DS – UNIT - 2

STACK

Definition

A stack is an ordered collection of homogeneous data elements where the insertion and deletion operation take place at one end only.

In short, it is a data structure in which insertion and deletion of elements occur only at one end.

Characteristics of a Stack

A stack data structure has following characteristics:

- It is non-primitive linear data structure.
- Its nature is LIFO (Last In First Out).
- The insertion and deletion operation occur only at one end.
- It is linear data structure of variable size.
- The elements are removed from the opposite order from that in which they are added to the stack.

Operations on Stack

They are:

1. Push
2. Pop
3. Peep
4. Change

Push	An insertion operation is known as Push. To insert an element on a stack.
Pop	A deletion operation is known as Pop. To delete an element from a stack.
Peep	To give the value of i^{th} element from a stack.
Change	To change the value of i^{th} element from a stack.

Top and Bottom of a stack

- The most accessible element of a stack is known as Top of a stack.
- The least accessible element of a stack is known as Bottom of a stack.

DS – UNIT - 2

Representation of a Stack

A stack may be mainly represented in the memory by two ways

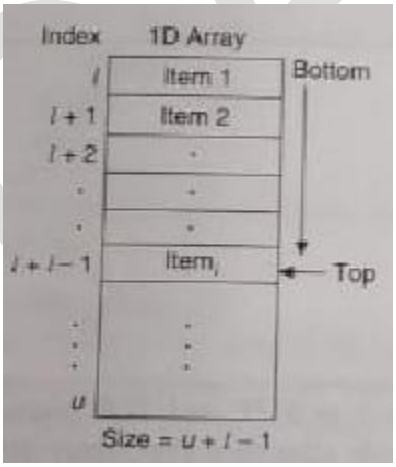
1. One dimensional array
2. Singly linked list

Array representation of Stack

It is very easy and convenient method to stored data of stack.

First we have to allocate a memory block of sufficient size to accommodate the full capacity of the stack. Then starting from the first location of the memory block, the items of the stack can be stored in a sequential fashion.

The following figure shows the array representation of a stack.



In above figure,

- ITEM$_i$ denotes the ith item in the stack,
- l and u denote the index range of the array whose values are 1 and SIZE respectively.
- TOP is a pointer that indicates current position of an item in array. The size of an array is u+l-1.

The Empty and full of a stack can be stated by following condition:

EMPTY : TOP < l
FULL : TOP >= u

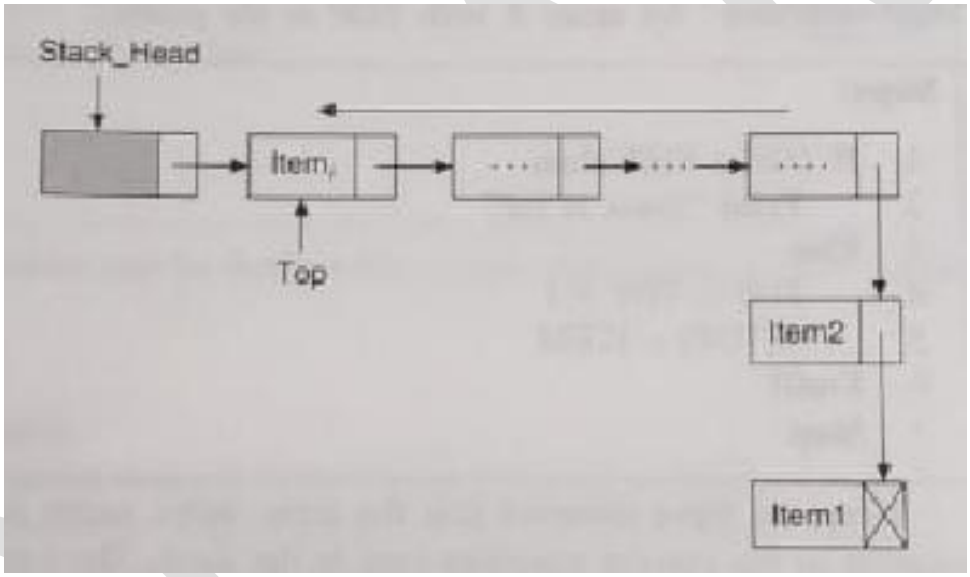
DS – UNIT - 2

Linked list representation of Stack

The limitation of stack representation using array is that it only represent fixed sized stack.

In some applications, the size of stack may vary during execution. In that case stack represented using linked list is more suitable.

In this representation, DATA field of node is for the ITEM and LINK field is point to the next ITEM. The following figure represents link representation of a Stack.



In the linked list representation, the first node on the list is the current item that is the item at the top of the stack and the last node is the node containing the bottom of the stack.

Thus, the PUSH operation will add a new node in the front and a POP operation will remove a node from the front of the list.

The SIZE of the stack is not important here because its representation allows dynamic stacks.

In this representation, whether a stack is EMPTY or not can be checked by LINK filed of Stack_Head node. Note that test of Overflow is not applicable in this case.

DS – UNIT - 2

ALGORITHM

1. An algorithm to insert an element into a stack.

PUSH (S, TOP, X, N): This procedure inserts an element X to the top of a stack which is represented by a vector S containing N elements with a pointer Top denoting the position of the top element in the stack.

Step -1 : [Check for Stack Overflow]
```
If ( TOP >= N )
Then
        Write    ('STACK    OVERFLOW')
        Return
```

Step -2 : [Increment Top]
```
TOP ←——TOP + 1
```

Step -3 : [Insert an Element]
```
S[ TOP ] ←—— X
```

Step -4 : [Finished **]**
```
Return
```

2. An algorithm to delete an element from a stack.

POP (S, TOP): This function removes the top element from a stack which is represented by a vector S and returns this element. Top is a pointer to the top element of the stack.

Step -1 : [Check for Underflow on Stack]
```
If ( TOP = 0 )
Then
        Write ('STACK UNDERFLOW ON POP')
        Take action in response to underflow
        Exit
```

Step -2 : [Decrement pointer]
```
TOP ←—— TOP -  1
```

Step -3 : [Return top Element of stack]

Return (S[TOP + 1])

3. An algorithm to give the value of i^{th} element from a stack.

PEEP (S, TOP, N, I): Given a vector S (consisting of N elements) representing a sequentially allocated stack and a pointer TOP denoting the top element of the stack, this function returns the value of the i^{th} element from the top of the stack. The element is not deleted by this function.

Step -1 : [Check for stack Underflow]
```
If ( (TOP – I + 1) <= 0 )
Then
    Write ('STACK UNDERFLOW ON PEEP')
    Take action in response to underflow
    Exit
```

Step -2 : [Return i^{th} Element from top of stack]
```
Return ( S[ TOP - I + 1] )
```

4. An algorithm to change the value of i^{th} element from a stack.

CHANGE (S, TOP, N, I, X): As before, a vector S (consisting of N elements) represents a sequentially allocated stack and a pointer TOP denotes the top element of the stack. This procedure change the value of the i^{th} element from the top of the stack to the value contained in X.

Step -1 : [Check for stack Underflow]
```
If ( (TOP – I + 1) <= 0 )
Then
    Write ('STACK UNDERFLOW ON CHANGE')
    Return
```

Step -2 : [Change i^{th} Element from top of stack]
```
S[ TOP – I + 1] ←—— X
```

Step -3 : [Finished]
```
Return
```

APPLICATIONS OF THE STACK

Infix, Postfix, Prefix Notation

DS – UNIT - 2

There are basically three types of notation for an expression (mathematical expression ; An expression is defined as a number of operands or data items combined using several operators.):

- Infix notation
- Prefix notation
- Postfix notation **INFIX**

The infix notation is what we come across in our general mathematics, where the operator is written in-between the operands.

For example: The expression to add two numbers A and B is written in infix notation as: A + B

Note that the operator '+' is written in-between the operands A and B. The reason why this notation is called infix, is the place of operator in the expression.

PREFIX

The prefix notation a notation in which the operator is written before the operands, it is also called **polish notation** in the honor of the mathematician Jan Lukasiewicz who developed this notation.

The same expression when written in prefix notation looks like: + AB

As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

POSTFIX

In the postfix notation the operators are written after the operands, so it is called the postfix notation (post means after), it is also known as **suffix notation or reverse polish notation**.

The above expression if written in postfix expression looks like follows: AB +

The prefix and postfix notations are not really as efficient to use as they might look at first. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction: add (A, B)

Note the operator **add** (name of the function) precedes the operands A and B.

DS – UNIT - 2

Because the postfix notation is a type of notation which is most suitable for a computer to calculate any expression (due to its reversing characteristic), and is the universally accepted notation for designing arithmetic and logical (ALU) unit of the CPU (processor). Therefore it is necessary for us to study the postfix notation.

Moreover the postfix notation is the way computer looks towards any arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated.

Advantage of using postfix notation

Although human beings are quite used to work with mathematical expressions in infix notation, which is rather complex, as using this notation one has to remember a set of non-trivial rules.

That set of rules must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS, and associativity.

Using infix notation one cannot tell the order in which operators should be applied by only looking at expression.

Whenever an infix expression consists of more than one operators, the precedence rules (BODMAS) should be applied to decide that which operator (and operands associated with that operator) are evaluated first.

As compared to postfix notation, which is much more easy to work with or evaluate. In a postfix expression operands appear before the operators, there is no need for operator precedence and other rules.

As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated applying the encountered operator. Place the result back onto the stack, doing so the stack will contain finally a single value at the end of process.

Conversion: Infix to Postfix using manually and stack for parenthesis and Non-parenthesis

Notation Conversions

Let an expression A + B * C is given in, which is in infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have right result.

For example: A + B * C = 4 + 3 * 7 = 7 * 7 = 49

Is this the right result? No, this is because the multiplication is to be done before addition, because it has higher precedence over addition.

This means that for an expression to be calculated we must have the knowledge of precedence of operators. The error in the above calculation occurs, as there are no braces to define the precedence of operators. Thus expression A + B * C is interpreted as, A + (B * C).

This is an alternative for us to convey the computer that multiplication has higher precedence over addition, as there is no other way to specify this. Operator precedence

Exponential operator	^	Highest precedence
Multiplication / Division	*, /	Next, precedence
Addition / Subtraction	+, -	Least precedence

Converting infix expression to postfix form

A + B * C	Infix Form
A+ (B * C)	Parenthesized expression
A + (BC*)	Convert the multiplication
A (BC*) +	Convert the addition
ABC* +	Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to right.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in above expression B * C is parenthesized first before A + B.
3. The sub-expression (part of expression) which has been converted into postfix is to be treated as single operand.
4. Once the expression is converted to postfix form remove the parenthesis.

EXAMPLE 1 Give postfix form for A + [(B + C) + (D + E) * F] / G

SOLUTION Evaluation order is

A + [{ (BC+) + (DE+) * F } / G]
A + [{ (BC+) + (DE+ F*) } / G]
A + [{ BC+ DE+ F* + } / G]
A + (BC+ DE + F* + G /)

ABC + DE + F* + G / + Postfix Form

EXAMPLE 2 Give postfix form for A + B - C.

SOLUTION

(A + B) – C
(AB +) – C Now as said-after converting a part of expression to postfix, treat it as single unit, therefore (AB+) now can be treated as T.
T – C
TC - Now as the final conversion is accomplished replace the value of T.
AB + C - Postfix Form
3 Convert the expression A * B + C / D to postfix form.

SOLUTION

In this particular example the * operator is encountered during left to right evaluation, which shares an equal precedence to operator.

When this happens (two operators with same precedence comes in same expression), the operator that is encountered first while evaluating from left to right is first parenthesized. Hence the above expression will be interpreted as:

(A * B) + C / D

(AB *) + C / D

(T) + C / D T = AB *

(T) + (C / D)

(T) + (CD /)

T + S S = CD /

TS +

AB * CD / Postfix expression

EXAMPLE 4 Give positive form for A * B + C

SOLUTION.

(A *B) + C

(AB *) + C
(T) + C
TC +
AB * C+ Postfix expression

EXAMPLE 5 Give postfix form for A + B / C - D

SOLUTION

A + (B / C) – D

A + (BC /) – D

A + T – D T = BC /

(A + T) – D

(AT +) – D

S – D S = AT +

SD -

AT + D -

ABC / + D - Postfix expression

6 Convert the expression (A + B) / (C - D) to postfix form.

SOLUTION In this expression the brackets are already specified. Therefore the conversion looks like:

(AB+) / (CD-)
T / S Where T = (AB+) and S = (CD-)
TS /
AB+ CD- / Postfix expression

EXAMPLE 7 Give postfix form for (A + B) * C / D.

SOLUTION

(AB +) * C / D

T * C / D T = (AB+)

(T * C) / D

(TC *) / D

S / D S = (TC*)

SD /

TC * D /

AB + C * D / Postfix expression

EXAMPLE 8 Give postfix form for (A + B) * C / D + E ^ F / G

SOLUTION

(AB +) * C / D + E ^ F / G

T * C / D + (E ^ F) / G T = (AB +) and ^ has the highest priority

T * C / D + (EF ^) / G

T * C / D + S / G S = (EF ^)

(T * C) / D + S / G

(TC *) / D + S / G

Q / D + S / G Q = (TC *)

(Q / D) + S / G

(QD /) + S / G

P + S / G P = (QD /)

P + (S / G)

P + (SG /)

P + O O = (SG /)

PO + Now we will expand the expression PO +

QD / O +

TC * D / O +

AB + C * D / S G / +

AB + C * D / EF ^ G / + Postfix expression

.

9 A + [(B + C) + (D + E) * F] / G. Give postfix form.

SOLUTION

A + [(BC+) + (DE+) * F] / G
A + [T + S * F] / G — where T = (BC+) and S = (DE+)

A + [T + (S * F)] / G

A + [T + (SF*)] / G

A + [T + Q] / G — where Q = (SF*)

A + (TQ+) / G

A + P / G — where P = (TQ+)

A + (PG /)

A + N — where N = (PG/)

AN+ — Expanding the expression AN+

A PG/ +

A TQ+ G/ +

A BC+ Q+ G/ +

A BC+ SF*+ G/ +

A BC+ DE+ F* +G/+ — Postfix expression.

EXAMPLE 10 A + (B * C - (D / E ^ F) *G) *H. Give postfix form.

SOLUTION

A + (B * C - (D / (EF ^)) * G) *H — (^ has the highest priority)
A + (B * C - (D / T) * G) * H — T = EF ^
A + (B * C - (DT/) * G) * H
A + (B * C - S * G) * H — S = DT /
A + ((B * C) - S * G) *H
A + ((BC *) – S * G) * H
A + (Q – S * G) * H — Q = (BC *)
A + (Q – (S * G)) * H
A + (Q – (SG *)) * H
A + (Q – P) * H — P = (SG *)
A + (QP -) * H
A + O * H — O = (QP -)
A + (O * H)

A + (OH *)
A + N — N = (OH *)
AN + — Expanding the expression AN+
A OH * +
A QP – H * +
A BC * P – H * +
A BC* SG * - H * +
A BC* DT/ G * - H * +
A BC* D EF^ / G * - H * + — Postfix expression

11 A – B / (C * D ^ E). Give postfix form.

SOLUTION

A – B / (C * (D ^ E))
A – B / (C * (DE ^))
A – B / (C * T) — T = (DE ^)
A – B / (CT *)
A – B / S — S = (CT *)
A – (B / S)
A – (BS /)
A – Q — Q = (BS /)
AQ - — Expanding the expression AQ -
A BS/ -
A B CT* / -
A B C DE^ * / - — Postfix expression

Recursion : Definition and example

Recursion is defined as defining anything in terms of itself. Recursion is used to solve problem involving iteration, in reverse order.

Recursion is an alternative to iteration in making a function execute repeatedly.

Recursion: when a method call itself:

Classic example: the factorial function
N!=1 * 2 * 3 * …… * (N-1) * N

Recursion function:
F(n)=1 if n=0 =n*f(n-1)
Otherwise

As a C method:

DS – UNIT - 2

```
int fact (int n)
{
        If      (n==0)
                return
                1;
        else
                return n*fact (n-1);
}
```

Example - Factorial

```
#include <stdio.h>
unsigned long long int factorial(unsigned int i)
{ if (i <= 1)
    {
     return 1;
    } return i * factorial (i -
1); }

int main() { int
  i = 12;
  printf("Factorial of %d is %d\n", i, factorial(i));
  return 0;
}
```

```
Factorial of 12 is 479001600
```

Example - Fibonacci

```
#include <stdio.h> int
fibonacci(int i)
{ if (i == 0)
    {
     return 0;
    } if (i
  == 1)
    {
     return 1;
```

```
0
1
1
2
3
5
8
13
21
```

```
    }
  return fibonacci(i-1) + fibonacci(i-2);
}
int main() { int
  i;
  for (i = 0; i < 10; i++) { printf("%d\t\n",
    fibonacci(i));
  }
  return 0;
}
```

```
34
```

Queue

INTRODUCTION

A queue is logically a First-In First-Out (FIFO) type of list.

In our day to day life we come across situations where we have to wait in a line for our turn. A new customer joins the queue from the rear end whereas the ones who are done with the work leave the queue from the front end.

This means that the customers are attended in the order in which they arrive at the service center. i.e. first come first serve (FCFS) type of service). This waiting queue is a Queue. Queue means a line. The above-mentioned characteristic applies to a Queue.

Thus, a queue is a non-primitive linear data structure.

It is an ordered collection of homogeneous elements in which new elements are added at one end called the REAR end, and the existing elements are deleted from other end called the FRONT end.

This linear data structure represents a linear list and permits deletion to be performed at one end of the list and the insertion at the other. The information in such a list is processed in the same order as it was received. A normal queue can be represented by one-dimensional array in C.

An example of a queue can be found in a time-sharing computer system where many users share the system simultaneously.

Since such a system typically has a single central processing unit (called the processor) and one main memory, these resources must be shared by allowing one user's program to execute for a short time, followed by the execution of another

user's program, etc., until there is a return to the execution of the initial user's program.

The user programs that are waiting to be processed form a waiting queue. This queue may not strictly operate on a strictly first-in first-out basis, but on some complex priority scheme based on such factors as what compiler is being used, the execution time required, the number of print lines desired, etc. **The resulting queue is sometimes called a priority queue.**
Suppose that at a service center, t1 units of time is needed to provide a service to a single customer and on an average a new customer arrives at service center in t2 units of time. Then there are following possibilities that may arise:

1. If t1<t2 then the service counter will be free for some time before a new customer arrives at the service counter. Hence there will be no customer standing at a particular time in the queue, or the queue will be empty.

2. If t1>t2 then the service counter will be busy for some time even after the arrival of a new customer. Therefore the new customers have to wait for some time. This procedure when continues for some time gives rise to a queue.

3. If t1=t2 then as one customer leaves the counter other will arrive at the same instant, hence queue will be single element long.

Example (Graphical) of Addition in a 5 element Queue using Array:

Stage-1
F/R=-1

0	1	2	3	4

Stage-2
F/R=-0

5				
0	1	2	3	4

Stage-3

5	9			
0	1	2	3	4

Stage-4

F=0	R=1			

5	9	3		
0	1	2	3	4
F=0		R=2		

Stage-5

5	9	3	7	
0	1	2	3	4
F=0			R=3	

It is clear from the above example that whenever we insert an element in the queue, the value of REAR is incremented by one i.e. Rear=Rear+1

Only during the insertion of the first element in the queue, we will always increment the FRONT by one i.e. Front=Front+1. Thereafter the FRONT will not be changed during the entire addition operation.

DS – UNIT - 2

Hence two pointers namely FRONT and REAR are used to indicate the two ends of the queue. For the insertion of the next element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant.
Example (Graphical) of Deletion from a 5 element Queue using Array:

Stage-1

5	9	3	7	
0	1	2	3	4

Stage-2

F=0			R=3	

	9	3	7	
0	1	2	3	4
	F=1		R=3	

Stage-3

		3	7	
0	1	2	3	4

Stage-4

		F=2	R=3	

			7	
0	1	2	3	4
			F=3 R=3	

Stage-5
F/R=-1

0	1	2	3	4

IMPLEMENTATION OF QUEUE

Queues can be implemented in two ways:

1. Static implementation using Arrays 2. Dynamic implementation using Pointers

If Queue is implemented using Arrays, we must be sure about the exact number of elements we want to store in the Queue, because we have to declare the size of the array at design time or before the processing starts.

In this case, the beginning of the array will become the FRONT and the last location of the array will be REAR for the queue. The total number of elements will be calculated as
FRONT-REAR+1

If Queue is implemented using pointers, the main disadvantage is that a node in a linked representation occupies more memory space than a corresponding element in the array representation.

The reason is that there are at least two fields in each node, the data field and a field to store the address of next node in the linked list. Whereas in an Array, only data field is there.

It must be understood that the memory space occupied by a node in a linked representation is not exactly twice the space used by an element of array representation.
The advantage is linked representation makes an effective utilization of memory. Another advantage would be when an element is needed to be inserted / deleted from in between two elements.

In case of arrays, insertion / deletion in between two elements will require to shift the elements. Also there is no space restriction with linked representation compared to Arrays where a fixed number of elements are allowed.

Hence linked representation allows us to stress on our primary aim i.e. addition or deletion and not on the overheads related to these processes, So the amount of work required is independent of the size of the list.

For a linked representation, the addition of new node requires creating a new node, inserting in the required position by adjusting two pointers and deletion requires adjusting pointers and pointing the node to be deleted and free it.

OPERATIONS ON A QUEUE

There are two basic operations that can be performed on a Queue:

1. **Insert an element in a Queue**
2. **Delete an element from the Queue**

ALGORITHM TO INSERT AN ELEMENT IN QUEUE (using Arrays)

Procedure QINSERT(Q, F, R, N, Y)

Given F and R pointers to the front and rear elements of a queue, a queue Q consisting of N elements, and an element Y, this procedure inserts Y at the rear of the queue. Prior to the first invocation of the procedure, F and R have been set to zero.

1. [Overflow?]

   If R □ N
   then W rite (‘OVERFLOW ’) Return

2. [Increment rear pointer]
   R ← R + 1
3. [Insert element]
   Q[R] ← Y
4. [Is front pointer properly set?]
   If F = 0
   then F ← 1
   Return

ALGORITHM TO DELETE AN ELEMENT FROM THE QUEUE (using Arrays)

Procedure QDELETE(Q, F, R)

Given F and R, the pointers to the front and rear elements of a queue respectively, and the queue Q to which they correspond, this function deletes and returns the last element of the queue. Y is a temporary variable.

1. [Underflow?]
   If F = 0
   then Write (' UNDERFLOW')
   Return (0) (0 denotes an empty queue)
2. [Delete element]
   Y ← Q[F]
3. [Queue Empty?]
   If F = R
   then F ← R ← 0 else F ← F + 1 (increment front pointer)
4. [Return element]
   Return(Y)

LIMITATIONS OF SIMPLE QUEUES

There is a problem associated with a simple queue when implemented using Arrays.

When we perform the insertions and deletions multiple times, we would reach a stage where Rear will point to last location of Array. If there are no elements in the

array, due to the multiple insertions and deletions, there may not be any element in the array and Rear might be pointing to the last location.

Hence if we try to insert and element, it will show the message “Queue is Full” though it is empty.

To remove this problem, one solution will be whenever an element is deleted, shift all remaining elements to the left by one position. But if the queue is too large then it will be difficult and time consuming.

Also after multiple insertions and deletions, the front would have moved beyond the size of the array, pointing to a non-existent location. To take this pointer again pointing to the right position in the queue, we always reset / check both the pointer Front and Rear in the Deletion algorithm using Arrays above.

Also this pair of algorithms can be very wasteful of storage if the front pointer F never manages to catch up to the rear pointer. This method of performing operations on a queue should only be used when the queue is emptied at certain intervals.

The solution to above issues would be to use a Circular Queue.

CIRCULAR QUEUE

A circular queue is one in which the insertion of a new element is put at the very first location of the queue if the last location of the queue is full.

In other words if we have a queue **Q** of **n** elements, then after inserting an element last i.e. in the **n-1th** location of the array, the next element will be inserted at the very first location i.e. location with subscript 0 of the array.

It is possible to insert new elements if and only if those locations are empty.

A circular queue is one in which the first element comes just after the last element. This technique essentially allows the queue to wrap around upon reaching the end of the array.

It can be viewed as a mesh or loop of wire, in which the two ends of the wire are connected together. The circular queue is implemented by visualizing the one-dimensional array as circular queue.

A circular queue overcomes the problem of unutilized space in linear queues implemented as arrays. A circular queue also has a Front and Rear to keep the track of the elements to be deleted and inserted and therefore to maintain the unique characteristic of the queue.

DS – UNIT - 2

The following assumptions are made:

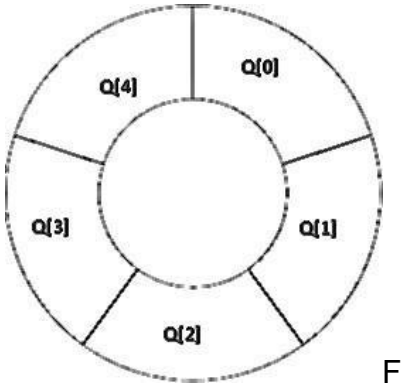


Fig. 1

1. Front will always be pointing to the first element (as in the linear queue)
2. If Front = Rear the queue will be empty
3. Each time a new element is inserted into the queue, the Rear is incremented by one. Rear=Rear+1
4. Each time an element is deleted from the queue the value of Front is incremented by one. Front=Front+1

Insertion in a Circular Queue

The insertion in a queue shown in the picture above (Fig. 1) is same as insertion in a linear queue. We only have to keep track of Front and Rear with some extra logic.

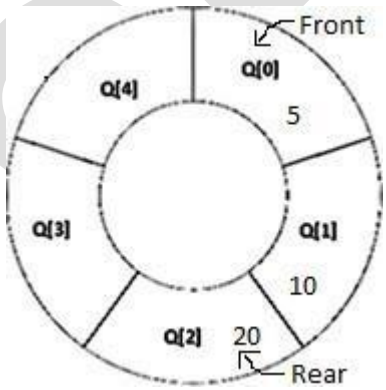


Fig. 2

A 5-element Circular Queue with 3 data (5, 10, 20) is shown in this figure (Fig. 2).
If more elements are added to this queue, it looks like the figure below (Fig. 3).
Now the queue will be full. If we now try to add another element to the queue, as the new element is inserted from the Rear end, the position of the element to be inserted will be calculated by the relation:
Rear = (Rear + 1) % MAXSIZE
Queue[Rear] = Value

DS – UNIT - 2

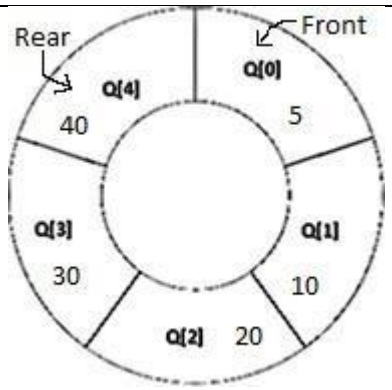


Fig. 3

For the Figure (Fig. 3), the value of Rear is 4, value of MAXSIZE is 5, hence
Rear = (Rear + 1) % MAXSIZE
Rear = (4 + 1) % 5
= 0

Note that Front is pointing to Q[0] (Front=0) and Rear also comes out to be 0 i.e. Rear is also pointing to Q[0]. Since Rear=Front, the Queue Overflow condition is satisfied, and our try to add new element will flash message "Queue Overflow".

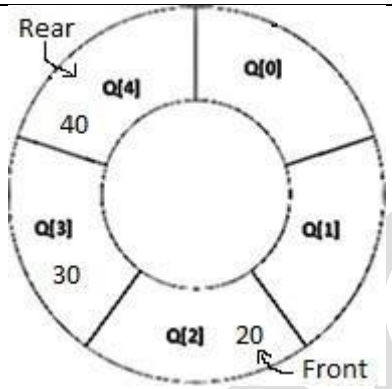


Fig. 4

In Figure (Fig. 4), Consider the situation when we add an element to the queue. The Rear is calculated as below:
Rear = (Rear + 1) % 5
Rear = (5 + 1) % 5
= 0

The new element will be added to Q[Rear] or Q[0] location of the array and Rear is increased by one i.e. Rear = Rear + 1 = 0 + 1. So the next element will be added to location Q[1] of the array.

Deletion in a Circular Queue

The deletion method for circular queue also requires some modification as compared to linear queues.

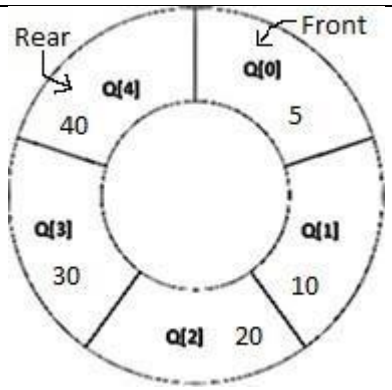


Fig. 5

In Fig. 5 the queue is full. Now if we delete one element from the queue, it will be deleted from the front end.

After deleting the front element, the front should be modified according to position of Front

i.e. if Front indicates to the last element of the circular queue then after deleting that element Front should be again reset to 0 (Front=0).

Otherwise after every deletion the new position which Front should indicate will be:
Front = (Front + 1) % MAXSIZE

ALGORITHM TO INSERT AN ELEMENT IN A CIRCULAR QUEUE (using Arrays)

Procedure CQINSERT(F, R, Q, N, Y)

Given pointers to the front and rear of a circular queue F and R, a vector Q consisting of N elements and an element Y, this procedure inserts Y at the rear of the queue. Initially F and R are set to zero.

1. [Reset rear Pointer?]
   If R = N
   then R ← 1
   else R ← R + 1
2. [Overflow?]
   If F = R
   then Write (' OVERFLOW ') Return
3. [Insert element]
   Q[R] ← Y
4. [Is front pointer properly set?]
   If F = 0

then F ← 1
Return

ALGORITHM TO DELETE AN ELEMENT FROM A CIRCULAR QUEUE (using Arrays)

Procedure CQDELETE(F, R, Q, N)

Given F and R, pointers to the front and rear of a circular queue, respectively, and a vector Q consisting of N elements, this function deletes and returns the last element of the queue. Y is a temporary variable.

1. [Underflow?]
   If F = 0
   then Write (' UNDERFLOW')
   Return (0)
2. [Delete element]
   Y ← Q[F]
3. [Queue Empty?]
   If F = R
   then F ← R ← 0 Return (Y)
4. [Increment front pointer]
   If F = N
   then F ← 1
   else F ← F + 1 Return (Y)

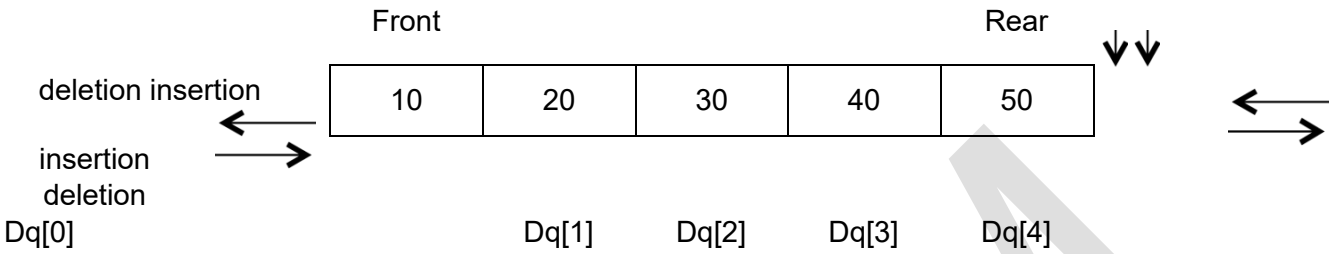
DOUBLE ENDED QUEUES (DEQUE)

It is a homogeneous linear list of elements in which insertion and deletion operations are performed from both ends i.e. we can insert elements from the rear end or from the front end. Hence it is called double-ended queue. It is commonly referred to as deque.

There are two types of deques. These two types are due to the restrictions put to perform either the insertions or deletions only at one end. They are:

DS – UNIT - 2

1. **Input-restricted deque:** It allows insertions at only one end.
2. **Output-restricted deque:** It permits deletions from only one end.



Since both insertion and deletion are performed from either end, it is necessary to design an algorithm to perform the following four operations:

1. Insertion of an element at the Rear end of the queue
2. Deletion of an element From the Front end of the queue
3. Insertion of an element at the Front end of the queue
4. Deletion of an element from the Rear end of the queue

For an input-restricted deque only the operations specified in 1, 2, 3 and 4 are valid.
For an output-restricted deque only the operations specified in 1, 2 and 3 are valid.

PRIORITY QUEUES

A priority queue is a collection of elements such that each element has been assigned a priority and the order in which elements are deleted and processed comes from the following rules:

1. An element of higher priority is processed before any element of lower priority
2. Two elements with the same priority are processed according to the order in which they were added to the queue

A queue in which we are able to insert items or remove items from any position based on some property (such as priority of the task to be processed) is often referred to as a priority queue.

A prototype of priority is processed first, and programs with the same priority form a standard Queue. Insertions in a Priority Queue need not occur at the absolute rear of the queue.

Hence it is clear that an array implementation may require moving a substantial number of data when an item is inserted at the rear of one of the higher priorities. To avoid this, you can use a linked list to great advantage when implementing a priority queue.

DS – UNIT - 2

There can be two types of implementations of priority queue:

1. Ascending Priority Queue
2. Descending Priority Queue

A collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed is called Ascending Priority Queue.

In Descending Priority Queue only the largest item is deleted. The elements of Priority Queue need not be numbers or characters that can be composed directly.

They may be complex structures that are ordered by one or several fields. Sometimes the field on which the element of a Priority Queue is ordered is not even part of the elements themselves.
The Priority Queue is a data structure in which intrinsic ordering of the elements determines the result of its basic operations.

An Ascending Priority Queue is a collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.

On the other hand a **Descending Priority Queue** allows only the largest item to be removed.

Insertion: The insertion in Priority Queues is the same as in non-priority queues.

Deletion: Deletion requires a search for the element of highest priority and deletes the element with highest priority.

The following methods can be used for deletion/removal from a given Priority Queue:

- An empty indicator replaces deleted elements
- After each deletion, elements can be moved up in the array decrementing the Rear
- The array in the queue can be maintained as an ordered circular array

APPLICATIONS OF QUEUES

1. Round robin technique for processor scheduling is implemented using queues.
2. All types of customer services (like railway ticket reservation) center software are designed using queues to store and service customers information.
3. Printer server routines are designed using queues. A number of users share a printer using printer server