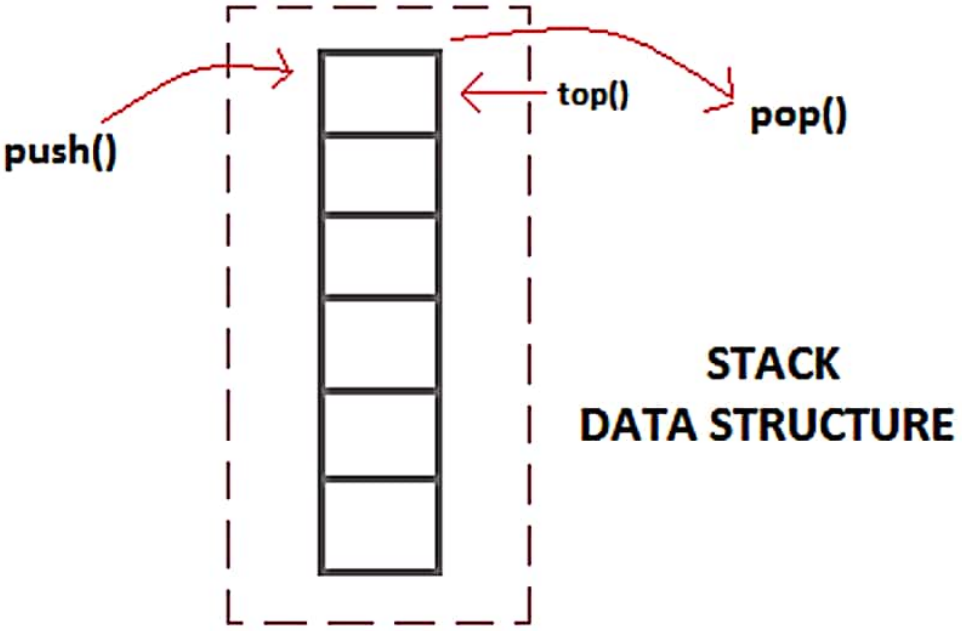
MODULE- II

What is Stack Data Structure?

Stack is an abstract data type with a bounded(predefined) capacity. It is a simple data structure that allows adding and removing elements in a particular order. Every time an element is added, it goes on the **top** of the stack and the only element that can be removed is the element that is at the top of the stack, just like a pile of objects.



Basic features of Stack

1. Stack is an **ordered list** of **similar data type.**
2. Stack is a **LIFO**(Last in First out) structure or we can say **FILO**(First in Last out).
3. push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called **Top.**
4. Stack is said to be in **Overflow** state when it is completely full and is said to be in **Underflow** state if it is completely empty.

Applications of Stack

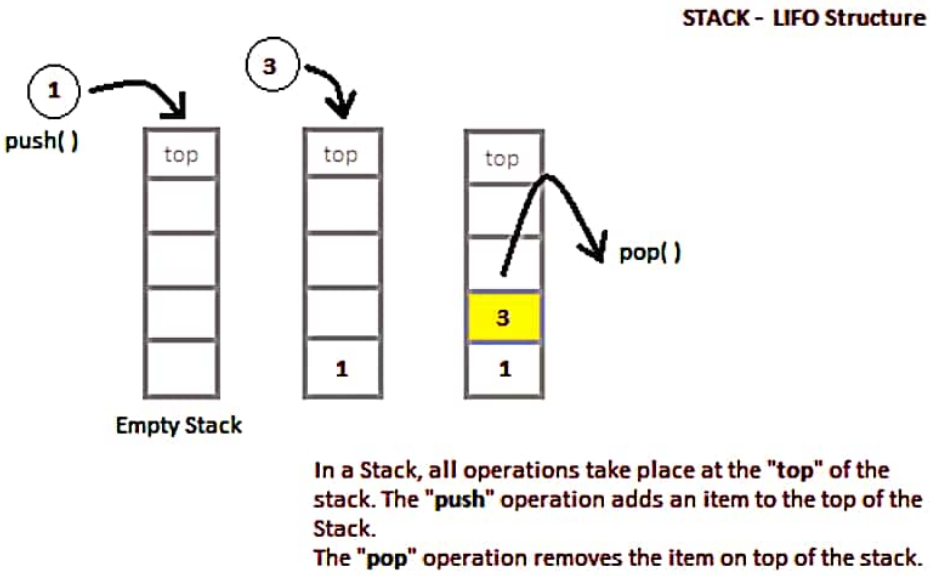
The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.

There are other uses also like:

1. Parsing
2. Expression Conversion(Infix to Postfix, Postfix to Prefix etc)

Implementation of Stack Data Structure

Stack can be easily implemented using an Array or a Linked List. Arrays are quick, but are limited in size and Linked List requires overhead to allocate, link, unlink, and deallocate, but is not limited in size. Here we will implement Stack using array.



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.

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.

Analysis of Stack Operations

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- **Push Operation** : O(1)
- **Pop Operation** : O(1)
- **Top Operation** : O(1)
- **Search Operation** : O(n)

The time complexities for push() and pop() functions are O(1) because we always have to insert or remove the data from the **top** of the stack, which is a one step process.

Conversion & Evaluation of Arithmetic Expressions

Arithmetic expression notations

In any arithmetic expression, each operator is placed in between two operands of it (ie mathematical representation) This way of representing an arithmetic expression is called as infix expression

eg. A+B.

Apart from usual mathematical representation of an arithmetic expression, an expression can also be represented in the following two ways:

1. Polish or Prefix Notation

2. Reverse Polish or Postfix Notation

Polish or Prefix Notation

The notation in which the operator symbol is placed before its operands, is referred as polish or prefix notation.

For Example: +AB. Reverse

Reverse or Postfix Notation

The notation, in which the operator symbol is placed after its operands, is referred as reverse polish or postfix notation.

For Example: AB+

Advantage of Prefix & Postfix over Infix notation is that there no need of parenthesis in the expression

Mathematical Procedure for conversion:

The possible conversions are:

1. Infix to Prefix

2. Prefix to Infix

3. Infix to Postfix

4. Postfix to Infix

5. Prefix to Postfix

6. Prefix to Postfix

Standard Arithmetic Operators and Precedence Levels :

^ (exponential)	Higher Level
***, /, %**	Middle Level
+, -	Lower Level

Infix to Prefix:

> Identify the inner most brackets
> Identify the operator according to the priority of evaluation
> Represent the operator and corresponding operator in prefix notation
> continue this process until the equivalent prefix expression is achieved.

Example : (A + B * (C – D ^ E) / F)

= (A + B * (C – [^ DE]) / F)

= (A + B * [– C ^ DE] / F)

= (A + [* B – C ^ DE] / F)

= A + [/ * B – C ^ DEF]

= + A / * B – C ^ DEF

Prefix to Infix:

> Identify the operator from right to left order
> The two operands which immediately follows the operator are for evaluation
> Represent the operator and operands in infix notation
> Continue this process until the equivalent infix expression is achieved.
> If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Example : + A / * B – C ^ DEF

= + A / * B – C [D ^ E] F

= + A / * B [C – D ^ E] F

= + A / [B * (C – D ^ E)] F

= + A [B * (C – D ^ E) / F]

= A + B * (C – D ^ E) / F

Infix to Postfix:

> Identify the inner most brackets
> Identify the operator according to the priority of evaluation
> Represent the operator and corresponding operator in postfix notation
> continue this process until the equivalent postfix expression is achieved.

Example : (A + B * (C – D ^ E) / F)

= (A + B * (C – [DE ^]) / F)

= (A + B * [CDE ^ –] / F)

= (A + [BCDE ^ – *] / F)

= A + [BCDE ^ – *F/]

= ABCDE ^ – *F/ +

Postfix to Infix:

> Identify the operator from left to right order

> The two operands which immediately preceeds the operator are for evaluation

> Represent the operator and operands in infix notation

> Continue this process until the equivalent infix expression is achieved

> If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Example : ABCDE ^ – *F/ +

= ABC[D ^ E] – *F/+

= AB [C – D ^ E] * F / +

= A [B *(C – D ^ E)] F / +

= A [B * (C – D ^ E) / F] +

= A + B * (C – D ^ E) / F

Prefix to Postfix:

> Identify the operator from right to left order

> The two operands which immediately follows the operator are for evaluation

> Represent the operator and operands in Postfix notation

> Continue this process until the equivalent Postfix expression is achieved.

> If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Example : + A / * B – C ^ DEF

= +.A / * B – C [DE ^] F

= + A / * B [CDE ^ –] F

= + A / [BCDE ^ – *] F

= + A [BCDE ^ – * F /]

= ABCDE ^ – * F / +

Postfix to Prefix:

> Identify the operator from left to right order
> The two operands which immediately preceeds the operator are for evaluation
> Represent the operator and operands in prefix notation
> Continue this process until the equivalent prefix expression is achieved
> If the priority of a scanned operator is less than any operators available with [ ], then put them within ( ).

Example : ABCDE ^ – *F/ +
= ABC[^ DE] – *F/+
= AB [– C ^ DE] * F / +
= A [* B – C ^ DE] F / +
= A [/ * B – C ^ DE F] +
= + A / * B – C ^ DE F

4.7.3.3 Importance of Postfix Expression

Although human beings are quite used to work with mathematical expression i.e. INFIX notation, which is rather complex as using this notation one has to remember a set of rules. The rules include BODMAS and ASSOCIATIVITY.

In case of POSTFIX notation, the expression, which is easier to work or evaluate the expression as compared to the INFIX expression. In a POSTFIX expression, operands appear before the operators, there is no need to follow the operator precedence and any other rules.

Actually, the processor represents the mathematical expression in postfix expression and uses it for evaluation. To do this it implements the concept of stack.

4.7.3.4 Conversion of an INFIX expression into POSTFIX expression using Stack

Algorithm for converting from INFIX to POSTFIX

[Assume Q is an Infix expression and P is the corresponding Postfix notation.]

Step 1: PUSH '(' onto STACK and Add ')' at the end of Q

Step 2: Repeatedly scan from Q Until STACK is Empty

Step 2.1: If Q[I] is an operand, Then Add it to P[J]
Step 2.2: Else if Q[I] is '(', Then PUSH Q[I] into STACK
Step 2.3: Else if Q[I] is an operator , Then

Step 2.3.1: While STACK[TOP] is an operator and has higher or equal priority compared to Q[I]
Pop operators from STACK and add to P[J]
[End Of While]

Step 2.3.2: Push operator Q [I] onto STACK

Step 2.4: Else if Q[I] is an ')', Then

Step 2.4.1: Repeatedly pop operators from STACK and add to P[J] until '(' is encountered

Step 2.4.2: Remove '(' from STACK

[End Of IF]

[End Of Loop Step – 2]

Step 3: Exit

Procedure of Conversion

e.g. Q = (B * C - (D / E ^ F))

Symbol Scanned from Q	Stack	Postfix Expression P
(	(	
B	(	B
*	(*	B
C	(*	BC
-	(-	BC*
(	(- (	BC*
D	(- (	BC*D
/	(- (/	BC*D
E	(- (/	BC * DE
^	(- (/ ^	BC * DE
F	(- (/ ^	BC * DEF
)	(- (/ ^	BC * DEF ^/
)	(-	BC * DEF ^ / -

- Infix to postfix conversion using stack

Q:- (B*C-(D/E^F))

Q[i]	Stack	P[j]
(	(	
B	(	B
*	(*	B
C	(*	BC
-	(-	BC*
(	(-(	BC*
D	(-(	BC*D
/	(-(/	BC*D
E	(-(/	BC*DE
^	(-(/^	BC*DE
F	(-(/^	BC*DEF
)	(-	BC*DEF^/
)	empty	BC*DEF^/-

Scanned with CamScanner

4.7.3.8 Conversion of infix expresion to prefix expression using stack

[Assume Q is an infix expression. Consider there are two stacks S1 and S2 exist. The following algorithm converts the infix expression Q into its equivalent Prefix notation.]

Algorithm :

Step 1: Add left parenthesis '(' at the beginning of the expression Q

Step 2: PUSH ')' onto Stack S1

Step 3: Repeatedly Scan Q in right to left order, Until Stack S1 is Empty

Step 3.1: If Q[I] is an operand, Then PUSH it onto Stack S2

Step 3.2: Else if Q [I] is ')', Then PUSH it onto Stack S1

Step 3.3: Else if Q [I] is an operator (OP) , Then

Step 3.3.1: Set X := POP (S1)

Step 3.3.2: Repeat while X is an Operator AND (Precedence(X) > Precedence(OP))

PUSH (X) onto Stack S2

Set X := POP (S1)

[End of While – Step 3.3.2]

Step 3.3.3: PUSH (X) onto Stack S1

Step 3.3.4: PUSH (OP) onto Stack S1

Step 3.4: Else if Q [I] is '(' , Then

Step 3.4.1: Set X := POP(S1)

Step 3.4.2: Repeat, While(X != ')') [Until right parenthesis found]

Step 3.4.2.1: PUSH (X) onto Stack S2

Step 3.4.2.2: Set X := POP(S1)

[End of While – Step 3.4.2]

[End of IF – Step 3.1]

[End of Loop – Step 3]

Step 4: Repeat, While Stack S2 is not Empty

Step 4.1: Set X := POP (S2)

Step 4.2: Display X

[End of While]

Step 5: Exit

Procedure of Conversion

e.g. Q = (A + B * C * (M * N ^ P + T) – G + H

Symbol Scanned from Q	Stack S1	Stack S2
H	)	H
+	)+	H
G	)+	HG
-	)+-	HG
)	)+-)	HG
T	)+-)	HGT
+	)+-)+	HGT
P	)+-)+	HGTP
^	)+-)+^	HGTP
N	)+-)+^	HGTPN
*	)+-)+*	HGTPN^
M	)+-)+*	HGTPN^M
(	)+-	HGTPN^M*+
*	)+-*	HGTPN^M*+
C	)+-*	HGTPN^M*+C
*	)+-**	HGTPN^M*+C
B	)+-**	HGTPN^M*+CB
+	)+-+	HGTPN^M*+CB**
A	)+-+	HGTPN^M*+CB**A
(		HGTPN^M*+CB**A+-+

So the Prefix Notation We can get by popping all the symbols from Stack S2.
i.e. +-+A**BC+*M^NPTGH

Infix to prefix using stack

Q = K+L-M*N+(O∧P)*W/U/V*T+Q

Input (Q)	Stack S_1	Prefix on stack S_2
Q		Q
+	+	
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
∧	+*//*)∧	QTVUWP
O	+*//*)∧	QTVUWP
(	+*//	QTVUWP∧
+	+*// +	QTVUWP∧*//*
N	++	QTVUWP∧*//*N
*	++*	QTVUWP∧*//*N
M	++*	QTVUWP∧*//*NM
-	++-	QTVUWP∧*//*NM*
L	++-	QTVUWP∧*//*NM*L
+	++-+	QTVUWP∧*//*NM*L
K	++-+	QTVUWP∧*//*NM*L+-++

←| ++-+KL*MN*//*∧OPWUV+Q

Scanned with CamScanner

Q. ((A + B) * C - (D - E) ^ (++G))

Q[i]	Stack S_1	Stack S_2
)	)	
)	))	
G	))	G
++	)) ++	G
(	)	G++
^	) ^	G++
)	) ^)	G++
E	) ^)	G++E
-	) ^) -	G++E
D	) ^) -	G++ED
(	) ^	G++ED-
^	) ^	G++ED-^
C	) -	G++ED-^C
*	) - *	G++ED-^C
)	) - *)	G++ED-^C
B	) - *)	G++ED-^CB
+	) - *) +	G++ED-^CB
A	) - *) +	G++ED-^CBA
(	) - *	G++ED-^CBA+
(	empty	G++ED-^CBA+*-

-*+ABC^-DE++G

Scanned with CamScanner

4.7.3.6 Evaluation of post fix expression using stack

[Assume P is a post fix expression]

Step 1: Add ')' at the end of P

Step 2: Repeatedly scan P from left to right until ')' encountered

Step 2.1: if P[I] is an operand, then

Push the operand onto STACK

Step 2.2: else If P[I] is an operator ⊗ then

Step 2.2.1: Pop two elements from STACK

(1st element is A and 2nd element is B)

Step 2.2.2: Evaluate B ⊗ A

Step 2.2.3: Push result back to STACK

[End of IF]

[End of loop – Step 2]

Step 3: VALUE: = STACK [TOP]

Step 4: Display VALUE

Step 5: Exit.

Q. Evaluate the following postfix expression.

4 6 2 + * 12 3 / -

P[i]	Stack	A	B	B (op) A
4	4			
6	4 | 6			
2	4 | 6 | 2			
+	4 | 8	2	6	6+2 = 8
*	32	8	4	4*8 = 32
12	32 | 12			
3	32 | 12 | 3			
/	32 | 4	3	12	12 / 3 = 4
-	empty (28)	4	32	32-4 = 28

Q

P[i]	Stack	A	B	B (op) A
5	5			
6	5 | 6			
*	30	6	5	6 * 5 = 30
24	30 | 24			
2	30 | 24 | 2			
3	30 | 24 | 2 | 3			
^	30 | 24 | 8	3	2	2^3
/	30 | 3	8	24	24 / 8 = 3
-	27	12	30	30 - 3

18

27

Scanned with CamScanner

4.7.3.10 Evaluation of prefix expression

[Assume P is a Prefix Expression]

Step 1: Add '(' at the beginning of the prefix expression

Step 2: Repeatedly scan from P in right to left order until '(' encountered

Step 2.1: If P [I] is an operand, then

PUSH the operand onto STACK

Step 2.2: Else if P [I] is operator (OP), then

Step 2.2.1: Pop two elements from STACK

(1st element is A and 2nd element is B)

Step 2.2.2: Evaluate A (OP) B

Step 2.2.3: Push result back to STACK

[End of if]

[End of loop – Step 2]

Step 3: VALUE: = STACK [TOP]

Step 4: Display VALUE

Step 5: Exit.

For Example :

Evaluate the following Prefix Expression using stack :

P = (- , * , 3 , + , 16 , 2 , / , 12 , 6

Symbol Scanned from Prefix Expression in Right to Left Order	Stack	A	B	A (OP) B
6	6			
12	6 , 12			
/	2	12	6	12 / 6 = 2
2	2 , 2			
16	2 , 2 , 16			
+	2 , 18	16	2	16 + 2 = 18
3	2 , 18 , 3			
*	2 , 54	3	18	3 * 18 = 54
-	52	54	2	54 – 2 = 52

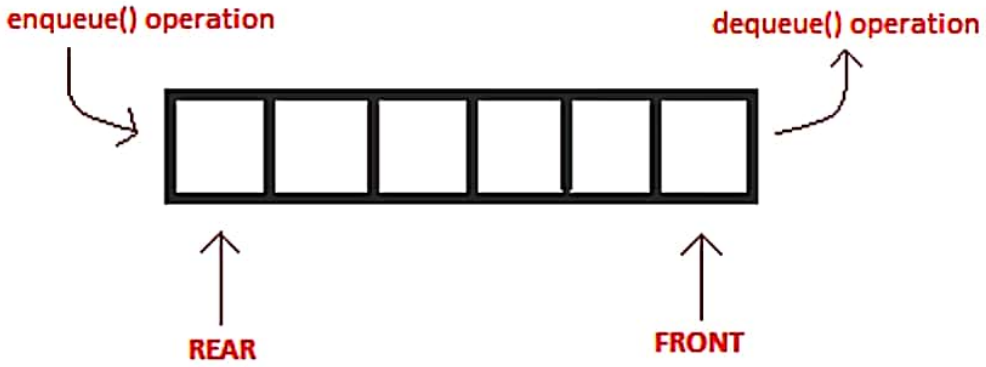
Finally the value in the Stack is 52.

Queue

Queue is also an abstract data type or a linear data structure, just like stack data structure, in which the first element is inserted from one end called the **REAR**(also called **rear**), and the removal of existing element takes place from the other end called as **FRONT**(also called **front**).

This makes queue as **FIFO**(First in First Out) data structure, which means that element inserted first will be removed first.

The process to add an element into queue is called **Enqueue** and the process of removal of an element from queue is called **Dequeue.**



enqueue( ) is the operation for adding an element into Queue.

dequeue( ) is the operation for removing an element from Queue .

QUEUE DATA STRUCTURE

Applications of Queue

Queue is used when things don't have to be processed immediately, but have to be processed in **F**irst **I**n **F**irst **O**ut order like Breadth First Search. This property of Queue makes it also useful in following kind of scenarios.

1) When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.

2) When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

Queue, as the name suggests is used whenever we need to manage any group of objects in an order in which the first one coming in, also gets out first while the others wait for their turn, like in the following scenarios:

3) Serving requests on a single shared resource, like a printer, CPU task scheduling etc.
4) In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
5) Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive i.e First come first served.

Basic features of Queue

- Like stack, queue is also an ordered list of elements of similar data types.
- Queue is a FIFO(First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- peek( ) function is oftenly used to return the value of first element without dequeuing it.

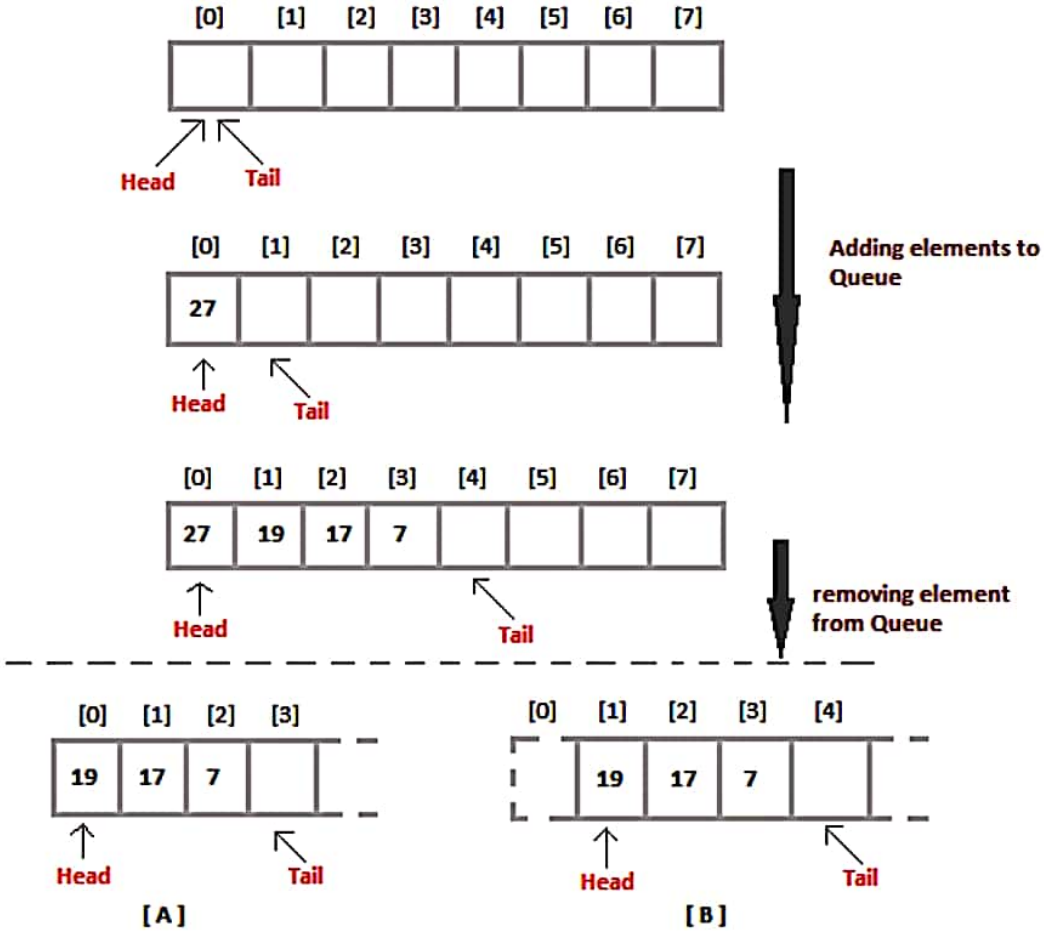
Types of Queue:

1. Linear Queue
2. Circular Queue
3. Doubled Ended Queue (D-Queue)
4. Priority Queue

Implementation of Queue Data Structure

Queue can be implemented using an Array, Stack or Linked List. The easiest way of implementing a queue is by using an Array.

Initially the front (FRONT) and the rear (REAR) of the queue points at the first index of the array (starting the index of array from 0). As we add elements to the queue, the rear keeps on moving afront, always pointing to the position where the next element will be inserted, while the front remains at the first index.



When we remove an element from Queue, we can follow two possible approaches (mentioned [A] and [B] in above diagram). In [A] approach, we remove the element at front position, and then one by one shift all the other elements in forward position.

In approach [B] we remove the element from front position and then move front to the next position.

In approach [A] there is an overfront of shifting the elements one position forward every time we remove the first element.

In approach [B] there is no such overfront, but whenever we move front one position afront, after removal of first element, the size on Queue is reduced by one space each time.

Algorithm for ENQUEUE operation

1) Check if the queue is full or not.
2) If the queue is full, then print overflow error and exit the program.
3) If the queue is not full, then increment the rear and add the element.

Algorithm for DEQUEUE operation

1) Check if the queue is empty or not.
2) If the queue is empty, then print underflow error and exit the program.
3) If the queue is not empty, then print the element at the front and increment the front.

Linear Queue Implementation

Insertion

Q insertion(Q, max, item, front, rear)

Step1: If (rear==max-1) then

Print "overflow"

Step2: Else

if(rear==-1 and front==-1)

set front=0

rear = 0

Step 3: Else

rear=rear+1

[End of if]

Step 4: Q[rear]=item

Step 5: Stop

Deletion

```
Q deletion(Q, max, item, front, rear)

Step1:  Start

Step2:  If (rear==-1 and front==-1)

            Print "underflow"

            End of if, exit.

Step3:  item = Q[Front]

      print  "The deleted item is" item

Step 4: If (rear==front)

            set rear =-1

            front=-1

Step 5:   else

            front=front+1

            [End of  if]

Step 6:  Stop
```

Display

```
Q display(Q, max, front, rear)
Step1:  Start
Step2:  If (front==-1 and rear==-1 )
            Print "no element for display"
Step3: else
Step 3.1: repeat for (i =front to rear by +1)
Step 3.1.1: print Q[i]
            [End of for]
            [End of if ]
Step 6:  Stop
```

Complexity Analysis of Queue Operations

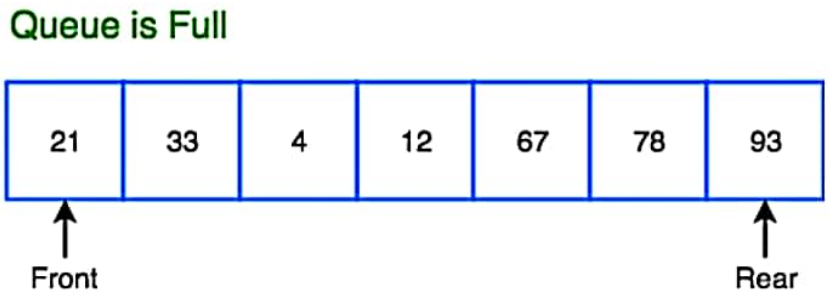
Just like Stack, in case of a Queue too, we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.

Enqueue: O(1)
Dequeue: O(1)
Size: O(1)

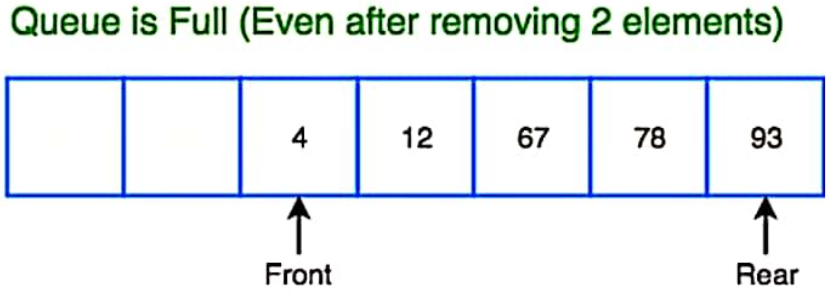
What is a Circular Queue?

Before we start to learn about Circular queue, we should first understand, why we need a circular queue, when we already have linear queue data structure.

In a Linear queue, once the queue is completely full, it's not possible to insert more elements. Even if we dequeue the queue to remove some of the elements, until the queue is reset, no new elements can be inserted.



When we dequeue any element to remove it from the queue, we are actually moving the front of the queue forward, thereby reducing the overall size of the queue. And we cannot insert new elements, because the rear pointer is still at the end of the queue.



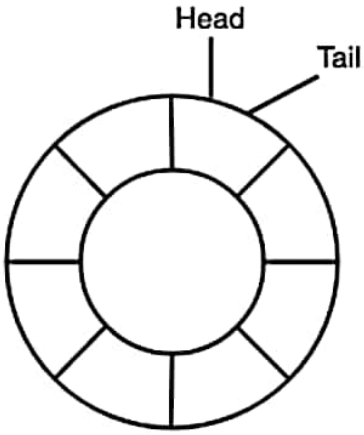
The only way is to reset the linear queue, for a fresh start.

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.

Basic features of Circular Queue

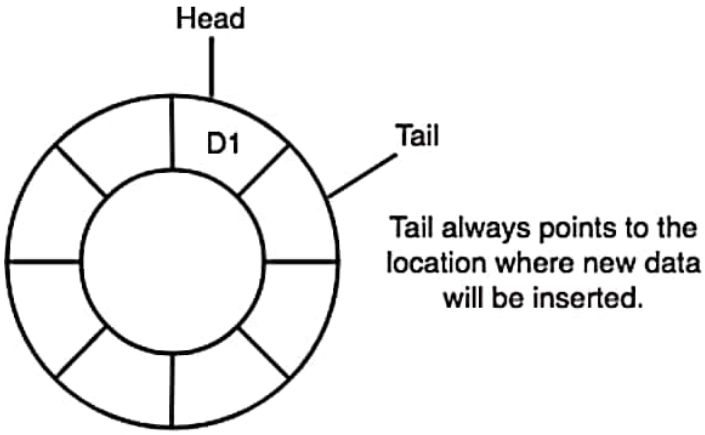
1. In case of a circular queue, front pointer will always point to the front of the queue, and rear pointer will always point to the end of the queue.
2. Initially, the front and the rear pointers will be pointing to the same location, this would mean that the queue is empty.

Initially the queue is empty, as Head and Tail are at same location

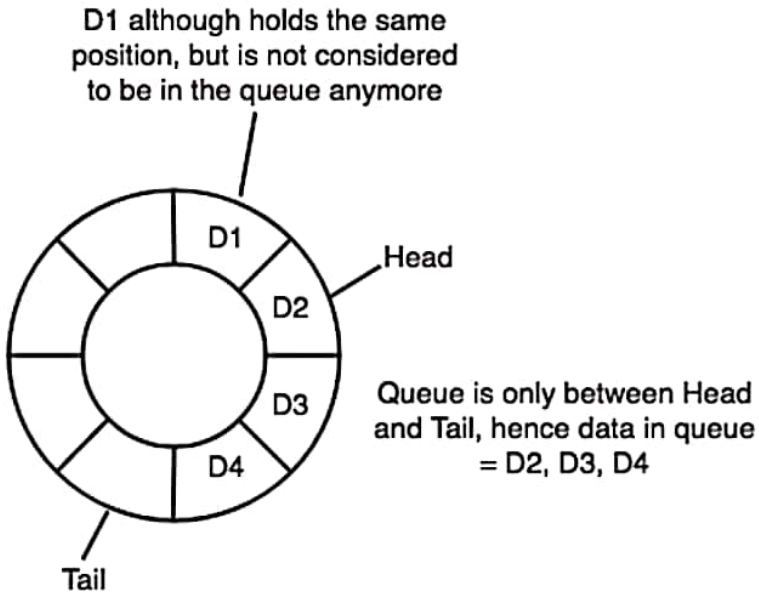


A simple circular queue with size 8

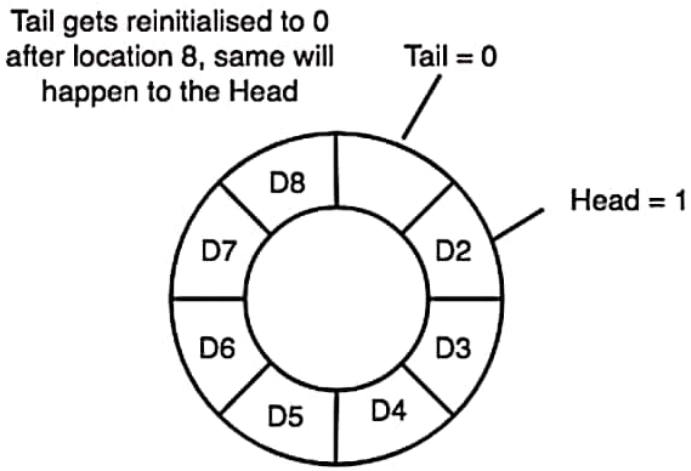
3. New data is always added to the location pointed by the rear pointer, and once the data is added, rear pointer is incremented to point to the next available location.



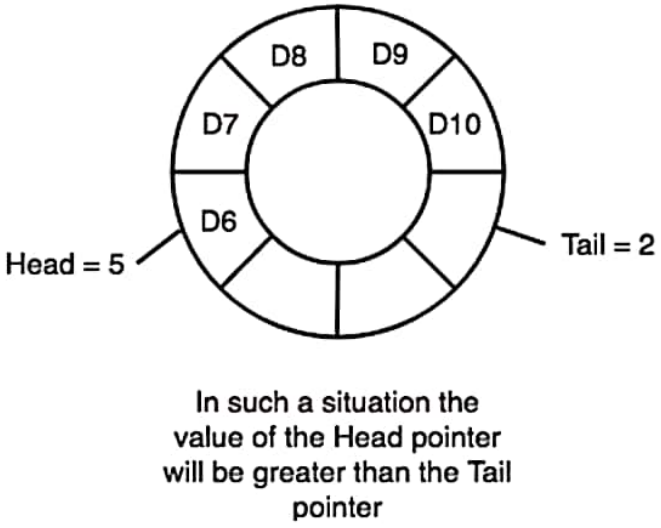
4. In a circular queue, data is not actually removed from the queue. Only the front pointer is incremented by one position when **dequeue** is executed. As the queue data is only the data between front and rear, hence the data left outside is not a part of the queue anymore, hence removed.



5. The front and the rear pointer will get reinitialised to **0** every time they reach the end of the queue.



6. Also, the front and the rear pointers can cross each other. In other words, front pointer can be greater than the rear. Sounds odd? This will happen when we dequeue the queue a couple of times and the rear pointer gets reinitialised upon reaching the end of the queue.



Going Round and Round

Another very important point is keeping the value of the rear and the front pointer within the maximum queue size.

In the diagrams above the queue has a size of 8, hence, the value of rear and front pointers will always be between 0 and 7.

This can be controlled either by checking everytime whether rear or front have reached the maxSize and then setting the value 0 or, we have a better way, which is, for a value x if we divide it by 8, the remainder will never be greater than 8, it will always be between 0 and 0, which is exactly what we want.

So the formula to increment the front and rear pointers to make them **go round and round** over and again will be, front = (front+1) % maxSize or rear = (rear+1) % maxSize

Application of Circular Queue

Below we have some common real-world examples where circular queues are used:

1. Computer controlled **Traffic Signal System** uses circular queue.
2. CPU scheduling and Memory management.

Implementation of Circular Queue

Below we have the implementation of a circular queue:

1. Initialize the queue, with size of the queue defined (maxSize), and front and rear pointers.
2. enqueue: Check if the number of elements is equal to maxSize - 1:
    - If **Yes**, then return **Queue is full.**
    - If **No**, then add the new data element to the location of rear pointer and increment the rear pointer.
3. dequeue: Check if the number of elements in the queue is zero:
    - If **Yes**, then return **Queue is empty.**
    - If **No**, then increment the front pointer.

4. Finding the size:
    - If, **rear >= front**, size = (rear - front) + 1
    - But if, **front > rear**, then size = maxSize - (front - rear) + 1

Insertion

```
CQ insertion(CQ, max, item, front, rear)
Step 1: Start
Step2:  If (front==0 and rear=max-1) or
(front==(rear+1)%max)
            Print "overflow"
Step3:  Else if (rear==-1 and front==-1)
            set  front=0,  rear = 0
Step 4: Else
            rear=(rear+1)%max
            [End of  if]
Step 5:   CQ[rear]=item
Step 6:  Stop
```

Deletion

```
CQ deletion(CQ, max, item, front, rear)
Step 1: Start
Step2:  If (front==-1 and rear==-1)
            Print "underflow"
Step3:  item =CQ[front]
            Print "the deleted item is", item
Step 4: if (rear==front)
            rear=-1 front=-1
Step 5:else
            front=(front+1)%max
            [end of if]
Step 6:  Stop
```

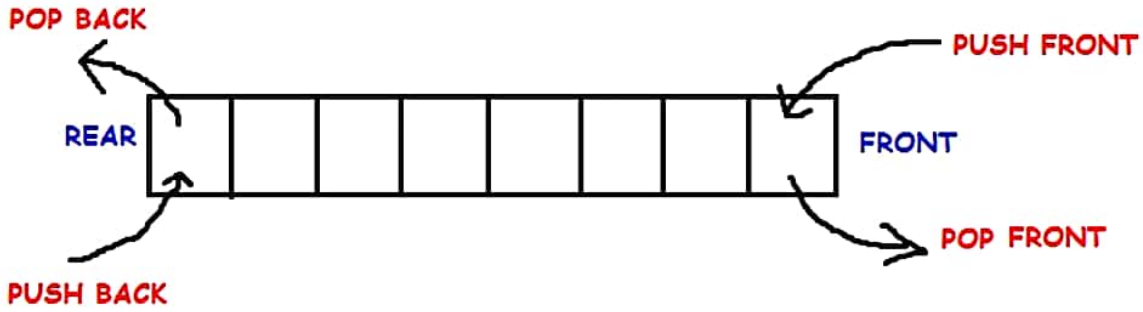
Display

```
CQ display(CQ, max, item, front, rear)
Step 1: Start
Step2:  If (front==-1 and rear==-1)
            Print ("no element to display")
Step3:  else
        step 3.1: if (rear >= front)
                    repeat (for i=front to rear by +1)
                display CQ[i]
        step 3.2.1: else
                repeat (for i=front to max-1 by+1)
                display CQ[i]
        step 3.2.2: repeat (for i=0 to rear by +1)
                display CQ[i]
Step 5:  Stop
```

Double Ended Queue

Double ended queue is a more generalized form of queue data structure which allows insertion and removal of elements from both the ends, i.e , front and back.

Thus, it does not follow FIFO rule (First In First Out).



There are two variants of a **double-ended queue.** They include:

Input restricted deque: In this dequeue,insertions can be done only at one of the ends, while deletions can be done from both ends.

Output restricted deque: In this dequeue,deletions can be done only at one of the ends, while insertions can be done on both ends.

There are four basic operations in usage of Deque that we will explore:

1. **Insertion at rear end**
2. **Insertion at front end**
3. **Deletion at front end**
4. **Deletion at rear end**

Algorithm for Insertion at rear end

Step1: If (rear==max-1) then

Print "overflow"

Step2: Else

if(rear==-1 and front==-1)

set front=0

rear = 0

Step 3: Else

rear=rear+1

[End of if]

Step 4: DQ[rear]=item

Step 5: Stop

Algorithm for Insertion at front end

Step-1 :Start

Step-2: If (front==0 and rear==max-1) then

print "overflow"

Step-3 : else

front = front-1

DQ[front]=item;

[End of if]

Step-3 :Stop

Algorithm for Deletion from front end

Step1: Start

Step2: If (rear==-1 and front==-1)

Print "underflow"

End of if, exit.

Step3: item = Q[Front]

print "The deleted item is" item

Step 4: If (rear==front)

set rear =-1

front=-1

Step 5: else

front=front+1

[End of if]

Step 6: Stop

Algorithm for Deletion from rear end

Step-1 : Start

Step-2: If (front==-1 and rear==-1) then

print "underflow"

Step 3: item = DQ[rear]

print "the deleted item is",item

Step 4: if (rear ==front)

set rear=-1

front=-1

Step 5: else

rear =rear-1

[End of if]

Step 6: Stop

priority queue

A priority queue is a special type of queue in which each element is associated with a priority and is served according to its priority. If elements with the same priority occur, they are served according to their order in the queue.

Generally, the value of the element itself is considered for assigning the priority.

For example, The element with the highest value is considered as the highest priority element. However, in other cases, we can assume the element with the lowest value as the highest priority element. In other cases, we can set priorities according to our needs.