

S4 DATA STRUCTURES (DS) Code: 4133
MODULE 1 - STACK AND QUEUE

Syllabus

1.1 Introduction to different Data Structures

- 1.1.1 Explain efficiency of algorithms, complexity and big O notation.
- 1.1.2 Describe different data structures-linear and non linear
- 1.1.3 Describe basic data structure operations – insertion, deletion, search, traverse
- 1.1.4 Explain about Abstract Data Types (ADTs) and C++ classes
- 1.1.5 Explain the use of iterators
- 1.1.6 Describe the Array as an ADT with printArray() operation.

1.2 Understanding Stack and its operations

- 1.2.1 Describe Stack and its operations - Push and Pop.
- 1.2.2 Explain about array representation of stacks
- 1.2.3 Describe Stack ADT with push(), pop(), stackfull() and stackempty()
- 1.2.4 Describe infix, prefix and postfix Expressions
- 1.2.5 Explain infix to postfix conversion using Stack ADT
- 1.2.6 Explain evaluation of postfix expression using stack ADT

1.3 Understanding Queues and its operations

- 1.3.1 Describe Queue and its operations – Insert and Delete.
 - 1.3.2 Describe circular queue and its array representation
 - 1.3.3 Describe Queue ADT (for circular queue) with insert(), delete(), QEmpty() and QFull().
 - 1.3.4 Describe Priority Queue and Dequeue
-

INTRODUCTION

Data

- It is a set of values.
- Information means meaningful data and processed data.

Instruction

- Commands given to the computer to perform an operation.

Program

- Set of instruction to perform an operation.

Algorithm

- Algorithm is a step by step procedure or method for solving a problem by a computer in a finite number of steps.
- Steps of an algorithm may include branching or repetition depending upon what problem the algorithm is being developed for.

Efficiency of an algorithm

- **Algorithmic efficiency is a property of an algorithm which relates to the number of computational resources used by the algorithm.**
- For maximum efficiency, minimize resource usage.
- Examples for resources: time, space etc.

Time efficiency: A measure of amount of time for an algorithm to execute.

Space efficiency: A measure of amount of memory for an algorithm to execute.

Complexity of an algorithm

- **Complexity of an algorithm is a measure of amount of time and/or space required by an algorithm for an input of a given size(n).**

Time complexity: amount of time taken by the algorithm.

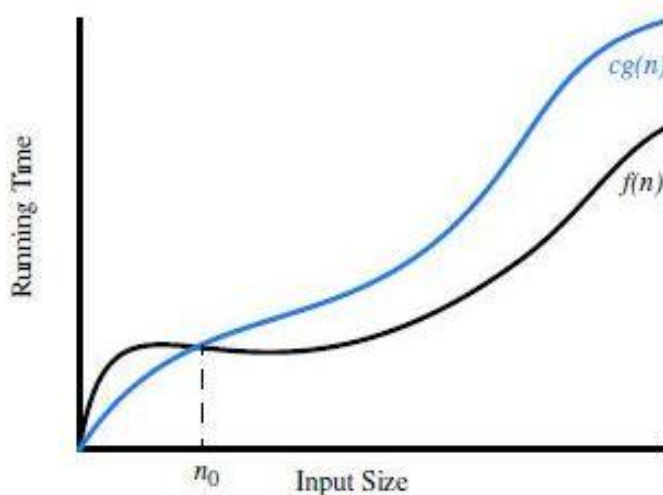
Space complexity: amount of memory taken by the algorithm.

Big O notation

- **Big O notation** is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity.
- It provides an upper bound on the growth rate of the function.
- It allows you to analyze algorithms in terms of overall efficiency and scalability.

Definition:

Let $f(n)$ and $g(n)$ be two functions, then $f(n)=O(g(n))$ or $f = O(g)$ (read “f of n is big oh of g of n” or “f is big oh of g”) if there is a positive integer C such that $f(n) \leq C * g(n)$ for all positive integers n .



Abstract data type

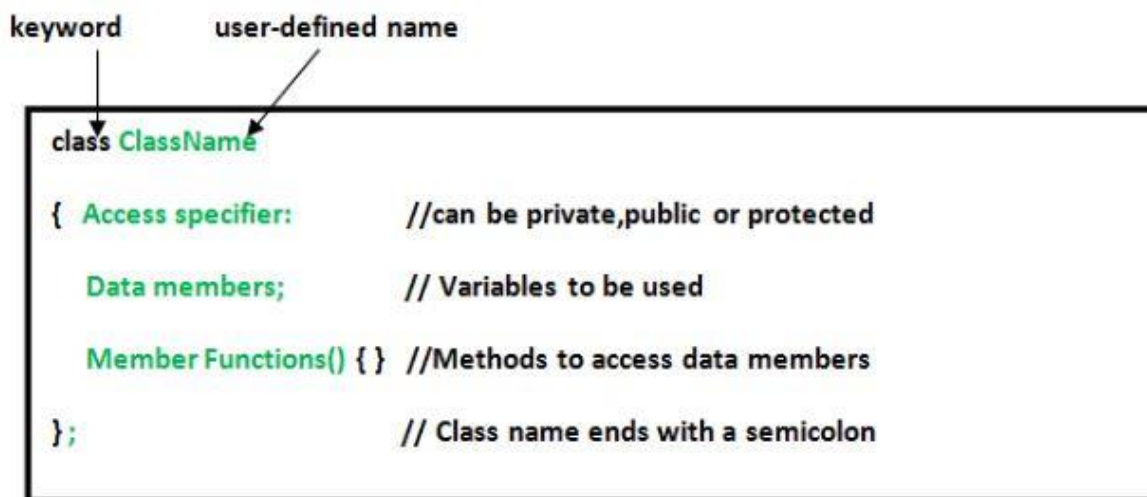
- Abstract data type is a mathematical model that contain specification of data object and the operation that can be performed on the object.

Syntax

```
Abstract datatype
{
    Instances;
}
{
    Operations;
}
```

C++ Classes and Objects

- **Class:** The building block of C++ that leads to Object Oriented programming is a **Class**.
- It is a user defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class..
- An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.



The diagram shows a C++ class definition with two annotations: 'keyword' pointing to 'class' and 'user-defined name' pointing to 'ClassName'. The class definition is enclosed in a box and contains the following code:

```
class ClassName
{
    Access specifier:           //can be private,public or protected
    Data members;               // Variables to be used
    Member Functions() { }     //Methods to access data members
};                             // Class name ends with a semicolon
```

Iterators

- ✚ **Iteration** is a process where a set of instructions or structures are repeated in a sequence a specified number of times or until a condition is met. When the first set of instructions is executed again, it is called an **iteration**.

DATA STRUCTURES (DS)

- It is an arrangement of data, either in the computer's memory or on the disk storage.
- **Examples** of data structures- Arrays, Linked lists, Queues, Stacks, binary trees and hash tables.

Application- widely applied in

- compiler design
- operating system
- statistical analysis package
- DBMS
- numerical analysis
- simulation
- artificial intelligence
- graphics

Types of data structures-linear and nonlinear

- **Linear DS**-if the **elements of a data structure are stored sequentially**, then it is a linear data structure. In linear data structure, we can traverse either forward or backward from a node. **e.g. arrays, stacks, queues and linked lists.**
- **Nonlinear DS**-if the **elements of a data structure are not stored in a sequential order**, then it is a nonlinear data structure. It branches to more than one node and cannot be traversed in a single run. **e.g. trees and graphs.**
- **The data structures used in c-**
 - 1. Arrays**- it is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index.
Arrays are declared by the following syntax
Array name[size];
Limitations- fixed size, data elements are stored in continuous memory locations which may not be always available, adding and removing of elements is problematic
 - 2. Linked lists**- in linked list, the linear relationship between the elements are represented by a link. Every node in the list points to the next node in the list. Every node contains the following information-value and a pointer or a link to the next node in the list.
 - 3. Stack**-every stack has a variable TOP associated with it. TOP is used to store the address of the topmost element of the stack.it is this position from where the element will be added or deleted.

4. **Queue**-it is FIFO DS in which the element that was inserted first is first one to be taken out. the elements in a queue are added at one end called the rear and removed from the other end called the front.
5. **Trees**-it is a collection of elements called the nodes. Every node contains a left pointer, a right pointer and a data element. Every binary tree has a root element pointed by a 'root' pointer.
6. **Graphs**-it is an abstract data structure that is used to implement the graph concept from mathematics. It is basically a collection of vertices and edges that connect these vertices.

➤ **Data structure operations**

There are 4 basic operations on any DS.

1. **Traversing**- *accessing each record exactly once* so that certain items in the record may be processed. e.g. in a graph may be traversed in BFS or DFS.
2. **Searching**- *finding the location of the record with a given key value* or finding the locations of all records which satisfy one or more conditions. Some data structure permits searching-tree, graph. But stack do not permit searching.
3. **Inserting**- *adding a new record to the new structure*. stack permits insertion at one end, but linked list permits in any position.
4. **Deleting**- *removing a record from the structure*. Queue permits from front end only. But linked list permits any position.

The following operations are used in special situations

- 1) **Sorting**- *arranging records in some logical order*
- 2) **Merging**- *combining records in two different sorted files into a single sorted file.*

Stacks

- It is a linear structure which can be implemented by either using an array or a linked list.
- The elements in a stack are added and removed only from one end, which is called the ***top***.
- So stack is called **LIFO** data structure-i.e-the element that was inserted last is taken out first.
- A stack has **3 basic operations: push, pop and peep**
 - Push-adds an element to the top of the stack.
 - Pop-removes the elements from the top of the stack.
 - Peep- returns the topmost element of the stack (without deleting it).
- **Advantage**-last in first out access
- **Disadvantage**- slow access to other elements

➤ Applications-

- 1) Reverse the order of data
- 2) Convert infix expression into postfix
- 3) Convert postfix expression into infix
- 4) Backtracking problem
- 5) System stack is used in every recursive function
- 6) Converting a decimal no. into its binary equivalent

Operations on stack

- Stack has 3 basic operations- push, pop and peep

Push operation

- ✓ It is used to insert an element into the stack
- ✓ The new element is added at the topmost position of the stack
- ✓ **Before inserting the value, we must first check if $\text{top} = \text{max}-1$, i.e. is the stack is full or not.** if the stack is already full, an overflow message is printed.

Algorithm-

Step 1: If $\text{Top} = \text{max}-1$, then print “overflow”

Step 2: set $\text{Top} = \text{Top} + 1$

Step 3: set $\text{stack}[\text{Top}] = \text{value}$

Step 4: end

10	20	30		
0	1	Top = 2	3	4 (max-1)

Here, size of stack = 5

Stackfull (Max-1) = 4

Top = 2

Stack: after insertion

10	20	30	40	
0	1	2	Top = 3	4 (max-1)

Here, size of stack = 5

Stackfull (Max-1) = 4

Top = 3

Pop operation

- ✓ It is used to delete the topmost element from the stack
- ✓ Before deleting the value, we must first check if $\text{top} = \text{NULL}$, i.e. is the stack is empty or not.
- ✓ If the stack is empty, an underflow message is printed
- ✓ If the stack is not empty, the value of the top is decrement

Algorithm-

Step 1: If $\text{top} = \text{NULL}$, then print "underflow"

Step 2: set $\text{val} = \text{stack}[\text{top}]$

Step 3: set $\text{top} = \text{top} - 1$

Step 4: end

10	20	30	40	
0	1	2	Top = 3	4 (max-1)

Here, size of stack = 5

Stackfull (Max-1) = 4

Top = 3

Stack: after deletion

10	20	30		
0	1	Top = 2	3	4 (max-1)

Here, size of stack = 5

Stackfull (Max-1) = 4

Top = 2

Peep operation

- ✓ This **returns** the value of the **topmost element of the stack without deleting it** from the stack
- ✓ If $\text{top} = \text{NULL}$, then an appropriate message is printed

Algorithm-

Step 1: if $\text{top} = \text{NULL}$, then print "stack is empty" go to step 3

Step 2: return $\text{stack}[\text{top}]$

Step 3: end

10	20	30		
0	1	Top = 2	3	4 (max-1)

Return 30

Array implementation of stack

- Stacks can be represented as a linear array.
- Every stack has a variable called **top** associated with it.
- Top is used to store the address of the topmost element** of the stack.
- There is another variable called **MAX**, which is used to store the **maximum number of elements** that the stack can hold.
- If top=NULL, then it indicates that the stack is empty and if top=MAX-1, then the stack is full.**

10	20	30		
0	1	Top = 2	3	4 (max-1)

Stack: after insertion

10	20	30	40	
0	1	2	Top = 3	4 (max-1)

Stack: after deletion

10	20	30		
0	1	Top = 2	3	4 (max-1)

Stack ADT

Abstract datatype stack

{

Instances

Linear list of elements;

Insertion and deletion takes place at one end called TOP;

}

{

Operations

stackEmpty() : return True if stack is empty, Return False otherwise

Stackfull() : : return True if stack is full, Return False otherwise.

size() : return number of elements in the stack.
Push() : insert element at the top of the stack.
Pop() : remove the topmost element from the stack.
Peep() : return the topmost element from the stack

}

Infix, postfix and prefix notation

- These notations are used to write the algebraic expressions.
- In **infix notation**, *the operator is placed in between the operands*.
For example, $A+B$; here the '+' operator is placed between two operands A and B
In computers, it is difficult to parse, because they need a lot of information to evaluate the expression. so computers work more efficiently with expressions written using prefix and postfix notations.
- In **postfix notation** (*reverse polish notation or RPN*)- **the operator is placed after the operands**.
E.g. the expression $A+B$ in infix notation can be written as $AB+$ in postfix notation. The order of evaluation of a postfix expression is always from left to right, even brackets cannot alter the order of evaluation.
- In **prefix notation** (*polish notation*), **the operators are placed before the operands**. E.g. the expression $A+B$ in infix notation can be written as $+AB$ in prefix notation.

Evaluation of an infix expression

There are two steps include in the evaluation of an infix expression

Step 1: Convert the infix expression into its equivalent postfix expression.

Step 2: Evaluate the postfix expression.

Convert the infix expression into its equivalent postfix expression

- Let 'I' be an algebraic expression written in infix notation. 'I' may contain parentheses, operands and operators.
- The precedence of the operators can be given as
Higher priority: $*,/,%$
Lower priority: $+,-$

- The order of evaluation of these operators can be changed by using parentheses.
- Algorithm: the first step is to push a left parenthesis on the stack and adding a corresponding right parentheses at the end of the infix expression.

Algorithm for infix to postfix conversion

Step 1: Add “)” to the end of the infix expression.

Step 2: push “(“ on to the stack.

Step 3: repeat until each character in the infix notation is scanned

 If a “)” is encountered , push it on the stack

 If an operand (whether a digit or an alphabet) is encountered , add it to the postfix expression

 If a “)” is encountered, then;

- a. Repeatedly pop from stack and add it to the postfix expression until a “(“ is encountered .
- b. Discard the “(“. That is , remove the “(“ from stack and do not add it to the postfix expression

 If an operator 0 is encountered, then;

- a. Repeatedly pop from the stack and add each operator (popped from the stack) to the postfix expression until it has the same precedence or a higher precedence than 0
- b. Push operator 0 to the stack

Step 4: repeatedly pop from the stack and add it to the postfix expression until the stack is empty

Step 5: exit

e.g.Convert the following infix expression into postfix expression using the algorithm.

(a) $A - (B / C + (D \% E * F) / G) * H$

(b) $A - (B / C + (D \% E * F) / G) * H$

Infix Character Scanned	Stack	Postfix Expression
	(
A	(A
-	(-	A
((- (A
B	(- (A B
/	(- (/	A B
C	(- (/	A B C
+	(- (+	A B C /
((- (+ (A B C /
D	(- (+ (A B C / D
%	(- (+ (%	A B C / D
E	(- (+ (%	A B C / D E
*	(- (+ (% *	A B C / D E
F	(- (+ (% *	A B C / D E F
)	(- (+	A B C / D E F * %
/	(- (+ /	A B C / D E F * %
G	(- (+ /	A B C / D E F * % G
)	(-	A B C / D E F * % G / +
*	(- *	A B C / D E F * % G / +
H	(- *	A B C / D E F * % G / + H
)		A B C / D E F * % G / + H * -

Table shows converting infix expression into postfix expression

Evaluate the postfix expression

- An algebraic expression written in infix notation, the computer **first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.**
- For this, **use stacks as the primary tool.**
- Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

Algorithm to evaluate a postfix expression

```

Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")" is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT

```

Example

Consider the infix expression given as $9 - ((3 * 4) + 8) / 4$. Evaluate the expression.

The infix expression $9 - ((3 * 4) + 8) / 4$ can be written as **9 3 4 * 8 + 4 / -** using postfix notation.

Character Scanned	Stack
9	9
3	9, 3
4	9, 3, 4
*	9, 12
8	9, 12, 8
+	9, 20
4	9, 20, 4
/	9, 5
-	4

Table shows evaluating the postfix expression.

Queues

- It is a data structure which stores its elements in an ordered manner
- A queue is a FIFO data structure. i.e. the element that was inserted first is the first one to be taken out
- The elements in a queue are **added at one end called the rear** and **removed from the other end called the front**
- It can be implemented by either using arrays or linked lists
- MAX specifies the maximum no. of elements in the queue
- **The queue is full, if $\text{rear} = \text{MAX} - 1$ and the queue is empty when $\text{front} = -1$ and $\text{rear} = -1$**

Advantage: provides FIFO data access

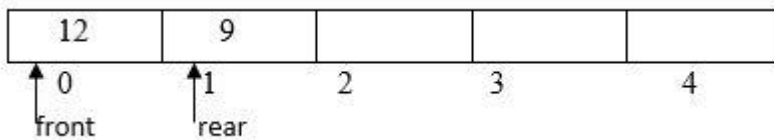
Disadvantage: slow access to other items.

Applications:

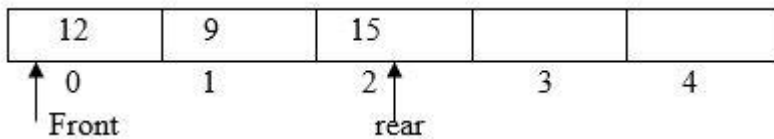
1. Priority queues
2. Graph operations
3. Tree operations
4. simulations
 - ✓ use queues in time sharing systems in which each process waits in a queue to get CPU time
 - ✓ use queues in computer networks. Data packets sent through network are put in a queue
 - ✓ operating systems often maintain a queue of processes

Array representation of queue

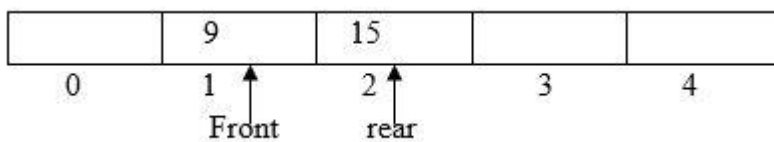
- Queues can be easily implemented using linear arrays.
- Every queue has a *front* and *rear* variables
- If we want to insert a value into the queue, the rear would be incremented by 1. then the value would be stored at the position pointed by the rear.
- If we want to delete a value from the queue, the front would be incremented by 1. then the value would be deleted at the position pointed by the front.
- Before inserting an element in the queue, we must check for overflow condition (if $\text{rear} = \text{MAX} - 1$, i.e the queue is full)
- Before deleting an element from the queue, we must check for underflow condition (if $\text{rear} = -1$, $\text{front} = -1$, i.e the queue is empty)



Queue



Queue after insertion of a new element



Queue after deletion of an element

Operations on queue

- A queue has two basic operations- **insertion and deletion**
- Apart from this , there is another operation peek-this returns the first element of the queue.

Insertion

- ✓ It **adds** an element **to the rear** of the queue.

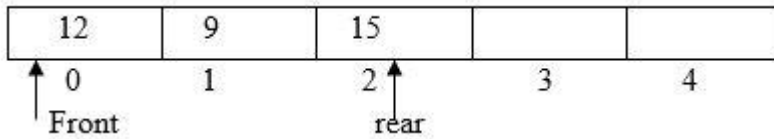
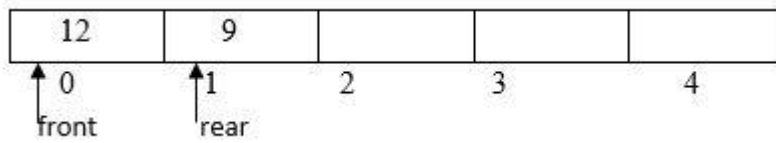
Algorithm

Step 1: IF REAR = MAX-1 Write OVERFLOW Goto step 4
[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1
SET FRONT = REAR = 0
ELSE
SET REAR = REAR + 1
[END OF IF]

Step 3: SET QUEUE[REAR] = VAL

Step 4: EXIT



Queue after insertion of a new element

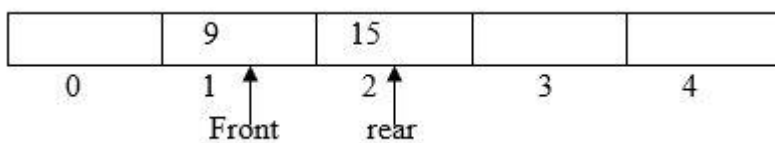
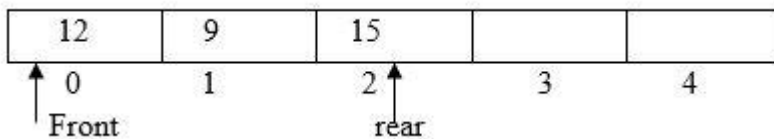
Deletion

- ✓ It **removes** the element **from the front** or the start of the queue.

Algorithm

```

Step 1: IF FRONT = -1 OR FRONT > REAR Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
  
```

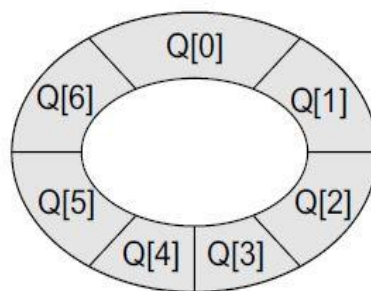


Queue after deletion of an element

Circular queues

- ❖ In a linear queue, the overflow condition will occur when $\text{rear} = \text{MAX} - 1$ and underflow condition will occur when $\text{rear} = -1$ & $\text{front} = -1$.
- ❖ There will be an another condition such as $\text{rear} = \text{max} - 1$ and $0 < \text{front} < \text{max} - 1$. In this situation, the queue is in overflow condition. At the same time there will be free spaces in the queue, because the value of front is greater than 0 and less than max-1. But in this case we cannot add new value into the linear queue. This is the drawback of a linear queue.
- ❖ This drawback can be avoided by using circular queue.

The circular queue is represented as shown below

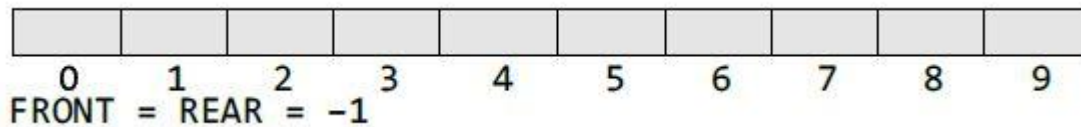


- The circular queue will be full only when $\text{front} = 0$ and $\text{rear} = \text{max} - 1$
- For insertion, we have to check the following conditions
 - ✓ If $\text{front} = 0$ and $\text{rear} = \text{MAX} - 1$, then print that circular queue is full
 - ✓ If $\text{rear} \neq \text{MAX} - 1$, then the value will be inserted and rear will be incremented
 - ✓ If $\text{front} \neq 0$ and $\text{rear} = \text{MAX} - 1$, then the queue is not full. So set $\text{rear} = 0$ and insert the new element there.

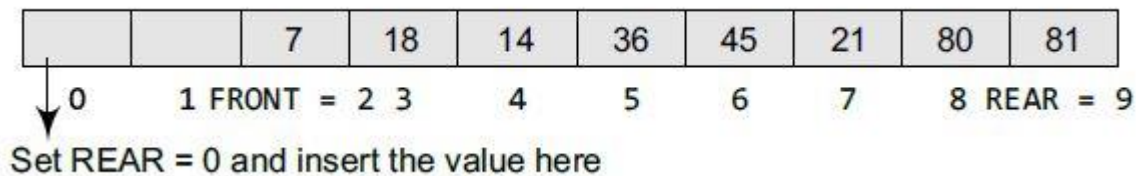
Array representation of circular queue

90	49	7	18	14	36	45	21	99	72
FRONT = 01	2	3	4	5	6	7	8	REAR = 9	

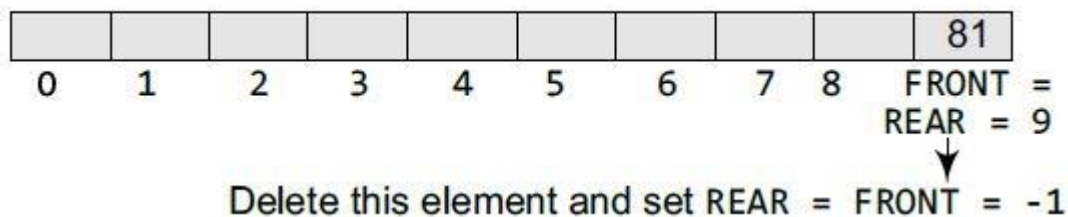
Queue Full



Empty Queue



Queue with vacant location



Queue with a single element

Algorithm to insert an element in a circular queue

Step 1: If front=0 and rear=MAX-1, then Write “overflow”

Else if front= -1 and rear= -1, then

Set front=rear=0

Else if rear=MAX-1 and front != 0

Set rear=0

Else

Set rear=rear+1

Step 2: set queue[rear]=val

Step 3: exit

90	49	7	18	14	36	45	21	99	72
0	1	2	3	4	5	6	7	8	9

FRONT = 0 REAR = 9

Queue full

0	1	2	3	4	5	6	7	8	9

FRONT = REAR = -1

Empty Queue

		7	18	14	36	45	21	80	81
0	1	2	3	4	5	6	7	8	9

FRONT = 2 REAR = 9

Set REAR = 0 and insert the value here

Queue with vacant location

Algorithm to delete an element

Step 1: if front=-1, then Write "underflow"

Step 2: set Val= queue[front]

Step 3: If front= rear, then

Set front=rear=-1

Else

If front=MAX-1

set front =0

Else

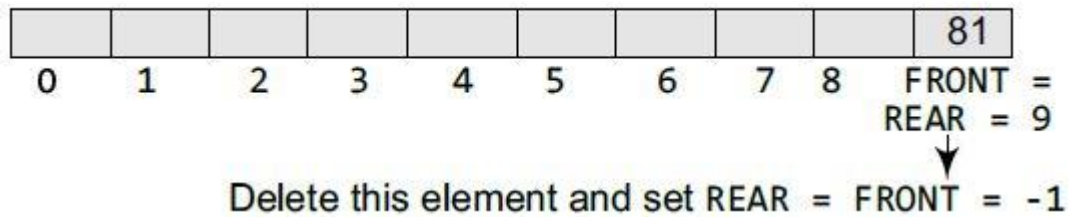
Set front=front+1

Step 4: exit

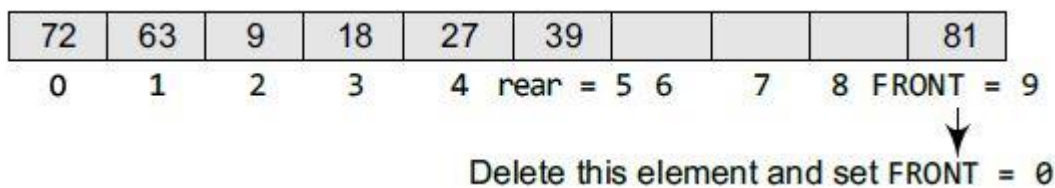
0	1	2	3	4	5	6	7	8	9

FRONT = REAR = -1

Empty queue



Queue with a single element



Queue where front = MAX-1 before deletion

Queue ADT or Circular Queue ADT

Abstract Datatype Queue

{

Instances

Linear list of element;

Insertion takes place at one end called Rear;

Deletion takes place at other end called Front;

}

{

Operations

QFull() : return True if Queue is full, Return False otherwise.

QEmpty() : return True if queue is empty, Return False otherwise.
herwise.

size() : return number of elements in the queue.

insert() : insert element at the Rear of the queue

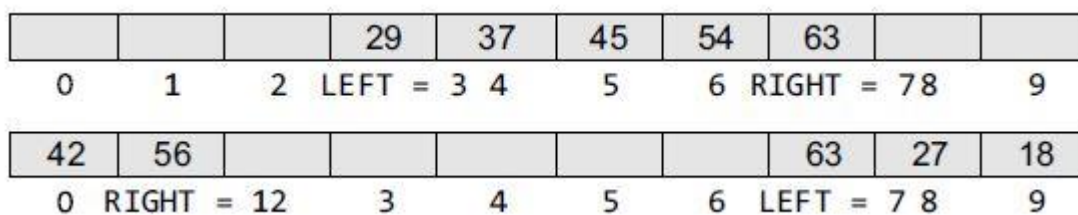
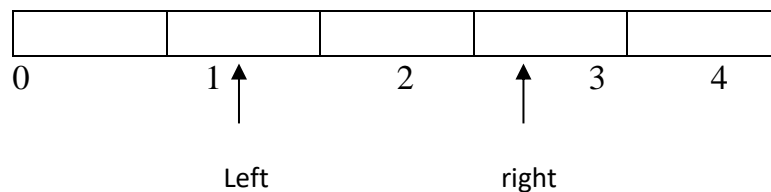
delete() : remove the element from front of queue.

Peek(), front() : return the front element from the queue.

}

Dequeues

- It is a list in which **the elements can be inserted or deleted at either end.**
- It is also known as a **head-tail linked list.**
- **No element can be added and deleted from the middle.**
- It can be implemented either by using a circular array or a circular doubly linked list.
- **In a deque, there are two pointers-LEFT and RIGHT**
- This pointers point to either end of the deque
- There are two type of deque
 - ✓ **Input restricted deque**-in this deque, insertions can be done only at one of the deque, while deletions can be done from both ends.
 - ✓
 - ✓ **Output restricted deque**- in this deque, deletions can be done only at one of the deque, while insertions can be done on both ends.



Double-ended queues

Priority queue

- It is a data structure in which **each element is assigned a priority**.
 - The priority determines the order in which the elements will be processed.
 - **The general rule** of processing the elements of a priority queue is
 - ✓ **The higher priority element is processed before the lower priority element.**
 - ✓ **The elements with the same priority are processed on a first-come-first –served basis.**
- **Priority queues are widely used in operating systems** to execute the highest priority process first.
- **The priority of the element can be set based on various factors.**
- The priority of the process may be set based on the **CPU time** it requires to get executed completely.
- For example, if there are three processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.
 - However, CPU time is not the only factor that determines the priority, rather it is just one among several factors.
- Another factor is the **importance of one process over another**.
- In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

Implementation of a priority queue

There are two ways to implement a priority queue

1. **Use a sorted list:** to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority.
2. **Use an unsorted list:** insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed.

Array representation of a priority queue

- ✓ A separate queue for each priority no. is maintained
- ✓ Each of these queues will be implemented using circular arrays or circular queues
- ✓ Every individual queue will have its own front and rear pointers
- ✓ We use two dimensional array for representing priority queue.

FRONT	REAR		1	2	3	4	5
3	3	1			A		
1	3	2	B	C	D		
4	5	3				E	F
4	1	4	I			G	H

Priority queue matrix

- FRONT[K] and REAR[K] contain the front and rear values of row K, where K is the priority number.
- The row and column indices start from 1, not 0