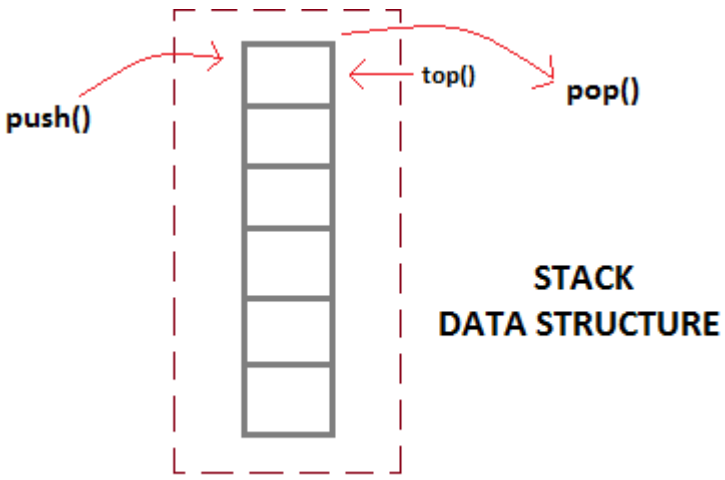
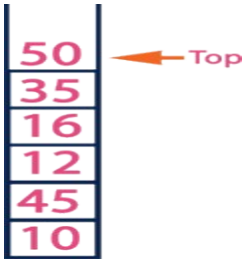
MODULE 2

STACKS: INTRODUCTION AND DEFINITION:

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



Example: If we want to create a stack by inserting 10,45,12,16,35 and 50. Then 10 becomes the bottom most element and 50 is the top most element. Top is at 50 as shown in the image below...



REPRESENTATION OF STACK AND OPERATIONS ON STACKS

Stack data structure can be implementing in two ways. They are as follows...

- Static representation Using Array
- Dynamic representation Using Linked List

When stack is implemented using array, that stack can organize only limited number of elements. When stack is implemented using linked list, that stack can organize unlimited number of elements.

Static representation Using Array:

A stack data structure can be implemented using one dimensional array. But stack implemented using array, can store only fixed number of data values. This implementation is very simple, just define a one dimensional array of specific size and insert or delete the values into that array by using LIFO principle with the help of a variable 'top'. Initially top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one

Dynamic representation using linked list:

The major problem with stack implemented using array is, it works only for fixed number of data values. The stack implemented using linked list can work for unlimited number of values. In linked list implementation of stack, every new element is inserted as top element.

OPERATIONS ON STACK:

Stack operation using array:

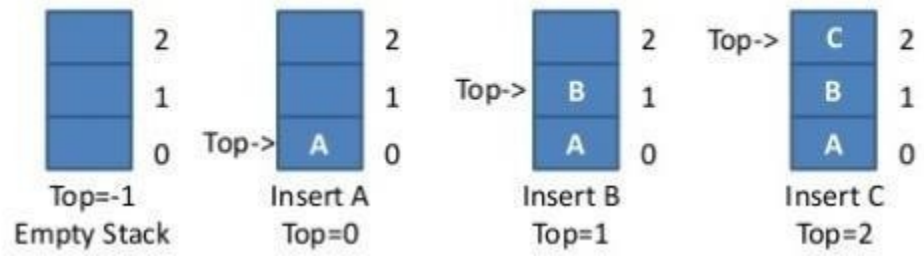
Push - Inserting value into the

stack Pop – Deleting value from

the stack **push operation**

- used to insert element into the stack
- each time new element is inserted into the stack, the value of top is incremented by one
- top >=N, It shows stack is overflow.

e.g.



Algorithm: push operation using Array

Push(stack, N, top, item)

Step 1: start

Step 2: check stack overflow

If top>=N then print “stack is overflow”

else

Step 3: set top = top + 1

Step 4: set stack[top] = item

Step 5: stop.

Note:

N = size of the stack

Item = data

Top = stack pointer

Stack = name of stack

Pop operation:

- Used to delete elements from the stack

- each time element is deleted from the stack, the value of top is decremented by one
- top =-1, It shows stack is underflow.

Algorithm: Pop operation using Array

Pop (stack, N, top, item)

Step 1: start

Step 2: check stack underflow

If stack = -1 then print “stack is underflow”

else

step 3: set item = stack[top]

step 4: set top=top-1

step 5: stop

Stack operations using linked list

Algorithm: push_linked list

Step 1: start

Step 2: Create a newNode with given value.

Step 3: Check whether stack is Empty

(top == NULL)

Step 4: If it is Empty, then

set newNode → next = NULL.

Step 5: If it is Not Empty, then

set newNode → next = top.

Step 6: Finally, set top = newNode.

Step 7: sto

Algorithm: pop_linked list

Step 1: start

Step 2: Check whether stack is Empty (top == NULL).

Step 3: If it is Empty, then display "Stack is Empty!!! Deletion is not possible!!!"

Step 4: If it is Not Empty, then

define a Node pointer 'temp' and set it to 'top'.

temp=top

Step 5: Then set 'top = top → next'.

Step 6: Finally, delete 'temp' (free(temp)).

Step 7: stop

APPLICATIONS ON STACKS

- Evaluation of Arithmetic expressions
- Code generation for stack machines
- Implementation of recursion
- Balancing of matching parenthesis
- Tower of Hanoi problem

Expression evaluation :

An expression is a collection of operators and operands that represents a specific value. In above definition, 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. Here operand can be a direct value or variable or address of memory location.

Expression types: Based on the operator position, expressions are divided into THREE types. They are as follows...

Infix Expression

Postfix Expression

Prefix Expression

Infix Expression

In infix expression, operator is used in between operands.

The general structure of an Infix expression is as follows...

Operand Operator Operand

Example: a+b

Postfix Expression

In postfix expression, operator is used after operands. We can say that "Operator follows the Operands". The general structure of Postfix expression is as follows...

Operand Operand Operator

Example: ab+

Prefix Expression

In prefix expression, operator is used before operands. We can say that "Operands follows the Operator".

The general structure of Prefix expression is as follows...

Operator Operand Operand

Example: +ab

Algorithm for infix to post fix conversion

Step 1: scan the expression from left to right.

Step 2: create empty stack.

Step 3: If scanned character is **operand**, then add it to **output.**

Step 4: If scanned character is **operator**, then add it to **stack.**

Step 5: compare scanned character with top of stack.

Step 6: If the reading symbol is left parentheses (, then Push it on to the Stack.

Step 7: If the reading symbol is right parenthesis ')', then Pop the contents of stack until respective left parenthesis is poped and print each poped symbol to the result.

Step 8: If top of stack has higher precedence than scanned character, delete higher precedence operator from the stack and add it to output and push the scanned character into stack.

Step 9: Repeat this process until all characters are scanned.

Problems:

1) Convert the following infix expression to postfix expression? (a+b)

Sol:

Scanned Character	Stack (operator)	Output
a	Empty	a
+	+	a
b	+	ab
		ab+

2) Convert the following infix expression to postfix expression? a+b*c+d

sol:

Scanned	Stack	Output

Character	(operator)	
a	Empty	A
+	+	A
b	+	Ab
*	+*	Ab
c	+*	Abc
+	++	abc*
d	++	abc*d
		abc*+d+

3) Convert the following infix expression to postfix expression? (A+B) * (C-D)

Sol:

Scanned Character	Stack (operator)	Output
(	(	
A	(	A
+	(+	A
B	(+	AB
)		AB+
*	*	AB+
(	*(	AB+

C	*(	AB+C
-	*(-	AB+C
D	*(-	AB+CD
)	*	AB+CD-
		AB+CD-*

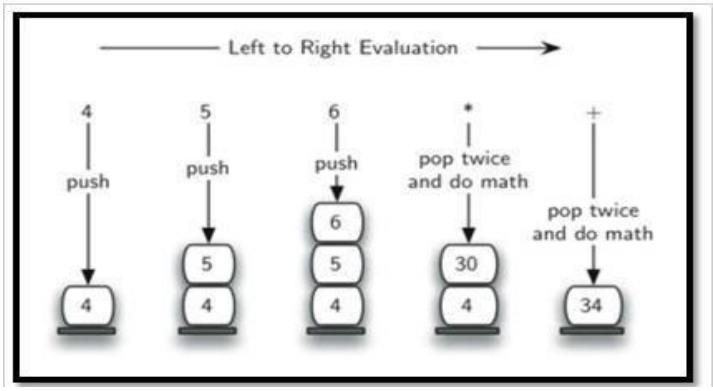
Postfix evaluation

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.

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then pop the required number of operands from the stack.
4. Evaluate the operator push the result back into the stack.
5. Finally! Perform a pop operation and display the popped value as final result.

Example:

Evaluate expression : 456*+



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Result: 34

Recursion:

Recursion is an important tool to describe a procedure having several repetitions of the same. A procedure is called recursive if the procedure is defined by itself.

Example:

Calculation of factorial value for an integer n.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 3 \times 2 \times 1$$

$$n! = n \times (n-1)!$$

Sample program:

```
fact(n)
{
if(n==0) then
```

```
return 1;
else
return(n*fact(n-1));
}
```

Balancing of matching parenthesis:

A balanced parenthesis means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. Consider the following correctly balanced strings of parentheses:

```
(( )( )( )( ))

(((( ))))

(( )((( ))( )))
```

Compare those with the following, which are not balanced:

```
(((((((( ))

( )))

(( )( )(( )
```

Code generation for stack machine

Generation for the codes for a given Arithmetic expression we are using registers. But some machines have limited number of registers. So, it is difficult to handle the storage of intermediate results. To overcome that problem, stacks are used instead of registers.

Stack has the following two instructions:

a) Push<name> : This instruction load the operands from memory location and place the content into the stack.
b) Pop<name>: The content on top of the stack is removed and stored in memory location.

To illustrate the hypothetical machine, let us consider the following arithmetic expressions:

A = B * C – A

The corresponding postfix notation can be obtained as follows:

A B C * A – =

The instruction code according to the stack machine is as given below:

```
PUSH A      // Load operand A into the stack
PUSH B      // Load operand B into the stack
PUSH C      // Load operand C into the stack
MUL         // Multiply B * C
PUSH A      // Load operand A into the stack
SUB         // Subtract B * C – A
POP A       // Store the result in the memory location for A
```

The various states of the stack are depicted in Figure 4.5.

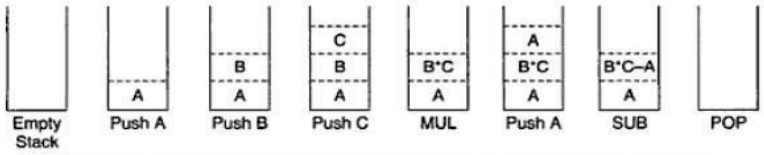


Figure 4.5 Evaluation of an arithmetic expression using a stack machine.

Towers of hanoi:

Another complex recursive problem is towers of Hanoi problem. Suppose there are three pillars: A,B,C. There are N discs of decreasing size so that no two disks are of the same size. Initially all the discs are stacked on one pillar in their decreasing order of size. Let this pillar be A. The other two pillars are empty. The problem is to move all discs from one pillar to another.

Conditions to move:

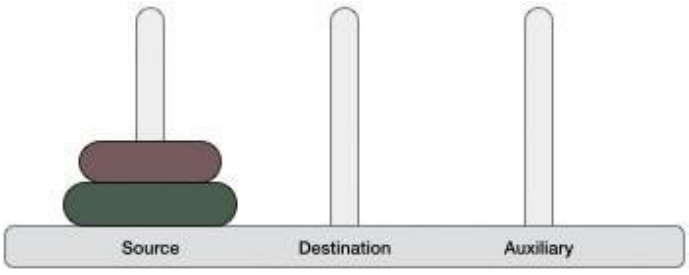
- Only one disc may be moved at a time.
- A disc may be moved from any pillar to another pillar
- A larger Disc cannot be placed on a smaller Disc.

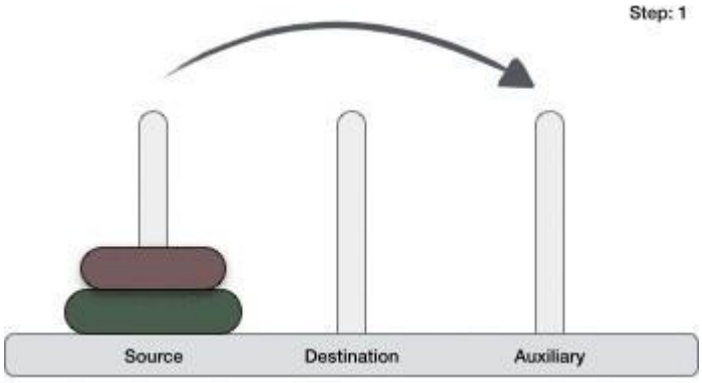
The steps to follow are −

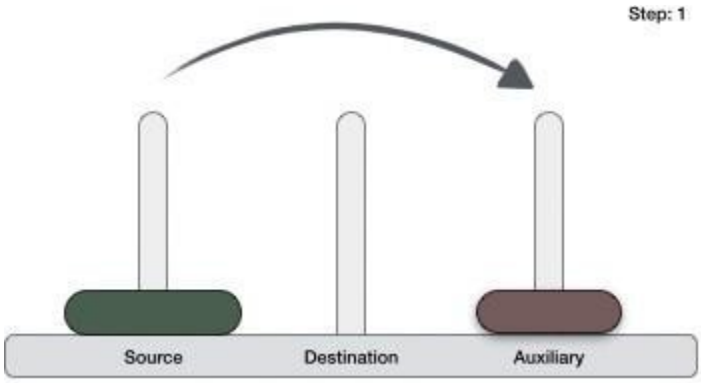
Step 1 − Move n-1 disks from **source** to **aux**
Step 2 − Move n^{th} disk from **source** to **dest**
Step 3 − Move n-1 disks from **aux** to **dest**

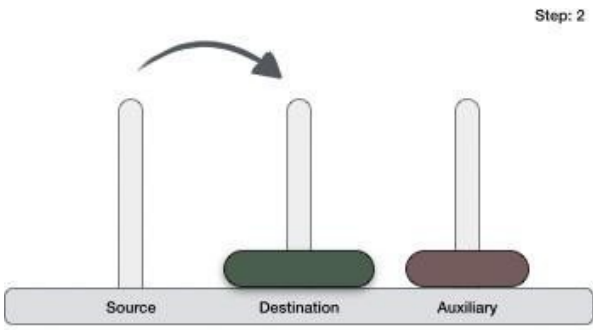
A recursive algorithm for Tower of Hanoi can be driven as follows −

```
Hanoi(disk, source, dest, aux)



Step 1: Start

Step 2: If disk == 1, then

    move disk from source to dest

  ELSE

    Hanoi(disk - 1, source, aux, dest)     // Step 1

    move disk from source to dest          // Step 2

    Hanoi(disk - 1, aux, dest, source)     // Step 3

Step 3: stop
```

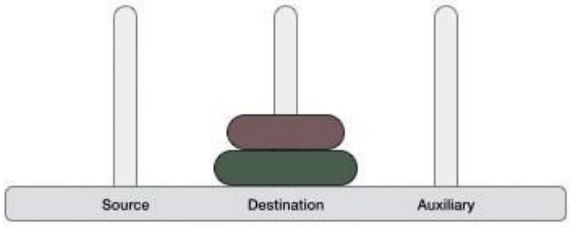



Step: 1
Source
Destination
Auxiliary


Step: 1
Source
Destination
Auxiliary


Step: 2
Source
Destination
Auxiliary

Step: 3



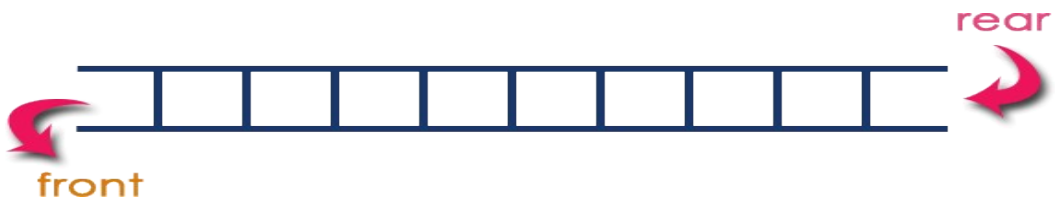
PART-II

QUEUES

QUEUES: INTRODUCTION

Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. The insertion is performed at one end and deletion is performed at other end. In a queue data structure, the insertion operation is performed at a position which is known as 'rear' and the deletion operation is performed at a position which is known as 'front'. In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle.

NOTE: In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called "deQueue()".



Example

Queue after inserting 25, 30, 51, 60 and 85.

REPRESENTATIONS OF QUEUES

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

- Static representation Using Array
- Dynamic representation Using Linked List

When a queue is implemented using array, that queue can organize only limited number of elements. When a queue is implemented using linked list, that queue can organize unlimited number of elements.

Static representation Using Array:

One dimensional array is used to represent a queue.

Two pointers front and rear represents two ends of queue.

To insert an element rear pointer is used and to delete an element front pointer is used.

Dynamic representation:

Queue is created dynamically by which size of the queue can be increased as per the necessity in Application program. For this purpose we are using single linked list representation of queue.

Operations on a Queue:

There are two basic operations on Queue: They are:

- Insertion
- Deletion

Insertion:

Insert elements into the queue by using rear pointer. For every insertion rear pointer is incremented by 1.

Deletion:

Delete an element from the queue by front pointer. For every deletion front pointer is incremented by 1.

Algorithm for insertion and deletion using Arrays (static representation)

Algorithm: insertion (Queue, N, Item, rear)

Step 1: start

Step 2: check Queue is full

if(rear==N) then

print “Queue

overflow” step 3: else

step 4: set rear=rear+1

step 5: set Queue[rear] = item

step 6: stop

Algorithm: Deletion (Queue, N, item, front)

Step 1: start

Step 2: check queue is empty

if(front = = rear) then

print “ Queue is underflow”

Step 3: else

Step 4: set item = Queue[front]

Step 5: set front = front + 1

Step 6: stop

Algorithm for insertion and deletion using linked list (Dynamic representation)

enQueue( ) - Inserting an element into the Queue

Step 1: Create a **newNode** with given value and

Step 2: Check whether queue is **Empty** (**rear** == **NULL**)

Step 3: If it is **Empty** then,

set **newNode** → **next** = **NULL**.

set **front** = **newNode**

set **rear** = **newNode**.

Step 4: If it is **Not Empty** then,

set **rear → next** = **newNode**

set **rear** = **newNode**.

deQueue( ) - Deleting an Element from Queue

Step 1: Check whether **queue** is **Empty** (**front == NULL**).

Step 2: If it is **Empty**, then

display **"Queue is Empty!!! Deletion is not possible!!!"**

Step 3: If it is **Not Empty** then, define a Node pointer '**temp**'

temp = front (set temp to **front**).

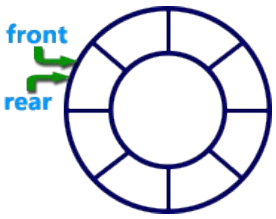
Step 4: set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

VARIOUS QUEUE STRUCTURES:

a) Circular queue
b) Deque
c) Priority queue

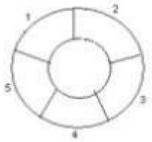
CIRCULAR QUEUE:

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.

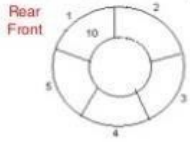


Example: Consider the following circular queue with N = 5.

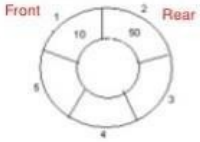
1. Initially, Rear = 0, Front = 0.



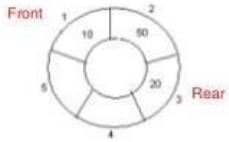
2. Insert 10, Rear = 1, Front = 1.



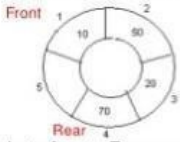
3. Insert 50, Rear = 2, Front = 1.



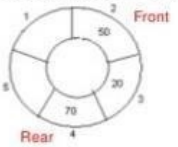
4. Insert 20, Rear = 3, Front = 0.



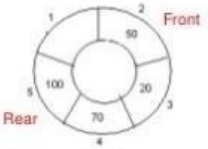
5. Insert 70, Rear = 4, Front = 1.



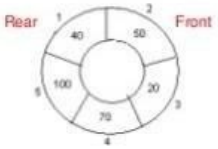
6. Delete front, Rear = 4, Front = 2.



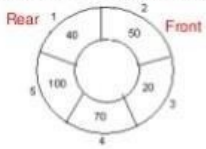
7. Insert 100, Rear = 5, Front = 2.



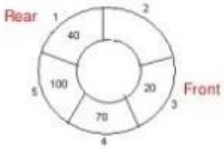
8. Insert 40, Rear = 1, Front = 2.



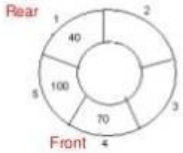
9. Insert 140, Rear = 1, Front = 2.
As Front = Rear + 1, so Queue overflow.



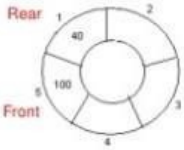
10. Delete front, Rear = 1, Front = 3.



11. Delete front, Rear = 1, Front = 4.



12. Delete front, Rear = 1, Front = 5.



How Circular Queue Works?

Circular Queue works by the process of circular increment i.e. when we try to increment any variable and we reach the end of queue, we start from the beginning of queue by modulo division with the queue size.

i.e.

```
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)
```

The second case happens when REAR starts from 0 due to circular increment and when its value is just 1 less than FRONT, the queue is full.

Queue operations work as follows:

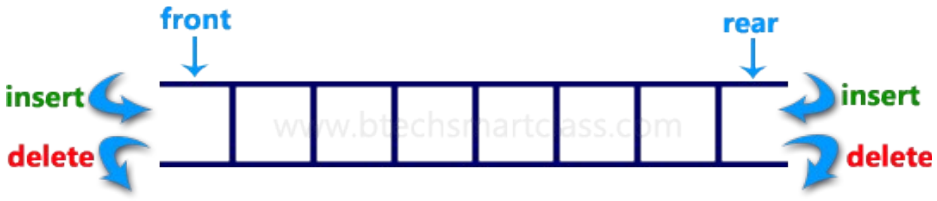
- Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.
- When initializing the queue, we set the value of FRONT and REAR to -1.
- On enqueing an element, we circularly increase the value of REAR index and place the new element in the position pointed to by REAR.
- On dequeueing an element, we return the value pointed to by FRONT and circularly increase the FRONT index.
- Before enqueing, we check if queue is already full.
- Before dequeuing, we check if queue is already empty.
- When enqueing the first element, we set the value of FRONT to 0.
- When dequeing the last element, we reset the values of FRONT and REAR to -1.

However, the check for full queue has a new additional case:

- Case 1: FRONT = 0 && REAR == SIZE - 1
- Case 2: FRONT = REAR + 1

DOUBLE ENDED QUEUE (DEQUEUE):

Double Ended Queue is also a Queue data structure in which the insertion and deletion operations are performed at both the ends (front and rear). That means, we can insert at both front and rear positions and can delete from both front and rear positions.

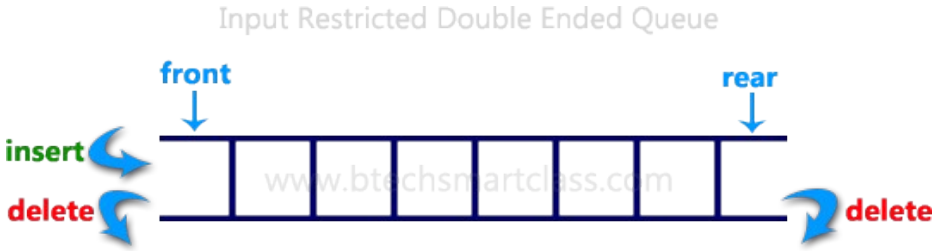


Double Ended Queue can be represented in TWO ways, those are as follows...

1. Input Restricted Double Ended Queue
2. Output Restricted Double Ended Queue

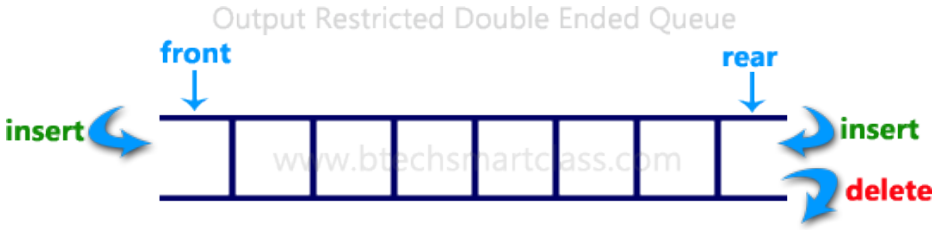
Input Restricted Double Ended Queue:

In input restricted double ended queue, the insertion operation is performed at only one end and deletion operation is performed at both the end



Output Restricted Double Ended Queue:

In output restricted double ended queue, the deletion operation is performed at only one end and insertion operation is performed at both the ends.



PRIORITY QUEUE: The priority queue is a data structure having a collection of elements which are associated with specific order. there are two types of priority queue.

Ascending priority queue: it is a collection of items in which the items can be inserted any order but only smallest element can be removed first.

Descending priority queue: it is collection of items in which insertion of elements can be in any order but only largest element can be removed first.

Application of priority queue:

1. The typical example of priority queue is scheduling the jobs in operation system. Typically, OS allocates priority to the jobs. The jobs are placed in the queue and position of the jobs in the priority queue determines their priority in OS. There are three kinds of jobs.

1. Real time jobs

2. Foreground jobs

3. Pending jobs

The OS always execute first real time jobs ,second foreground jobs, third pending jobs.

1. Priority queue is used in network communication.

2. used in simulation modeling

2.7. APPLICATIONS OF QUEUES:

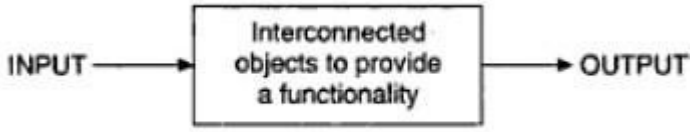
1. Simulation

2. CPU scheduling in multiprogramming environment

3. Round Robin algorithm

Simulation:

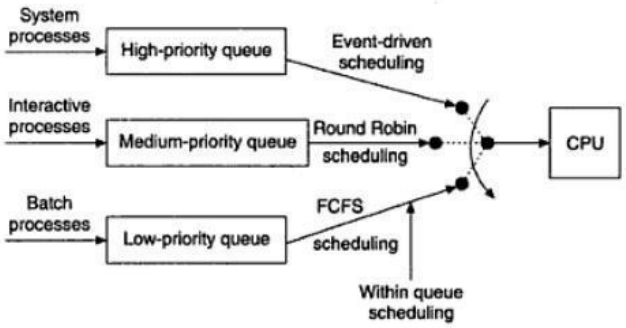
- Simulation is the model of a real life simulation in the form of computer program.
- The main objective of simulation program is to study the real life situation under the control of various parameters which affect the real problem.
- Based on the results of simulation, the actual problem is solved in an optimized way.
- **For example:** A computer program is a system where instructions are the interconnected objects and inputs and the results obtained during the execution constitute the output.
- Similarly, a ticket reservation counter is also a system.



(a) Outline of a system

CPU scheduling in multiprogramming environment

- In a multi-programming environment, a single CPU has to serve more than one program simultaneously.
- Possible jobs for the CPU is categorized into three groups:
  - System process
  - Interactive process
  - Batch process



Process scheduling with multi-level queues.

Round- robin algorithm: A round robin is a simple scheduling algorithm in which the queue data structure is used. A small unit of time called time quantum is defined. All the runnable processes are kept in the queue. The CPU Scheduler goes around this queue and allocates the CPU for each process one by one for that time interval. New processes are added at the rear end of the queue.

The CPU Scheduler chooses the first process from the queue and assigns the CPU to it. If the process does not finish its job, then that process will be added at the rear end, if the process finishes its job in given time interval then it is deleted from the queue. The CPU Scheduler assigns the CPU to the next process in the queue.

Round Robin with Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

Gantt chart is: