Unit - III

3 Stack and Queue

3.1 : Concept of Stack

Q.1 Define the term stack.

Ans. : A stack is an ordered list in which all insertions and deletions are made at one end, called the top. If we have to make stack of elements 10, 20, 30, 40, 50, 60 then 10 will be the bottommost element and 60 will be the topmost element in the stack. A stack is shown in Fig. Q.1.1.

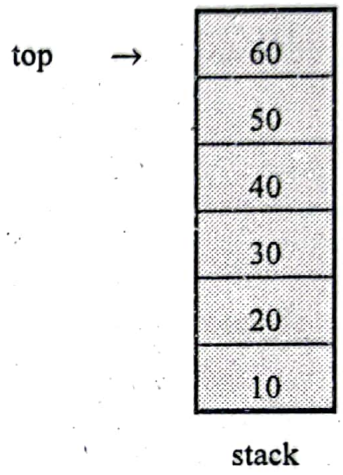


Fig. Q.1.1 Stack

Q.2 Explain the different ways by which the stack can be represented using sequential organization.

OR Explain sequential memory organization using suitable data structure. ☞ **[SPPU : May-17, Marks 6, May-18, Marks 4]**

Ans. : Declaration 1 :

```
#define size 100
int stack[size], top = -1;
```

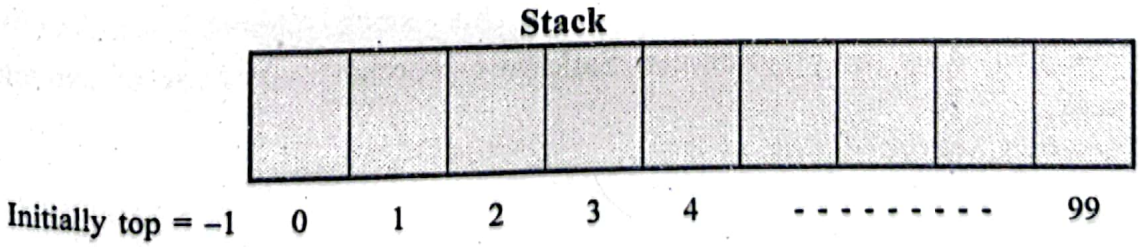


Fig. Q.2.1 Stack using one dimensional array

Declaration 2 :

```
#define size 10
struct stack {
        int s[size];
            int top;
           }st;
```

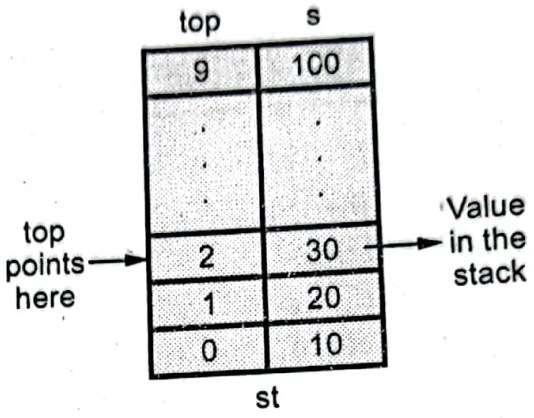


Fig. Q.2.2 Stack using structure

In the above declaration stack is declared as a structure.

Refer Fig. Q.2.2.

3.2 : Concept Implicit and Explicit Stack

Q.3 Explain the Concept of implicit and explicit stack and give its importance. ☞ **[SPPU : May-11, Dec.-12, Marks 4, May-14, Marks 2]**

Ans. : When we apply recursion an **implicit stacks** is used. Implicit stack is a special kind of stack used by **runtime memory environment.** This stack is used to hold the information about function CALL. This type of stack is used implicitly by the compiler to support function CALL and RETURN from the function. This stack cannot be directly manipulated by the programmer. Following program makes use of explicit stack.

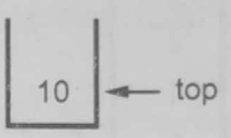
```
#include<iostream>
#include<stdlib.h>
using namespace std;
int main()
{

    int fun(int);
    cout<<"\n Use of Implicit stack";
    int a=10;
    fun(a);
    return 0;
}
int fun(int a)
{
 if(a>0)
 {
    fun(a-1);
    cout<<" " <<a;
 }
 else
    return a;
}
```

Output

Use of Implicit stack 1 2 3 4 5 6 7 8 9 10

(I) When **main ( )** CALLs fun (a) implicit stack is invoked and value of a = 10 is pushed onto the stack.

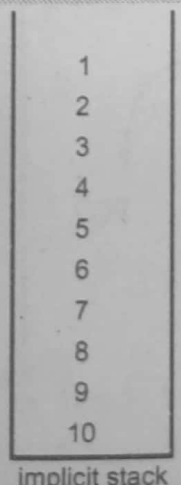
∴



implicit stack.

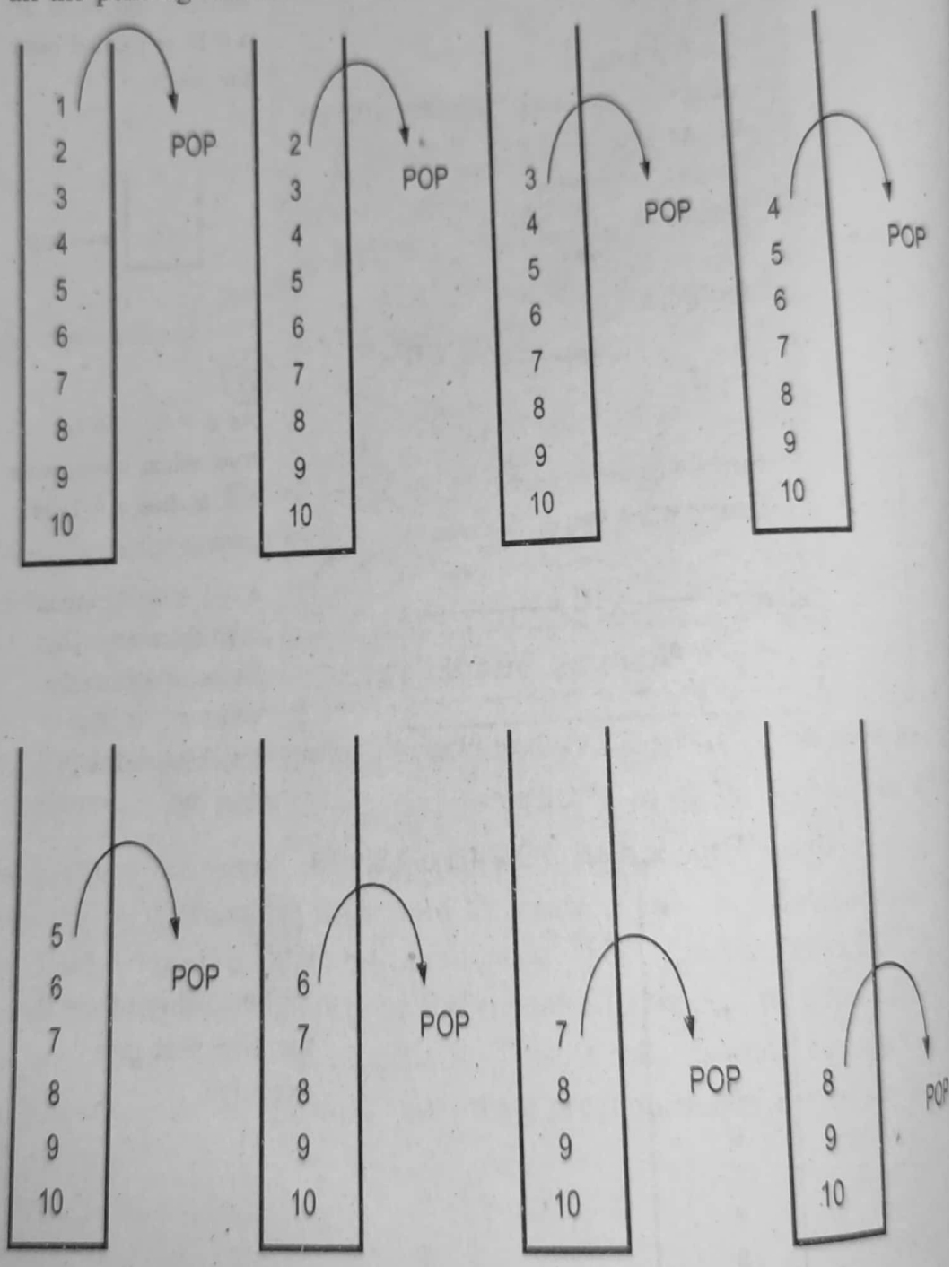
(II) As a = 10 > 0 is true, when a recursive call to **fun(a – 1)** is given

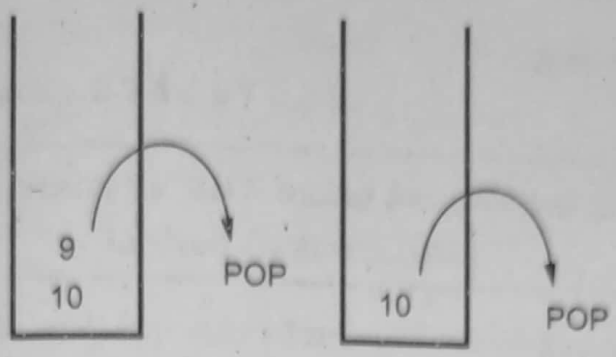
a – 1 = 9 is pushed onto the stack. This happens repeatedly when a > 0. The implicit stack will then be



(III) When value of a = 0 · the **else** part gets executed

(IV) Now, the control of the program reaches to **cout** statement. This is the time when recursive function wants to return to the **main( )**. At this instance, the stack is popped off and a = 1 gets printed. The control then reaches to the } bracket at the end. Thus control switches between **cout** and } and each time stack is popped off. This repetition occurs because all the pending call are in stack.





(V) Finally since all the pending or incomplete calls are getting fulfilled, the control returns back the **main( )** function.

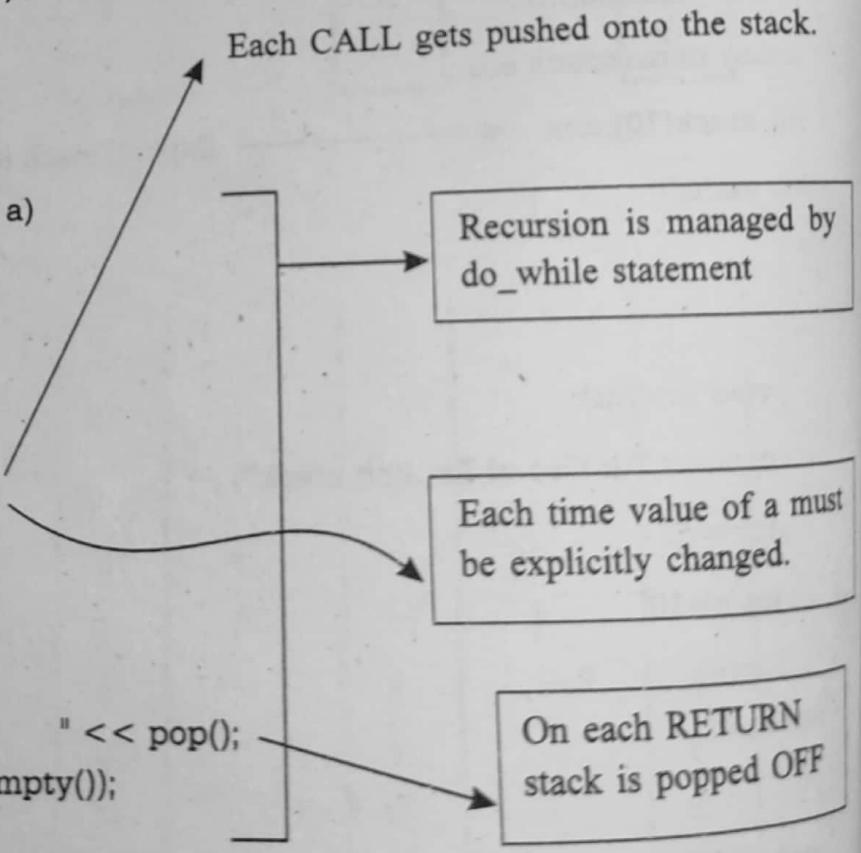
The implicit stack is mainly used to store data that is local to a function. Space for local data items is allocated from the stack during the execution of that function, and is de-allocated when the function exits.

Explicit stack

The above program can be shown below with **explicit usage of stack** -

```
#include<iostream>
  using namespace std ;
  int stack[10],top;  <-------- Explicit stack is maintained
  int main()
  {

     void fun(int);
     cout<<"\n Use of Explicit stack";
     top=-1;
     int a=10;
     fun(a);
  return 0;
  }
  void push(int item)
  {
```

```
    top++;
    stack[top]=item;
   }
   int pop()
   {
    int item;
    item=stack[top];
    top--;
    return item;
   }


int stempty()
   {
    if(top==--1)
      return 1;
    else
      return 0;
   }
   void fun(int a)
   {
    do
    {
    if(a>0)
    {
       push(a);
       a--;
    }

    else
       cout<<"     " << pop();
   }while(!stempty());

  }
```



Output

```
Use of Explicit stack 1 2 3 4 5 6 7 8 9 10
```

3.3 : Stack as ADT using Sequential and Linked Organization

Q.4 Write a Push and Pop operations with suitable example.

Ans. : We will now discuss the two important functions which are carried out on a stack. push is a function which inserts new element at the top of the stack. The function is as follows.

```
void push(int item)
{
 st.top++; / * top pointer is set to next location */
 st.s[st.top] =item; /* placing the element at that location */
}
```

Note that the push function takes the parameter **item** which is actually the element which we want to insert into the stack - means we are pushing the element onto the stack. In the function we have checked whether the stack is full or not, if the stack is not full then only the insertion of the element can be achieved by means of push operation.

Now let us discuss the operation pop, which deletes the element at the top of the stack. The function pop is as given below -

Note that always top element can be deleted.

```
int pop()
{
    int item;
    item = st.s[st.top];
    st.top – –;
     return(item);
}
```

In the choice of pop- it invokes the function 'stempty' to determine whether the stack is empty or not. If it is empty, then the function generates an error as stack underflow ! If not, then pop function returns the element which is at the top of the stack. The value at the top is stored in some variable as item and it then decrements the value of the top,

which now points to the element which is just under the element being retrieved from the stack. Finally it returns the value of the element stored in the variable item. Note that this is what called as logical deletion and not a physical deletion, i.e. even when we decrement the top, the element just retrieved from the stack remains there itself, but it no longer belongs to the stack. Any subsequent push will overwrite this element.

The push operation can be shown by following Fig. Q.4.1.

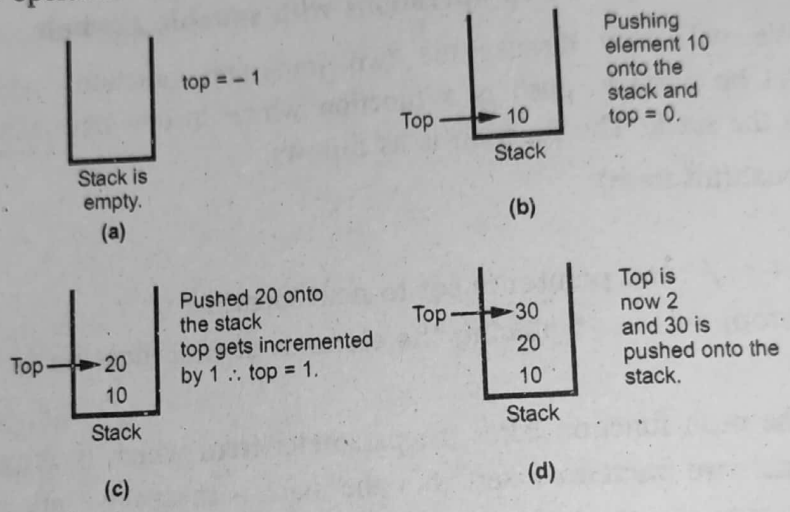


Fig. Q.4.1 Performing push operation

The pop operation can be shown by following Fig. Q.4.2.

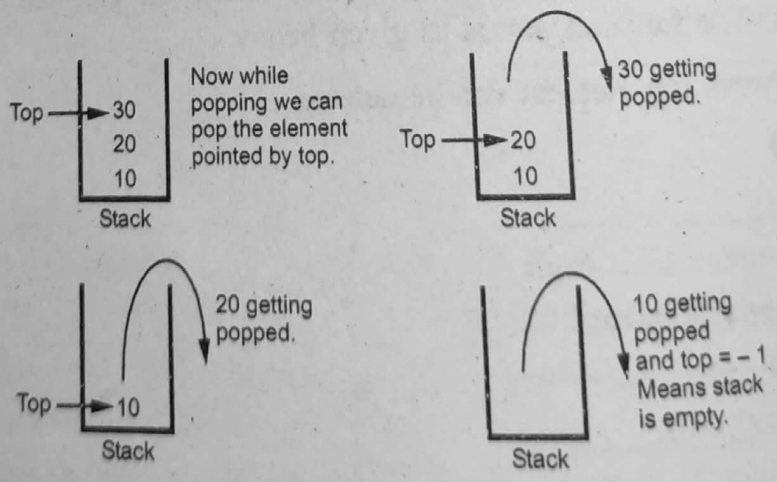


Fig. Q.4.2 Performing pop operation

Q.5 Explain stack implementation using linked list.

Ans. : Stack is a special case of list and therefore we can represent stack using arrays as well as using linked list. The advantage of implementing stack using linked list is that we need not have to worry about the size of the stack. Since we are using linked list as many elements we want to insert those many nodes can be created. And the nodes are dynamically getting created so there won't be any stack full condition.

The typical structure for linked stack can be

```
struct stack
{
   int data ;
   struct stack * next ;
}node ;
```

Each node consists of data and the next field. Such a node will be inserted in the stack. Following Fig. Q.5.1 represents stack using linked list.

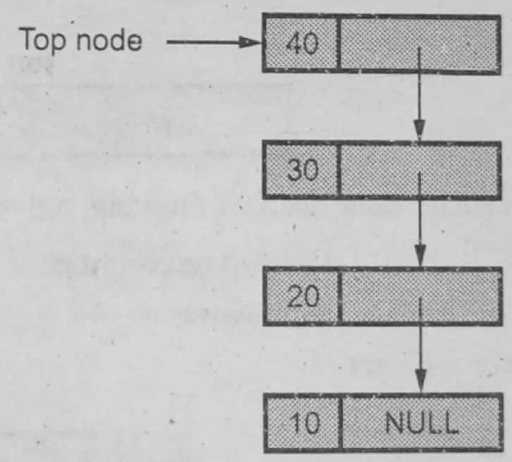


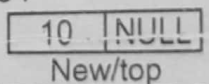
Fig. Q.5.1 Representing linked stack

1. Push operation

The pseudo code for push operation is as given below.

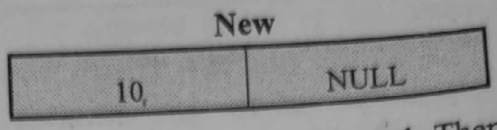
```
void Lstack::Push(int Item, node **top)
{
   node *New;
   New = new node;
   New->next=NULL;
   New->data=Item;
   New-> next = *top;
   *top = New;
}
```

Using new operator the memory can be allocated.
For example :



New/top

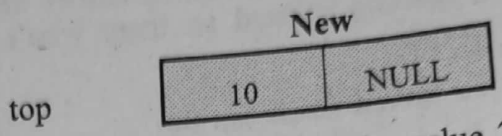
In the main function, when the item to be inserted is entered, a call to push function is given.



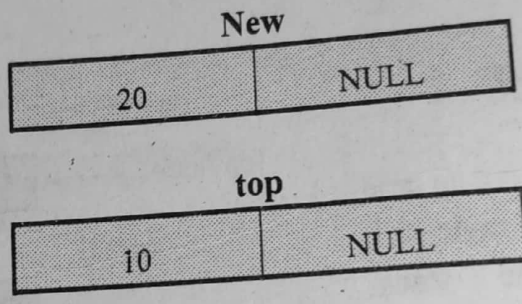
If item = 10, then by new node will be allocated. Then

```
*top    = New
```

This is our new **top** node. Hence stack will look like this



If again **Push** function is called for pushing the value 20 then



This can be done using following statements -

```
New->next=*top;
top=New;
```

And we will get

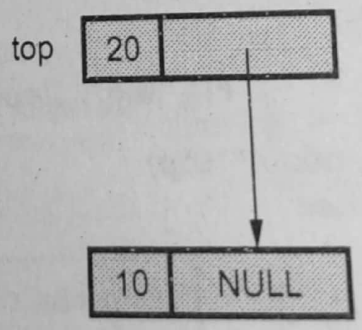


Fig. Q.5.2

When 30 is pushed then

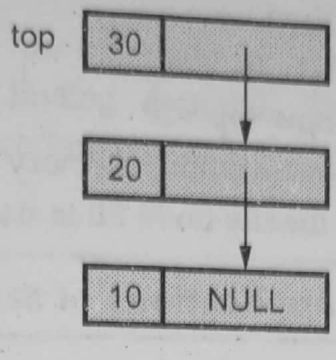


Fig. Q.5.3

Thus by pushing the items repeatedly we can create a stack using linked list.

2. Pop operation

```
int Lstack::Pop(node **top)
{
  int item;
  node *temp;
  item = (*top) ->data ;
  temp = *top;
  *top = (*top) -> next ;
  delete temp;
  return (item);
}
```

We want to delete top node. Hence in order to remember the data being deleted it is already stored in item

Taking the address of **top** in **temp** node and assigning new **top** to the next node.

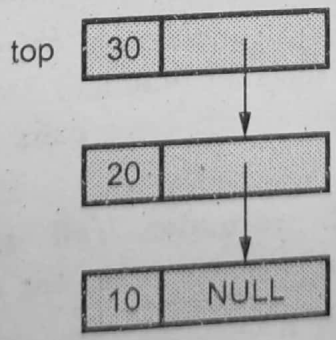


Fig. Q.5.4

If stack is as follows.

If we write

item = *(top)->data	then item=30
temp=*top	
*top=*top->next	now top =20
delete temp;	deallocating memory of node 30. That means node 30 is deleted.

3.4 : Applications of Stack

Q.6 What are various applications of stack ?

Ans. : Various applications of stack are

1. Expression conversion.
2. Expression evaluation.
3. Parsing well formed parenthesis.
4. Decimal to binary conversion.
5. Reversing a string.
6. Storing function calls.

3.5 : Recursion

Q.7 What is recursion ? Explain the properties of recursion.

Ans. : Definition : Recursion is a programming technique in which the function calls itself repeatedly for some input.

By recursion the same task can be performed repeatedly.

Properties of Recursion

Following are two fundamental principles of recursion

1. There must be at least one condition in recursive function which do not involve the call to recursive routine. This condition is called a "way out" of the sequence of recursive calls. The is called **base case property.**
2. The invoking of **each recursive call must reduce** to some manipulation and must go closer to base case condition.

For example - While computing factorial using recursive method -

```
if(n==0)
  return 1;  //This is a base case
else
  return n*fact(n-1);//on each call the computation will go closer to
base case
```

Q.8 Explain the use of stack in recursion.

Ans. : For illustrating the use of stack in recursive functions, we will consider an example of finding factorial. The recursive routine for obtaining factorial is as given below -

```
int fact (int num)
{
   int a, b;
   if (num = = 0)
      return 1 ;
      else
   {
      a = num – 1 ;
      b = fact (a) ;
      f = a * b;
      return f ;
   }
}
/* Call to the function */
ans = fact (4) ;
```

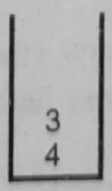
For the above recursive routine we will see the use of stack. For the stack the only principle idea is that – when there is a call to a function the values are pushed onto the stack and at the end of the function or at the return the stack is popped

Suppose we have num = 4. Then in function **fact** as num ! = 0 else part will be executed. We get

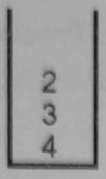
a = num – 1

a = 3

And a recursive call to fact (3) will be given. Thus internal stack stores



Next fact (2) will be invoked. Then

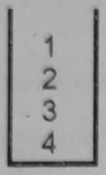


$\because$ a = num – 1

a = 3 – 1

b = fact (2)

Then,



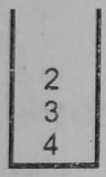
$\because$ a = num – 1

a = 2 – 1

b = fact (1)

Now next as num = 0, we return the value 1.

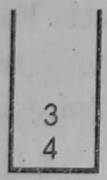
$\therefore$ a = 1, b = 1, we return f = a * b = 1 from function **fact**. Then 1 will be popped off.



Now the top of the stack is the value of a

Then we get b = fact (2) = 1

$\therefore$ f = a * b = 2 * 1 = 2 will be returned.

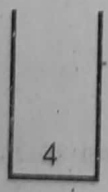


Here a = 3

Then b = fact (3)

b = 2

The stack will be

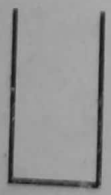


Here a = 4

Then b = fact (4)

= 6

$\therefore$ f = a * b = 24 will be returned.



As now stack is empty. We return f = 24 from the function **fact**.

3.6 : Expression Conversion

Q.9 Write an algorithm for conversion of infix to postfix expression.

☞[SPPU : May-14, Marks 4]

Ans. : The algorithm is as follows -

Read an expression from left to right each character one by one

1. If an operand is encountered then store it in postfix array.
2. If '(' is read, then simply push it onto the stack. Because the (has highest priority when read as an input.
3. If ')' is reads, then pop all the operators until (is read. Discard (. Store the popped characters in the postfix array.
4. If operator is read then
   1. If instack operator has greatest precedence (or equal to) over the incoming operator then pop the operator and add it to postfix expression(postfix array). Repeat this step until we get the instack operator of higher priority than the current incoming operator. Finally push the incoming operator onto the stack
   2. Else push the operator.
5. If stack is not empty then pop all the operators and store in postfix array.
6. Finally print the array of postfix expression.

Q.10 Write a function to convert an infix to postfix expression.

☞ [SPPU : June-22, Marks 9]

Ans. :

```
void postfix()
{
  int i,j=0;
  char ch,next_ch;
  for(i=0;i<strlen(in);i++)
  {
    ch= in[i];
    switch(ch)
    {
```

```
        case '(': push(ch);
                  break;
        case ')':while((next_ch=pop())!='(')
                  post[j++] = next_ch;
                  break;
        case '+':
        case '-':
        case '*':
        case '/':
        case '^':while(!stempty()&&(precedence(stk[top])>
            =precedence(ch)))
                  post[j++]=pop();
                 push(ch);//else part
                 break;
        default:post[j++]=ch;//if operand is encountered simply
        add it to postfix
       }
    }
    while(!stempty())
      post[j++]=pop();
    post[j]='\0';
    for(i=0;i<strlen(post);i++)
      printf("%c",post[i]);
}
```

Q.11 Clearly indicate the contents of stack during conversion of given infix expression to prefix : Infix expression :

A + (B * ((D – E % F) / H))

[SPPU : May-16, Marks 6, Similar June-22, Marks 9]

Ans. : Step 1 : Reverse the given infix expression.

)) H /) F % E – D ((* B (+ A

Step 2 : Make (as), and) as (

((H / (F % E – D)) * B) + A

Step 3 : Now convert this expression to postfix.

Expression	Current Symbol	Action Taken	Stack	Output
↑((H/(F%E-D))*B)+A	(	Push (	(	
↑(H/F%E-D))*B)+A	(	Push (	((	
↑H/(F%E-D))*B)+A	H	Print H	((	H
↑/(F%E-D))*B)+A	/	Push /	((/	H
↑(F%E-D))*B)+A	(	Push (	((/(	H
↑F%E-D))*B)+A	F	Print F	((/(	HF
↑%E-D))*B)+A	%	Push %	((/(%	HF
↑E-D))*B)+A	E	Print E	((/(%	HFE
↑-D))*B)+A	-	POP %, Print then Push -	((/(-	HFE %
↑D))*B)+A	D	Print D	((/(-	HFE % D
↑))*B)+A	)	POP -, Print POP (	((/	HFE % D -

)*B)+A ↑	)	POP /, Print. POP (	(	HFE % D – /
*B)+A ↑	*	Push *	(*	HFE % D – /
B)+A ↑	B	Print B	(*	HFE % D – / B
)+A ↑	)	Pop *, Print . POP (	empty	HFE % D – / B *
+A ↑	+	Push +	+	HFE % D – / B *
A ↑	A	Print A	+	HFE % D – / B * A
End of Input		POP + Print	empty	HFE % D – / B * A +

Now reverse the obtained postfix expression.

The + A * B / – D % EFH be the prefix expression.

Q.12 Convert the following expression from infix to Postfix and Prefix. Make use of appropriate data structure which can be used for conversion.

a. 2*3/(2–1)+5*3 **☞[SPPU : Dec.-18, Marks 6]**

Ans. : Infix to Postfix conversion

Input character read	action	stack	Output
2	Print 2		2
*	Push	*	2
3	Print 3	*	23
/	Pop *, print, push	/	23*

(	Push (	/ (	23*
2	Print 2	/ (	23*2
–	Push –	/ (–	23*2
1	Print	/ (–	23*21
+	POP–, Print POP (POP / , Print Push +	+	23*21 – 1
5	Print 5	+	23*21 – /5
*	Push *	+*	23*21 – /5
3	Print 3	+*	23*21 – /53
End of input	POP *, Print POP +, Print		23*21 – /53*+

The required Postfix expression is,

23 * 21 – / 53 * +

Inflix to Prefix conversion

Reverse given expression

3* 5+) 1–2 (/ 3* 2

Now make) as (and (as)

3* 5+ (1–2)/ 3* 2

Now convert this expression to Postfix form.

Input character read	action	stack	Output
3	Print		3
*	Push *	*	3
5	Print	*	35
+	POP *, Print, Push +	+	35*
(	Push (	+ (	35*
1	Print 1	+ (	35*1
–	Push	+ (–	35*1

2	Print 2	+ (−	35*12
)	POP −, Print POP (	+	35*12 −
/	Push /	+ /	35*12 −
3	Print 3	+ /	35*12 − 3
*	POP /, Print Push *	+ *	35*12 − 3 /
2	Print 2	+ *	35*12 − 3 / 2
End of input	POP *, Print POP +, Print	empty	35*12 − 3 / 2* +

Now reverse the expression obtained.

+ *2/ 3−21 *53

This is required prefix expression.

Q.13 Convert the following expression from Infix to Postfix and Prefix. Make use of appropriate data structure :

2 * 3/(2 – 1) + 5 * 3 ☞[SPPU : May-19, Marks 6]

Ans. : Infix and Postfix

Input Read	Action	Stack	Output
2 * 3 / (2 – 1) + 5 * 3 ↑	Print 2	Empty	2
2 * 3 / (2 – 1) + 5 * 3 ↑	Push *	*	
2 * 3 / (2 – 1) + 5 * 3 ↑	Print 3	*	2 3
2 * 3 / (2 – 1) + 5 * 3 ↑	POP *, Print then push /	/	23 *
2 * 3 /(2 – 1) + 5 * 3 ↑	Push (	/ (	

2 * 3 /(2 – 1) + 5 * 3 ↑	Print 2	/ (	23 * 2
2 * 3 /(2 – 1) + 5 * 3 ↑	Push –	/ (–	23 * 2
2 * 3 /(2 – 1) + 5 * 3 ↑	Print 1	/ (–	23 * 21
2 * 3 /(2 – 1) + 5 * 3 ↑	POP –, Print, POP (	/	23 * 21 –
2 * 3 /(2 – 1) + 5 * 3 ↑	POP /, Print, Push +	+	23 * 21 – /
2 * 3 /(2 – 1) + 5 * 3 ↑	Print 5	+	23 * 21 – / 5
2 * 3 /(2 – 1) + 5 * 3 ↑	Push *	+ *	23 * 21 – / 5
2 * 3 /(2 – 1) + 5 * 3 ↑	Print 3	+ *	23 * 21 – / 5 3
2 * 3 /(2 – 1) + 5 * 3 ↑	POP *, Print POP +, Print	Empty	**23 * 21 – / 53*+**

Infix to prefix

Step 1 : Reverse the given infix expression 3*5 +)1 – 2 (/3*2

Step 2 : Make (as) and) as (3*5 + (1 – 2)/3*2.

Step 3 : Now convert this expression to postfix.

Input Read	Action	Stack	Output
3*5+(1 – 2)/3*2 ↑	Print 3	Empty	3
3*5+(1 – 2)/3*2 ↑	Push	*	3

3*5+(1 – 2)/3*2 ↑	Print 5	*	3 5
3*5+(1 – 2)/3*2 ↑	POP *, Print Push +	+	3 5 *
3*5+(1 – 2)/3*2 ↑	Push (	+ (	3 5 *
3*5+(1 – 2)/3*2 ↑	Print 1	+ (	3 5 * 1
3*5+(1 – 2)/3*2 ↑	Push –	+ (–	3 5 * 1
3*5+(1 – 2)/3*2 ↑	Print 2	+ (–	3 5 * 1 2
3*5+(1 – 2)/3*2 ↑	POP –, Print	+	35 *12 –
3*5+(1 – 2)/3*2 ↑	Push /	+ /	35 *12 –
3*5+(1 – 2)/3*2 ↑	Print 3	+ /	35 *12 – 3
3*5+(1 – 2)/3*2 ↑	Push *	+/*	35 *12 – 3
3*5+(1 – 2)/3*2 ↑	Print 2	+/*	35 *12 – 32
3*5+(1 – 2)/3*2 ↑	POP * Print, POP, /Print POP +, Print	Empty	35 *12 – 32*/+

Now reverse the obtained expression + / * 2 3 – 2 1 * 5 3 is the required prefix expression.

3.7 : Expression Evaluation

Q.14 Write an algorithm for evaluation of postfix expression.

Ans. : Algorithm for evaluation of postfix

1. Read the postfix expression from left to right.
2. If the input symbol read is an operand then push it onto the stack.
3. If the operator is read POP two operands and perform arithmetic operations if operator is

+	then result	=	operand 1 + operand 2
–	then result	=	operand 1 – operand 2
*	then result	=	operand 1 * operand 2
/	then result	=	operand 1 / operand 2

4. Push the result onto the stack.
5. Repeat steps 1-4 till the postfix expression is not over.

Q.15 Write a function to evaluate a postfix expression.

Ans. :

```
*/
double EVAL::post(char exp[])
{
   char  ch,*type;
   double  result, val, op1, op2;
   int i;
   st.top = 0;
   i=0;
   ch = exp[i];

   while ( ch !='$' )
   {
      if ( ch >= '0' && ch <= '9')
      type ="operand";
      else if ( ch == '+' ch == '-'||
       ch == '*'|| ch == '/' ||
```

```
        ch == '^' )
        type="operator";
        if( strcmp(type,"operand")==0)/*if the character is operand*/
        {
        val = ch - 48;
        push( val);
        }
        else
        if (strcmp(type,"operator")==0)/*if it is operator*/
        {
         op2 = pop();
         op1 = pop(); //popping two operands to perform arithmetic
                        operation
         switch(ch)
        {
        case '+' : result = op1 + op2;
          break;
        case '-' : result = op1 - op2;
        break;
        case '*' : result = op1 * op2;
        break;
        case '/' : result = op1 / op2;
        break;
        case '^' : result = pow(op1,op2);
        break;
      }/* switch */
      push(result);
      }
         i++;
         ch=exp[i];
      } /* while */
      result = pop(); /*pop the result*/
      return(result);
}
/*
```

The characters '0', '1', ... '9' will be converted to their values, so that they will perform arithmetic operation.

Finally result will be pushed onto the stack.

Q.16 Clearly indicate the content of stack during evaluation of postfix expression : ab-cd/*e+, where a=8, b=6, c=10, d=5 and e=7.

☞[SPPU : Dec.-14, Marks 6]

Ans. :

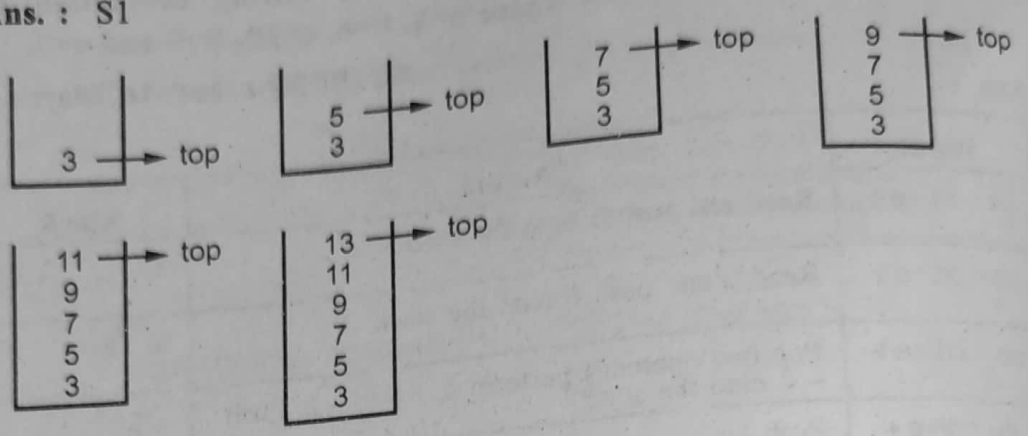
Input	Action	Stack
ab - cd/*e +	Read and push 8 onto the stack	8
ab - cd/*e +	Read b and push 6 onto the stack	8 6
ab - cd/*e +	Pop two operands perform 6 – 8 = – 2. Push – 2 onto the stack	– 2
ab - cd/*e +	Push 10	– 2 10
ab - cd/*e +	Push 5	– 2 10 5
ab - cd/*e +	Pop two operands perform 10/5 = 2. Push 2 onto the stack	– 2 2
ab - cd/*e +	Pop two operands perform – 2 * 2 = – 4 push – 4 onto the stack	– 4
ab - cd/*e+	Push 7	– 4 7
ab - cd/*e+	Pop two operands perform 7 + (– 4) = 3. Push 3 onto stack	3

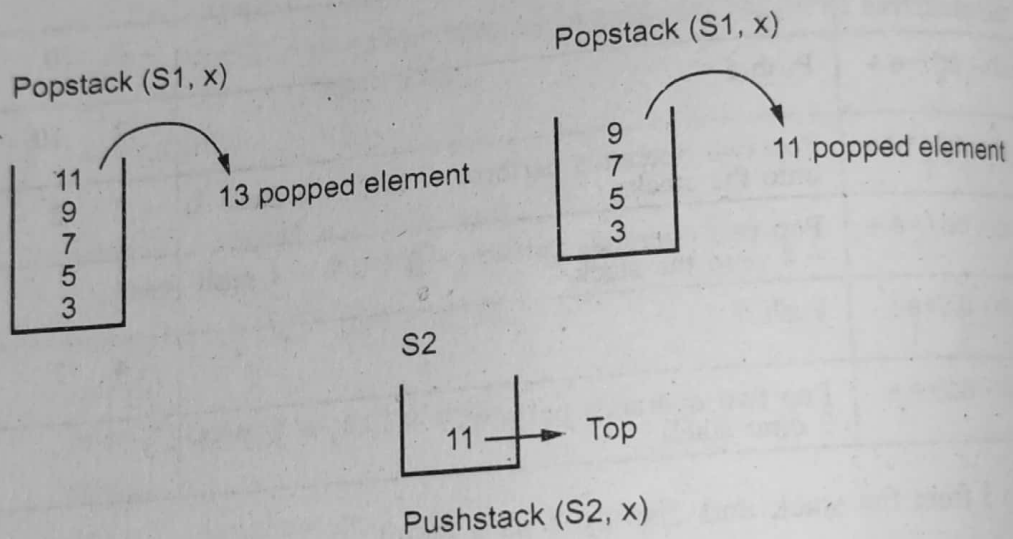
Pop 3 from the stack and display **3** as a result.

Q.17 Imagine we have two empty stacks of integers, S1 and S2. Draw a picture of each stack after the following operations :

```
Pushstack (S1, 3);
Pushstack(S1, 5)
Pushstack(S1, 7);
Pushstack(S1, 9);
Pushstack(S1, 11);
Pushstack(S1, 13);
While(!emptystack(S1))
{
Popstack(S1, x);
Popstack(S1, x);
Puphstack(S2, x);
}
```

Ans. : S1





Pushstack (S2, x)

Q.18 Evaluate given expressions : i) 2 3 * 2 1 – / 5 3 * +
ii) + * 2 / 3 – 2 1 * 5 3

Ans. : i) 2 3 * 2 1 – / 5 3 * +

Input Read	Action Taken	Stack
2	Push	2
3	Push	2 3
*	POP 3, POP 2, perform 2*3 = 6. Push 6	6
2	Push 2	6 2
1	Push 1	6 2 1

-	POP 1, POP 2, perform 2 – 1 = 1. Push 1	6 1
/	POP 1, POP 6, perform 6/1 = 6. Push 6	6
5	Push 5,	6 5
3	Push 3	6 5 3
*	POP 3, POP 5 perform 5*3 =15. Push 15	6 15
+	POP 15, POP 6 perform 6+15 = 21. Push 21	21
End of input	POP 21 and Print it as result	

Ans. : 21

ii) + * 2 / 3 – 21 * 53

+*2/3–21*53	
+*2/3(–21)*53	(-21) = (2-1) = 1
+*2/3 1(*53)	(*5 3) = (5*3) = 15
+*2 (/ 3 1)15	(/ 3 1) = (3/1) = 3
+(*2 3)15	(*23) = (2*3) = 6
(+6 15)	(+ 6 15) = (6+15) = 21
21	

Q.19 If the values of A, B, C and D are 2, 3, 4 and 5 respectively, calculate the value of the following postfix expressions :
i) AB * C – D + ii) ABC + * D – ☞**[SPPU : May-19, Marks 6]**

Solution : i) AB * C – D +

Input	Action	Stack
A	Read A and Push 2	2
B	Read B and Push 3	2 3
*	POP two operands and perform 2 * 3 = 6. Push 6.	6
C	Read C and Push 4	6 4

–	POP two operands and perform 6 – 4 = 2. Push 2.	2
D	Read D and Push 5	2 5
+	POP two operands and perform 2 + 5 = 7. Push 7.	7
End of Input	POP and Print 7 as result	Empty

ii) ABC + * D –

Input	Action	Stack
A	Read A, and Push 2	2
B	Read B, and Push 3	2 3
C	Read C, and Push 4	2 3 4
+	POP two operands and perform 3 + 4 = 7. Push 7.	2 7
*	POP two operands and perform 2 * 7 = 14. Push 14.	14
D	Read D, and Push 5	14 5
–	POP two operands and perform 14 – 5 = 9 Push 9.	9
End of Input	POP and Print 9 as result	Empty

3.8 : Concept of Queue as ADT

Q.20 Explain the Linear data structure - Queue.

☞[SPPU : May-19, Marks 4]

Ans. : The queue can be formally defined as ordered collection of elements that has two ends named as **front** and **rear**. From the front end one can delete the elements and from the rear end one can insert the elements.

For Example :

The typical example can be a queue of people who are waiting for a city bus at the bus stop. Any new person is joining at one end of the queue, you can call it as the rear end. When the bus arrives the person at the other end first enters in the bus. You can call it as the front end of the queue.

Following Fig. Q.20.1 represents the queue of few elements.

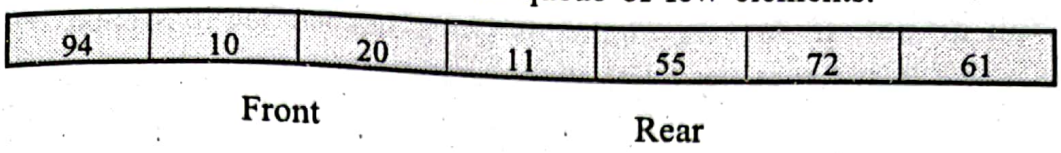


Fig. Q.20.1 Queue

Q.21 Give an ADT for queues.

Ans. :

AbstractDataType Queue

{

Instances : The queue is a collection of elements in which the element can be inserted by one end called rear and elements can be deleted by other end called front.

Operations :

1. Insert : The insertion of the element in the queue is done by the end called rear. Before the insertion of the element in the queue it is checked whether or not the queue is full.
2. Delete : The deletion of the element from the queue is done by the end called front. Before performing the delete operation it checked whether the queue is empty or not.

}

3.9 : Implementation of Queue using Arrays and Linked Organization

Q.22 Explain the insertion and deletion operation of queue implemented using arrays.

Ans. : 1. Insertion of element into the queue

The insertion of any element in the queue will always take place from the rear end.

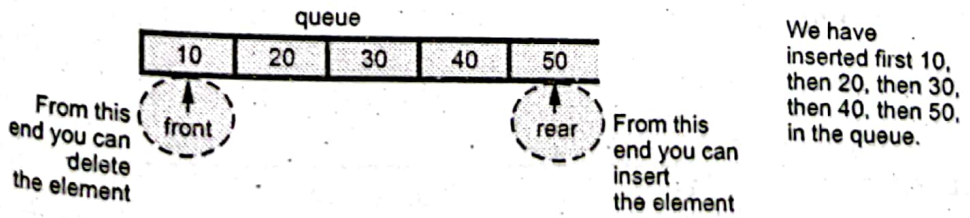


Fig. Q.22.1 Representing the insertion

Before performing insert operation you must check whether the queue is full or not. If the rear pointer is going beyond the maximum size of the queue then the queue overflow occurs.

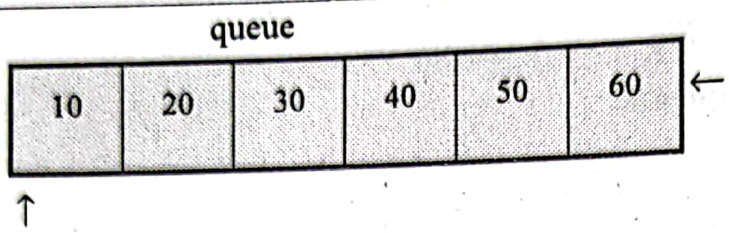


Fig. Q.22.2 Representing the queue overflow

2. Deletion of element from the queue

The deletion of any element in the queue takes place by the front end always.

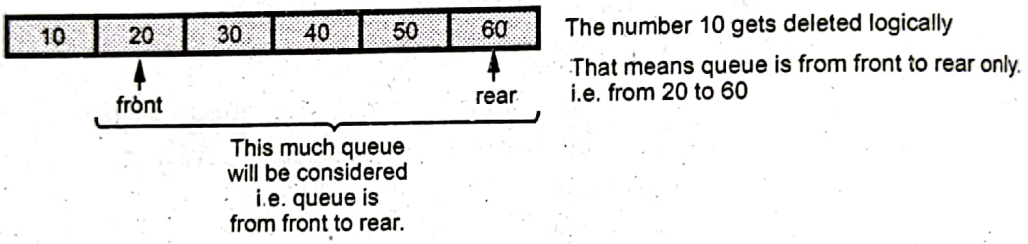


Fig. Q.23.3 Representing the deletion

Before performing any delete operation one must check whether the queue is empty or not. If the queue is empty, you can not perform the deletion. The result of illegal attempt to delete an element from the empty queue is called the queue underflow condition.

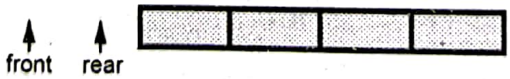


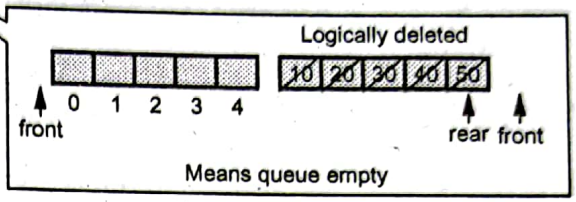
Fig. Q.24.4 Representing the queue underflow

```
int MyQ::insert(int item)
{
 if(Q.front ==-1)
 Q.front++;
```

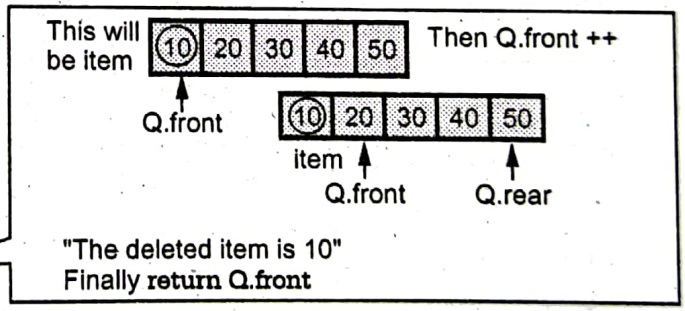
This condition will occur initially when queue is empty

```
 Q.que[++Q.rear] = item;
 return Q.rear;
}
int MyQ::Qempty()

{
if((Q.front == – 1) || (Q.front > Q.rear))
return 1;
else
return 0;
}
```

Always increment the rear pointer and place the element in the queue.




```
/*
The delete Function
Input:none
Output:front value
Called By:main
Calls:none
*/
int MyQ::delet()

{
int item;
item = Q.que[Q.front];
Q.front++;
cout<<"\n The deleted item is "<<item;
return Q.front;
}
```



Q.23 Write insertion and deletion function to implement a linked queue.

Ans. :

```
void Lqueue::insert()
   {
    char ch;
```

```
Q *temp;
clrscr();
temp =new Q;//allocates memory for temp node
temp->next=NULL;
cout<<"\n\n\n\tInsert the element in the Queue\n";
cin>>temp->data;

if(front == NULL)//creating first node
{
 front= temp;
 rear=temp;
 }
 else              //attaching other nodes
  {
    rear->next=temp;
    rear=rear->next;
    }
}
-------------------------------------------------------
The delet Function
-------------------------------------------------------
*/
Q *Lqueue::delet()
{
 Q *temp;
 temp=front;
 if(Qempty(front))
  {
    cout<<"\n\n\t\tSorry!The Queue Is Empty\n";
    cout<<"\n Can not delete the element";
  }
  else
   {
     cout<<"\n\tThe deleted Element Is  "<<temp->data;
     front=front->next;
```

```
    temp->next=NULL;
    delete temp;
  }
return front;
}
```

3.10 : Concept of Circular Queue

Q.24 What is circular queue ?

Ans. : Circular queue is a data structure in which rear is adjacent to front element.

There is a formula which has to be applied for setting the front and rear pointers, for a circular queue.

```
rear = (rear + 1) % size
front = (front + 1) % size
```

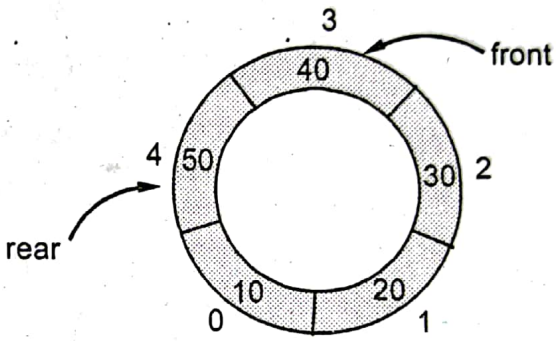


Fig. Q.24.1 Circular queue

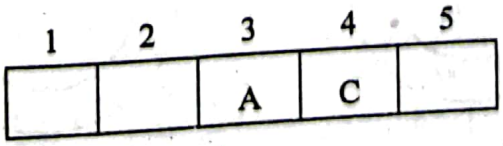
$$rear = (rear + 1) \% size$$
$$= (4 + 1) \% 5$$
$$rear = 0$$

So we can store the element 60 at 0^{th} location similarly while deleting the element.

$$front = (front + 1) \% size$$
$$= (3 + 1) \% 5$$
$$front = 4$$

So delete the element at 4^{th} location i.e. element 50

Q.25 Consider following circular queue of characters and size 5.



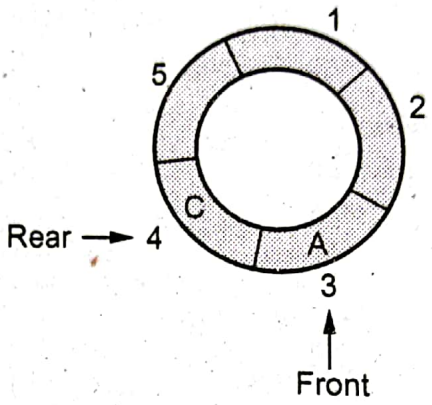
Front point to A and rear points to C

Show the queue contents as per the following operations at every step.

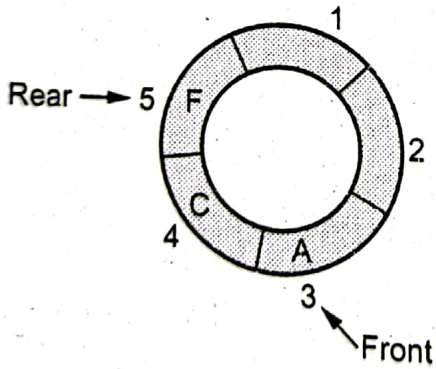
i) F is added to the queue.
ii) Two letters are deleted.
iii) K, L, M are added to the queue.
iv) Two letters are deleted.
v) R is added to the queue.
vi) Two letters are deleted.
vii) R is added to the queue.
viii) Two letters are deleted.

☞[SPPU : May-14,16, Marks 8]

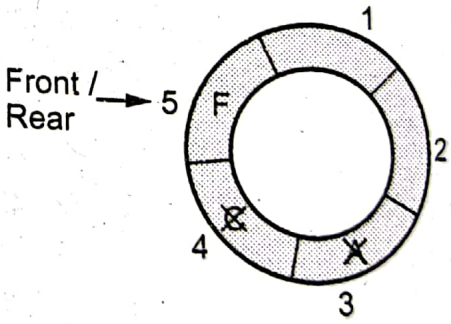
Ans. : Consider the circular representation as,



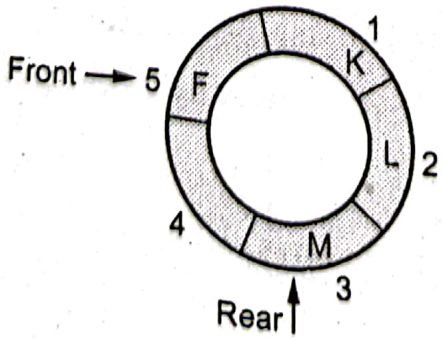
Step 1 : F is added



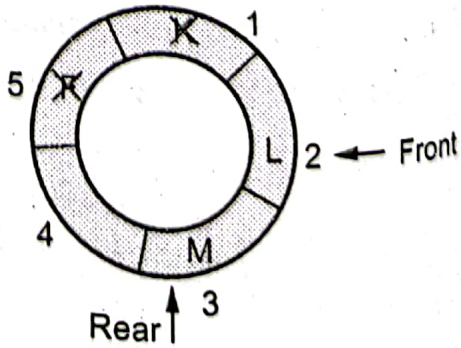
Step 2 : Two letters deleted



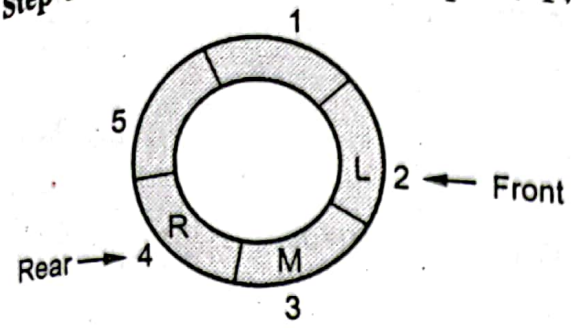
Step 3 : K, L, M are added



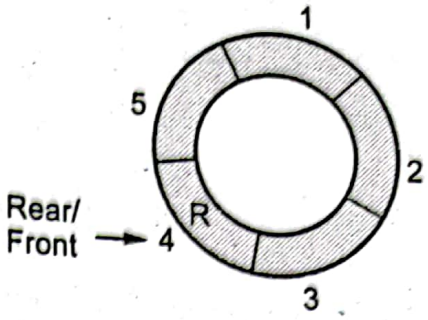
Step 4 : Two letters are deleted



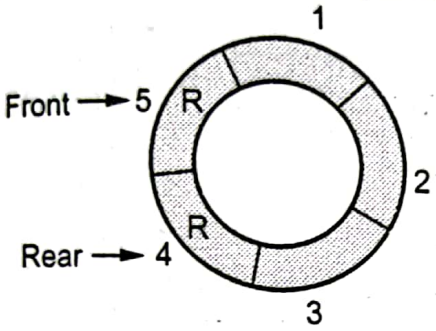
Step 5 : R is added



Step 6 : Two letters deleted



Step 7 : R is added to the queue



Step 8 : Two letters are deleted

The queue becomes empty.

Q.26 What is the difference between circular queue and linear queue ?

Ans. :

Sr. No.	Linear Queue	Circular Queue
1.	The elements are arranged in linear sequential manner, one after the other.	The elements are arranged in circular manner the last element is adjacent to the first element of the queue.
2.	It requires more memory.	It requires less memory.
3.	Less efficient	More efficient

3.11 : Double Ended Queue

Q.27 Write a short note on - Double ended queue.

Ans. : Double ended queue is a data structure in which data can be inserted from both the rear and front end. Similarly data can be deleted from both the ends rear and front.

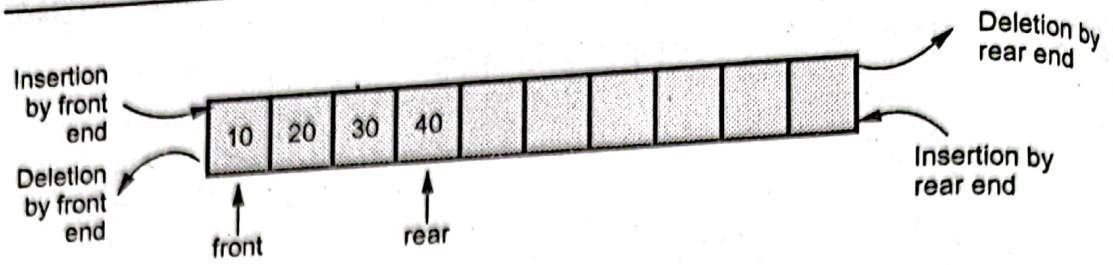
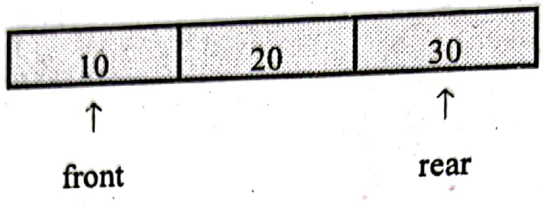


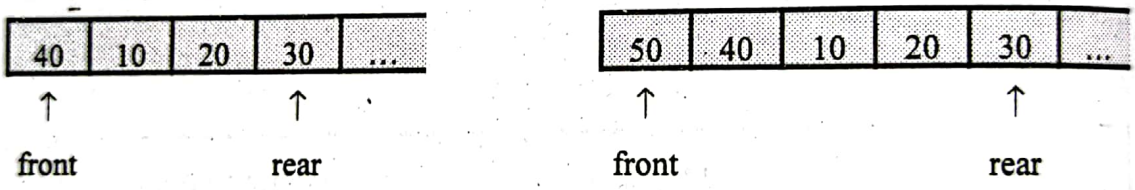
Fig. Q.27.1 Doubly ended queue

As we know, normally we insert the elements by rear end and delete the elements from front end. Let us say we have inserted the elements 10, 20, 30 by rear end.



Now if we wish to insert any element from front end then first we have to shift all the elements to the right.

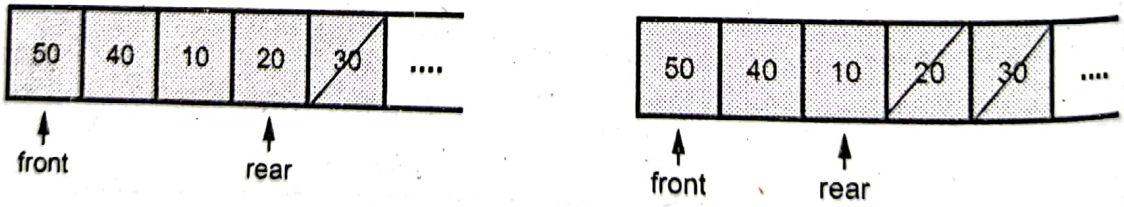
For example if we want to insert 40 by front end then, the deque will be



Inserted 40 in the queue by front end

Inserted 50 in the queue by front end

(a) Insertion by front end



(b) Deletion by rear end

Fig. Q.27.2 Operations on deque

We can place –1 for the element which has to be deleted.

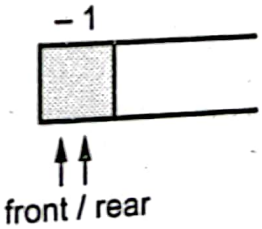
Q.28 The dq is an input restricted doubly ended queue, implemented as a linear queue. The delete functions return the element that is

deleted and also stores 0 (zero) at the location of deleted element. Clearly indicate the contents of queue after each add and delete operation given below :

```
struct dqueue
{
    int front
    in arr[10];
    int rear;
};
Struct dque dq = {–1, {0}, –1};
int i = 0;
addqatend (&dq, 11);
addqatend (&dq, 12);
addqatend (&dq, 13);
i = delqatbeg (&dq);
i = delqatbeg (&dq);
i = delqatend (&dq);
addqatend (&dq, 22);
addqatend (&dq, 23);
i = delqatbeg (&dq);
i = delqatend (&dq);
addqatend (&dq, 24);
```

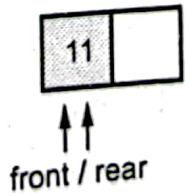
☞[SPPU : May-16, Marks 6]

Ans. : Step 1 : Initial Configuration



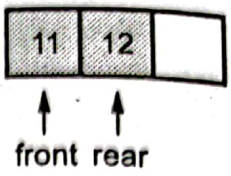
Step 2 :

addatend (& dq, 11)



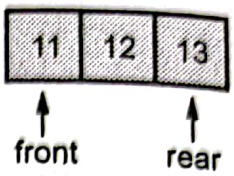
Step 3 :

addatend (& dq, 12)



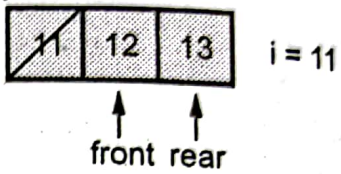
Step 4 :

addatend (& dq, 13)



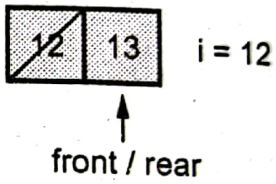
Step 5 :

i = delatbeg (& dq)



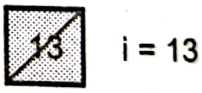
Step 6 :

i = delatbeg (& dq)



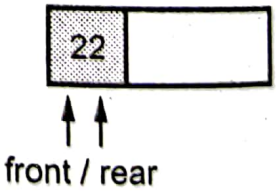
Step 7 :

i = delatbeg (& dq)



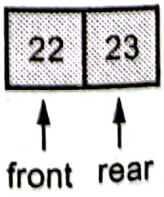
Step 8 :

addatend (& dq, 22)



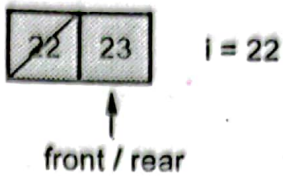
Step 9 :

addatend (& dq, 23)



Step 10 :

i = delqatend (& dq)

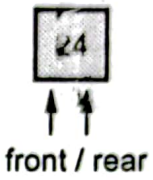


Step 11 :

i = delqatend (& dq)

Step 12 :

i = addatend (& dq, 24)



3.12 : Applications of Queue : Priority Queue

Q.29 Write short note on - Priority queue.

☞[SPPU : Dec.-19, Marks 3]

Ans. : The priority queue is a data structure having a collection of elements which are associated with specific ordering.

Application of Priority Queue

1. The typical example of priority queue is scheduling the jobs in operating system. Typically operating system allocates priority to jobs. The jobs are placed in the queue and position1 of the job in priority queue determines their priority. In operating system there are three kinds of jobs. These are real time jobs, foreground jobs and background jobs. The operating system always schedules the real time jobs first. If there is no real time job pending then it schedules foreground jobs. Lastly if no real time or foreground jobs are pending then operating system schedules the background jobs.
2. In network communication, to manage limited bandwidth for transmission the priority queue is used.

3. In simulation modeling, to manage the discrete events the priority queue is used.

Types of Priority Queue

The elements in the priority queue have specific ordering. There are two types of priority queues –

1. **Ascending Priority Queue** – It is a collection of items in which the items can be inserted arbitarily but only smallest element can be removed.
2. **Descending Priority Queue** – It is a collection of items in which insertion of items can be in any order but only largest element can be removed.

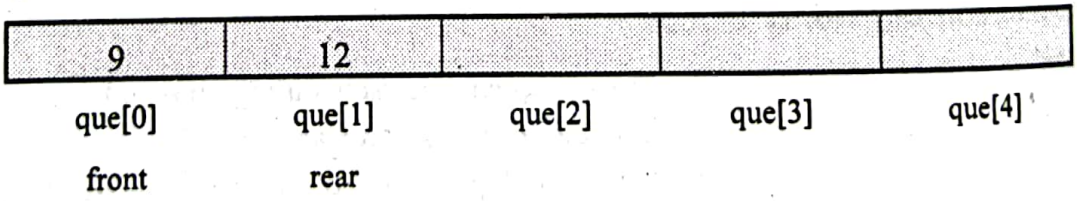
In priority queue, the elements are arranged in any order and out of which only the smallest or largest element allowed to delete each time.

Q.30 What is priority queue ? What is its use ? Give the function to add an element in priority queue.

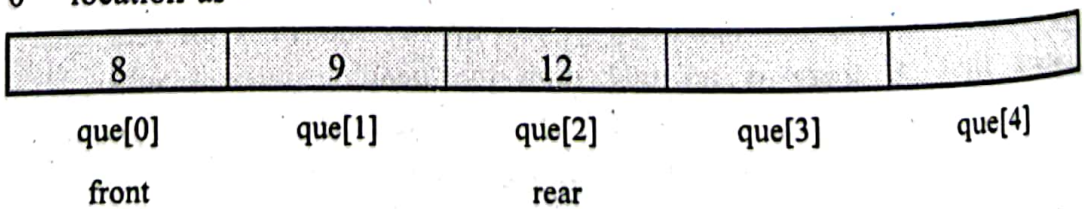
☞[SPPU : Dec.-15, Marks 6, May-10, Marks 10]

Ans. : Refer Q.29.

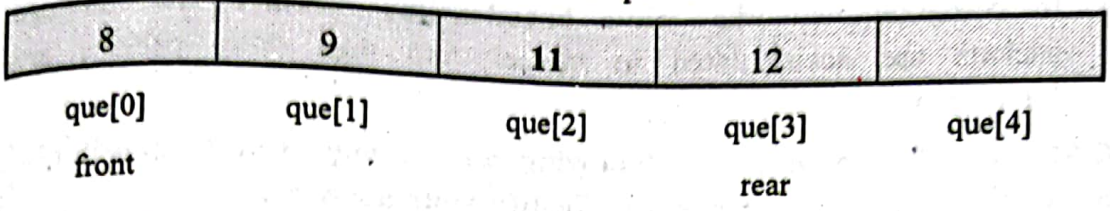
While implementing the priority queue we will apply a simple logic. That is while inserting the element we will insert the element in the array at the proper position. For example if the elements are placed in the queue as -



And now if an element 8 is to be inserted in the queue then it will be at 0^{th} location as -



If the next element comes as 11 then the queue will be -



The C++ function for this operation is as given below -

```
int Pr_Q::insert(int rear,int front)
{
   int item,j;
   cout<<"\nEnter the element: ";
   cin>>item;
   if(front ==-1)
         front++;
   j=rear;
   while(j>=0 && item<que[j])
{
    que[j+1]=que[j];
    j--;
   }
   que[j+1]=item;
   rear=rear+1;
   return rear;
}
```

3.13 : Applications of Queue

Q.31 List down applications of queues. ☞ **[SPPU : May-13, Marks 4]**

Ans. : 1. In operating system for **scheduling jobs**, priority queues are used.

2. In Local Area Network (LAN) multiple computers share few printers. Then these **printers** accumulate jobs in a queue. Thus printers can process multiple print commands with the help of queues.

3. For **categorizing data**, queues are used.

4. In **simulation and modeling** queues are used.
5. In computer networks, while **broadcasting message**, the message packets are accumulated in queue. And then these packets are forwarded.

Q.32 Specify which of the following applications would be suitable for a first-in-first-out queue and Justify your answer :
i) A program is to keep track of patients as they check into a clinic, assigning them to doctors on a first come, first-served basis.
ii) An inventory of parts is to be processed by part number.
iii) A dictonary of words used by spelling checker is to be created.
iv) Customers are to take numbers at a bakery and be served in order when their numbers come-up.

☞[SPPU : May-11, Dec.-12, Marks 4, May-15, Marks 6]

Ans. :

i) No, this application will not be suitable for first come first serve queue because sometimes some serious patients may be come and he/she needs doctors in urgent.

ii) Yes, for this application queue is suitable because here the part no must be stored in specific manner (order).

iii) No, because words can be chosen in random order.

iv) Yes, at bakery customers must be serve if first come first serve basis.

END...