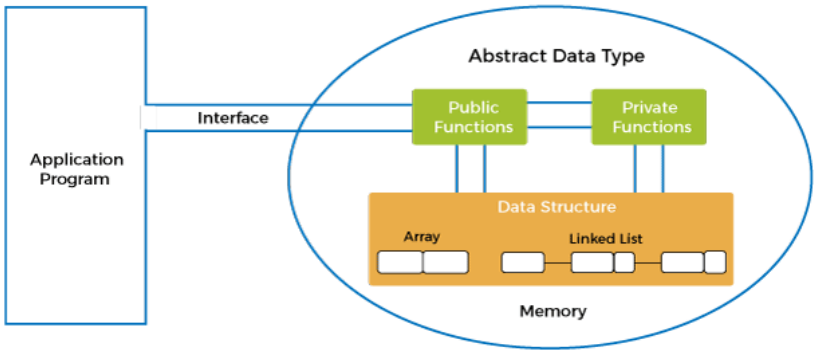
Unit-2

Abstract Data Types: Introduction, List ADT, Stack ADT, Queue ADT.
Stacks: Introduction, stack operations, applications.
Queues: Introduction, Operations on queues, circular queues, Priority queues, applications.

Abstract data type:

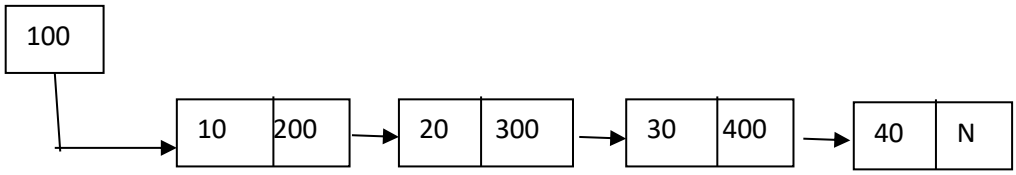
An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere. The interface does not give any specific details about something should be implemented or in what programming language.

In other words, we can say that abstract data types are the entities that are definitions of data and operations but do not have implementation details. In this case, we know the data that we are storing and the operations that can be performed on the data, but we don't know about the implementation details. The reason for not having implementation details is that every programming language has a different implementation strategy for example; a C data structure is implemented using structures while a C++ data structure is implemented using objects and classes.



List ADT:

The data is generally stored in key sequence in a list which has a head structure consisting of *address of initial node* needed to compare the data in the list.

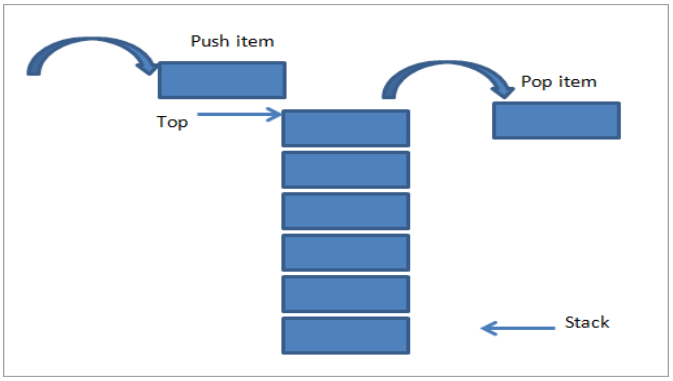


A list contains elements of the same type arranged in sequential order and following operations can be performed on the list.

- get() – Return an element from the list at any given position.
- insert() – Insert an element at any position of the list.
- remove() – Remove the first occurrence of any element from a non-empty list.
- removeAt() – Remove the element at a specified location from a non-empty list.
- replace() – Replace an element at any position by another element.
- size() – Return the number of elements in the list.
- isEmpty() – Return true if the list is empty, otherwise return false.
- isFull() – Return true if the list is full, otherwise return false.

Stack ADT

- A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.
- Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
- A stack contains elements of the same type arranged in sequential order, all operations take place at a single end that is top of the stack

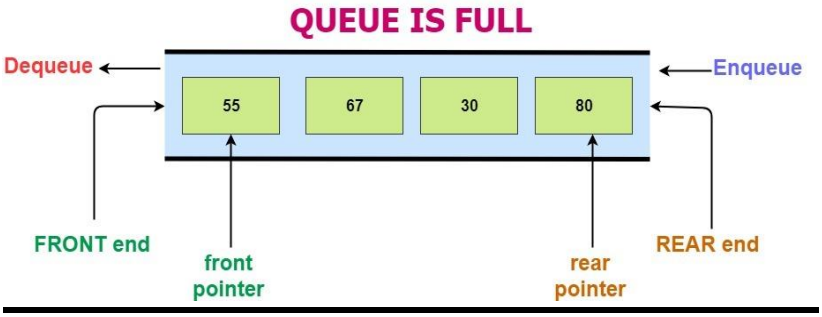


Operations can be performed:

- push() – Insert an element at one end of the stack called top.
- pop() – Remove and return the element at the top of the stack, if it is not empty.
- peek() – Return the element at the top of the stack without removing it, if the stack is not empty.
- size() – Return the number of elements in the stack.
- isEmpty() – Return true if the stack is empty, otherwise return false.
- isFull() – Return true if the stack is full, otherwise return false.

Queue ADT:

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



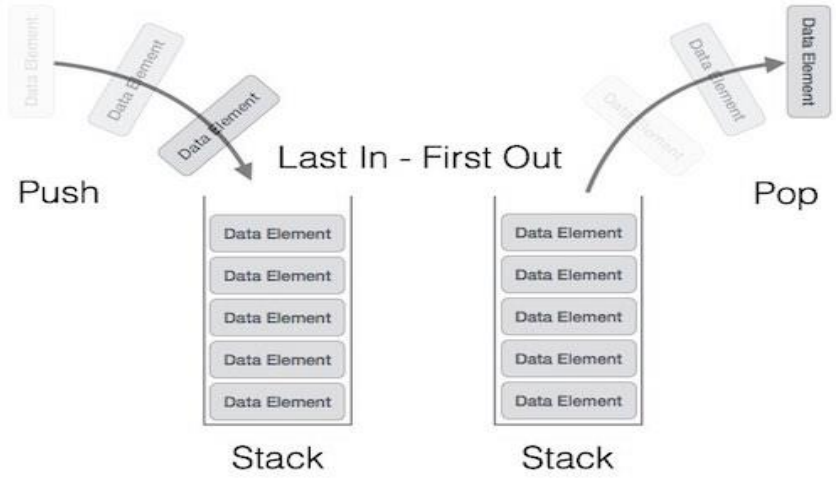
Operations to be performed:

- enqueue() – Insert an element at the end of the queue.
- dequeue() – Remove and return the first element of the queue, if the queue is not empty.
- peek() – Return the element of the queue without removing it, if the queue is not empty.
- size() – Return the number of elements in the queue.
- isEmpty() – Return true if the queue is empty, otherwise return false.
- isFull() – Return true if the queue is full, otherwise return false

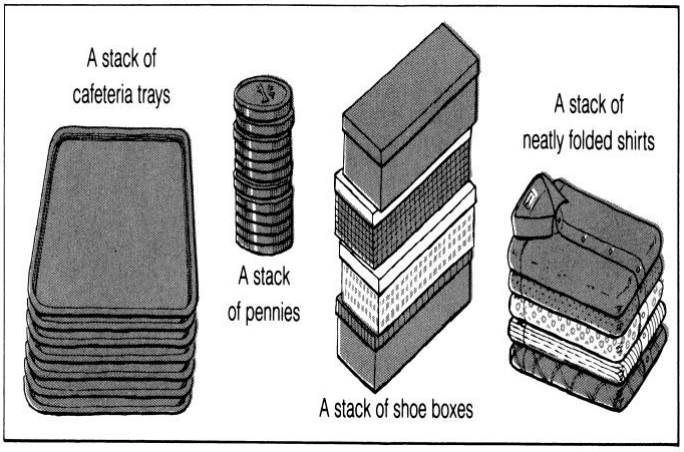
Stacks

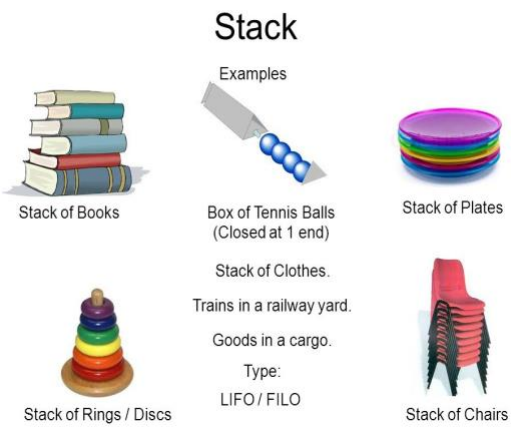
What is a stack?

- Stack is a linear data structure in which items can be inserted only from one end and get items back from the same end.
- There, the last item inserted into stack, is the first item to be taken out from the stack. This mechanism we call it as Last In First Out (LIFO)/ (FILO).



Examples of Stack:





Basic stack operations:

- **Push( )**: To insert an item from Top of stack is called push operation. The push operation change the position of Top in stack.
- **Pop( )**: To remove some item from top of the stack is the pop operation, we can pop only from top of the stack.
- **IsEmpty( )**: Stack considered empty when there is no item on Top. IsEmpty operation return true when no item in stack else false.
- **IsFull( )**: stack considered full if no other element can be inserted on top of the stack. This condition normally occur when stack implemented through array.
- **Top / peek( )** : Open end of the stack is called TOP, From this end item can be inserted.

Algorithm for Push ( ) operation:

Step 1 − Checks if the **stack** is full.
Step 2 − If the **stack** is full, produces an error and exit.
Step 3 − If the **stack** is not full, increments top to point next empty space.
Step 4 − Add data element to the **stack** location, where top is pointing.

Step 5 − Returns success.

Pseudo code for push ():

```
begin:
    if stack is full
        return "stack is full";
    else
        top = top + 1;
        stack[top] = data ;
end procedure
```

Algorithm for POP operation:

Step 1 − Checks if the stack is empty.
Step 2 − If the stack is empty, produces an error and exit.
Step 3 − If the stack is not empty, accesses the data element at which top is pointing.
Step 4 − Decreases the value of top by 1.
Step 5 − Returns success.

Procedure pop:

```
begin:
    if stack is empty
        return "stack is empty";
    else
        data = stack[top];
        top = top-1 ;
        return data
end procedure
```

Pocedure for isFull():

```
begin:
    if top == MAXSIZE-1
        return "true"
    else
        return "false"
end procedure
```

Procedure for isEmpty()

```
begin:
    if top<=1
        return true
    else
        return false
end procedure
```

Procedure for peek()

```
begin:
    return stack[top]
end procedure
```

Algorithm for Display() operation:

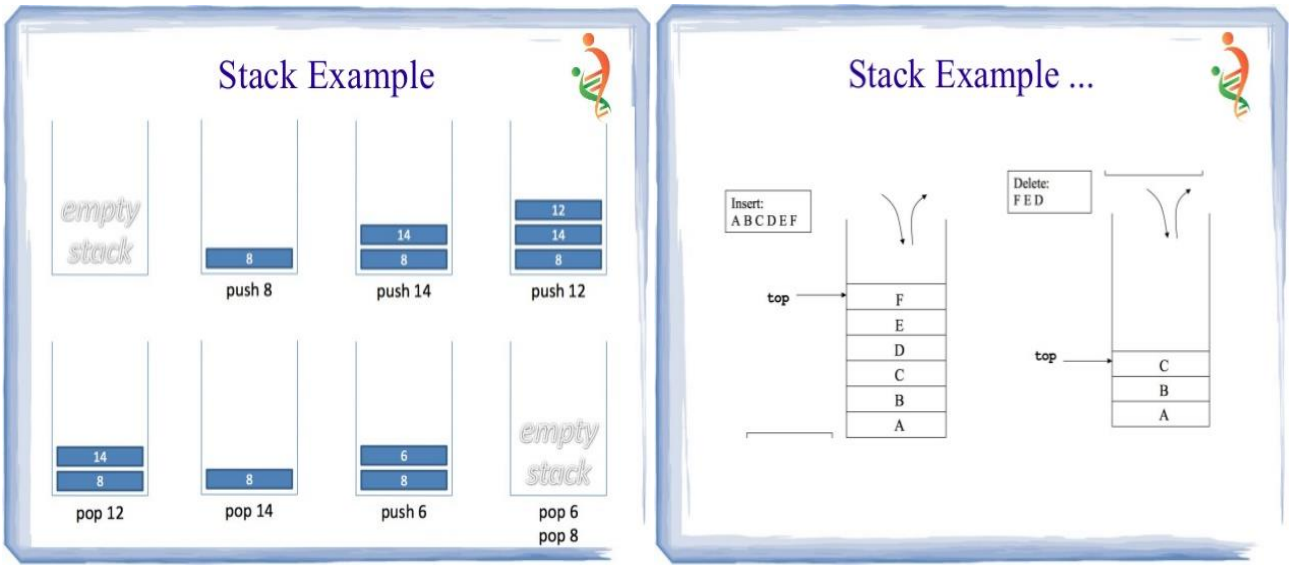
Step1- Check whether stack is empty.

Step2- If it is empty, then display "stack is empty" and terminate the function.

Step3- if it is not empty, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).

Step4- Repeat above step until 'i' value becomes '0'.

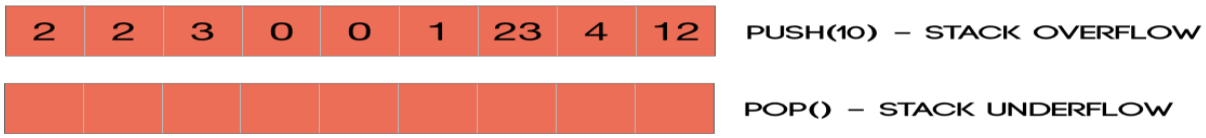
Stack Example:



Errors in stack:

We also handle two errors with a stack. They are:

- **stack underflow:** When we try to pop an element from an empty stack, it is said that the stack underflowed.
- **stack overflow:** However, if the number of elements exceeds the stated size of a stack, the stack is said to be overflowed.



Stack implementation:

Stack can be implemented in two ways. They are as follows...

1. **Using Array:** When a stack is implemented using an array, that stack can organize an only limited number of elements. The operations we perform on arrays are create(), push(), pop() and display().
2. **Using Linked List:** When a stack is implemented using a linked list, that stack can organize an unlimited number of elements.

(i) Stack using Arrays:

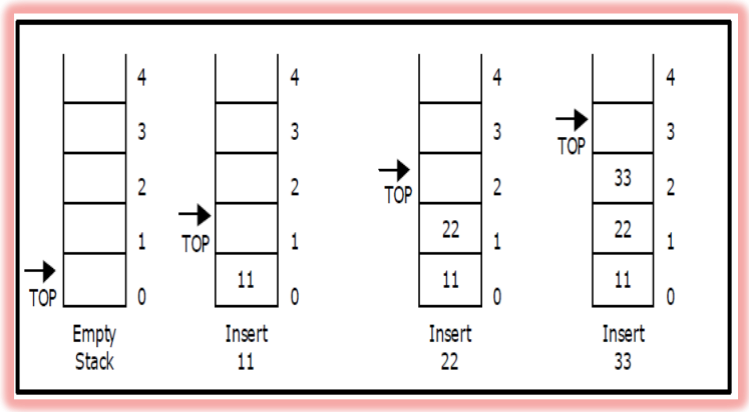
Before implementing actual operations, first follow the below steps to create an empty stack.

Create() :- Create an empty stack to insert elements or delete the elements.

Step 1 - Include all the **header files** which are used in the program and define a constant "SIZE" with specific value.
Step 2 - Declare all the **functions** used in stack implementation.
Step 3 - Create a one dimensional array with fixed size (**int stack[SIZE]**)
Step 4 - Define a integer variable **'top'** and initialize with **'-1'**. (**int top = -1**)
Step 5 - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack

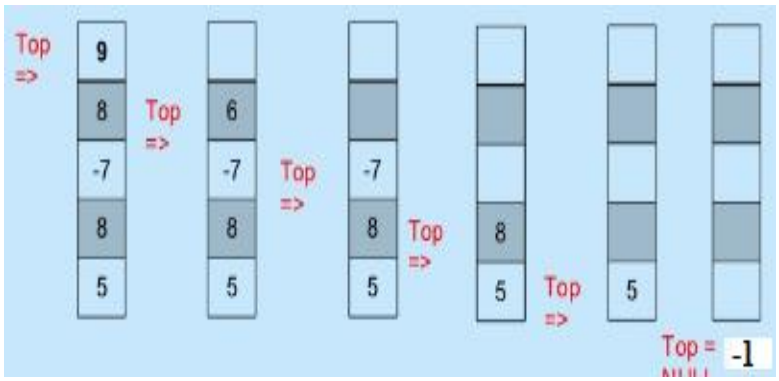
Push(value) - Inserting value into the stack

Step 1 - Check whether stack is FULL. (top == SIZE-1)
Step 2 - If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.
Step 3 - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).



Pop() - Delete a value from the Stack

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)
Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
Step 3 - If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).



Display() - Displays the elements of a Stack

Step 1 - Check whether **stack** is **EMPTY**. (**top == -1**)
Step 2 - If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.

Step 3 - If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top.
Display **stack[i]** value and decrement **i** value by one (**i--**).

Step 3 - Repeat above step until **i** value becomes '0'.

Recursion:

What is Recursion?

- The process in which a function calls itself directly or indirectly is called "Recursion" and the corresponding function is called as "Recursive function".
- Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), In-order/Pre-order/Post-order tree traversals. DFS of graphs etc.
- The Recursion can be classified into two types

Direct Recursion: It is a simpler way as it only involves a single step of calling the original function or method or subroutine.

Indirect Recursion: On the other hand, indirect recursion involves several steps.

Example with recursion:

```
#include<stdio.h>
int factorial(int);
int main(){
  int num, res;
  printf("\n Enter any integer number:");
  scanf("%d",&num);
  res =factorial(num);
  printf("\n factorial of %d is: %d", num, res);
  return 0;
}
int factorial(int n)
{
  if(n==0)
        {
    return(1);
        }
 //Function calling itself: recursion
        else
                return(n*factorial(n-1));
}
```

Applications of stack:

1. String reversal
2. Recursions
3. Depth First search graph traversal
4. Backtracking
5. Expression conversions

Evaluation of Expressions:

What is an Expression?

An expression is a collection of operators and operands that represents a specific value.

- Operator is a symbol which performs a particular task like arithmetic operation or logical operation or conditional operation etc.,
- Operands are the values on which the operators can perform the task.

Expression Types:

Based on the operator position, expressions are divided into 3 types. They are as follows...

- Infix Expression: Operator is used in between the operands.
  Syntax: operand1 operator operand2
  Example: a + b
- Postfix Expression: Operator is used after operands. We can say that "Operator follows the Operands".
  Syntax: operand1 operand2 operator
  Example: a b +
- Prefix Expression: Operator is used before operands. We can say that "Operands follows the Operator".
  Syntax: operator operand1 operand2
  Example: + a b

Conversions:

Every expression can be represented using infix, prefix and postfix notations. And we can convert an expression from one form to another form like

- **Infix to Postfix**,
- **Infix to prefix**
- **Postfix evaluation,**
- **Prefix evaluation**

Infix to postfix conversion:

Algorithm:

1. Print operands as they arrive.
2. If the stack is empty or contains a '(' on the top, push the incoming operator on to the stack.
3. If incoming symbol is '(', push it on the stack.
4. If the incoming symbol is a ')', pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Let us consider an infix expression: **D = A + B * C**

CHARACTER	STACK	POSTFIX EXPRESSION
D	Empty	D
=	=	D
A	=	D A
+	=+	D A
B	=+	D A B
*	=+*	D A B
C	=+*	D A B C
	Empty	D A B C * + =

(ii) (A + B) * (E – D)

CHARACTER	STACK	POSTFIX EXPRESSION
Initially	Stack is empty	Empty
(	(	Empty
A	(	A
+	(+	A
B	(+	A B
)	~~(+)~~	A B +
*	*	A B +
(	* (	A B +
E	* (	A B + E
-	* (-	A B + E
D	* (-	A B + E D
)	* ~~(-)~~	A B + E D -
	Stack is Empty	A B + E D - *

(iii) **X ^ Y / (5 * Z) + 2**

CHARACTER	STACK	POSTFIX EXPRESSION
Initially	Stack is empty	Empty
X	Stack is empty	X
^	^	X
Y	^	X Y
/	/	X Y ^
(	/ (	X Y ^
5	/ (	X Y ^ 5
*	/ (*	X Y ^ 5
Z	/ (*	X Y ^ 5 Z
)	/ ~~(*~~)	X Y ^ 5 Z *
+	+	X Y ^ 5 Z * /
2	+	X Y ^ 5 Z * / 2
	Stack is Empty	X Y ^ 5 Z * / 2 +

Postfix evaluation:

Algorithm:

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

- Step 1: Read all the symbols one by one from left to right in the given postfix expression.
- Step 2: If the reading symbol is "operand" then push it onto the stack
- Step 3: if the reading symbol is "operator" then perform two pop operations and store the two popped operands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 then push result to stack again.
- Step 4: Finally! perform a pop operation and display the popped value as final result.

Consider an example : (5 + 3) * (8 – 2)

CHARACTER	STACK	POSTFIX EXPRESSION
Initially	Stack is empty	Empty
(	(	Empty
5	(	5
+	(+	5
3	(+	5 3
)	~~(+)~~	5 3 +
*****	*	5 3 +
(	* (	5 3 +
8	* (	5 3 + 8
2	* (	5 3 + 8 2
-	* (-	5 3 + 8 2
)	* ~~(-)~~	5 3 + 8 2 -
	Stack is empty	5 3 + 8 2 - *

Reading symbol	Stack Operation	Evaluated part of Expression
Initially	Stack is empty	Empty
5	Push(5) **5** | | |	nothing
3	Push (3) **5** | 3 | |	Nothing

+	opr1= pop()
	opr2= pop()
	res = opr2 + opr1
	push(res)
	8 | | |
		opr1= pop() // 5
		opr2= pop() //3
		res = 5 + 3
		push(8)
		(5+3)

8	Push(8) **8** | 8 | |	(5 + 3)
2	Push(2) **8** | 8 | 2 |	(5 + 3)

–	opr1= pop()
	opr2= pop()
	res = opr2 – opr1
	push(res)
	8 | 6 | |
		opr1= pop() // 2
		opr2= pop() //8
		res = 8-2
		push(6)
		(5+3) , (8-2)

*****	opr1= pop()
	opr2= pop()
	res = opr2 * opr1
	push(res)
	48 | | |
		opr1= pop() // 6
		opr2= pop() //8
		res = 8 * 6
		push(48)
		(5+3) *(8-2)

S **(end of expression)**	result= pop()	Display(result) **48**

Let us consider postfix expression: 10 2 8 * + 3 –

Reading symbol	Stack Operation	Evaluated part of Expression
Initially	Stack is empty	Empty
10	Push(10) **10** | | |	nothing
2	Push (2) **10** | 2 | |	Nothing
8	push(8) **10** | 2 | **8** |	Nothing

*	opr1= pop()
	opr2= pop()
	res = opr2 * opr1
	push(res)
	10 | 16 | |
		opr1= pop() // 8
		opr2= pop() //2
		res = 2 * 8
		push(16)
		2 * 8

+	opr1= pop()
	opr2=pop()
	res= opr2 + opr1
	Push(res)
	26 | | |
		opr1= pop() // 16
		opr2= pop() //10
		res = 10 + 16
		push(26)
		10+(2*8)

3	push(3) **26** | 3 | |	10+(2*8)

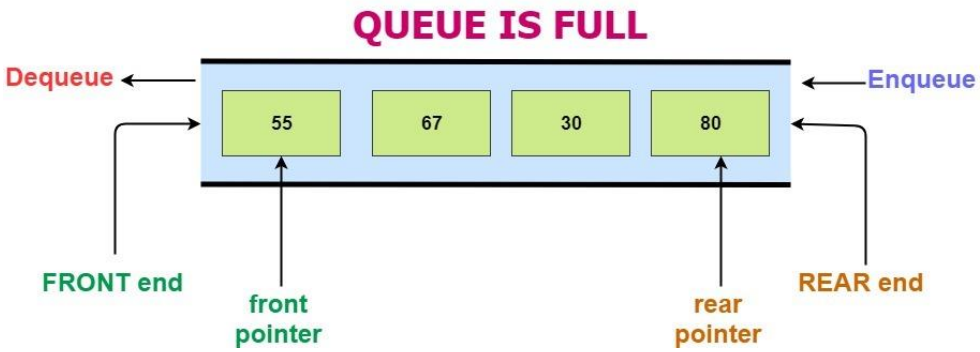
-	opr1= pop()
	opr2= pop()
	res = opr2 - opr1
	push(res)
	23 | | |
		opr1= pop() // 3
		opr2= pop() //26
		res = 26 - 3
		push(23)
		10+(2*8)-3

$ **(end of expression)**	result= pop()	Display(result) **23**

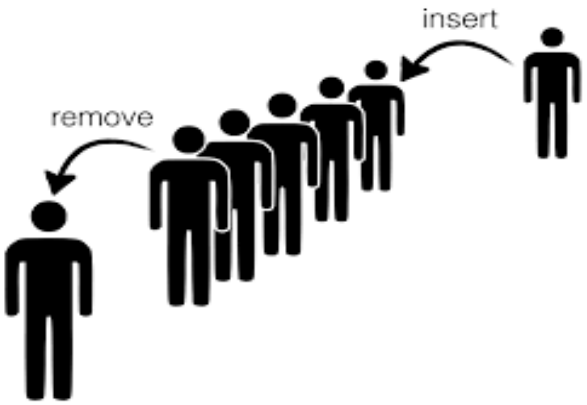
QUEUES

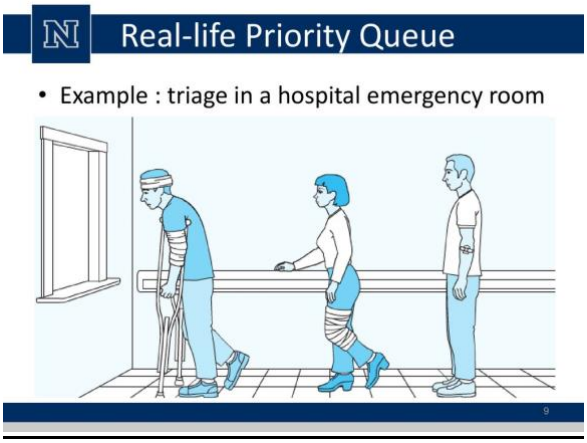
What is a Queue?

A Queue is another special kind of list, where items are inserted at one end called the 'Rear' and deleted at the other end called the 'Front'. Another name for a queue is a "FIFO" or "First-in-first-out" list.



Examples of Queue:





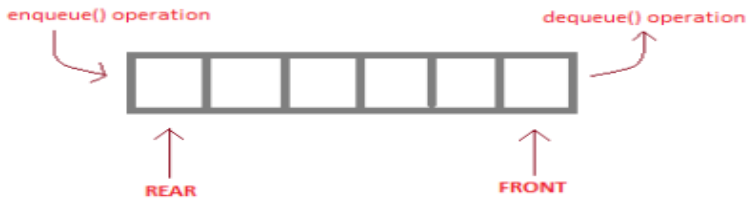
Basic Queue operations:

- **Enqueue()**: Add an element to the end of the queue is called Enqueue operation.
- **Dequeue()**: To remove some elements from the front of the queue is called Dequeue operation.
- **IsEmpty()**: Queue considered empty when there is no elements. IsEmpty operation return true when no elements in queue otherwise returns false.
- **IsFull()**: Queue considered full if no other element can be inserted at rear position. This condition normally occur when queue is implemented through array.
- **Peek()**: Get the value of the front of the queue without removing it.

Queue implementation:

Queue can be implemented in two ways. They are as follows...

1. **Using Array:** When a queue is implemented using an array, that queue can organize only limited number of elements. The operations we can perform like create(), enqueue(), dequeue() and display().



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

2. **Using Linked List:** When a queue is implemented using a linked list, that queue can organize an unlimited number of elements.

(i) Operations on Queue using Array:

Algorithm for create() an empty queue:

Before we implement actual operations, first follow the below steps to create an empty queue.

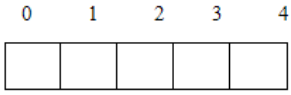
- **Step 1 -** Include all the **header files** which are used in the program and define a constant '**SIZE**' with specific value.
- **Step 2 -** Declare all the **user defined functions** which are used in queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4 -** Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**)
- **Step 5 -** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

Algorithm for Enqueue() operation:

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

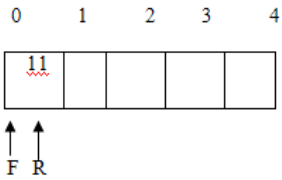
Example for queue:

Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.

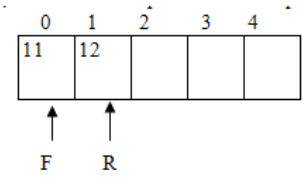


Front=Rear=-1

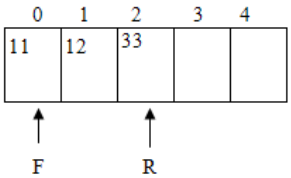
- Now, insert 11 to the queue. Then queue status will be:



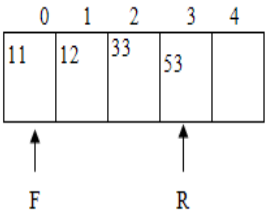
- Next, insert 12 to the queue. Then the queue status is:



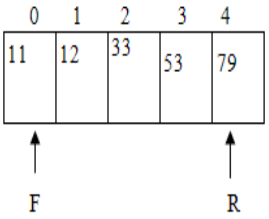
- Again insert another element 33 to the queue. The status of the queue is:



- Again insert another element 53 to the queue. The status of the queue is:



- Again insert another element 79 to the queue. The status of the queue is:



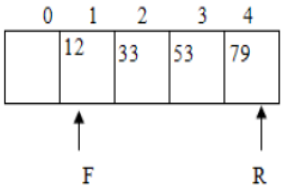
- If you want to insert another element, you cannot insert the element as queue's REAR reaches to maxsize-1, this is overflow condition.

Algorithm for Dequeue() operation:

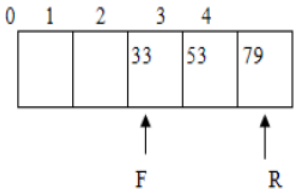
- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then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

DEQUEUE:

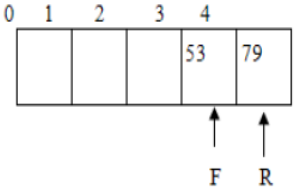
Now, delete an element. The element deleted is the element at the front of the queue. So the status of the queue is:



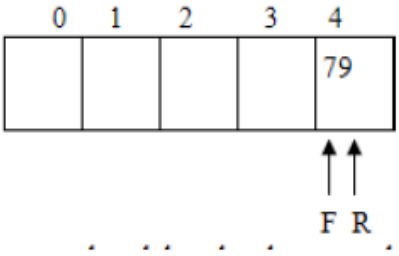
- After deleting second element ,the queue is,



- After deleting third element ,the queue is,

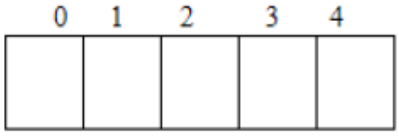


- After deleting fourth element ,the queue is,



When front=rear, then delete the element and set FRONT=REAR=-1.

- After deleting fifth element ,the queue is,



Algorithm for peek() operation:

- **Step1 :** Check the queue is empty (front==rear) if empty return queue is empty
- **Step2:** if queue is not empty then return queue[front]
-

Algorithm for isEmpty() operation:

- **Step1 :** if front < 0 || front > rear then queue is empty
- **Step2:** otherwise queue is not empty

Algorithm for isFull() operation:

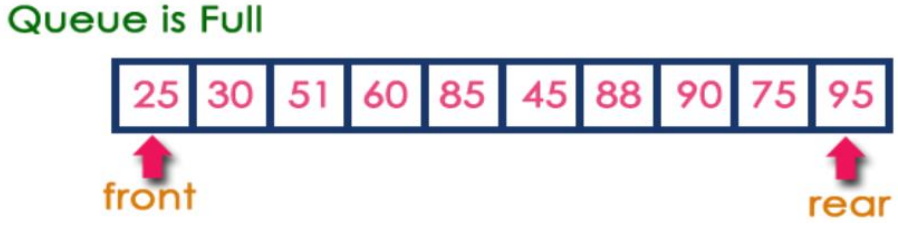
- **Step1 :** if rear == mazxsize-1 then return queue is full
- **Step2:** otherwise queue is not full

Algorithm for Display() operation:

- **Step 1** - Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2** - If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3** - If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.
- **Step 4** - Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

Circular Queues

In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example, consider the queue below...



Now consider the following situation after deleting three elements from the queue...

This situation also says that Queue is Full and we cannot insert the new element because '**rear**' is still at last position. In the above situation, even though we have empty positions in the queue we cannot make use of them to insert the new element. This is the major problem in a normal queue data structure. To overcome this problem we use a circular queue data structure.

What is a Circular Queue?

- A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.
- It is also called **'Ring Buffer'**.

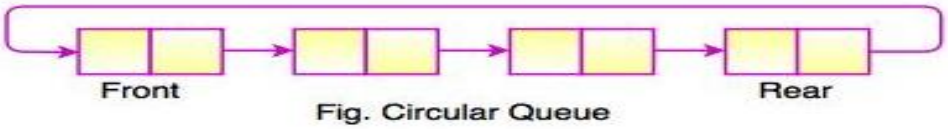


Fig. Circular Queue

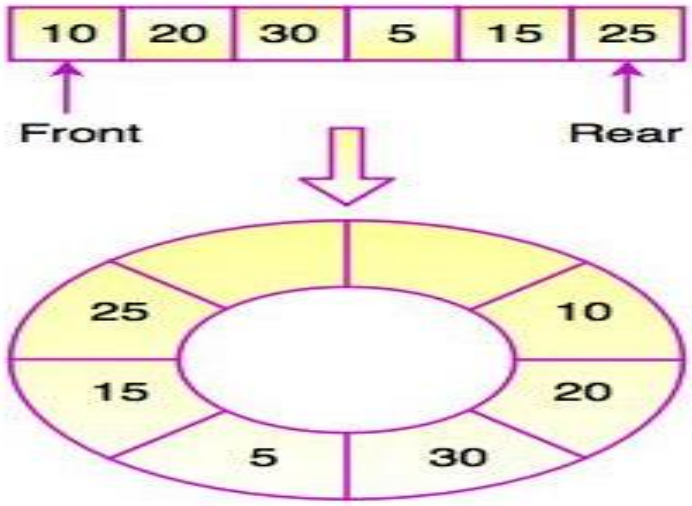


Fig. Circular Queue

Implementation of Circular Queue:

Create(): To implement a circular queue data structure using an array, we first create an empty queue.

- **Step 1 -** Include all the **header files** which are used in the program and define a constant 'SIZE' with specific value.
- **Step 2 -** Declare all **user defined functions.** .
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 4 -** Define two integer variables '**front**' and '**rear**' and initialize both with '**-1**'. (**int front = -1, rear = -1**)
- **Step 5 -** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

- In a circular queue, enQueue() is a function which is used to insert an element into the circular queue.
- In a circular queue, the new element is always inserted at **rear** position. The enQueue() function takes one integer value as parameter and inserts that value into the circular queue.

We can use the following steps to insert an element into the circular queue...

- **Step 1 -** Check whether queue is FULL.
  ((rear == SIZE-1 && front == 0) || (front == rear+1))
- **Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If queue is not full, **f**or the first element, set value of FRONT to 0.
- **Step 4 -** Circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)
- **Step 5 -** add the new element in the position pointed to by REAR.

deQueue() - Deleting a value from the Circular Queue

- In a circular queue, deQueue() is a function used to delete an element from the circular queue.
- In a circular queue, the element is always deleted from **front** position. The deQueue() function doesn't take any value as a parameter

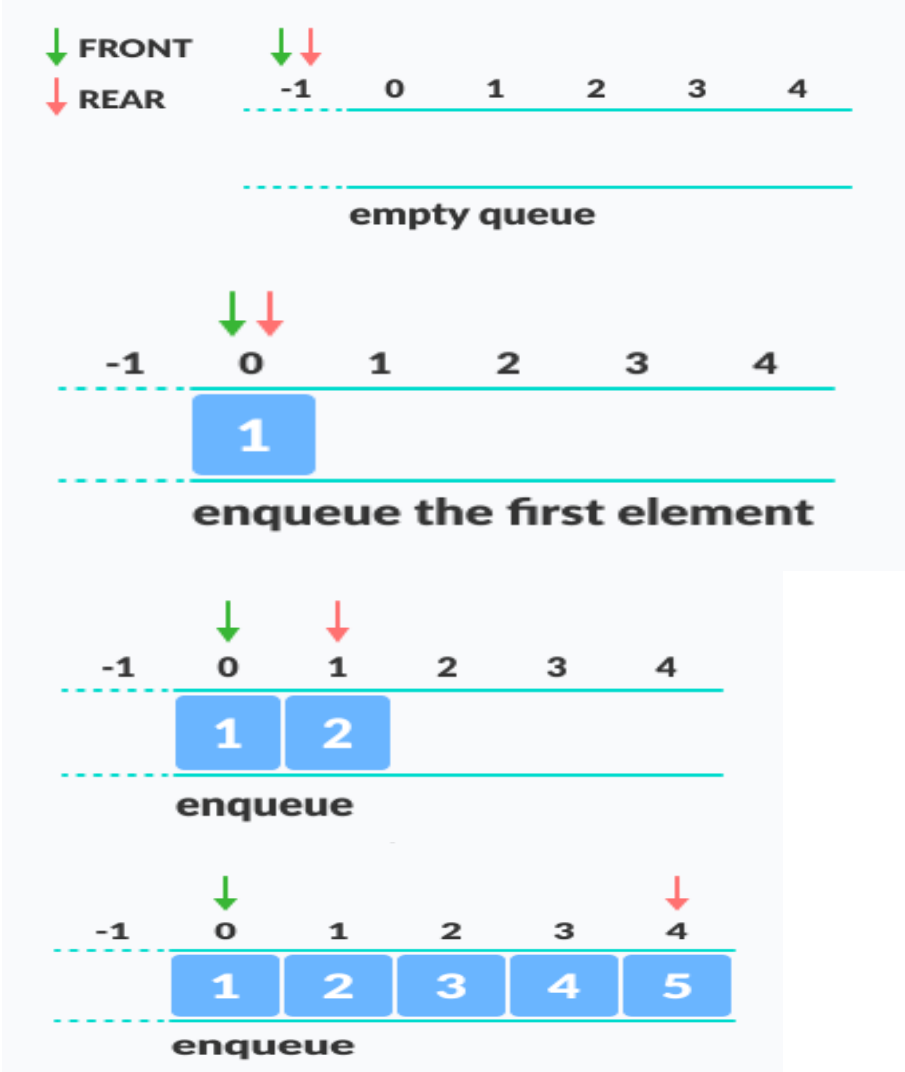
We can use the following steps to delete an element from the circular queue...

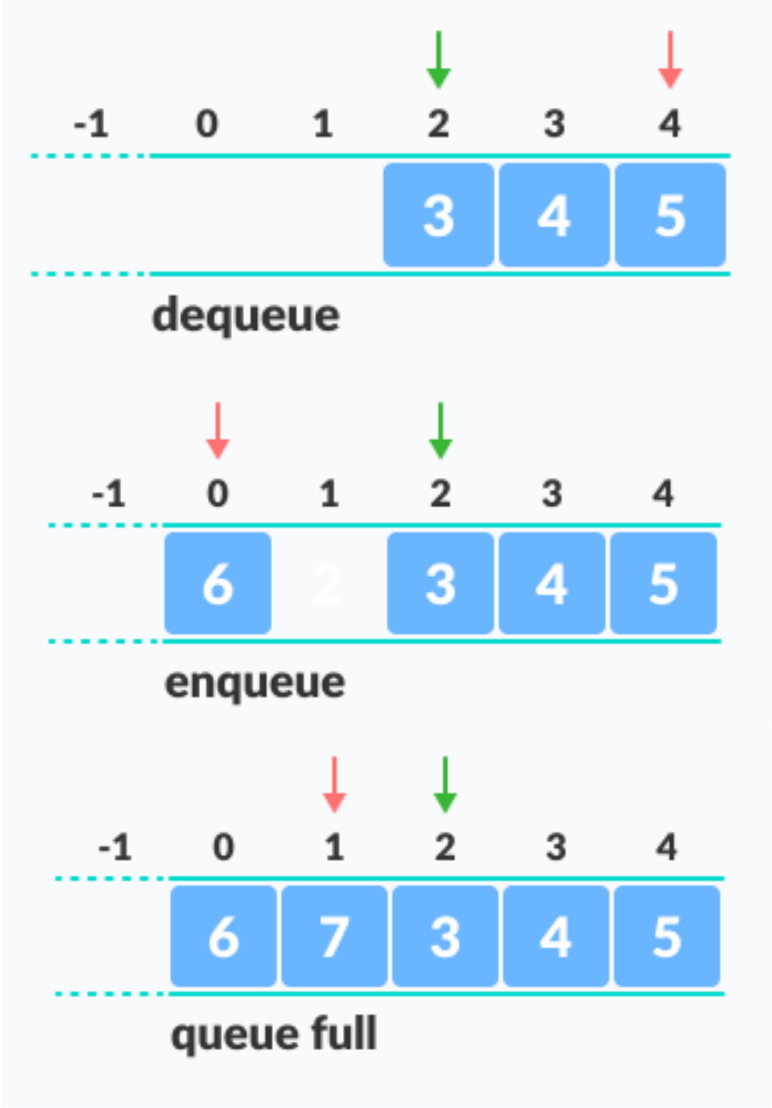
- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, return the value pointed by FRONT
- **Step 4 -** circularly increase the FRONT index by 1
- **Step 5 -** for the last element, reset the values of FRONT and REAR to -1

display() - Displays the elements of a Circular Queue

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4 -** Check whether '**front** <= **rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i** <= **rear**' becomes **FALSE**.
- **Step 5 -** If '**front** <= **rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i** <= **SIZE - 1**' becomes **FALSE**.
- **Step 6 -** Set **i** to **0**.
- **Step 7 -** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i** <= **rear**' becomes **FALSE**.

Example:


FRONT
REAR
-1
0
1
2
3
4
empty queue
-1
0
1
2
3
4
1
enqueue the first element
-1
0
1
2
3
4
1
2
enqueue
-1
0
1
2
3
4
1
2
3
4
5
enqueue



Priority queue:

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in an ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, 22 inserted in a priority queue with an ordering imposed on the values is from least to the greatest. Therefore, the 1 number would be having the highest priority while 22 will be having the lowest priority.

Operations of priority queue:

- **insert / enqueue** − add an item to the rear of the queue.
- **remove / dequeue** − remove an item from the front of the queue.
- **Peek** − get the element at front of the queue.
- **isFull** − check if queue is full.
- **isEmpty** − check if queue is empty.

Characteristics of a Priority queue:

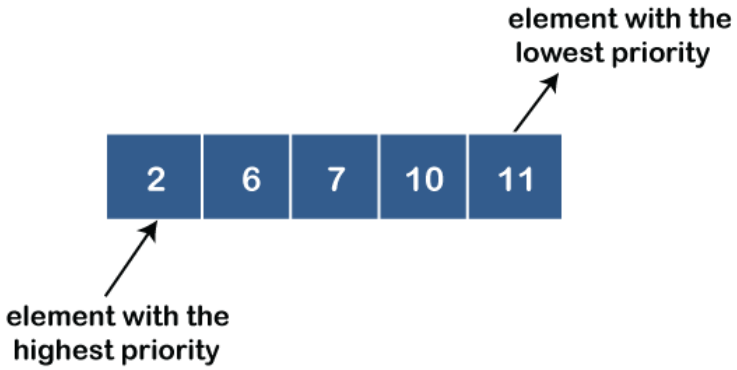
A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with the higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

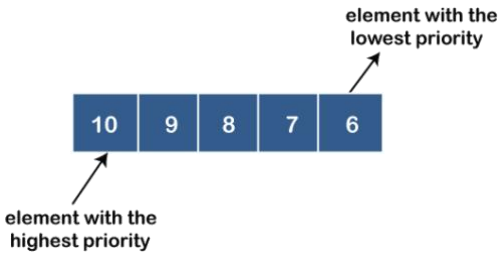
Types of Priority Queue

There are two types of priority queue:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.



- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number, i.e., 5 is given as the highest priority in a priority queue



Implementation of Priority Queue:
The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

The following are the applications of QUEUE data structure

- Serving request on a single shared resource, like a printer, CPU task scheduling.
- In real life scenario, call center phone systems uses queues to hold people calling them in an order, until a service representation is free.
- Handling of interrupt in real time systems the interrupts are handled in the same order as they arrive ie. First come first serve.

The following are the applications of Circular queue:

- Traffic light functioning is the best example for the circular queue. The colours in the traffic light follow a circular pattern.
- Memory management: circular queue is used in memory management.
- Process Scheduling: A CPU uses a queue to schedule processes.

Double Ended Queue (Dequeue)

The dequeue in data structure stands for Double Ended Queue. We can say the dequeue in data structure is a generalized version of the queue A dequeue in data structure is a linear data structure that does not follow the FIFO rule (First in first out), that is, in a dequeue in data structure, the data can be inserted and deleted from both front and rear ends. However, in a queue, we may only insert and remove data from one end. The representation of a dequeue in data structure is represented below

REPRESENTATION IN DEQUE

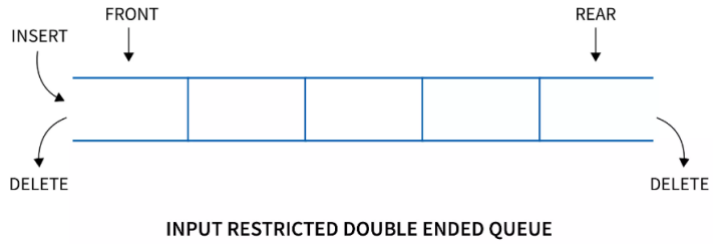
Types of Dequeue in Data Structure

Basically, there are two types of dequeue in data structure

1. Input restricted queue
2. Output restricted queue

Input restricted queue:

In an input restricted queue, the data can be inserted from only one end, while it can be deleted from both the ends.



INPUT RESTRICTED DOUBLE ENDED QUEUE

Output restricted Queue:

In an Output restricted queue, the data can be deleted from only one end, while it can be inserted from both the ends.

OUTPUT RESTRICTED DOUBLE ENDED QUEUE

Operations on Dequeue in Data structure:

In this section, we are going to implement a dequeue in data structure using the circular queue or array. So before going further let's get a brief idea about what is a circular array.

Circular array: It is an array where the last element of the array is connected to the first element. Thus, it forms a circle like structure, that makes it reusable for the empty spaces in the previous indexes caused due to deletion operations.

In a circular array, if the array is full, we again start from the beginning. However, in a linear array, if the array is full, we can't insert elements anymore. In each of the operations explained below, if the array is full, an "overflow message" will be thrown.
In a dequeue in data structure we can perform the following operations:

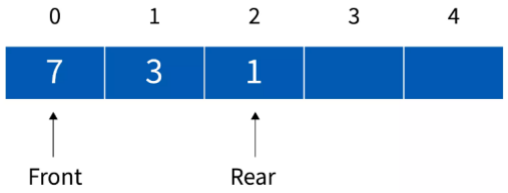
1. Insertion at front
2. Insertion at rear
3. Deletion at front
4. Deletion at rear

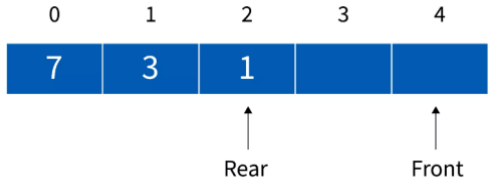
Insertion at front:

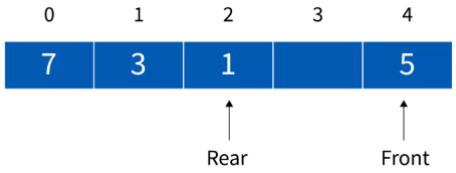
With the help of the Insertion at the front-end operation we can insert the element from the front end of the dequeue in data structure. There is a criterion before implementing the operation, at first we need to check whether the dequeue is full or not. If the dequeue is not full, then we can insert the element from the front end, by following the below conditions –

1. Initially, we will check the position of the front variable in our array

2. In case, the front variable is less than 1 (front < 1), we will reinitialize the front as the last index of the array (front = n-1).
3. Otherwise, we will decrease the front by 1.
4. Add the new key for example 5 here into our array at the index front - array[front].
5. Every time we insert a new element inside the dequeue the size increases by 1





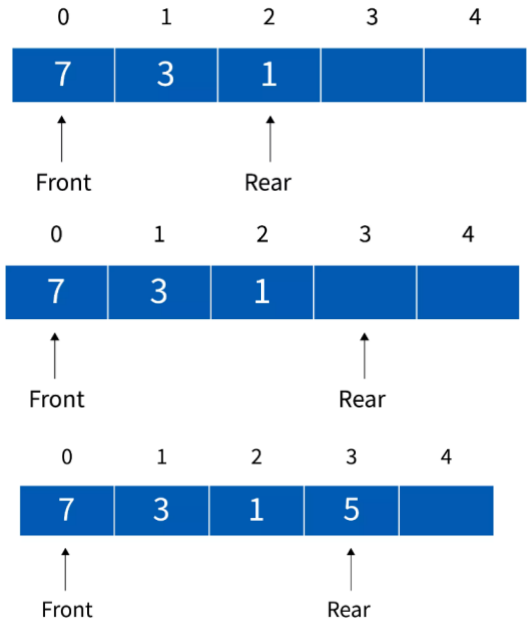



```
void insert_front(int key)
{
   if(full())
   {
      Printf( "OVERFLOW\n");
   }
   else
   {
            //If queue is empty
            if(front == -1)
                  front = rear = 0;
           //If front points to the first position element
            else if(front == 0)
                  front = SIZE-1;
            else
                  --front;

            arr[front] = key;
   }
}
```

Insertion at the rear end:

With the help of the **Insertion at the rear end** operation, we can insert the element from the **rear end** of the dequeue in data structure. We can insert the element from the rear end by following the below conditions -

1. At first, we need to check whether the dequeue in data structure is full or not.
2. If the dequeue data structure is full, we have to reinitialize the rear with 0 (rear = 0).
3. Else increase the rear by 1.
4. Add the new key for example 5 here into our array at the index rear - array[rear].
5. Every time we insert a new element inside the dequeue the size increases by 1.

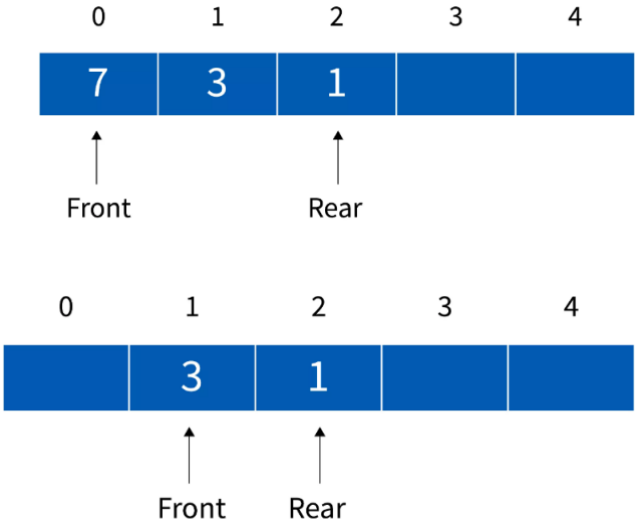



```
void insert_back(int key)
{
   if(full())
   {
      Printf( "OVERFLOW\n");
   }
   else
   {
      //If queue is empty
                  if(front == -1)
                       front = rear = 0;
                         //If rear points to the last element
                  else if(rear == SIZE-1)
                       rear = 0;

                   else
                       ++rear;
                  arr[rear] = key;
   }
}
```

Delete First Element:

With the help of the **Deletion at the front-end** operation, we can delete the element from the **front end** of the dequeue in data structure. We can delete the element from the front end by following the below conditions -

1. At first, we need to check whether the dequeue in data structure is empty or not.
2. If the dequeue data structure is empty i.e. front = -1, we cannot perform the deletion process and it will throw an error of **underflow condition**.
3. If the dequeue data structure contains only one element i.e. front = rear, set front = -1 and rear = -1.
4. Else if the front is at the last index i.e. front = n - 1, we point the front to the starting index of the dequeue data structure i.e. front = 0.
5. If none of the cases satisfy we simply increment our front by 1, front = front + 1.
6. Every time we delete any element from the dequeue data structure the size decreases by 1.

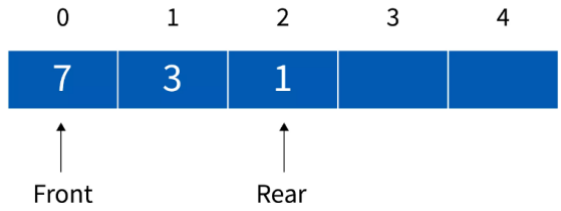


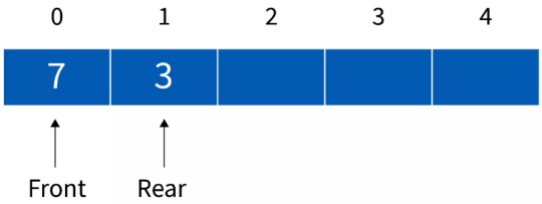

```
void delete_front()
{
   if(empty())
   {
      Printf( "UNDERFLOW\n");
   }
   else
   {
                                  //If only one element is present
      if(front == rear)
                                  front = rear = -1;

      //If front points to the last element
      else if(front == SIZE-1)
                                  front = 0;

      else
                                  ++front;
   }
}
```

Deletion at the rear end

With the help of **Deletion at the rear end** operation, we can delete the element from the **rear end** of the dequeue in data structure. We can delete the element from the rear end by following the below conditions -

4. At first, we need to check whether the deque data structure is empty or not
5. If the dequeue data structure is empty i.e. front = -1, we cannot perform the deletion process and it will throw an error of **underflow condition**.
6. If the dequeue data structure contains only one element i.e. front = rear, set front = -1 and rear = -1.
7. Else if the rear is at the starting index of the dequeue i.e. rear = 0, point the rear to the last index of the dequeue data structure i.e. rear = n-1.
8. If none of the cases satisfy we simply decrement our rear by 1, rear = rear - 1.
9. Every time we delete any element from the dequeue the size decreases by 1






```
void delete_back()
{
   if(empty())
   {
      Printf( "UNDERFLOW\n");
   }
   else
   {
                                  //If only one element is present
      if(front == rear)
                                  front = rear = -1;

      //If rear points to the first position element
      else if(rear == 0)
                                  rear = SIZE-1;

      else
                                  --rear;
   }
}
```

Check if Queue is empty:

It can be simply checked by looking where front points to. If front is still intialized with -1, the queue is empty.

```
bool is_empty()
{
   if(front == -1)
          return true;
   else
          return false;
}
```

Check if Queue is full:

Since we are using circular array, we check for following conditions as shown in code to check if queue is full.

```
bool is_full()
{
   if((front == 0 && rear == SIZE-1)  || (front == rear + 1))
      return true;
   else
      return false;
}
```