UNIT-2

STACKS AND QUEUES

Stack Abstract Data type :-

Stack is a linear data structure and is an ordered collection of homogenous data elements, where the Insertion and deletion operations takes place at one end called top of the stack. That means new element is added at top of the stack and an element is removed from the top of the stack.

In stack the Insertion and deletion operations are Performed based on "LIFO" (Last In First out) Principle.

STACK Operations :-

They are mainly 5 operations are applied on stack They are ① Push ② Pop ③ Peek ④ isfull ⑤ isempty

① **Push** :- Insertion of an element into the stack or writing a value to the stack is called Push operation.

② **Pop** :- Remove of an element from the stack or reading a value from the stack is called Pop operation.

③ **Peek** :- It is displays the top of the stack element.

④ **stack overflow** (isfull) :- when the stack is full and we try to Push an elements Into the stack, stack overflow occurs

⑤ **stack underflow** (isempty) :- when the stack is empty and we try to Pop (remove) an element from the stack, Then it is called stack underflow.

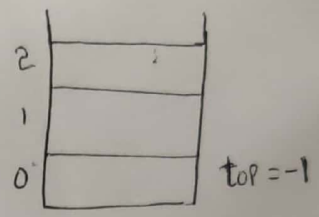
Array representation of stack :-

A stack can be Implemented using array as follows. Before Implementing actual operations first follows

Step1 :- Initialize The top and stack size:

Initially top = -1

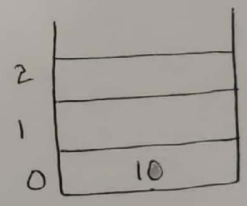
stack size = 3



Empty stack

Step2 :- Insert Item Into The stack item = 10

(i) Push (10)

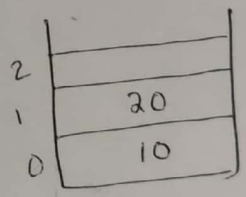


TOP = 0

stack [top] = item
10

(ii) Insert item = 20

Push (20)

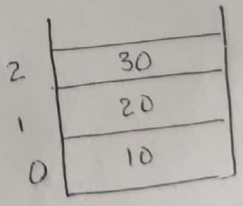


top = 1

stack [top] = 20

(iii) Insert item = 30

Push (30)



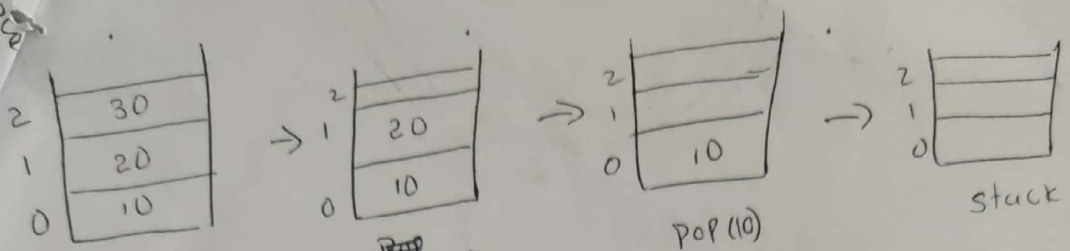
top = 2

stack [top] = 30

if (top == size-1) → stack is full the last value if we want to Insert another element into the stack stack overflow occurs because The size of The stack is full

if (top == size-1) → it occurs stack is full

3 - Remove element from The stack.



If we want to delete another element from the stack, stack underflow occuss because There is no elements In The stack.

Algorithms for stack operations :-

① isempty() Algorithm :-

```
begin
int isempty()
{
 if (top == -1)
   return true;
 else
   return false;
}
end
```

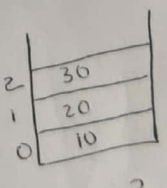
Implementation of isempty() function In c++ Programming language is slightly different.

we Initialize top = -1 as The Index In array starts from 0. So we check if The top is below zero or -1 to determine if The statck is empty. we try to Remove an element from The stack Then it is called "stack underflow"

② isfull() Algorithm :-

```
begin
int isfull()
{
 if (top = size-1)
   return true;
 else
   return false
}
end
```

Ex :-

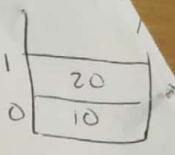


size=3
top=2

when The stack is full and we try to ~~Push~~ Insert an element into The stack, stack overflow occuss

③ Peek() Algorithm:-

Ex:-



⇒ return top;

```
begin
  int Peek (·)
  {
    if (top == null)
    { return "Stack is empty"
    else
      return stack[top];
    }
  end
```

Peek is an operation that returns the value of the topmost element of the Stack without deleting it from the Stack.

④ Algorithm for Push() operation:-

The Process of putting a new data elements onto stack is known as a Push operation. Push operation Involves a series of steps.

Step1:- check if the stack is full

Step2:- if the stack is full, Produces an error and exit

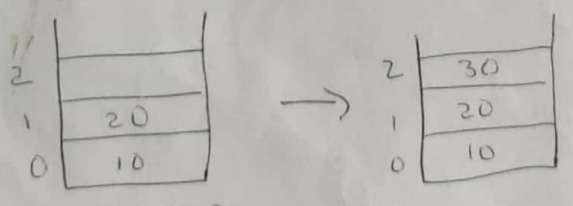
Step3:- if the stack is not full Increments the top to Point next empty space.

Step4:- Adds data element to the stack location where is top Pointing.

Step5:- returns success.

Ex:-

size=3



Push(30)

Push() :-

```
begin
push (stack, n, top, item)
{
if (top == size-1)
 {
 return stack full
 }
else
 {
   top = top++
   stack[top]= item ;
 }
 end
```

(5) POP() Operation :-

Accessing the content while removing it from the stack is known as a POP() Operation. In an array Implementation of POP() operation, the data element is not actually removed, instead top is decremented to lower position in the stack to point to the next value.

A Pop operation may involves the following steps.

Step1 :- Checks if the stack is empty.

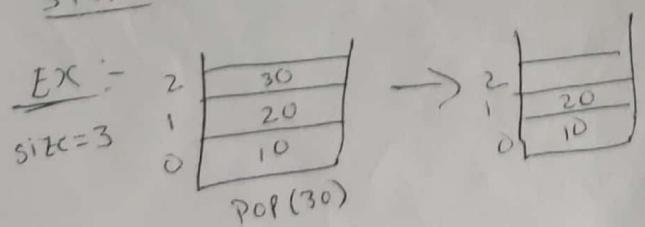
Step2 :- if the stack is empty, produces an error and exit

Step3 :- if the stack is not empty access the data element at which top is pointing.

Step4 :- decreases the value of top by 1

Step5 :- Return success

EX :-



Algorithm for POP() operation:-

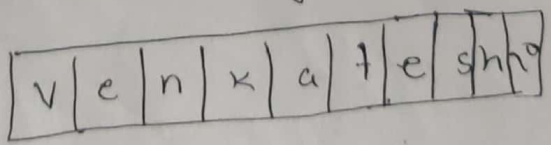
```
begin
POP (stack, top, item).
{
  if (top == -1)
    return stack is empty

  else
     item = stack[top];
      top--;
      return item;
  }
end
```

Stack Applications:-

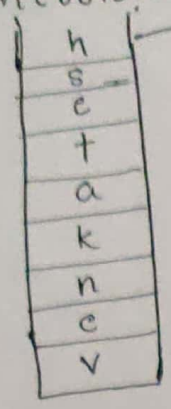
1. Reversing a list
2. factorial calculation
3. Infix to Postfix transformation
4. Evaluating Arithmetic Expressions.

① **Reversing a list:-** to reverse a string stack can be used The simple mechanism is to Push all characters of a string onto a stack and Then POP all The characters from The stack and Print Them.

Ex:- If The Input string is Venkatesh



Push all the characters onto the stack till '0' is not encountered.



Now if we POP each characters from the stack and Print it we get.

h	s	c	t	a	k	n	e	v

which is a reverse string

(2) Factorial calculation:-

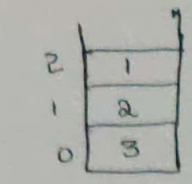
a Sample recursive Program how stacks helpful in handling recussion.

Ex:- Recursive factorial calculation.

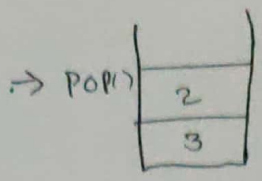
factorial can be computed using stack.

(1) Enter the number for which the factorial is to be computed say n
(2) Push the numbers from 1 to n onto the stack
(3) Initialize fact = 1
(4) Perform the multiplication with fact by poping the element each time
(5) Store the multiplication in fact variable
(6) Repeate the step 4 and 5 for n element
(7) finally display the factorial value.

Ex :- n = 3



push 1 to n

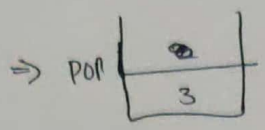


POP (1)

→ fact = fact * POP val

⇒ 1 * 1

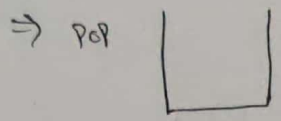
fact = 1



POP (2)

⇒ fact = 1 * 2

fact = 2



POP (3)

⇒ fact = 2 * 3

fact = 6

It displays The fact value = 6

(3) Evaluating Arithmetic Expressions :-

* Expressions :- Expressions is a string of operands and operators

Operands → It is some numeric (or) alphabets → Ex :- a, b, 1, 2, 3

Operators → Symbols → +, -, /, *, ^

Ex :- a + b + c , 1 + 3 + 4 - 2 5

Expressions are general divide into 3 types

1. Infix Expression
2. Postfix Expression
3. Prefix Expression

① Infix Expression :- In this notation, the operator is placed between its operands

In This type of Expressions The arrangement of operands and operator is as follows

Syntax: Infix Expression = operand1 operator operand2
(or)
⇒ OP1 operator OP2

Ex:- (a+b), (a+b) * (c+d), (1+2)+(3*5)

Parenthesis can be used in these expressions.

② Postfix Expressions:-

In this notation, The operator is placed after its operands

In This type of expressions the arrangement of operands and operators is as follows

Syntax:- Postfix Expression = operand1 operand2 operator
(or)
OP1 OP2 operator

Example:- ① ab +
② ab + cd - *
③ ab + e / df + *

In Postfix Expressions there is no Parenthesis used. All corresponding operands come first and then operator can placed.

③ Prefix Expressions:-

In this notation, The operator is placed before its operand. In Prefix Expression The arrangement of operators and operands as follows.

Syntax:- Prefix Expression = operator operand1 operand2
(or)
⇒ operator OP1 OP2

In This no Parenthesis used all The corres... operators came first and Then operands are ar...

Ex:- ① + a b
② + + a b c
③ + a * b c

Evaluating Arithmetic Expressions are represent by using 2 operations.
① Infix to Postfix
② Infix to Prefix

Order of operation:-
① Parenthesis → ( )
② Exponents → ^ (Right to left)
③ multiplication and division, (*, /) } left to right
④ Addition and Substraction, (+, -) }

① Infix to Postfix:-

① a + b → a b +

② a + b + c
(a b +) + (c)
a b + c +

③ a + (b + c)
a + b c +
a b c + +

④ a + b * c
a + b c *
a b c * +

⑤ a * b + c
a b * + c
a b * c +

⑥ (A - B) * (C + D)
(A B -) * (C D +)
A B - C D + *

⑦ (A + B) / (C + D) - (D * E)
(A B +) / (C D +) - (D E *)
A B + C D + / - (D E *)
A B + C D + / D E * -

fix to Prefix :-

① a+b

+ab

② a+b+C

+ab+C

++abC

③ a+(b+C)

a++bC

+a+bC

④ a+b*C

a+*bC

+a*bC

⑤ a*b+C

*ab+C

+*abC

⑥ (A-B) * (C-D)

-AB * -CD

*-AB-CD

⑦ (A+B) / (C+D) - (D*E)

(+AB) / (+CD) - (*DE)

/+AB+CD - *DE

-/+AB+CD*DE

⑧ (A+B) * C

(+AB) * C

*+ABC

Evaluation of Postfix Expression (using stack) :-

Ex: 562+*84/- equivalent Infix Expression is

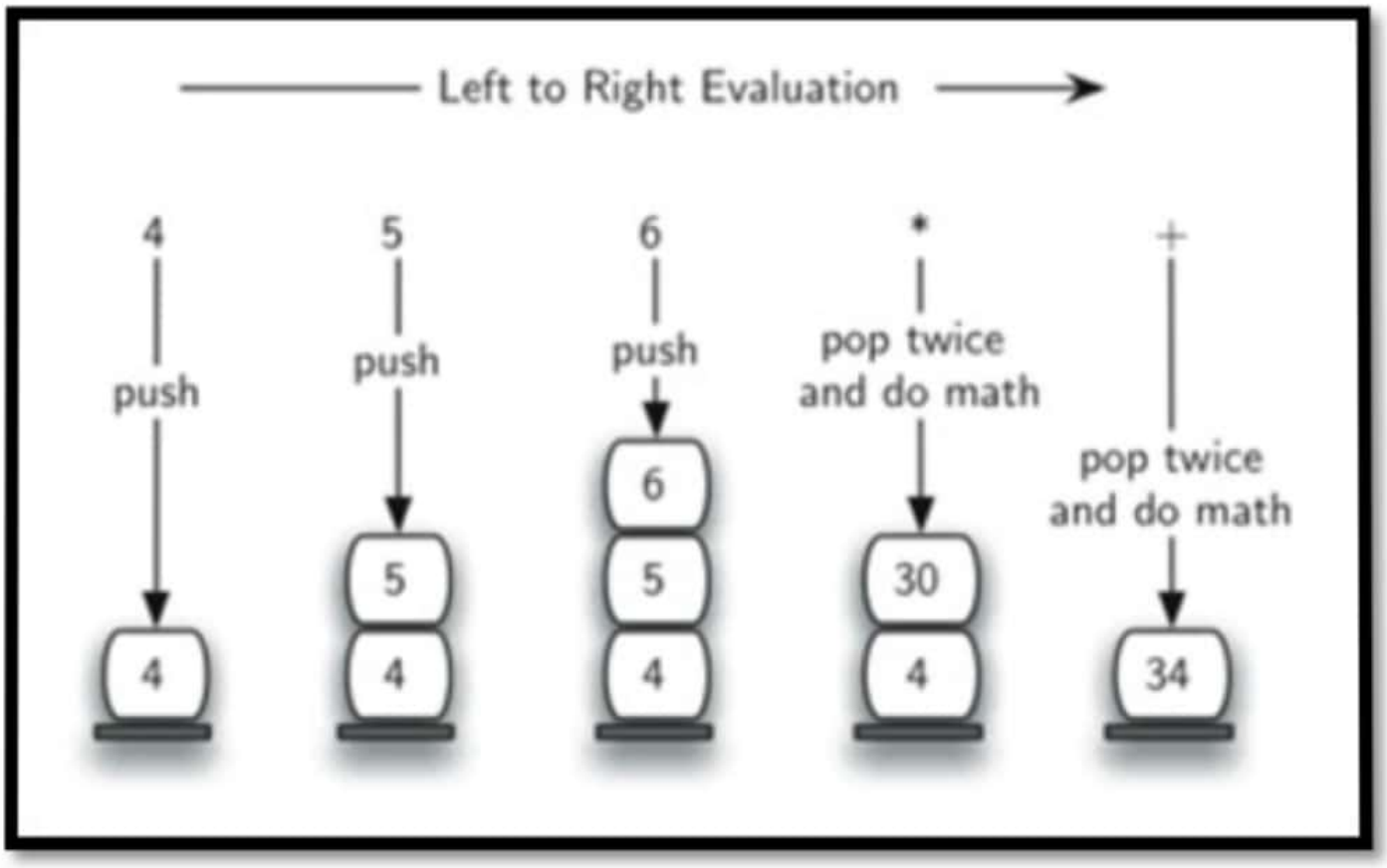
5*(6+2)-8/4.

Postfix: 562+*84/-$

Symbol scanned	stack
5	5
6	5, 6
2	5, 6, 2
+	5, 8
*	40
8	40, 8
4	40, 8, 4
/	40, 2
-	38

O/P :- 38

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 functions:

Recursion is the Process or Technique by which a function calls itself. A recursive function contains a statement within its body, which calls the same function. Thus, it is also called as circular definition. A recursion can be classified as direct recursion and indirect recursion.

The function calls itself is called direct recursion.
a function (f_1) calls another function (f_2) called indirect recursion. called function(f_2), calling function (f_1).

Ex: /* Program to find factorial of an number using recursion */

```
#include <stdio.h>
#include <conio.h>
void main()
int fact (int x)
{
   if (x<=1)
    return(1);
   else
     return (x * fact(x-1));
}
void main()
{
  int n, res;
  clrscr();
  Printf ("Enter a number");
  Scanf ("%d", &n);
   res = fact(n);
   Printf ("%d", res);
   getch();
}
```

O/P Enter a number 5
120

Ex = x = 5

5 × fact (5-1)
5 × fact (4),
5 × 4 × fact (4-1)
5 × 4 × fact (3),
5 × 4 × 3 × fact (3-1)
5 × 4 × 3 × fact (2),
5 × 4 × 3 × 2 × fact (2-1)
5 × 4 × 3 × 2 × fact (1)
5 × 4 × 3 × 2 × 1
=> 120

Program for Tower of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.

Approach :

```
Take an example for 2 disks :
Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.

Step 1 : Shift first disk from 'A' to 'B'.
Step 2 : Shift second disk from 'A' to 'C'.
Step 3 : Shift first disk from 'B' to 'C'.
```

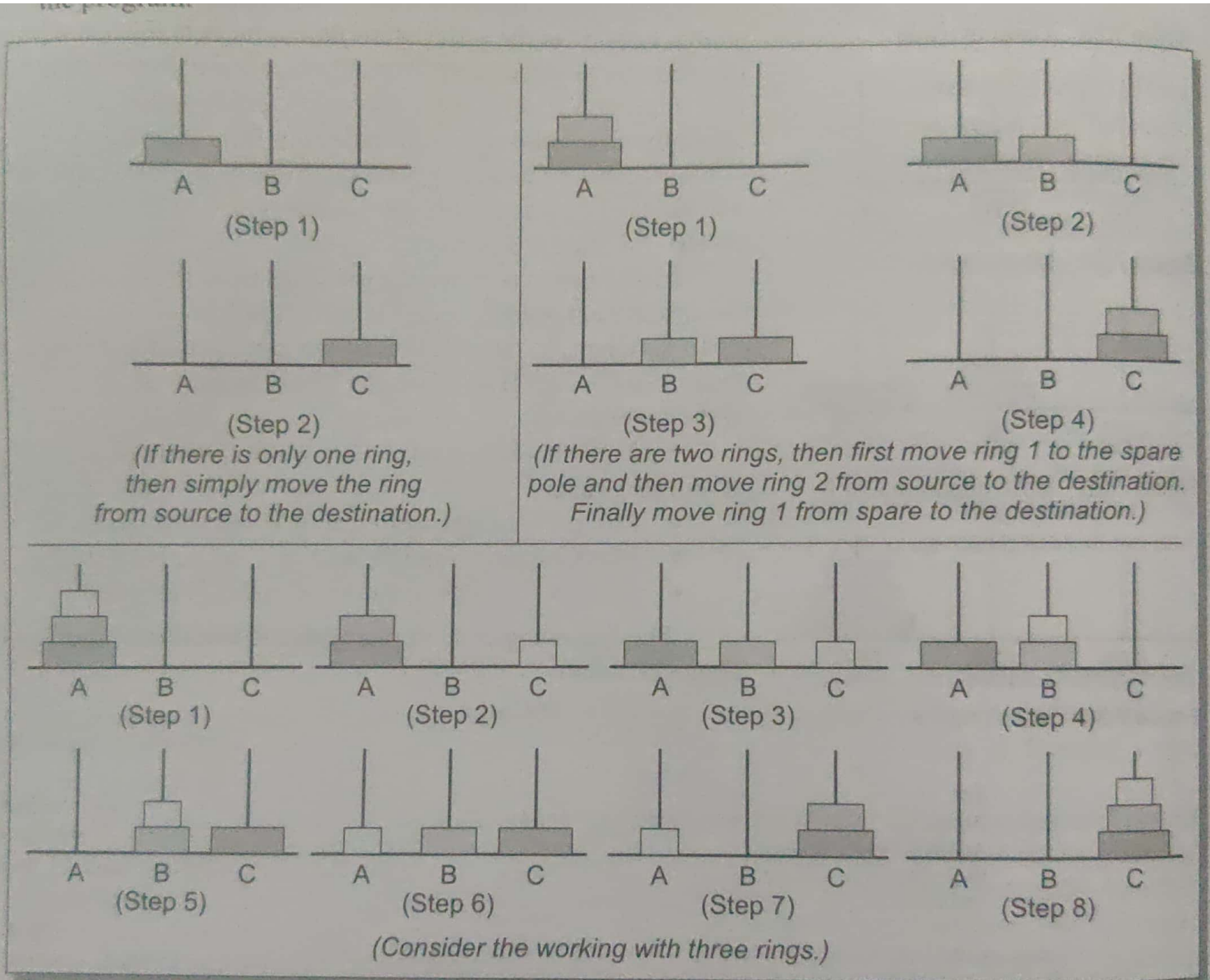


Figure 7.37 Working of Tower of Hanoi with one, two, and three rings

Queue ADT

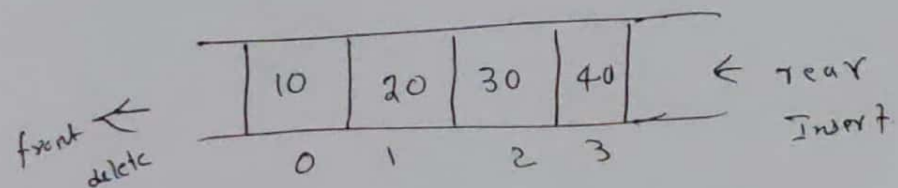
A Queue is a special type of data structure In which formally defined as ordered collection of elements That has two ends

① front (Deletions takesplace from The front end)

② rear (Insertions takesplace from The rearend)

Queue works on The basis of FIFO (First In First out) because The first element entering The Queue will be The first element to be deleted.

a Queue can be represented using the Sequential allocation (using arrays)



Queue operations:- Queue is nothing but collection of items. All The elements In The Queue are Stored sequentially. The various operations on The Queue are

① Queue underflow (queue Empty)

② Queue overflow (queue full)

③ Insert of The element Into The Queue (Enqueue)

④ Delete of The element ~~In~~ from The Queue (Dequeue)

⑤ display of The Queue (traverse)

Array representation of Queue :-

A Queue can be Implemented using array as follows.

Before Implementing actual operations first as follows.

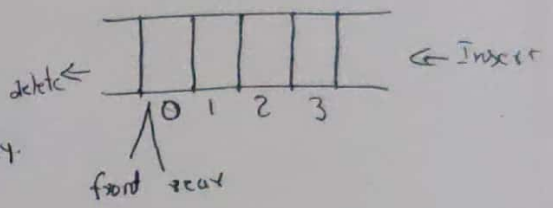
Step 1: declare the Queue using array.

Ex:- int queue[4]

Initialize the front = 0

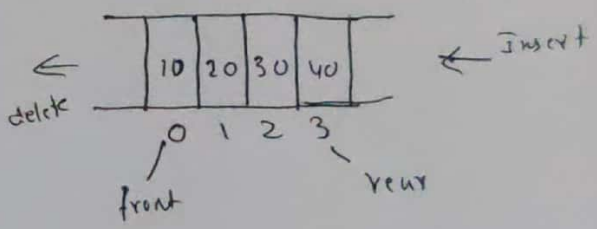
rear = 0

front == rear is nothing but Queue is empty.



Step 2: Insert a value Into the Queue — rear Increment until size-1

1. Enqueue (10) (rear 0)
   Enqueue (20) (rear 1)
   Enqueue (30) (rear 2)
   Enqueue (40) (rear 3)



if (rear == size-1) Then Queue is full. if we want to Insert another Element Into the Queue. The Queue Over flow occurs because the size of the queue is full.

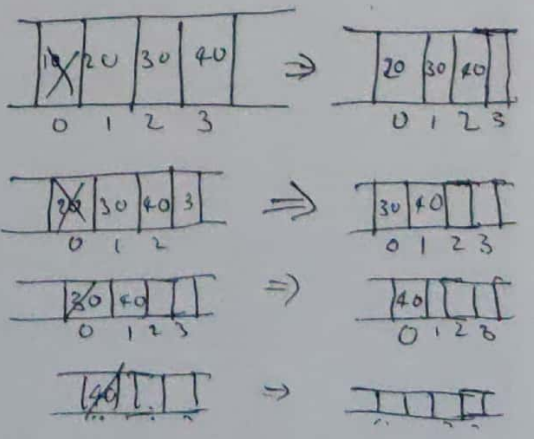
Step 3: Remove element from the Queue

dequeue () →

dequeue () →

de queue() →

de queue() →



Queue empty

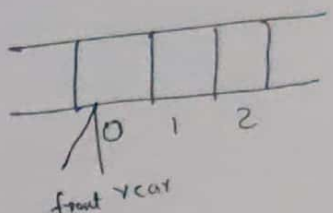
If we want to delete another element from the Queue, Queue is underflow occurs because there is no elements In The Queue.

Alogrithm for Queue ~~overflow~~ operations :-

(1) Queue underflow() (empty) :-

```
begin
int Queueempty()
{
  if (front == rear)
  {
    cout << "queue is empty";
  }
  else
  {
    cout << "queue contains value";
  }
}
end
```

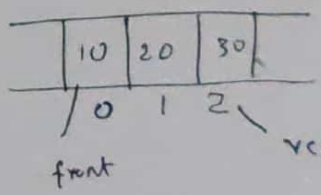
size = 3



(2) Queue is overflow() (full) :-

```
begin
  int Queuefull()
  {
    if (rear == size-1)
    {
      cout << "queue is full";
    }
    else
    {
      cout << "Queue is not full";
    }
  }
end
```

size = 3



rear = size-1

2 == 3-1

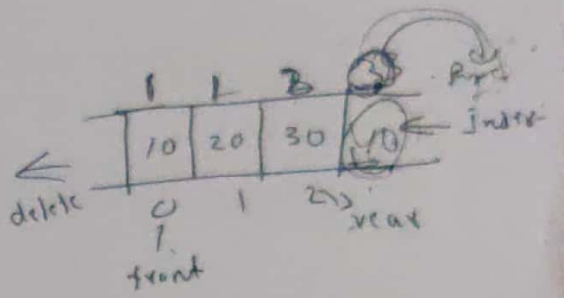
2 == 2

Queue is full

(3) Insert of The element Into The Queue:-

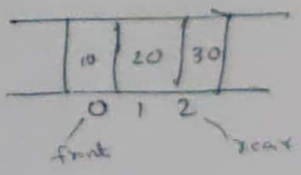
```
begin.
Void insert ( )
{
 if (rear == size-1)
 {
  cout<<" Queue is full & can't Insert value";
 }
 else
 {
  int element;
  cout <<" Enter element";
   cin >> element;
   Queue [rear] = element;
     rear++
 }
}
end.
```

size = 3



(4) display The element In Queue (or) traverse element:-

```
begin
void display ( )
{
  if ( front == rear )
  {
   cout<<" Queue is empty";
  }
  else
  {
   cout<<" queue element are:";
   for ( i= front; i<=rear; i++ )
    {
      cout << queue [i];
    }
  }
}
```

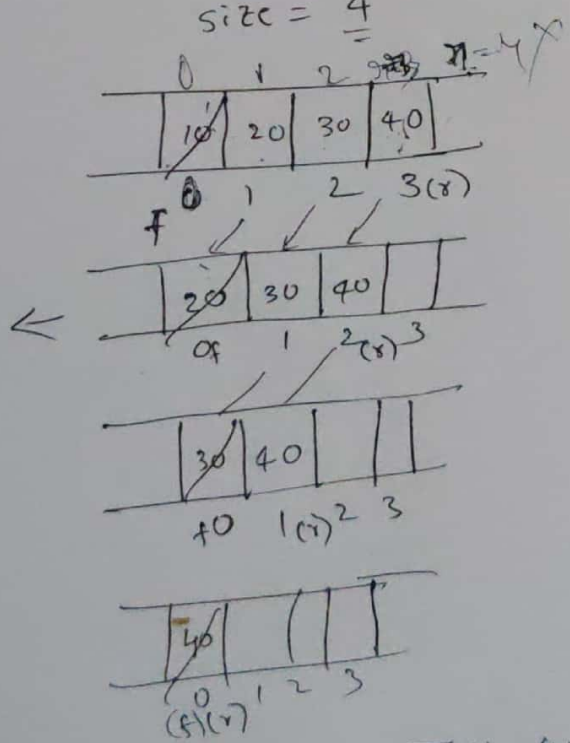


Delete Element In The Queue :-

- begin

```
Void delete ( )
{
 it ( front == rear)
 {
   cout << " Queue is Empty (can't delete element" ;
 }
 else
 {
 cout << " delete element is " << Queue [front] ;
  for ( i=0 ; i<=rear ; i++ )
  {
    Queue [i] = Queue [i+1] ;
  }
    rear -- ;
 }
}
```

Ex :-



delete front value = 10

delete front value = 20

delete front value = 30

delete front value = 40

Then front = rear Then Queue Empty

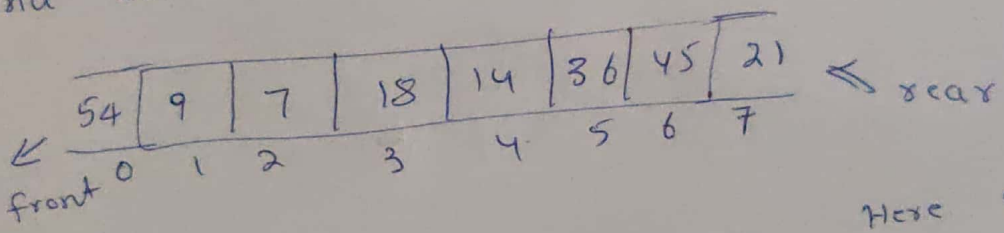
Types of Queues :-

A Queue data structure can be classified into the following types

1. Circular Queue
2. Deque
3. Priority Queue.

1. Circular Queue :-

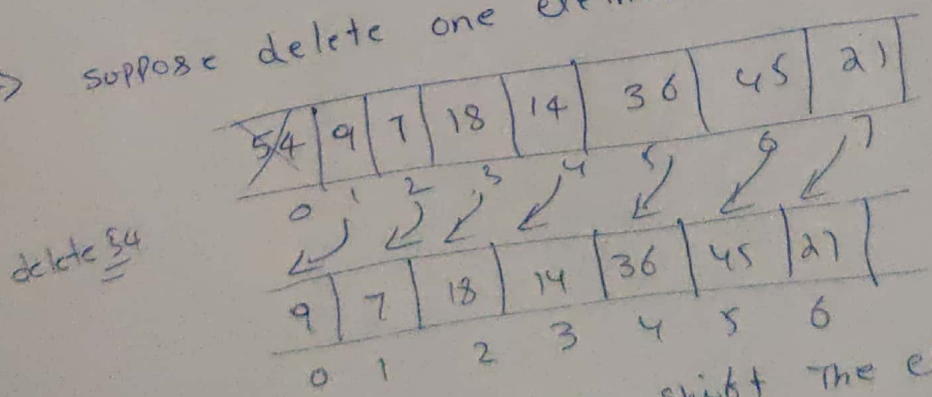
In linear Queue, Insertions can be done only at one end called Rear and deletions are always done from the other end called the front



Here size = 8
front = 0
Rear = 7

→ Now if you want to insert another value, it will not be possible because the queue is completely full.

→ suppose delete one element front(0) = 54



Main Problem In queue shift the elements to the left. But this can be very Time consuming especially when the queue is quite large.

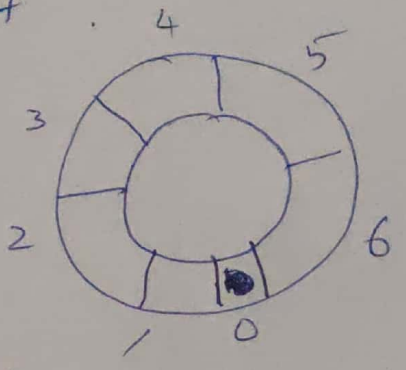
→ To resolve this Problem, we have one solution i.e Circular Queue.

Definition: Circular Queue is a linear Data structre In which the operations are Performed based on FIFO (First In First out) principle and last Position is connected back to the first Position To make a circle.

" It is also called "Ring Buffer".

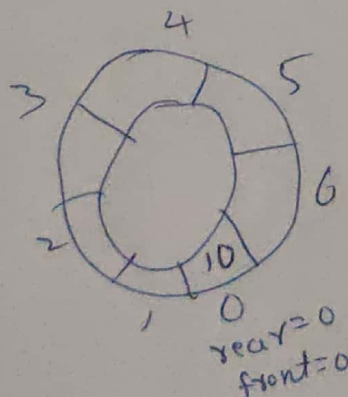
Invertion:

Here size = 7

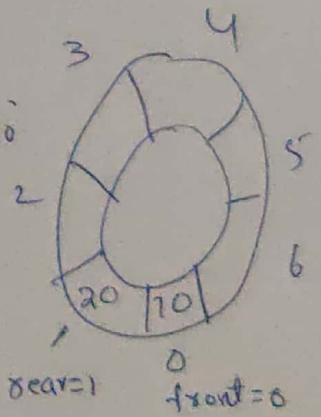


Initially
front = -1
rear = -1
" empty "

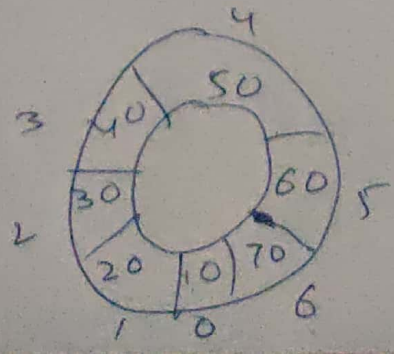
① Insert element '10'
Then Increment rear & front
rear=0 front=0



Insert element '20'
⇒

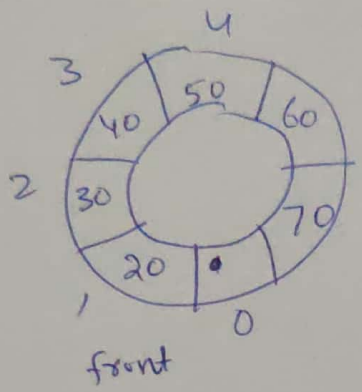


Insert element, 30, 40, 50, 60, 70



Here → rear = 6
front = 0

Suppose Delete The element front=0 ie 10 and Then front incremented

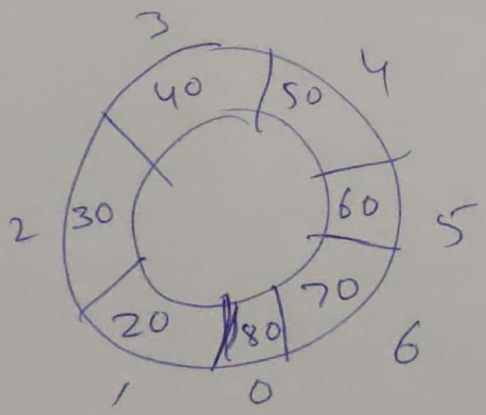


size = 7

rear = 6

front = 1

Insert The element :80 Then set rear = 0 and insert The element.



The circular queue will be full only when front = 0 and rear = Max – 1. A circular queue is implemented in the same manner as a linear queue is implemented.
The only difference will be in the code that performs insertion and deletion operations.

For insertion, we now have to check for the following three conditions:
1. If front = 0 and rear = MAX – 1, then the circular queue is full.
2. If rear != MAX – 1, then rear will be incremented and the value will be inserted.
3. If front != 0 and rear = MAX – 1, then it means that the queue is not full. So, set rear = 0 and insert the new element there.

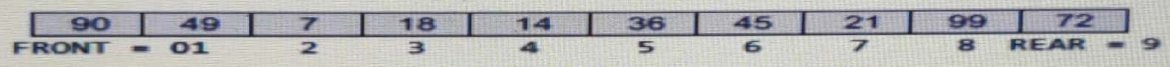


Figure Full queue

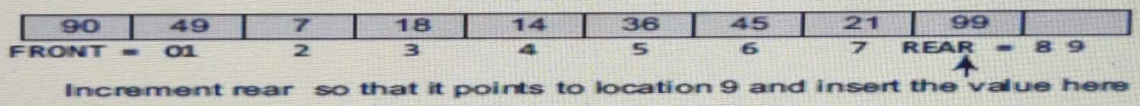


Figure Queue with vacant locations

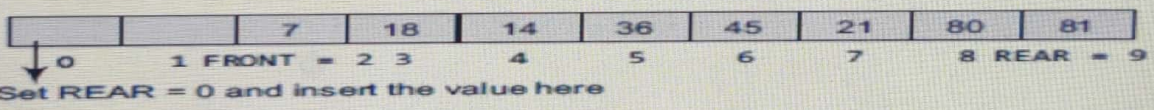


Figure Inserting an element in a circular queue

To delete an element, again we check for three conditions.

1. If front = –1, then there are no elements in the queue.
   So, an underflow condition will be reported.
2. If the queue is not empty and front = rear, then after deleting the element at the front the queue becomes empty and so front and rear are set to –1.
3. If the queue is not empty and front = MAX–1, then after deleting the element at the front, front is set to 0.

Figure Empty queue

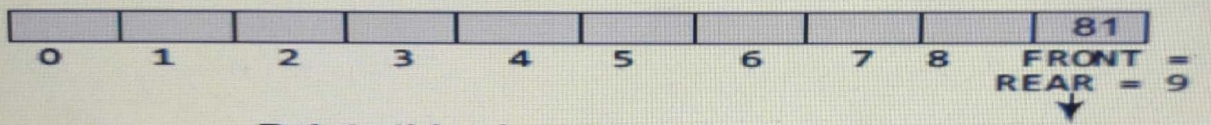


Figure Queue with a single element

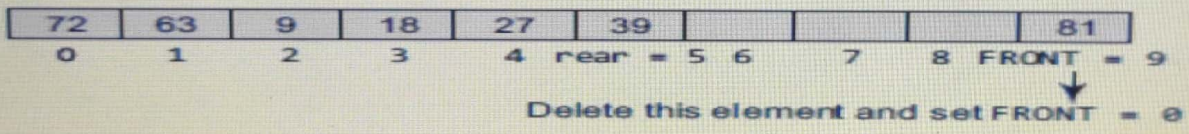
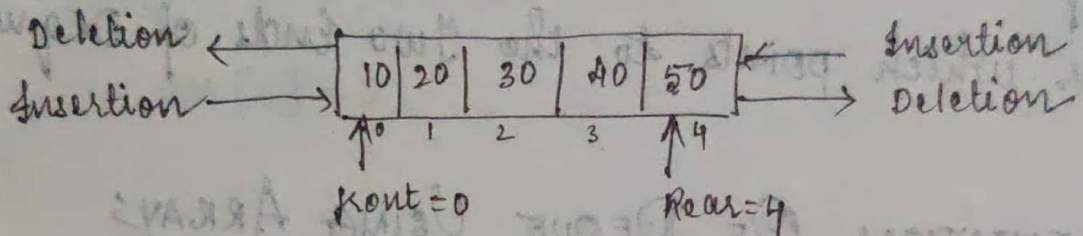


Figure Queue where FRONT = MAX–1 before deletion

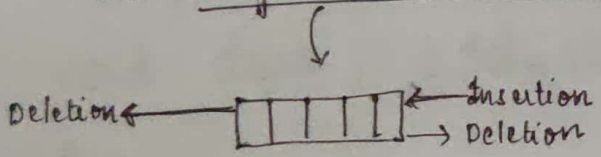
DE-QUES

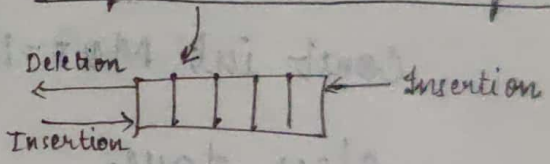
- A 'Deque' is a Linear List in which elements can be added or removed at either End but not in the middle

- The term Deque is a contraction of the name Double-Ended Queue



- There are two variables of deque, namely, the Input Restricted deque & Output Restricted deque





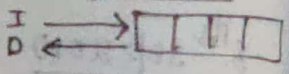
- An Input Restricted Deque is a Deque, which allows Insertions at only one End of the List

but allows deletion at both ends of List

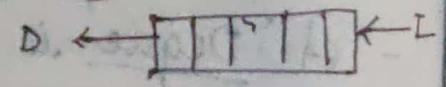
- An Output Restricted Deque is a Deque, that allows deletions at only one end of the list but allows Insertion at both ends of the list

- ~~A Deque is a double ended~~

- If you restrict yourself to InsertFront() and DeleteFront() then the Deque acts like an Stack



- If we restrict ourself to InsertFront() and DeleteRear() then it acts like a Queue



- A Deque provides a more versatile datastructure than either a stack or a Queue

- The Deque is maintained by either an array (or) Linked List with pointers 'Front' and 'Rear' which points to the Two Ends of Deque

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.

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.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.

3, 4, 8, 14, 22

- **add(2):** This function will insert '2' element in a priority queue. As 2 is the smallest element among all the numbers so it will obtain the highest priority.

2, 3, 4, 8, 14, 22

- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.

3, 4, 8, 14, 22

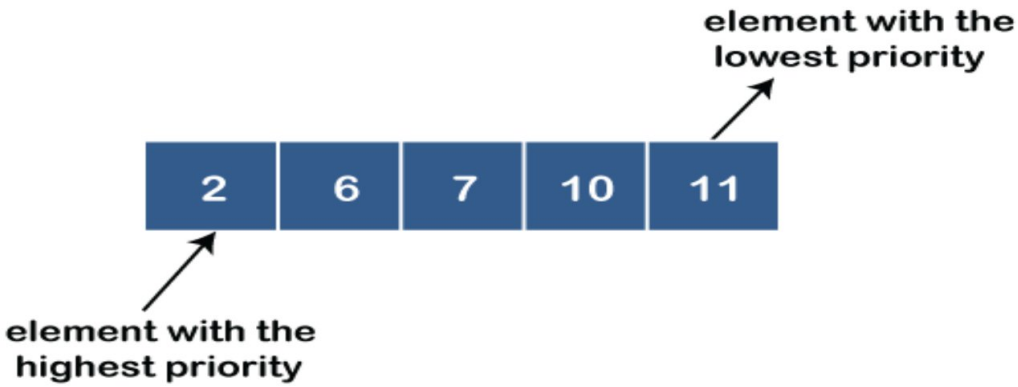
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

3, 4, 5, 8, 14, 22

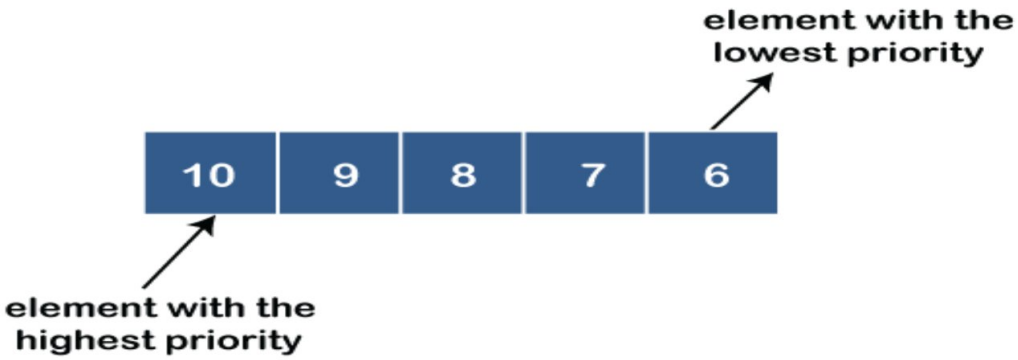
Types of Priority Queue

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.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node. 5

VALUE (INFO) **PNR(Priority)**

	INFO	PNR
0	200	2
1	400	4
2	500	4
3	300	1
4	100	2
5	600	3
6	700	4

Let's create the priority queue step by step.

In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 300, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 300, priority number 2 is having a higher priority, and data values associated with this priority are 200 and 100. So, this data will be inserted based on the FIFO principle; therefore 200 will be added first and then 100.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 400, 500, 700. In this case, elements would be inserted based on the FIFO principle; therefore, 400 will be added first, then 500, and then 700.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 600, so it will be inserted at the end of the queue.