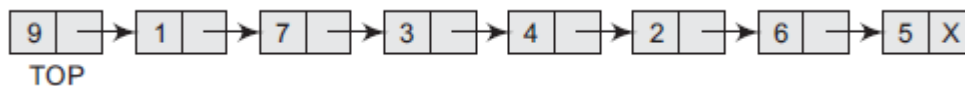
Algorithm for PEEK operation

1. Check if the stack is empty or not.
2. If the stack is empty, then print error of underflow and exit the program.
3. If the stack is not empty, then print the element at the top without removing it.

Linked Representation of Stacks:

- The drawback in that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance
- In a linked stack, every node has two parts, one that stores *data* and another that stores the *address* of the next node. The START pointer of the linked list is used as TOP.
- All insertions and deletions are done at the TOP (similar to **insertion at beginning**).

- If TOP = NULL, then it indicates that the stack is empty.
- The linked representation of a stack is



Linked Implementation of Stack:

- In a linked stack, each node of the stack consists of two parts i.e. data part and the next part. Each element of the stack points to its immediate next element in the memory.
- In the linked stack, there one pointer maintained in the memory i.e. TOP pointer. The TOP pointer contains the address of the starting element of the STACK.
- Both Insertion and deletions are performed at only one end called TOP. If TOP is NULL, it indicates that the stack is empty. Initially

```

struct node
{
    int data;
    struct node *next;
};
  
```

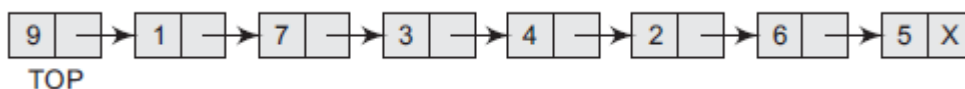
```

struct node *TOP = NULL ;
  
```

Operation on Linked STACK: There are two basic operations which can be implemented on the linked queues. The operations are PUSH and POP.

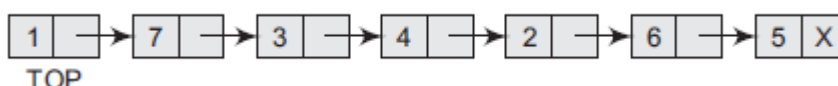
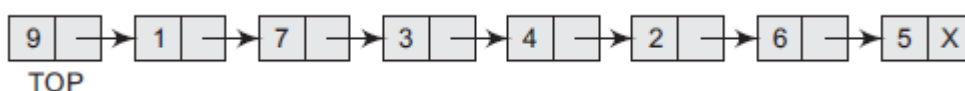
PUSH function: PUSH function will add the element at the beginning of the linked list.

1. Declare a new node and allocate memory for it.
2. If TOP == NULL, make TOP points to the new node.
3. Otherwise, add the new node at TOP end and makes the next of new node is previous TOP.



POP function: POP function will remove the TOP element from the STACK.

1. Check whether the stack is empty or not
2. If it is the stack is empty (TOP == NULL), We can't POP the element.
3. Otherwise, Make the TOP node points to the next node. i.e TOP = TOP->next; Free the TOP node's memory.



Algorithms:

void PUSH(int Value)

```
{
struct node *new_node=(struct node *)
malloc(sizeof(struct node));

new_node->data = Value;
new_node->Next = TOP
TOP = new_node;
}
```

void POP()

```
{
if(TOP==NULL)
printf("UNDERFLOW")
else
{
PTR=TOP;
TOP=TOP->Next;
free(PTR)
}
}
```

Applications of Stacks

- ✓ Stack is used to reversing the given string.
- ✓ Stack is used to evaluate a postfix expression.
- ✓ Stack is used to convert an infix expression into postfix/prefix form.
- ✓ Stack is used to matching the parentheses in an expression.

Reversing list:

A list of numbers or string can be reversed by using the stack, perform the following steps

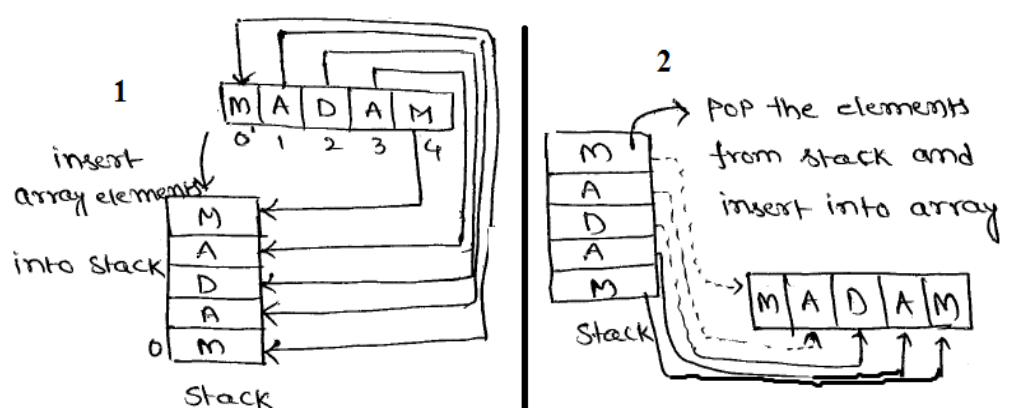
1. Reading the elements from the array and pushed into the stack.
2. Pop the elements and again stored into the array starting from first index

Algorithm:

```
void reverse()
{
// input: array A[ ], size N
// top = -1, Stack S
// output: Reversing array A[ ]
for i=0 to N-1
TOP = TOP+1;
S[TOP] = A[i];

for i=0 to N-1
A[i]=S[TOP];
TOP = TOP-1;
}
```

Example:



Factorial Calculation:

- To find the solution of larger problem, a general method is reduce the larger problem into one or more sub problems. This process will continuous until the sub problem is finding the solution. Finally using all the sub problems solutions we will find the solution for the larger problem.

- A *Recursion* is defined as a function that calls itself.
- To understand recursion, let us take an example of calculating factorial of a number.
- To calculate $n!$, we multiply the number with factorial of the number that is 1 less than that number. $n! = n \times (n-1) \times (n-2) \times \dots \times 2 \times 1$

- In other words, $n! = n \times (n-1)!$
- Let us say we need to find the value of $5!$

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

- This can be written as $5! = 5 \times 4!$,

$$\text{where } 4! = 4 \times 3!$$

Therefore, $5! = 5 \times 4 \times 3!$

Similarly, $5! = 5 \times 4 \times 3 \times 2!$

Expanding further $5! = 5 \times 4 \times 3 \times 2 \times 1!$

We know, $1! = 1$

- Now if you look at the problem carefully, you can see that we can write a recursive function to calculate the factorial of a number. Every recursive function must have a base case and a recursive case. For the factorial function,

Base case is when $n = 1$, because if $n = 1$, the result will be 1 as $1! = 1$.

Recursive case of the factorial function will call itself but with a smaller value of n , this case can be given as $\text{factorial}(n) = n \times \text{factorial}(n-1)$

PROBLEM	SOLUTION
$5!$	$5 \times 4 \times 3 \times 2 \times 1!$
$= 5 \times 4!$	$= 5 \times 4 \times 3 \times 2 \times 1$
$= 5 \times 4 \times 3!$	$= 5 \times 4 \times 3 \times 2$
$= 5 \times 4 \times 3 \times 2!$	$= 5 \times 4 \times 6$
$= 5 \times 4 \times 3 \times 2 \times 1!$	$= 5 \times 24$
	$= 120$

Evaluation of Expressions:-

- An expression is defined as the combination of operands (variables, constants) and operators arranged as per the syntax of the language.
- An expression can be represented using three different notations. They are infix, postfix and prefix notations:

Prefix: An arithmetic expression in which we fix (place) the arithmetic operator before (pre) its two operands. The prefix notation is called as polish notation. Example: $+ A B$

Infix: An arithmetic expression in which we fix (place) the arithmetic operator in between the two operands. Example: $A + B$

Postfix: An arithmetic expression in which we fix (place) the arithmetic operator after (post) its two operands. The postfix notation is called as suffix notation OR reverse polish notation.

Example: $A B +$

Operator Precedence: When an expression consist different level of operators we follow it. We consider five binary operations: $+$, $-$, $*$, $/$ and $^$ (exponentiation). For these binary operations, the following in the order of precedence (highest to lowest): $^$, $*$, $/$, $+$, $-$

Operator Associativity: When an expression consist more than one same level precedence operators we follow it.

Basically we have Left to Right associativity and Right to Left Associativity. Most of the operators are follows Left to Right but some of the operators are follow Right to left Associativity like Unary (+/-), ++/-- , Logical negation (!), Pointer and address (*,&), Conditional Operators and Assignment operators(=,+=-,*=,/=,%=).

Example: $x = a / b - c + d * e - a * c$

Let a = 4, b = c = 2, d = e = 3 then the value of x is found as



$$= ((4 / 2) - 2) + (3 * 3) - (4 * 2) = 0 + 9 - 8 = 1$$

EVALUATION OF POSTFIX EXPRESSION:

The standard representation for writing expressions is infix notation. But the compiler uses the postfix notation for evaluating the expression rather than the infix notation. It is an easy task for evaluating the postfix expression than infix expression because there are no parentheses. To evaluate an expression we scan it from left to right. The postfix expression is evaluated easily by the use of a stack.

To evaluate a postfix expression use the following steps...

1. Read the **poststring** from left to right
2. Initialize an **empty Stack**
3. Repeat until end of the **poststring**
 - i. If the scanned character is operand, then push it on to the Stack.
 - ii. If the scanned character is operator (+ , - , * , / etc.), then pop top two elements from the stack, perform the operation with the operator then push result back on to the Stack.
4. Finally! We have one element in the stack, perform a pop operation and display the popped value as **final result**.

Postfix Expression is 5 3 + 8 2 - *		
Symbol	Stack	Evaluation
Initially	 <p>Stack is empty</p>	
5	 <p>Push(5)</p>	

3	<div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 0 auto;"> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">3</div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">5</div> </div> <p style="text-align: center;">Push(3)</p>	
+	<div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 0 auto;"> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">8</div> </div> <p style="text-align: center;">Value1=pop() Value2=pop() Result=Value2+Value1 Push(Result)</p>	<p style="text-align: center;">Value1=3 Value2=5 Result=5+3=8 Push(8)</p>
8	<div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 0 auto;"> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">8</div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">8</div> </div> <p style="text-align: center;">Push(8)</p>	
2	<div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 0 auto;"> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">2</div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">8</div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">8</div> </div> <p style="text-align: center;">Push(2)</p>	
-	<div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 0 auto;"> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">6</div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">8</div> </div> <p style="text-align: center;">Value1=pop() Value2=Pop() Result=Value2-Value1 Push(Result)</p>	<p style="text-align: center;">Value1=2 Value2=8 Result=8-2=6 Push(6)</p>
*	<div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 0 auto;"> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">48</div> </div> <p style="text-align: center;">Value1=pop() Value2=Pop() Result=Value2*Value1 Push(Result)</p>	<p style="text-align: center;">Value1=6 Value2=8 Result=8*6=48 Push(48)</p>
End of Expression	<div style="border: 1px solid black; padding: 2px; width: fit-content; margin: 0 auto;"> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%;"></div> <div style="border: 1px solid black; width: 100%; height: 100%; text-align: center;">48</div> </div> <p style="text-align: center;">Result=pop()</p>	<p style="text-align: center;">Final Result is 48</p>

Conversion of INFIX to POSTFIX:

Procedure to convert from infix expression to postfix expression is as follows.

1. Initialize an empty stack
2. Push “(“ onto Stack, and add “)” to the end of Infix string.
3. Scan the Infix string from left to right until end of the infix
 - i. If the scanned character is “(“, pushed into the stack.
 - ii. If the scanned character is “)”, pop the elements from the stack up to encountering the “(“, and add the popped elements to postfix string except parentheses.
 - iii. If the scanned character is an operand, add it to the Postfix string.
 - iv. If the scanned character is an Operator, compare the precedence of the character with the element on top of the stack. If top of Stack has lower precedence over the scanned character then push the operator into the stack else pop the element from the stack and add it to postfix string and push the scanned character to stack.

Example: a * (b + c) *d

Token	Stack				Postfix String
	(
a	(a
*	(*			A
((*	(A
b	(*	(Ab
+	(*	(+	Ab
c	(*	(+	Abc
)	(*			abc+
*	(*			abc+*
d	(*			abc+*d
)					abc+*d*