CHAPTER 2 Stack and Queues

University Prescribed Syllabus

Introduction, ADT of Stack, Operations on Stack, Array Implementation of Stack, Applications of Stack-Well form-ness of Parenthesis, Infix to Postfix Conversion and Postfix Evaluation, Recursion.

Introduction, ADT of Queue, Operations on Queue, Array Implementation of Queue, Types of Queue-Circular Queue, Priority Queue, Introduction of Double Ended Queue, Applications of Queue.

Module 2

Syllabus Topic : Introduction to Stack

2.1 INTRODUCTION TO STACK

GQ. 2.1.1 *What is stack ?* **(1 Mark)**

- In C programming language, we have seen the concept of an array in which we can store multiple elements at a time.
- Array is considered as a flexible data structure. That means we can add or remove elements anywhere in the array.
- But such flexibility is problematic because at the end there will be no track that how, where and when elements are added or removed.
- Hence to restrict the transactions on data elements, Data Structure provides various standards formats such as stack, queue, tree, graph etc.
- These formats have their own rules and regulations while performing operations on data elements. Hence track of records is maintained properly.

> **Definition :** *Stack is data structure in which addition and removal of an element is allowed at the same end called as top of the stack.*

- We can imagine the example of book shelf in which books are placed one above other. Now if we want to keep any new book in the shelf, then that can be easily kept at the top and also the topmost book can be picked first.
- Stack is also known as LIFO (Last In First Out) list. That means the element which gets added at last will be removed first.
- Fig. 2.1.1 illustrates the structure and operations of stack.

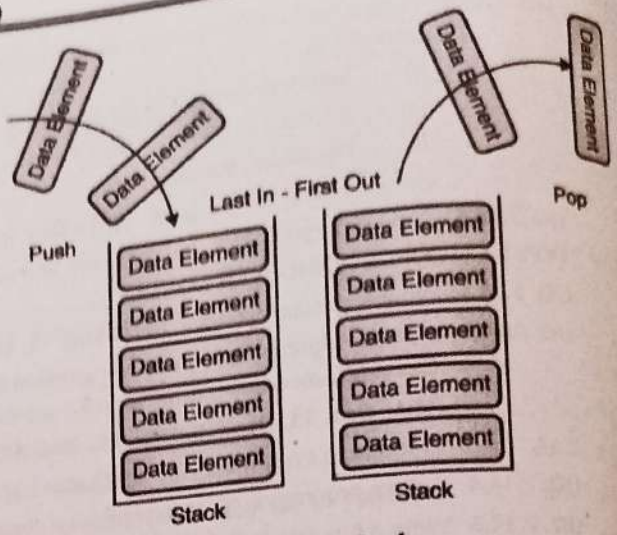


Fig. 2.1.1 : Stack

Syllabus Topic : ADT of Stack : Operations on Stack

2.2 ADT OF STACK : OPERATIONS ON STACK

UQ. 2.2.1 *Explain STACK as ADT.*

MU - Dec. 15, Dec. 16, 3 Marks

- The stack is an abstract data type which is defined by the following structure and operations.
- Following operations are performed on stack :

☞ **Operations on Stack**

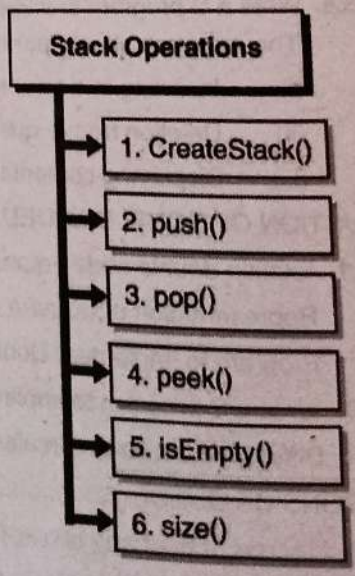


Fig. 2.2.1 : Stack Operations

1. CreateStack()

It creates a new empty stack. It does not require any parameter and returns an empty stack.

2. push(item)

It inserts new element at the top of the stack. It needs the item to be added and returns nothing.

☞ **Algorithm of push**

```
Step 1 : If TOP >= SIZE – 1 then
         Write "Stack is Overflow"
Step 2 : TOP = TOP + 1
Step 3 : STACK [TOP] = X
```

3. pop()

It removes the topmost element from the stack. It does not require any parameter and returns the item. The stack is modified.

☞ **Algorithm of pop**

```
Step 1 : If TOP = -1 then
         Write "Stack is Underflow"
Step 2 : Return STACK [TOP]
Step 3 : TOP = TOP - 1
```

4. peek()

It returns the topmost element from the stack but does not remove it. It does not require any parameter. The stack is not modified.

5. isEmpty()

– It is used to test whether the stack is empty or not. It does not require any parameter and returns a Boolean value.

6. size()

– returns the number of items in the stack. It does not require any parameter and returns an integer.

☞ **Example**

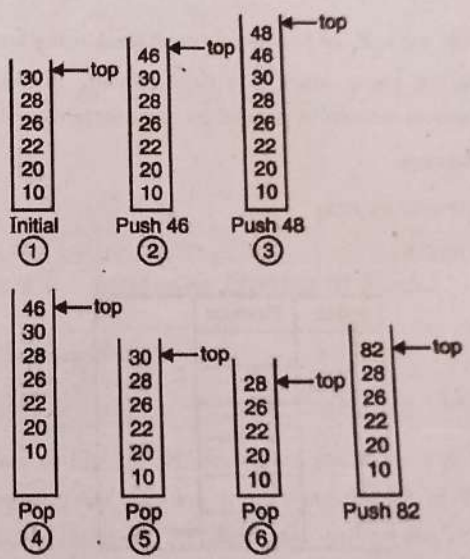
– Consider s is a stack that has been created and starts out empty, then in the following table we can observe the results of a sequence of **stack operations.**

Table 2.2.1 : Stack operations

Stack Operation	Stack Content	Return Value
s.isEmpty()	[ ]	True
s.push("A")	[A]	
s.push(5)	[A, 5]	
s.peek()	[A, 5]	5
s.push("B")	[A, 5, B]	
s.size()	[A, 5, B]	3
s.pop()	[A, 5]	B

Module 2

GQ. 2.2.2 *The stack contain 10, 20, 22, 26, 28, 30, with 30 being at top of the stack. Show diagrammatically the effect of PUSH 46 PUSH 48 POP POP POP PUSH 82.* **(3 Marks)**



Syllabus Topic : Array Implementation of Stack

2.3 ARRAY IMPLEMENTATION OF STACK

GQ. 2.3.1 *Explain the concept of representing stack through arrays.* **(4 Marks)**

– Stack is generally represented in two ways : **static** and **dynamic.**

– The static implementation of stack is done using array.

- Dynamic representation is done using linked list.
- Array is considered as a static data structure because size of array is fixed.

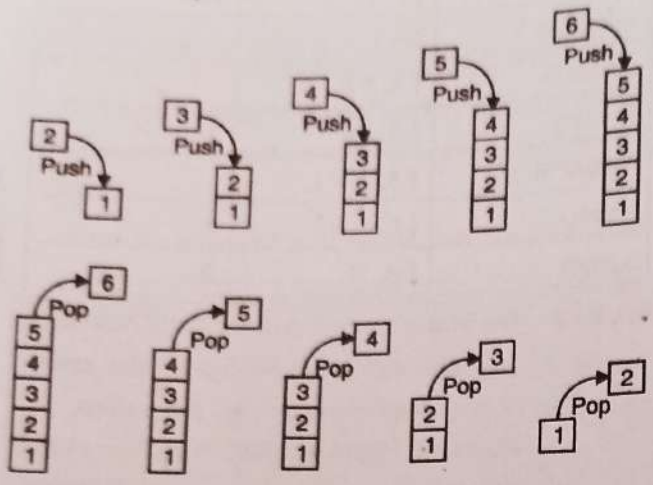


Fig. 2.3.1 : Representation of stack through array

- Now we will see how to represent stack using array :

  As we know, stack is a data structure in which the topmost element is pointed by the integer variable **top**.

Example

Consider an array

```
int arr[5];
```

Index	Element	
4		
3		
2		
1		
0		
-1		← top

Fig. 2.3.2 : Empty Stack

- Now initially as stack is empty, the top is pointing to -1 position. That means we can consider that if top is at -1 position, stack is empty.
- As elements are added in the stack, the top is incremented by one every time. Hence as per stack convention, the top will always points to the upper element.
- Consider we have added three elements in array say 11,12,13 then the structure will be :

3		
2	13	← top
1	12	
0	11	
-1		

Fig. 2.3.3 : Partially filled Stack

- If the top is at size_of_arry-1 position, then stack is considered as full. In this array we have added two more elements say 14 and 15.
- Now as the top is at position 4 i.e size of array – 1, the stack is considered as full.

Index	Element	
4	15	← top
3	14	
2	13	
1	12	
0	11	
-1		

Fig. 2.3.4 : Full Stack

- The above explanation indicates that when top is at arry_size-1 position, we cannot add element in it as it is full and if top is at position -1, we cannot remove element from it as it is empty.

2.3.1 Initializing Stack

- After stack creation, we initialize it by setting the top at the position -1.

```
void initstack ()
{
        stack[top] = -1 ;
}
```

2.3.2 Inserting Element in the Stack

GQ. 2.3.2 Write a procedure to PUSH on element on stack. **(5 Marks)**

- The insertion of element in the stack is known as push. Before inserting element in stack, first we have to check whether it is full or not.
- While adding an element, if the stack is found full, then this situation is considered as **stack overflow.** Refer Fig. 2.3.4.

Function to check whether stack is full or not

```
int is_full()
{
 if(top == SIZE-1)
        return(1);
    else
    return(0);
}
```

Function to add (push) element

```
void push()
{
  if(is_full() == 1)
  {
     printf("\n\tSTACK is over flow");
  }
  else
  {
     printf(" Enter an element to add in the stack :");
     scanf("%d",&ele);
     top++;
     stack[top] = ele;
  }
}
```

2.3.3 Deleting Element from the Stack

GQ. 2.3.3 *Explain POP operation on stack using array representation.* **(5 Marks)**

- The deletion of element from the stack is known as pop. Before deleting an element from the stack, we have to check whether it is empty or not.
- While deleting an element, if the stack is found empty, then this situation is considered as **stack underflow.** See Fig. 2.3.2.

Function to check whether stack is empty or not

```
int is_empty()
{
 if(top == -1)
        return(1);
  else
     return(0);
}
```

Function to delete (pop) an element

```
void pop()
{
  if(is_empty() == 1)
  {
     printf("\n\tSTACK is under flow");
  }
  else
  {
          printf("\n Element popped : %d",stack[top]);
     top--;
  }
}
```

2.3.4 Displaying Element of Stack

GQ. 2.3.4 *Explain "Displaying stack elements".* **(5 Marks)**

- Now to display elements from stack, we use loop and begins from position top to position 0 in the array. Hence elements are displayed in reverse order from which they are added and because of it stack is known **LIFO** structure.
- Here first we have to check whether stack is empty or not.

```
void display()
{
     int i;
  if(is_empty() == 1)
  {
     printf("\n\tSTACK is under flow");
```

Module 2

```
    }
    else
    {
            printf("\n Stack elements : ");
            for(i = top;i >=0;i--)
             {
              printf(" %d ",stack[i]);
             }
    }
}
```

Now we will implement all the operations in a single program.

2.3.5 Algorithm of Program to Demonstrate Stack Implementation using Array

Step 1 : Start

Step 2 : Display Menu : 1:push, 2:pop, 3:display, 4 :exit.

Step 3 : Read choice

Step 4 : if choice 1

Call push function

if choice 2

Call pop function

if choice 3

Call push display

if choice 4

exit from program

default : Display invalid choice

Step 5 : Read choice again

Step 6 : if choice between 1 – 3

Repeat step 4

Else

Stop

2.3.6 Program to Demonstrate Stack Implementation using Array

GQ. 2.3.5 Write a menu driven 'C' program for implementing stack using array. **(10 Marks)**

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
int stack[SIZE],choice,top,ele,i;
int is_full();
int is_empty();
void push();
void pop();
void display();

void main()
{

  top = -1; // initialization

  printf("\n\t STACK OPERATIONS USING ARRAY");
  printf("\n\t----------------------------");
  printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");

   do
{
   printf("\n Enter your Choice : ");
   scanf("%d",&choice);
   switch(choice)
   {
   case 1;
   {
     push();
     break;
   }
   case 2:
   {
```

Function prototyping

```
        pop();
        break;
      }
      case 3:
      {
        display();
        break;
      }
      case 4:
      {
        printf("\n\t EXIT POINT ");
        break;
      }
      default:
      {
        printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
      }
      getch();
      }
   }
   while(choice!=4);
}
```

```
int is_full()
{
    if(top == SIZE-1)
        return(1);
    else
    return(0);
}
```

If top is at last position, means stack is full.

```
void push()
{
     if(is_full() == 1)
     {
         printf("\n\tSTACK is over flow");
     }
     else
     {
     printf(" Enter an element to add in the stack :");
     scanf("%d",&ele);
     top++;
     stack[top] = ele;
     }
}
```

Increments top by one and element is added above last element

```
int is_empty()

{
    if(top == -1)
        return(1);
    else
    return(0);
}
```

If top is at -1 position, means stack is empty.

```
void pop()
{
   if(is_empty() == 1)
   {
      printf("\n\tSTACK is under flow");
   }
   else
   {
         printf("\n Element popped : %d",stack[top]);
      top--;
   }
}
```

Decrements top by one. Element is logically removed from the stack because last element is always considered at position top.

```
void display()
{

   if(is_empty() == 1)
   {
      printf("\n\tSTACK is under flow");
   }
   else
```

Module 2

```
    {
        printf("\n Stack elements : ");
for(i = top;i >=0;i--)
    {
        printf(" %d ",stack[i]);
    }
  }
}
```

Displays element from top to 0

Output

```
        STACK OPERATIONS USING ARRAY
        ---------------------------
        1.PUSH
        2.POP
        3.DISPLAY
        4.EXIT
Enter your Choice : 1
Enter an element to add in the stack :11

Enter your Choice : 1
Enter an element to add in the stack :12

Enter your Choice : 3

Stack elements :  12  11
Enter your Choice : 2

Element popped : 12
Enter your Choice : 3

Stack elements :  11
Enter your Choice : 4_
```

UQ. 2.3.6 *Write a function in C to maintain 2 stacks in a single array.*

MU - Dec. 15, 10 Marks

```
#include<stdio.h>
#include<stdlib.h>
main()
{
    int n,top1,top2,ch=1,a,i,arr[100];
    printf("Enter size of array you want to use\n");
    scanf("%d",&n);
    top1=-1;
    top2=n;
    while(ch!=0)
    {
        printf("What do u want to do?\n");
        printf("1.Push element in stack 1\n");
        printf("2.Push element in stack 2\n");
        printf("3.Pop element from stack 1\n");
        printf("4.Pop element from stack 2\n");
        printf("5.Display stack 1\n");
        printf("6.Display stack 2\n");
        printf("0.EXIT\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
            {
                printf("Enter the element\n");
                scanf("%d",&a);
                if(top1!=(top2-1))
                arr[++top1]=a;
                else
                printf("Overflow\n");
                break;
            }
case 2:
{
    printf("Enter the element\n");
    scanf("%d",&a);
    if(top2!=(top1+1))
    arr[--top2]=a;
    else
    printf("Overflow\n");
    break;
}
case 3:
{
    if(top1==-1)
    printf("Stack1 is empty\n");
    else
    {
        a=arr[top1--];
        printf("%d\n",a);
    }
break;
}
```

```
case 4:
    {
        if(top2==n)
        printf("Stack2 is empty\n");
        else
        {
            s=arr[top2++];
            printf("%d\n",a);
        }
break;
}
case 5:
{
    if(top1==-1)
    printf("Stack1 is empty\n");
    else
    {
        printf("Stack1 is-->>\n");
        for(i=0;i<=top1;i++)
        printf("%d ",arr[i]);
        printf("\n");
    }
break;
}
case 6:
{
    if(top2==n)
    printf("Stack2 is empty\n");
    else
    {
        printf("Stack2 is-->>\n");
        for(i=(n-1);i>=top2;i--)
        printf("%d ",arr[i]);
        printf("\n");
    }
break;
}
case 0:
break;
    }
}
getch();
}
```

Output

```
Enter size of array you want to use
5
What do u want to do?
1.Push element in stack 1
2.Push element in stack 2
3.Pop element from stack 1
4.Pop element from stack 2
5.Display stack 1
6.Display stack 2
0.EXIT
1
Enter the element
11
What do u want to do?
1.Push element in stack 1
2.Push element in stack 2
3.Pop element from stack 1
4.Pop element from stack 2
5.Display stack 1
6.Display stack 2
0.EXIT
```

Module 2

Syllabus Topic : Applications of Stack

2.4 APPLICATIONS OF STACK

UQ. 2.4.1 *Give applications of stack.*

MU - Dec. 16, 3 Marks

- Up till now we understood the basic concept and structure of stack. Stack is a very useful data structure in managing data.
- There are various types of applications of stack :

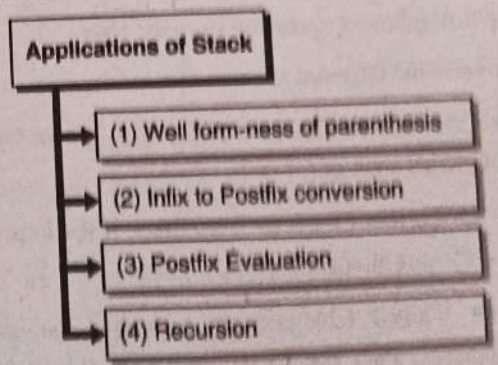


Fig. 2.4.1 : Applications of Stack

Syllabus Topic : Well Form-ness of Parenthesis

2.5 WELL FORM-NESS OF PARENTHESIS

- A classic use of a stack is for determining whether a set of parenthesis is "well formed".
- What exactly do we mean by well formed? In the case of parenthesis pairs, for an expression to be well formed, a closing parenthesis symbol must match the last unmatched opening parenthesis symbol and all parenthesis symbols must be matched when the input is finished.
- Why do we care about balancing parenthesis? Because the compiler looks for balanced parenthesis in such situations as:
- Nested components (such as for loops, if-then statements, and while loops). The compiler checks for matching {} pairs to denote blocks of code. If the pairs do not match, a compiler error will occur.
- Mathematical expressions.

  For example, a = b - {(x + y) * z + 3};
- array notation [ ].
- Function calls and function definitions. Specifically, the parameters must be surrounded by balanced "(" ")".

2.5.1 Algorithm to Check Well Form-ness of an Expression

GQ. 2.5.1 *Write an Algorithm to check well form-ness of an expression.* **(5 Marks)**

- Declare A Stack.
- Input Algebraic Expression from the User
- Traverse the Expression
- Push the Current Character to Stack if it is an Opening Parantheses such as (, [or {.
- Pop the Current Character from Stack if the Expression has a Closing Bracket such as),] or }.
- If the Popped Character is matching the starting parantheses, then the Expression is Balanced else it includes Unbalanced Parantheses.
- After Traversal is completed, if there is any remaining Left Bracket in the Stack, then the Parantheses are not Balanced.

2.5.2 Program to Check Well Form-ness of an Expression

UQ. 2.5.2 *Use stack data structure to check well-formedness of parentheses in an algebraic expression. Write C program for the same* **MU - Dec. 18, 10 Marks**

UQ. 2.5.3 *Write a program in 'C' to check for balanced parenthesis in an expression using stack.* **MU - May 19, 10 Marks**

Soln. :

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>

#define MAX 25

int top = -1;
int stack[MAX];

void push(char item)
{
    if(top == (MAX - 1))
    {
        printf("Stack is Full\n");
        return;
    }
    top++;
    stack[top] = item;
}

char pop()
{
    if(top == -1)
```

```
    {
        printf("Stack is Empty\n");
        exit(1);
    }
    return(stack[top--]);
}

int match_paranthesis(char a, char b)
{
    if(a == '[' && b == ']')
    {
        return 1;
    }
    else if(a == '{' && b == '}')
    {
        return 1;
    }
    else if(a == '(' && b == ')')
    {
        return 1;
    }
    return 0;
}

int check_paranthesis(char expression[])
{
    int count;
    char temp;
    for(count = 0; count < strlen(expression); count++)
    {
        if(expression[count] == '(' || expression[count] ==
'{' || expression[count] == '[')
        {
            push(expression[count]);
        }
        if(expression[count] == ')' || expression[count] ==
'}' || expression[count] == ']')
        {
            if(top == -1)
            {
                printf("The Right Parentheses are more than
the Left Parentheses\n");
                return 0;
            }
            else
            {
                temp = pop();
                if(!match_paranthesis(temp,
expression[count]))
                {
                    printf("The Mismatched Parentheses in the
Expression are:\t%c and %c\n", temp, expression[count]);
                    return 0;
                }
            }
        }
    }
    if(top == -1)
    {
        printf("\nThe Expression has Balanced
Parentheses\n");
        return 1;
    }
    else
    {
        printf("The Expression has unbalanced
parentheses\n");
        return 0;
    }
}
int main()
{
    char expression[MAX];
    int validity;
    printf("\nEnter an Algebraic Expression:\t");
    scanf("%s", expression);
    validity = check_paranthesis(expression);
    if(validity == 1)
```

Module 2

```
    {
        printf("The Expression is Valid\n");
    }
    else
    {
        printf("The Expression is Invalid\n");
    }
getch();
}
```

2.6 POLISH NOTATIONS

UQ. 2.6.1 *Explain infix, postfix and prefix expressions with an example.*

MU - May 15, 6 Marks

GQ. 2.6.2 *Discuss the polish notation of arithmetic expression in application of stacks.*

(4 Marks)

- Notation is the way of writing arithmetic expression.
- **Concept : Polish Notation** is a way of expressing arithmetic expressions that avoids the use of brackets to define priorities for evaluation of operators.
- Polish Notation was devised by the Polish philosopher and mathematician Jan Łukasiewicz (1878-1956) for use in symbolic logic.
- There are three ways to write any arithmetic expression. All these ways are different but equivalent notations, i.e. even though they are different, they do not change the essence or output of an expression.
- These notations are as follows:

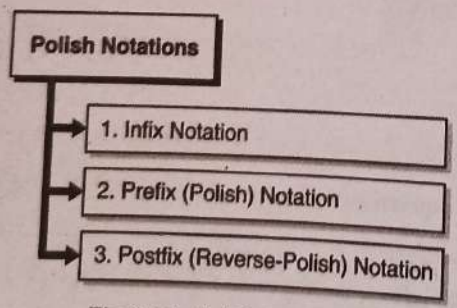
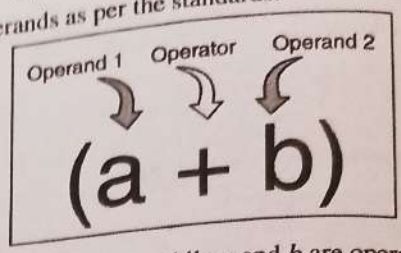


Fig. 2.6.1 : Polish Notations

- The way of using operators in the expression decides the name of notation.

1. Infix Notation

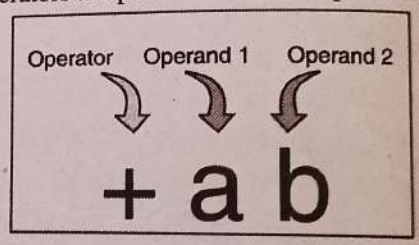
- This is basic and usual way of writing an expression.
- In this expression the operators are placed in between the operands as per the standards.



- Here + is an operator while *a* and *b* are operands.
- This is human readable format which is easy to read, write, and speak. But sometimes this expression is not so easy for computing devices which leads to conversion of infix format into other formats such as prefix or postfix.

2. Prefix (Polish) Notation

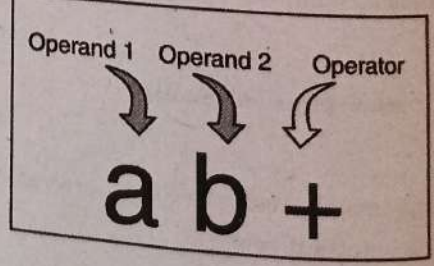
- This is another way to write an arithmetic expression in which operators are placed before the operands.



- This notation is considered as equivalent to its infix notation.
- Prefix notation is also known as **Polish Notation**.

3. Postfix (reverse-polish) Notation

- This is the last but not least way to write an arithmetic expression in which operators are placed after the operands.



- This notation is considered as equivalent to its infix notation.
- Postfix notation is also known as **Reversed Polish Notation**.

Syllabus Topic : Conversion of Infix to Postfix Expression

2.7 CONVERSION OF INFIX TO POSTFIX EXPRESSION

- We have seen that the infix is a regular format while postfix means operators are placed after operands.
- But this rule is just like ; consider a company declares that to appoint a software engineer the qualification is Engineering. But this does not indicate that every person completed Engineering will be appointed. At the time of interview, number of elements will be considered such as knowledge, sincerity etc. of the candidate.
- Just like that keeping operators at the end is the basic rule, rather than that number of elements will be considered.
- In this conversion, stack is used as a mediator between source string (infix) and target string (postfix). Stack will hold operators for some time.
- Now to understand the conversion exactly we will see the algorithm of this conversion.

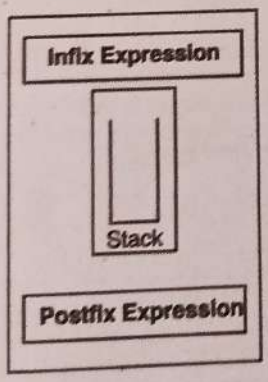


Fig. 2.7.1 : Infix to Postfix Conversion

2.7.1 Algorithm of Infix to Postfix Conversion

GQ. 2.7.1 *Write an algorithm to convert infix string to postfix (fully parenthesized).* **(5 Marks)**

Step 1 : The input string (infix notation) is scanned from left to right.

Step 2 : If the scanned character (ch) is space or tab, it is skipped.

Step 3 : If the scanned character (ch) is digit or alphabet, it is appended to postfix string.

Step 4 : If the scanned character (ch) is opening parenthesis '(', it is pushed to stack.

Step 5 : If the scanned character (ch) is operator :

All operators from top of the stack are popped having more or same priority than ch and appended to postfix string.

When less priority operator is found, then it is pushed in the stack again with ch.

Step 6 : If the scanned character (ch) is closing parenthesis ')', then all the operators above the opening parenthesis are appended to postfix string.

Step 7 : After evaluations of all the characters, the remaining operators from stack are appended to postfix string.

2.7.2 Program of Infix to Postfix Conversion

UQ. 2.7.2 *Write a program in 'C' to convert infix expression to postfix expression using stacks.* **MU - May 14, Dec. 17, 10 Marks**

UQ. 2.7.3 *Write a C program to convert infix expression to postfix expression.* **MU - Dec. 19, 10 Marks**

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 50
void convert () ;
```

```
int priority ( char ) ;
char stack[MAX] ;
char s[MAX], t[MAX] ;
int top ;
void main( )
{
        top = -1 ;
        printf ( "\nEnter an expression in infix form: " ) ;
        gets (s) ;
        convert () ;
        printf ( "\nThe postfix expression is: " ) ;
        printf ( " %s", t ) ;
        getch( ) ;
}
void convert ()
{
  char opr ;
int i =0,j=0;
    while ( s[i]!='\0' )
    {
        if ( s[i] == ' ' || s[i] == '\t' )
        {
            i++;
            continue ;
        }
        if ( isdigit ( s[i] ) || isalpha ( s[i] ) )
        {

            while ( isdigit ( s[i] ) || isalpha ( s[i] ) )
            {
                t[j] = s[i] ;
                i++;
                j++;
            }
        }
        if ( s[i] == '(' )
        {
            stack[++top] = s[i] ;
                        i++;
        }
        if ( s[i] == '*' || s[i] == '+' || s[i] == '/' || s[i]
== '%' || s[i] == '-' || s[i] == '$' )
        {
            if ( top != -1 )
            {
                opr = stack[top--] ;
                while ( priority ( opr ) >= priority ( s[i] ))
                {
                    t[j] = opr ;
                    j++;
                    opr = stack[top--];
                }
                stack[++top] = opr ;
                stack[++top] = s[i] ;
            }
            else
                stack[++top] = s[i] ;
            i++;
        }
    if ( s[i] == ')' )
    {
        opr = stack[top--];
        while ( ( opr ) != '(' )
        {
            t[j] = opr ;
            j++;
            opr = stack[top--];
            }
            i++;
        }
    }
while ( top != -1 )
{
    char opr = stack[top--];
    t[j] = opr ;
    j++;
```

Spaces are skipped

Alphabets and digits are added in postfix string

For operators priorities are compared

When found ')', all operators above '(' from stack are added to postfix

All remaining operators from stack are added to postfix

```
    }
t[j] = '\0';
}
int priority ( char c )
{
    if ( c == '$' )
        return 3 ;
    if ( c == '*' || c == '/' || c == '%' )
        return 2 ;
    else
    {
        if ( c == '+' || c == '-' )
            return 1 ;
        else
            return 0 ;} }
```

Priority is returned as per the operator

Output

```
Enter an expression in infix form: ((A+B) * (C-D)) / E
The postfix expression is:  AB+CD-*E/
```

☞ **Example**

GQ. 2.7.4 *Consider infix string : ((A+B) * (C-D)) / E.*

We will see step by step conversion of this expression into postfix form :

((A + B) * (C – D))/E

Table 2.7.1 : Infix to Postfix Conversion

Character from Infix	Stack	Postfix Expression
(	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
C	(*	AB+C
-	(*(-	AB+C
D	(*(-	AB+CD
)	(*	AB+CD-
)		AB+CD-*
/	/	AB+CD-*
E	/	AB+CD-*E
		AB+CD-*E/

GQ. 2.7.5 *Convert the given infix expression to postfix expression using stack and the details of stack at each step of conversion Expression*

*a ↑ b * c-d+e/f /(g + h)*

Note : ↑ indicates exponent operator.

(5 Marks)

a ↑ b * c-d+e/f /(g + h)

Table 2.7.2 : Infix to Postfix Conversion

Character Scanned	Stack	Postfix
a	–	a
↑	↑	a
B	↑	ab
*	↑	ab
C	*	ab ↑c
-	–	ab ↑ c*
D	–	ab ↑ c * d
+	+	ab ↑ c * d –
E	+	ab ↑ c * d – e
/	+/	ab ↑ c * d – e
F	+/	ab ↑ c * d – ef
/	+/	ab ↑ c * d – ef /

Module 2

Character Scanned	Stack	Postfix
(	+/(	ab ↑ c * d – ef /
G	+/(	ab ↑ c * d – ef / g
+	+/(+	ab ↑ c * d – ef / g
h	+/(+	ab ↑ c * d – ef / gh
	+/	ab ↑ c * d – ef /gh +
	Empty	ab ↑ c * d – ef/ gh + / +

∴ Postfix expression is = ab ↑ c * d – ef / gh + /+

GQ. 2.7.6 *Translate the given infix expression to postfix expression using stack and show the details of stack at each step.*

*Expression : ((A + B) * D)↑ (E – F)*

(5 Marks)

((A + B) * D)↑ (E – F)

Table 2.7.3 : Infix to Postfix Conversion

Character Scanned	Stack	Postfix
(	(	–
(	((	–
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
D	(*	AB+D
)	Empty	AB+D*
↑	↑	AB+D*
(	↑(	AB+D*E
E	↑(	AB+D*E
-	↑(–	AB+D*E
F	↑(–	AB+D*EF
	↑	AB+D*EF–
	Empty	AB+D*EF–↑

∴ Postfix expression = AB+D*EF–↑

Syllabus Topic : Evaluation of Postfix Expression

2.8 EVALUATION OF POSTFIX EXPRESSION

- The evaluation of postfix expression is easy.
- Postfix expression is without parenthesis. As Postfix expression can be evaluated as two operands and one operator at a time, it is easier for the compiler and the computer to handle this evaluation.

Rules regarding evaluation postfix expression

- The basic rules regarding evaluation of a Postfix Expression are :
  - Read elements from left to right and push the element in the stack if it is an operand.
  - If the element is found to be operator, then pop two elements from stack. Evaluate the expression which is formed by inserting the operator in operands.
  - Push the result of the evaluation into the stack. Repeat it till the end of the expression.
  - Print the popped element from stack as a result.

2.8.1 Algorithm to Evaluate Postfix Expression

GQ. 2.8.1 *Write an algorithm to evaluate postfix expression.* **(5 Marks)**

Step 1 : Read postfix expression from Left to Right

Step 2 : If operand is encountered, push it in Stack.

Step 3 : If operator is encountered, Pop two elements
A → Top element B → Next Top element
Evaluate B operator A

Step 4 : Push result into stack

Step 5 : Read next postfix element if not end of postfix string

Step 6 : Repeat from Step 2

Step 7 : Print the result popped from stack.

2.8.2 Program to Evaluate Postfix Expression

UQ. 2.8.2 *Write program in 'C' to evaluate a postfix expression using STACK ADT.*

MU - Dec. 13, May 18, 10 Marks

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>

#define MAX 50

int stack[MAX] ;
int top, nn ;
char s[20] ;
void push ( char ) ;
char pop () ;
void calculate () ;

void main( )
{
    top = -1 ;
  printf ( "\nEnter an expression in postfix form : " ) ;
    gets (s) ;
    calculate() ;
  getch( ) ;
}
void push(char ch)
{
  if(top==MAX-1)
  {
    printf("\n\tSTACK is over flow");
  }
  else
  {
    top++;
    stack[top] = ch;
  }
}
char pop()
{
  if(top==-1)
  {
    printf("\n\tSTACK is under flow");
    return(-1);
  }
  else
  {
    char ch = stack[top];
    top--;
    return(ch);
  }
}
void calculate()
{
    int n1, n2, n3,i ;
    i = 0;
    while ( s[i] )
    {

      if ( s[i] == ' ' || s[i] == '\t' )
      {
          i++;
          continue ;
      }
      if ( isdigit ( s[i] ) )
      {
          nn = s[i] - '0' ;
          push ( nn ) ;
      }
      else
      {

      n1 = pop () ;
      n2 = pop () ;
      switch ( s[i] )
          {
              case '+' :
```

Starting from first character

Converts char to int

If character is operand push to stack, if it is operator, then operations on two stack elements and result is pushed to stack

```
                n3 = n2 + n1 ;
                break ;
            case '-' :
                n3 = n2 - n1 ;
                break ;
            case '/' :
                n3 = n2 / n1 ;
                break ;
            case '*' :
                n3 = n2 * n1 ;
                break ;
            case '%' :
                n3 = n2 % n1 ;
                break ;
            case '$' :
                n3 = pow ( n2 , n1 ) ;
                break ;
            default :
                printf ( "Unknown operator" ) ;
                exit ( 1 ) ;
        }
        push ( n3 ) ;
    }
    i++ ;
}
printf("\nResult is %d",n3);
}
```

Output

```
Enter an expression in postfix form : 53+82-*
Result is 48
```

GQ. 2.8.3 *Consider infix expression : 53+82-*.* **(5 Marks)**

We will see step by step evaluation of this expression :

Table 2.8.1 : Evaluate Postfix Expression

Reading symbol	Stack operations	Evaluated part of expression
Initially	Stack is Empty	Nothing
5	push(5)	Nothing
3	push(3)	Nothing

+	Value 1 = pop()
	Value 2 = pop()
	result = Value 2 + Value 1
	push(result)
		Value 1 = pop(); // 3
		Value 2 = pop(); // 5
		result = 5 + 3; // 8
		Push(8)
		(5 + 3)

8	push(8)	(5 + 3)
2	push(2)	(5 + 3)

–	Value 1 = pop()
	Value 2 = pop()
	result = Value 2 – Value 1
	push(result)
		Value 1 = pop(); // 2
		Value 2 = pop(); // 8
		result = 8 – 2; // 6
		Push(6)
		(8 – 2)
		(5 + 3),(8 – 2)

*	Value 1 = pop()
	Value 2 = pop()
	result = Value 2 + Value 1
	push(result)
		Value 1 = pop(); // 6
		Value 2 = pop(); // 8
		result = 8 * 6; // 48
		Push(48)
		(6 * 8)
		(5 + 3),(8 – 2)

End of expression	result = pop()	Display (result) 48 as final result

GQ. 2.8.4 *Evaluate the following postfix expression and show stack after every step in tabular form. Given A = 5, B = 6, C = 2, D = 12, E = 4 ABC + *DE\- -* **(5 Marks)**

ABC+*DE/-

This will become

5 6 2 + * 12 4 / -

Table 2.8.2 : Evaluate Postfix Expression

Reading	Stack	Evaluated
5	5	
6	5,6	
2	5,6,2	
+	5,8	6 + 2 = 8
*	40	5*8 = 40
12	40,12	
4	40,12,4	
/	40,3	12 / 4 = 3
-	37	40-3 = 37

Result is 37

GQ. 2.8.5 *Evaluate following postfix expression. A : 6, 2, 3, +, –, 3, 8, 2, +, +, *, 2, ^, 3, +* **(5 Marks)**

Table 2.8.3 : Evaluate Postfix Expression

Reading	Stack	Evaluated
6	6	
2	6 2	
3	6 2 3	
+	6 5	2 + 3 = 5
-	1	6 – 5
3	1 3	
8	1 3 8	
2	1 3 8 2	
+	1 3 10	8 + 2 = 10
+	1 13	3 + 10 = 13
*	13	1 * 13
2	13 2	
^	169	$13^2 = 169$
3	169 3	
+	172	169 + 3 = 172

Result 172

GQ. 2.8.6 *Consider the following arithmetic expression written in postfix notation : 10, 2, *, 15, 3, /, +, 12, 3, 2, ↑, +, + Evaluate this expression to find its value.* **(5 Marks)**

Table 2.8.4 : Evaluate Postfix Expression

Reading	Stack	Evaluated
10	10	
2	10 2	
*	20	10 * 2 = 20
15	20 15	
3	20 15 3	
/	20 5	15 / 3 = 5
+	25	20 + 5
12	25 12	
3	25 12 3	
2	25 12 3 2	
↑	25 12 9	$3^2 = 9$
+	25 21	12 + 9 = 21
+	46	25 + 21 = 46

Module 2

Syllabus Topic : Recursion

2.9 RECURSION

UQ. 2.9.1 What is recursion ?
MU - Dec. 14, May 17, Dec. 17, 2 Marks

UQ. 2.9.2 Explain the term recursion with an example.
MU - May 15, 7 Marks

UQ. 2.9.3 Explain recursion as an application of stack with examples.
MU - May 16, 10 Marks

Definition : Calling a function inside itself is called as recursion. Such function is called as recursive function.

How recursion works?

- The execution of recursion continues unless and until some specific condition is met to prevent its repetition.
- For the purpose of preventing infinite recursion, *if...else* statement (or similar approach) can be used in which one branch go for the recursive call while other doesn't.

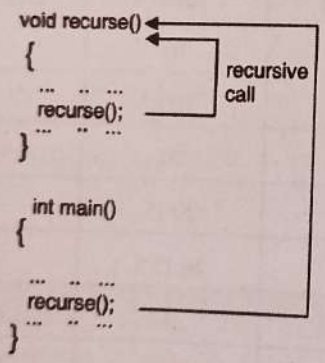


Fig. 2.9.1

2.9.1 Advantages of Recursion

GQ. 2.9.4 Write any four advantages of recursion. (4 Marks)

1. It helps to reduce the size of program by minimizing the code.
2. It makes easy to maintain function calling related information.
3. Evaluation of stack can be implemented through recursion.
4. Also infix, prefix, post-fix notation can be evaluated with the help of recursion.

2.9.2 Disadvantages of Recursion

GQ. 2.9.5 Write any four disadvantages of recursion. (4 Marks)

1. It takes more time because of stack overlapping.
2. Stack overflow may occur in recursive programs.
3. Memory requirement is more as at every recursive call, separate memory address is allocated for the variables.
4. It may leads to indefinite call if exit condition does not work.
5. Efficiency is less.

Algorithm of program to calculate the factorial of number using recursion

GQ. 2.9.6 Write an algorithm of program to calculate the factorial of number using recursion. (5 Marks)

Factorial is calculates as follows :

$$5! = 5 * 4 * 3 * 2 * 1 = 120$$

Step 1 : Start
Step 2 : Read number num
Step 3 : Call factorial(num)
Step 4 : Print factorial fact
Step 5 : Stop

factorial(no)
Step 1 : If no<0 then return -1
Step 2 : Else if no=0 then return 1
Step 3 : Else

return(no*factorial(no-1))

☞ **Program to calculate factorial of a number**

GQ. 2.9.7 *Write a 'C' program to calculate the factorial of number using recursion.* **(5 Marks)**

```
#include<stdio.h>
#include<conio.h>

int factorial(int);
void main()
{
int i,fact,num;

printf("Enter a number to get factorial : ");
scanf("%d",&num);
fact=factorial(num);
printf("Factorial of %d is %d",num,fact);
getch();
}
int factorial(int no)
{
if(no<0)
return(-1);
if(no==0)
return(1);

return(no*factorial(no-1));
}
```

If value of no is > 1, function is called recursively

Output

```
C:\eval.exe
Enter a number to get factorial : 4
Factorial of 4 is 24
```

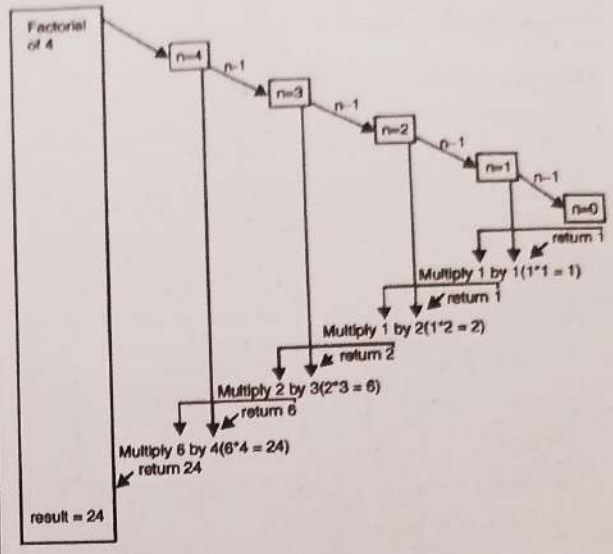


Fig. 2.9.2 : Factorial using recursion

☞ **Program to print Fibonacci series using recursion**

– In Fibonacci series every number is summation of its previous two numbers. Only the first two terms are directly considered as 0 and 1.

☞ **Algorithm to print Fibonacci series using recursion**

GQ. 2.9.8 *Write an algorithm of program to print Fibonacci series using recursion.* **(5 Marks)**

Fibonacci is calculated as :

0, 1, 1, 2, 3, 5, ...

Step 1 : Start
Step 2 : Initialize x= 0, y = 1
Step 3 : Read number of terms n
Step 4 : print x,y and Call fibonacii(x, y, n)
Step 5: Stop

fibonacii(x, y, n)

Step 1 : If n<0 then exit
Step 2 : sum = x + y
Step 3 : Print sum
Step 4 : x = y and y = sum
Step 5: fibonacii(x,y,n-1);

GQ. 2.9.9 *Write program to print Fibonacci series using recursion.* **(5 Marks)**

```
#include <stdio.h>
void fibonacii(int x,int y, int n)
{
  int sum;
  if(n>0)
  {
    sum=x+y;
    printf("%d ",sum);
    x=y;
    y=sum;
    fibonacii(x,y,n-1);
  }
  }
int main()
{
  int x,y,n;

  x=0;
  y=1;
  printf("Enter total number of terms: ");
  scanf("%d",&n);

  printf("Fibonacii series is : ");
  printf(" %d %d ",x,y);

fibonacii(x,y,n-2);
  getch();
}
```

Number is sum of its previous two nos. and then second becomes first and sum becomes second

First term

Second term

print x and y as first and second terms of series

call function with (n-2) terms

Output

```
Enter total number of terms: 10
Fibonacii series is :  0 1 1 2 3 5 8 13 21 34
```

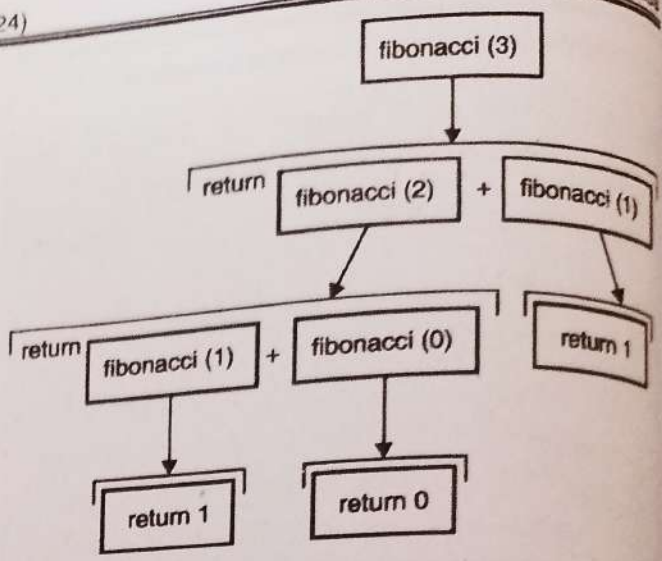


Fig. 2.9.3 : Fibonacci series using recursion

☞ **Algorithm to multiply natural numbers using recursion**

GQ. 2.9.10 *Write an algorithm to multiply natural numbers using recursion.* **(5 Marks)**

Step 1 : Start
Step 2 : Read two numbers a and b
Step 3 : Call multiply(a,b)
Step 4 : Print multiplication m
Step 5 : Stop

multiply(a,b)
Step 1 : product=0,i=0;
Step 2 : if i <a then product = product + b ad i++
Step 3 : multiply(a,b);
Step 4 : return product

GQ. 2.9.11 *Write a 'C' program for multiplication of natural numbers using recursion.* **(5 Marks)**

```
#include<stdio.h>
#include<conio.h>
int multiply(int,int);
int main(){

  int a,b,m;
  printf("\n Enter two numbers : ");
```

```
  scanf("%d%d",&a,&b);

  m = multiply(a,b);

  printf("\n Multiplication is %d",m);

  getch();
}

int multiply(int a,int b)
{
  static int product=0,i=0;

  if(i < a){
    product = product + b;
    i++;
    multiply(a,b);
  }
  return product;
}
```

Value of b is added in product and function is called recursively

Output

```
C:\tnd.exe
Enter two numbers : 10 5
Multiplication is 50
```

UQ. 2.9.12 *Write recursive function to calculate GCD of 2 numbers.* MU - May 14, 5 Marks

Program

```
#include <stdio.h>
int hcf(int n1, int n2);

main()
{
  int n1, n2;
  printf("Enter two positive integers: ");
  scanf("%d %d", &n1, &n2);

  printf("G.C.D of %d and %d is %d.", n1, n2, hcf(n1,n2));
  getch();
}

int hcf(int n1, int n2)
{
  if (n2 != 0)
    return hcf(n2, n1%n2);
  else
    return n1;
}
```

Output

```
C:\bl.exe
Enter two positive integers: 100 70
G.C.D of 100 and 70 is 10.
```

UQ. 2.9.13 *Write a 'C' program to calculate sum of 'n' natural numbers using recursion.* MU - Dec. 14, Dec. 17, 3 Marks.

Program

```
#include <stdio.h>
int sum(int n);

main()
{
  int num;
  printf("Enter a positive integer: ");
  scanf("%d", &num);
  printf("Sum = %d",sum(num));
  getch();
}
```

```
int sum(int n)
{
  if(n != 0)
    return n + sum(n-1);
  else
    return n;
}
```

Output

```
C:\bl.exe
Enter a positive integer: 5
Sum = 15
```

UQ. 2.9.14 Write recursive function in 'C' to find sum of digits of a number.

MU - May 17, 5 Marks.

Program

```
#include <stdio.h>

int sum (int a);

main()
{
  int num, result;

  printf("\n Enter the number: ");
  scanf("%d", &num);
  result = sum(num);
  printf("\n Sum of digits in %d is %d\n", num, result);
    getch();
}

int sum (int num)
{
  if (num != 0)
  {
    return (num % 10 + sum (num / 10));
  }
  else
  {
    return 0;
  }
}
```

Output

```
C:\bl.exe
Enter the number: 123
Sum of digits in 123 is 6
```

UQ. 2.9.15 Write a 'C' program to convert decimal to binary using any appropriate data structure you have studied.

MU - Dec. 13, 7 Marks

Program

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
#define MAX 10

typedef struct mystack
{
 int st_data[MAX];
 int st_top;
}mystack;

int isEmpty(mystack *s)
{
 if(s->st_top==-1)
    return(1);
 return(0);
}
```

```
int isFull(mystack *s)
{
if(s->st_top==MAX-1)
  return(1);
return(0);
}

void push(mystack *s,int x)
{
s->st_top=s->st_top+1;
s->st_data[s->st_top]=x;
}

int pop(mystack *s)
{
int x;
x=s->st_data[s->st_top];
s->st_top=s->st_top-1;
return(x);
}
//_____________
void main()
{
mystack s;
int num;

s.st_top=-1;
  printf("\nEnter decimal number : ");
  scanf("%d",&num);

  while((num!=0))
  {
   if(!isFull(&s))
     {
     push(&s,num%2);
     num=num/2;
     }
   else
     {
     printf("\nStack overflow");
     exit(0);
     }
  }

printf("\n ");
while(!isEmpty(&s))
  {
  num=pop(&s);
  printf("%d",num);
  }
  getch();
}
```

Output

```
C:\bi.exe
Enter decimal number : 13
 1101
```

UQ. 2.9.16 *Write a function in C to convert prefix expression to postfix expression.*

MU - Dec. 15, 7 Marks

Program

```
#include<stdio.h>
#include<string.h>
#include<math.h>
#include<stdlib.h>

#define CH_BLNK' '
#define CH_TB '\t'
#define SIZE 50

char *pop();
char prefix_expr[SIZE];
char stack[SIZE][SIZE];
```

Module 2

```
void push(char *str);
int isempty();
int white_space(char ch);
void convert();
int top;

int main()
{
    top = -1;
    printf("\n Enter Expression in Prefix form : ");
    gets(prefix_expr);
    convert();
getch();
}/*End of main()*/

void convert()
{
    int i;
    char op1[SIZE], op2[SIZE];
    char ch;
    char temp[2];
    char strin[SIZE];
    for(i=strlen(prefix_expr)-1;i>=0;i--)
    {
        ch=prefix_expr[i];
        temp[0]=ch;
        temp[1]='\0';

        if(!white_space(ch))
        {
            switch(ch)
            {
            case '+':
            case '-':
            case '*':
            case '/':
            case '%':
            case '^':
                strcpy(op1,pop());
                strcpy(op2,pop());
                strcpy(strin,op1);
                strcat(strin,op2);
                strcat(strin,temp);
                push(strin);
                break;
            default: /*if an operand comes*/
                push(temp);
            }
        }
    }
    printf("\n Postfix Expression : ");
    puts(stack[0]);
}/*End of convert()*/


void push(char *str)
{
    if(top > SIZE)
    {
        printf("\nStack overflow\n");
        exit(1);
    }
    else
    {
        top=top+1;
        strcpy( stack[top], str);
    }
}/*End of push()*/

char *pop()
{
    if(top == -1 )
    {
        printf("\nStack underflow \n");
        exit(2);
    }
    else
```

```
        return (stack[top--]);
}/*End of pop()*/
int isempty()
{
    if(top==-1)
        return 1;
    else
        return 0;
}
int white_space(char ch)
{
    if(ch==CH_BLNK|| ch==CH_TB || ch=='\0')
        return 1;
    else
        return 0;
}/*End of white_space()*/
```

Output

```
C:\bin\exe
Enter Expression in Prefix form : ++A*BCD
Postfix Expression :: ABC*+D+
```

Syllabus Topic : Introduction : ADT of Queue

2.10 INTRODUCTION : ADT OF QUEUE

UQ. 2.10.1 *Give ADT for the queue data.*

MU - Dec. 14, 2 Marks

UQ. 2.10.2 *Explain Queue as ADT.*

MU - Dec. 15, 2 Marks

❑ ***Definition : Queue is a data structure in which addition of an element is allowed at one end (called as rear) while removal of an element is allowed at another end (called as front).***

- Queue is an **Abstract Data Type**.
- Queue is also known as **FIFO** (First In First Out List). It indicates that the first inserted element will be removed first and last inserted element will be removed last.
- There are number of real life examples of queue. The most common example is queue of people waiting to get movie ticket at ticket counter of a theatre.
- The first person in the queue will get the first ticket and will go outside the queue first.
- The new person will always stand next to the last person to become new last one.
- Practical Examples : Refer Fig. 2.10.1 and 2.10.2.

Module 2

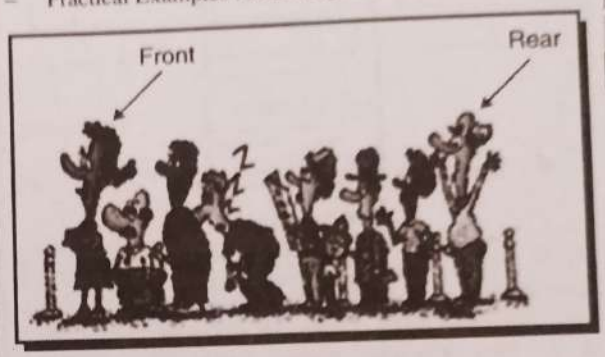


Fig. 2.10.1 : Queue

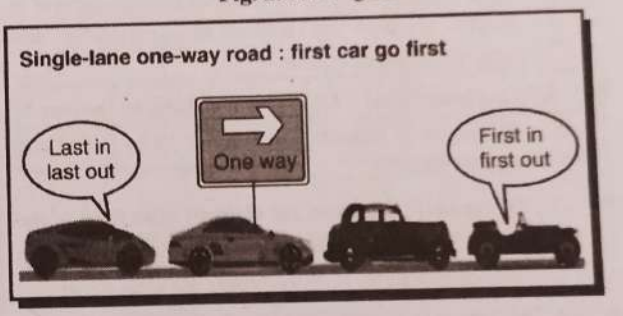


Fig. 2.10.2 : Queue

☞ **Some other Real Life Examples of Queue are**

- OS processes.
- Queue of packets in data communication.

Syllabus Topic : Operations on Queue

2.11 OPERATIONS ON QUEUE

GQ. 2.11.1 *Write the implementation procedure of basic primitive operations of the Queue using Linear array.* **(5 Marks)**

- The queue is an **abstract data type** which is defined by the following structure and operations.

- As we have seen, queue is structured as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front."

The queue operations are as follows :

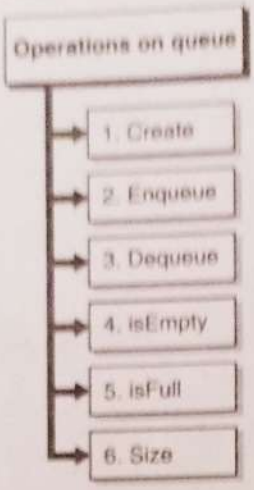


Fig. 2.11.1 : Operations on Queue

1. **Create()** : Creates and initializes new queue that is empty. It does not require any parameter and returns an empty queue.
2. **Enqueue(item)** : Adds a new element to the rear of the queue. It requires the element to be added and returns nothing.
3. **Dequeue()** : Removes the element from front of the queue It does not require any parameter and returns the deleted item.
4. **isEmpty()** : Checks whether the queue is empty or not. It does not require any parameter and returns a boolean value.
5. **Size()** : Returns the total number of elements present in the queue. It does not require any parameter and returns an integer.

- **Example** : Consider q is a queue that has been created and starts out empty, then in the following table we can observe the results of a sequence of **queue operations**.

Table 2.11.1 : Queue Operations

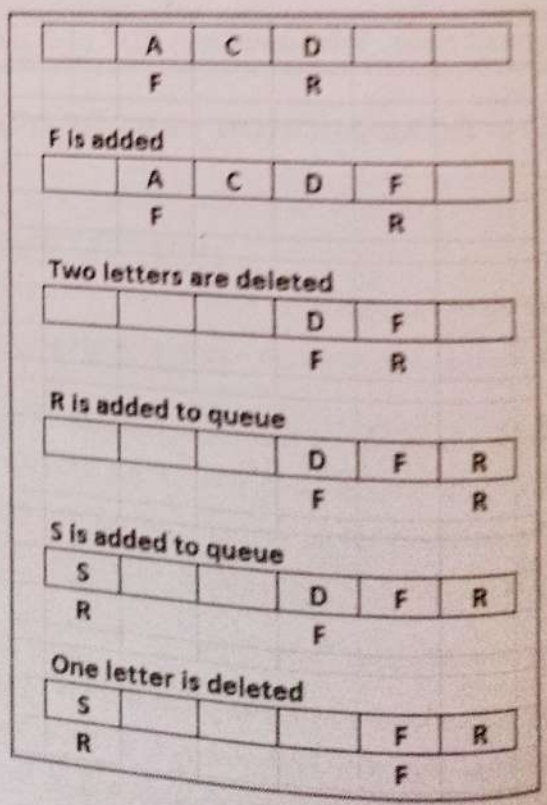
Queue Operation	Queue Content	Return Value
q.isEmpty()	[ ]	True
q.enqueue("A")	[A]	
q.enqueue (5)	[A, 5]	
q.dequeue()	[5]	A
q.enqueue("B")	[5 B]	
q.size()	[5 B]	2

Example

UQ. 2.11.2 *Consider the following queue, where queue is a circular queue having 6 memory cells.*
Front = 2, Rear = 4.
Queue : _, A, C, D, _, _
Describe queue as following operations take place :
F is added to the queue
Two letters are deleted
R is added to the queue
S is added to the queue
One letter is deleted

MU - Dec. 10, 2 Marks

Program



2.12 DIFFERENCE BETWEEN STACK AND QUEUE

GQ. 2.12.1 *Explain difference between Stack and Queue.* **(5 Marks)**

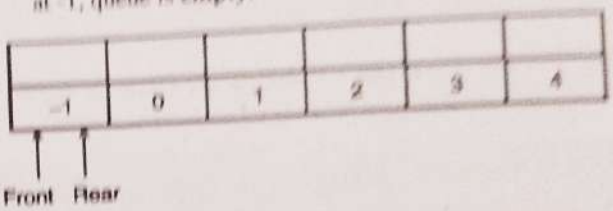
Parameter	Stack	Queue
Operation end	Elements are added and deleted from same end	Elements are added and deleted from different ends
Pointer	In stack single pointer is used to point top of the element	In queue, two pointers are used to point front and rear ends.
Order	Stack follows Last In First Out (LIFO) order.	Queue follows First In First Out (FIFO) order.
Operation names	Stack operations are known as push and pop.	Queue operations are known as enqueue and dequeue.
Visualization	The visualization of Stack is of vertical collection.	The visualization of Queue is of horizontal collection.
Examples	Books kept one above other in shelf	Queue for tickets at theatre

Syllabus Topic : Array Implementation of Queue

2.13 ARRAY IMPLEMENTATION OF QUEUE

GQ. 2.13.1 *Explain 'Queue Full' and 'Queue Empty' condition with suitable example.* **(4 Marks)**

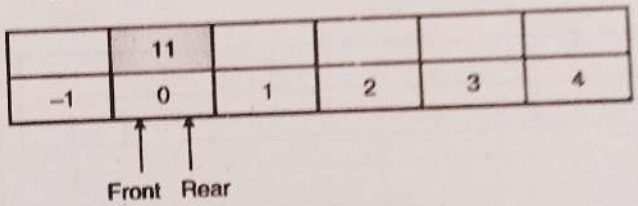
- Queue is considered as a linear data structure, hence it can be represented using the array. It is also known as **static implementation of queue.**
- Here two pointers will work : ***front*** **and** ***rear***.
- *Front* will point to the first element while *rear* will point to the last element.
- Consider an int arr[5];
- Initially queue is empty, hence *front* and *rear* both will point to -1. To check emptiness of queue we can use this condition. That means if front and rear both are at -1, queue is **empty**.



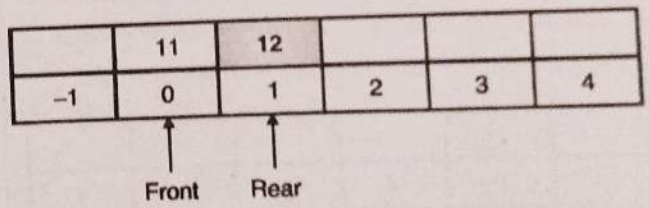
2.13.1 Enqueue : Inserting an Element in Queue

GQ. 2.13.2 *Write an algorithm for simple queue with ENQUEUE operation.* **(5 Marks)**

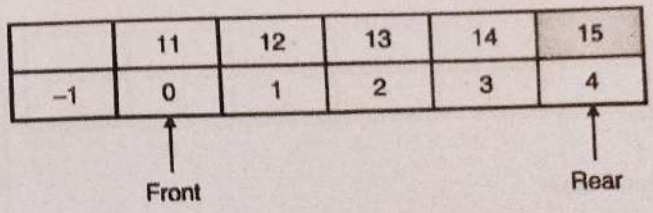
- While adding first element, both *front* and *rear* are incremented by one and at rear position new element will be added.



- From here for every addition of new element, rear will be incremented and at rear position new element will be added.



- When rear is reached to SIZE minus 1 (SIZE-1) position, the queue is considered as **full**.



Algorithm to insert an element in queue

Step 1: If rear= SIZE – 1 then Write "Queue is Overflow"

Step 2: rear = rear + 1

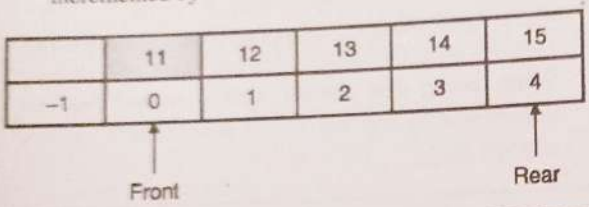
Step 3 : QUEUE [rear] = X

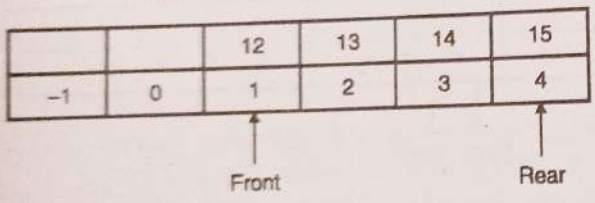
Step 4: If front = -1 then front = 0

2.13.2 Dequeue : Deleting an Element from Queue

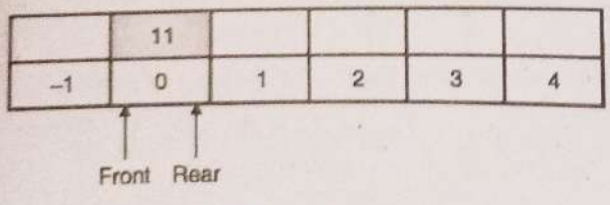
GQ. 2.13.3 Write an algorithm for simple queue with DEQUEUE operation. **(3 Mark)**

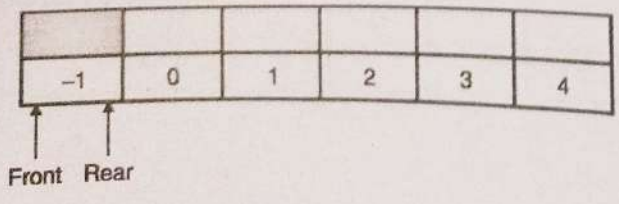
- The element at position *front* is deleted and front is incremented by 1.





- If a single element is present in the queue at the time of deletion (*front* and *rear* both are at same position), then after deletion both *front* and *rear* are set at position -1 which indicates that now queue is empty.





2.13.3 Algorithm to Delete an Element from Queue

Step 1 : If front = -1 then Write "Queue is Underflow"

Step 2 : Return QUEUE [front]

Step 3: If front = rear then front = -1, rear = -1 Else front = front + 1

2.13.4 Program to Perform Operations on Queue

UQ. 2.13.4 Write a menu driven program in C to implement QUEUE ADT. The program should perform the following operations.

(i) Inserting element in the beginning

(ii) Deleting an Element from the Queue

(iii) Displaying the Queue

(iv) Exiting the program

MU - May 16, 12 Marks

UQ. 2.13.5 Write a program in 'C' to implement Queue using array.

MU - May 19, 10 Marks

Program

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define SIZE 5
void enqueue( int x);
void dequeue( );
void display ();
int FRONT=-1;
int REAR=-1;
int QUEUE[SIZE];
void main()
{
    int x,ch;

    while(1)
    {
```

Queue is empty

```
printf("\n1 : Insert\n");
printf("2 : Delete\n");
printf("3 : Display\n");
printf("4 : Exit\n");
printf("Enter Your Choice : ");
scanf("%d",&ch);

    switch(ch)
{
  case 1:
     printf("Enter Element:");
     scanf("%d",&x);
     enqueue(x);
     break;
  case 2:
     dequeue( );
     break;
  case 3:
     display( );
     break;
  case 4:
     exit(0);
  }
}
getch();
}
void enqueue( int x)
{
  if(REAR>=SIZE-1)
  printf("Queue is Overflow");
  else
  {
    REAR=REAR+1;
    QUEUE[REAR]=x;

      if(FRONT==-1)
    FRONT=0;
  }
}
void dequeue( )
{
  if(FRONT==-1)
  printf("Queue is Underflow");
  else
  {
    printf("Deleted Element is %d\n",QUEUE[FRONT]);
    if(FRONT==REAR)
    {
      FRONT=-1;
      REAR=-1;
    }
    else
    {
      FRONT=FRONT+1;
    }
  }
}
void display()
{
  int i;
  if(FRONT==-1)
  printf("Queue is Empty\n");
  else
  {
    for(i=FRONT; i<=REAR; i++)
    printf(" %d ",QUEUE[i]);
  }
}
```

Menu

Queue is empty

Deleting remaining single element

If front is not at -1, then elements from front to rear are printed

Rear is incremented and element placed at rear position

Adding first element

Output

```
1 : Insert
2 : Delete
3 : Display
4 : Exit
Enter Your Choice : 1
Enter Element:11

1 : Insert
2 : Delete
3 : Display
4 : Exit
Enter Your Choice : 1
Enter Element:12

1 : Insert
2 : Delete
3 : Display
4 : Exit
Enter Your Choice : 3
 11  12
1 : Insert
2 : Delete
3 : Display
4 : Exit
Enter Your Choice : 2
Deleted Element is 11

1 : Insert
2 : Delete
3 : Display
4 : Exit
Enter Your Choice : 3
 12
1 : Insert
2 : Delete
3 : Display
4 : Exit
Enter Your Choice :
```

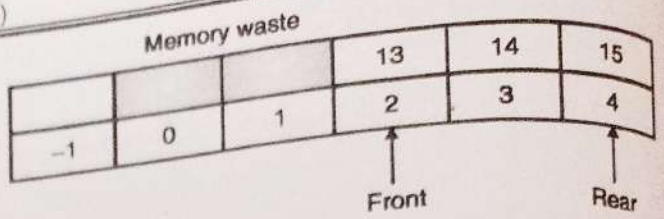
Syllabus Topic : Types of Queue : Circular Queue

2.14 TYPES OF QUEUE : CIRCULAR QUEUE

UQ. 2.14.1 *What is circular queue ?*

MU - May 15, Dec. 16, 7 Marks

- There are different types of queue. We have already seen linear queue. Other types of queue are Circular Queue, Priority Queue and Double Ended Queue.
- To understand circular queue, first we will understand the **disadvantage** of **linear queue** :
- In a simple queue, we have seen that when elements are deleted, the pointer front is incremented.
- The spaces which get free after deletion cannot be reutilized in this type of queue.



- In the above situation, first two locations are free because of deletion of first two elements, but as rear is at SIZE-1 position, the queue is considered as full and we cannot add new elements. It leads to wastage of memory.
- So to solve this problem, DS provides concept of circular queue.

> **Definition** : *Circular Queue is a linear data structure in which the operations are carried out on the basis of FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.*

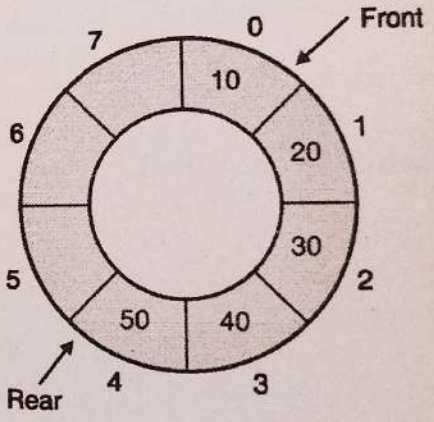
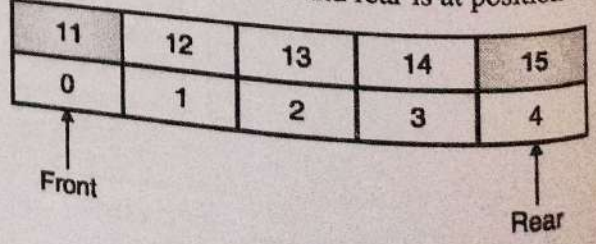


Fig. 2.14.1 : Circular Queue

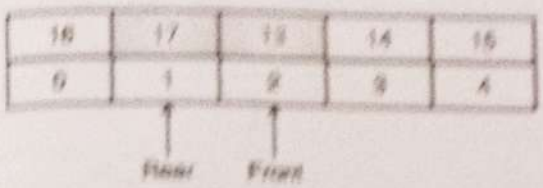
- Now we will see array representation of circular queue.
- Here when the rear reaches at SIZE-1 position, it is not considered that queue is full.

Only in **two situations** circular **queue** is considered as **full** :

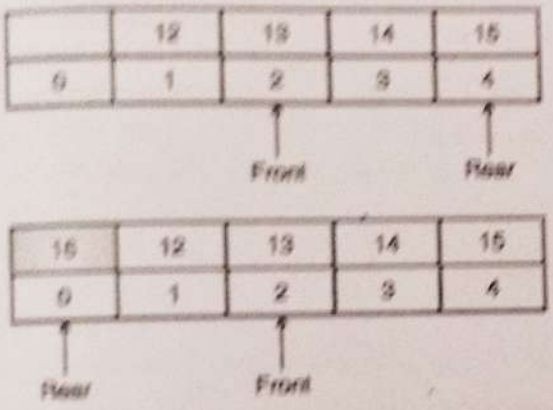
1. Front is at position 0 and rear is at position SIZE-1



2. Front is one place next to rear



Here in the process of adding elements, if rear reaches to last position and if there are vacant positions at the beginning of array, then rear can be shifted to first position of array and new element can be added at rear.



Algorithm to implement circular queue

Step 1 : Start

Step 2 : Show options: 1. Insert 2. Delete 3. Display 4. Exit

Step 3 : Accept choice

Step 4 : As per user's choice call a function out of enQueue(), deQueue(), display(), exit()

Step 5 : Exit

Algorithm of Enqueue (Insertion)

GQ. 2.14.2 Write an algorithm for circular queue that insert an element at rear end. **(4 Marks)**

Step 1 : If queue is full print the message and exit

Step 2 : Accept element

Step 3 : If rear is not at last position increment it by 1 Else set it at position 0

Step 4 : Add element at rear location

Algorithm of Dequeue (Deletion)

UQ. 2.14.3 Write an algorithm/program to implement Delete operation into a Circular Queue using array representation of Queue. **MU - Dec. 11, Dec. 12, Dec. 14, Dec. 16, 6 Marks**

Step 1 : If queue is empty print the message and exit

Step 2 : Print front element as deleted

Step 3 : Increment front by one

Step 4 : If front reaches to SIZE position, set it at 0. If element deleted is last then set front and rear to 0

Algorithm of Display

Step 1 : If queue is empty print the message and exit

Step 2 : Print elements from front to rear

Module 2

2.14.1 Program to Implement Circular Queue

UQ. 2.14.4 Write a program in 'C' to implement a circular queue. **MU - May 15, 5 Marks.**

UQ. 2.14.5 Write a function in C to insert, delete and display elements in Circular Queue. **MU - Dec. 15, 10 Marks**

UQ. 2.14.6 Write a program in 'C' to implement circular queue using arrays. **MU - May 18, 10 Marks**

```
#include<stdio.h>
#include<conio.h>
#define SIZE 3

void enQueue();
void deQueue();
void display();

int MyQueue[SIZE], front = -1, rear = -1,element;

void main()
```

```
{
 int choice;

 while(1){
  printf("\n******* MENU *******\n");
  printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
  printf("Enter your choice: ");
  scanf("%d",&choice);
  switch(choice){
   case 1:
        enQueue();
        break;
   case 2: deQueue();
        break;
   case 3: display();
        break;
   case 4: exit(0);
   default: printf("\nInvalid choice!!!\n");
  }
 }
}

void enQueue()
{
```

Condition to check whether Queue is full

```
if((front == 0 && rear == SIZE - 1) || (front == rear+1))
    printf("\nCircular Queue is Full! Insertion not
   possible!!!\n");
 else
   {
   printf("\nEnter the element to be insert: ");
   scanf("%d",&element);
   if(rear == SIZE-1 && front != 0)
   rear = -1;
  MyQueue[++rear] = element;

   if(front == -1)
   front = 0;
 }
}
void deQueue()
{
 if(front == -1 && rear == -1)
   printf("\nCircular Queue is Empty! Deletion is not
   possible!!!\n");
 else
   {
   printf("\nDeleted element : %d\n",MyQueue[front++]);
   if(front == SIZE)
   front = 0;
   if(front-1 == rear)
   front = rear = -1;
 }
}
void display()
{
 if(front == -1)
   printf("\n Circular Queue is Empty!!!\n");
 else{
   int i = front;
   printf("\nCircular Queue Elements are : \n");
   if(front <= rear){
    while(i <= rear)
      printf("%d\t",MyQueue[i++]);
   }
   else{
    while(i <= SIZE - 1)
      printf("%d\t", MyQueue[i++]);
    i = 0;
    while(i <= rear)
      printf("%d\t", MyQueue[i++]);
   }
 }
}
```

Shifting rear at beginning empty position when reached to end

Deleting last element

Output

```
C:\qi.exe
****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be insert:  11

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be insert:  12

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be insert:  13

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

Circular Queue Elements are :
11      12      13
****** MENU ******
```

```
C:\qi.exe
****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 2

Deleted element : 11

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 1

Enter the element to be insert:  14

****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice: 3

Circular Queue Elements are :
12      13      14
****** MENU ******
1. Insert
2. Delete
3. Display
4. Exit
Enter your choice:
```

2.14.2 Difference between Simple Queue and Circular Queue

GQ. 2.14.7 *Give the difference between Simple Queue and Circular Queue.* **(3 Marks)**

Parameter	Simple Queue	Circular Queue
Presentation	Linear	Circular
Memory	Memory is wasted	No memory wastage
Front	Rear cannot shift from last to first position	Rear can shift from last to first position
Position	Rear is always next to Front	Rear can be set previous to Front

Module 2

Syllabus Topic : Priority Queue

2.15 PRIORITY QUEUE

UQ. 2.15.1 *Explain Priority Queue with example.* **MU -May 19, 5 Marks**

– Priority Queue is considered as an extension of queue with following given properties.

1. Every element in the queue has a priority associated with it.
2. An element which has high priority is dequeued before an element which has low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

2.15.1 Advantages of Priority Queue

GQ. 2.15.2 *List the advantages of priority queue* **(3 Marks)**

1. Simplicity
2. Reasonable support for priority
3. The most important tasks having higher priority are performed first.
4. Important tasks do not have to wait for completion of less priority tasks.

5. Suitable for applications with varying fast and slower requirements.

2.15.2 Applications of Priority Queue

GQ. 2.15.3 Write applications of priority queue. (2 Marks)

1. Priority queues can be used in Operating System for the purpose of job scheduling. Here the job with higher priority gets processed first.
2. Simulation systems in which priority corresponds to event times.
3. Graph algorithms such as Dijkstra's shortest path algorithm, Prim's Minimum Spanning Tree, etc.
4. All queue applications where priority is involved.

2.15.3 Types of Priority Queue

GQ. 2.15.4 Enlist types of priority Queue. (2 Marks)

- There are two types of priority queue :

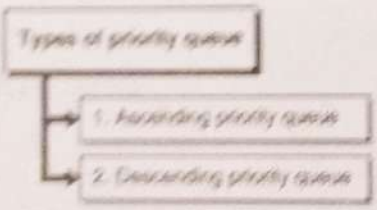


Fig. 2.15.1 : Types of Priority Queue

- **1. Ascending Priority Queue :** In this type of priority queue, elements can be added randomly but only the removal of smallest element is allowed first.
- **2. Descending Priority Queue :** In this type of priority queue, removal of largest element is allowed first.

2.15.4 Elements of Priority Queue

GQ. 2.15.5 What are the elements of priority queue? (2 Marks)

- As an element in priority queue, there may be numbers, characters or various types of complex structures which can be ordered on specific fields. For example records of Telephone Directory.
- The priority values used can be part of whole data elements. These may be external values.

2.15.5 Implementation of Priority Queue

- **Using Array**
- A simple implementation of priority queue can be done by using array of following structure.

```
struct data {
  int element;
  int priority;
};
```

- insert() operation can be carried out by adding an element at the location corresponding upon priority.
- delete() operation can be carried by deleting element from position.

- **Using Heap**
- Heap is a specialized tree-based data structure.
- It is generally preferred rather than array for priority queue implementation since the performance of heaps is better as compared to arrays or linked list.

2.15.6 Array Representation of Priority Queue

GQ. 2.15.6 With a neat sketch explain working of priority queue. (10 Marks)

Rules to be followed are :

1. The lower priority number indicates the higher priority.
2. Add jobs in the queue and arrange them priority wise.
3. When two jobs have same priority, they will be added order-wise.

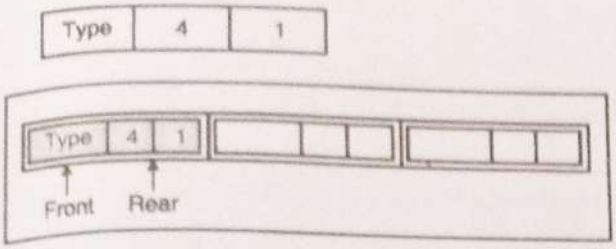
We have to add following jobs :

TYPE 4 1

SWAP 3 2

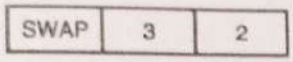
COPY 5 3

(a) First we will add

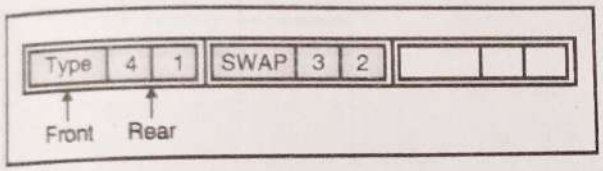


– Front and rear both points to the job TYPE with priority 4

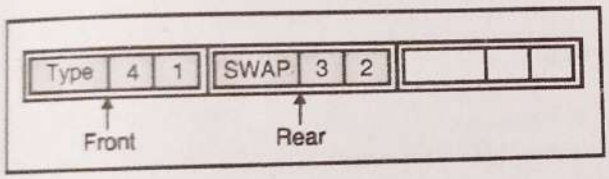
(b) Now we will add



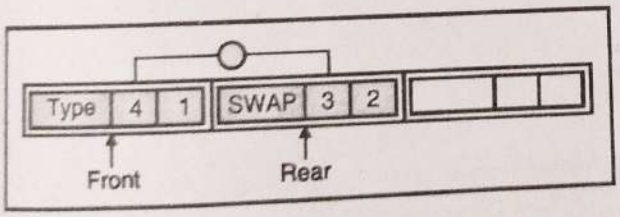
i.e. Job SWAP with priority 3.

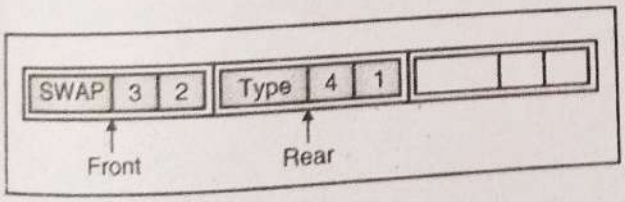


Rear would point to SWAP

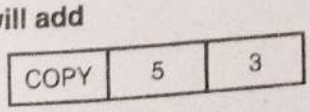


– The priority of SWAP is more that TYPE. Hence SWAP should be placed before TYPE.

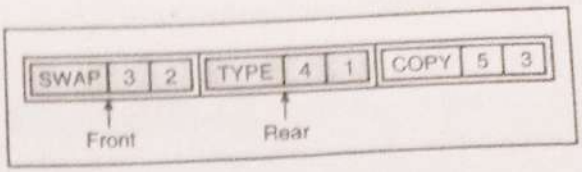




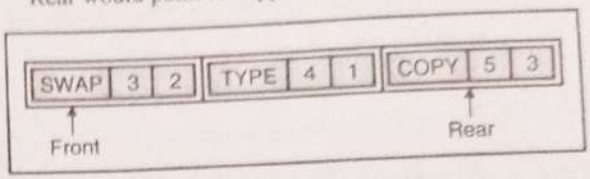
(c) Now we will add



i.e. Job COPY with priority 5.



– Rear would point to copy with priority 5



Module 2

☞ **Algorithm of program on priority Queue**

Step 1 : Start

Step 2 : Show options: 1. Insert 2. Delete 3. Display 4. Exit

Step 3 : Accept choice

Step 4 : As per user's choice call a function out of InsertPQ(), DeletePQ(), display(), exit()

Step 5 : Exit

☞ **Algorithm of insert element in priority queue**

Step 1 : If queue is full print the message and exit

Step 2 : Accept element

Step 3 : Search position for element as per priority by moving existing elements

Step 4 : Set new element at proper position

☞ **Algorithm to delete element from priority queue**

Step 1 : If queue is empty print the message and exit

Step 2 : Print front element as deleted

Step 3 : P = PQ[front]

Step 4 : Increment front by one

Step 5 : return P

☞ **Algorithm of Display**

Step 1 : If queue is empty print the message and exit

Step 2 : print elements from front to rear

2.15.7 Program on Priority Queue

UQ. 2.15.7 *Write program to implement priority queue.* **MU - Dec. 14, 8 Marks**

UQ. 2.15.8 *Write a C program to implement priority queue using arrays. The program should perform the following operations :*

(i) Inserting in a priority queue

(ii) Deletion from a queue

(iii) Displaying contents of the queue

MU - Dec. 18, 12 Marks

```
#define SIZE 5
int front=0,rear=-1;
typedef struct PRQ
{
  int element;
  int priority;
}PriorityQ;

PriorityQ PQ[SIZE];

InsertPQ(int elem, int pre)
{
  int i;
  if( Qfull())
  printf("\n\n Queue is Overflow\n");
  else
  {
    i=rear;
    ++rear;
    while(PQ[i].priority >= pre && i >= 0)
    {
      PQ[i+1]=PQ[i];
      i--;
    }
    PQ[i+1].element=elem;
    PQ[i+1].priority=pre;
  }
}

PriorityQ DeletePQ()
{
  PriorityQ p;
  if(Qempty())
  {
  printf("\n\nQueue is Underflow\n\n");
  p.element=-1;
  p.priority=-1;
  return(p); }
  else
  {
    p=PQ[front];
    front=front+1;
    return(p);
  }
}
int Qfull()
{
  if(rear==SIZE-1) return 1;
  return 0;
}

int Qempty()
{
  if(front > rear)
  return 1;
  else
  return 0;
}

display()
{
  int i;
```

Define typedef with members element and priority

Shifts elements with more priority to next position to make place for new element

If Queue not empty, then remove front element and increment front

```
if(Qempty()) printf(" \n Queue is empty \n");
else
{
  printf("Front->");
  for(i=front;i<=rear;i++)
    printf("[%d,%d] ",PQ[i].element,PQ[i].priority);
  printf("<-Rear");
}
}

main()
{
  int opn;
  PriorityQ p;
  do
  {
    clrscr();
    printf("\n Operations on Priority Queue \n\n");
    printf("\n 1-Insert, 2-Delete,3-Display,4-Exit\n");
    printf("\n Select your choice : ");
    scanf("%d",&opn);
    switch(opn)
    {
    case 1: printf("\n\nEnter the element and its Priority : ");
      scanf("%d%d",&p.element,&p.priority);
      InsertPQ(p.element,p.priority); break;
    case 2: p=DeletePQ();
      if( p.element != -1)
       printf("\n\nElement deleted : %d \n",p.element);
      break;
    case 3: printf("\n\nElements of Queue : \n");
      display(); break;
    case 4: printf("\n\n Exiting.... \n\n"); break;
    default: printf("\n\nInvalid choice !!! Try Again !! \n\n");
      break;
    }
    printf("\n\n\n\n Press a Key to Continue . . . ");
    getch();
  }while(opn != 4);
}
```

Display elements and priorities from front to rear

Menu, as per choice operation will be performed

Output

```
Operations on Priority Queue

1-Insert, 2-Delete,3-Display,4-Exit
Select your choice : 1

Enter the element and its Priority : 11 3


 Press a Key to Continue . . .
Operations on Priority Queue

1-Insert, 2-Delete,3-Display,4-Exit
Select your choice :
```

Module 2

Syllabus Topic : Introduction of Double Ended Queue / Dequeues

2.16 INTRODUCTION OF DOUBLE ENDED QUEUE (DQUEUE)

UQ. 2.16.1 *Explain double ended queue.*

MU - May 16, Dec. 16, Dec.19, 3 Marks

- **Double Ended Queue** is data structure in which the insertion and deletion operations are allowed at both the ends (front and rear).
- That means, we can insert at both front and rear positions and as well we can delete from both front and rear positions.

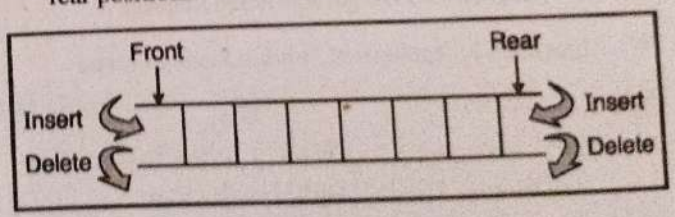


Fig. 2.16.1 : Dequeue

2.16.1 Representation of Dqueue

SQ. 2.16.2 What are the ways to represent Dqueue? **(4 Marks)**

- There are two ways to represent Double Ended Queue, those are as follows :

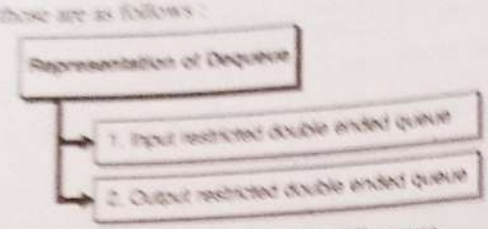


Fig. 2.16.2 : Representation of Dequeue

1. **Input Restricted Double Ended Queue**
   - In this type of queue, the insertion operation is allowed at only one end while deletion operation is allowed at both the ends.

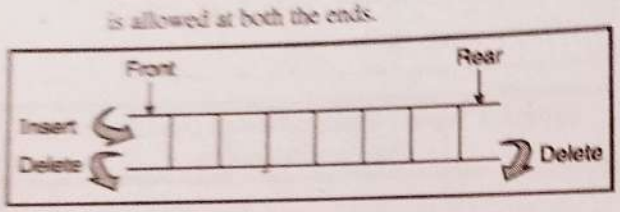


Fig. 2.16.3 : Input Restricted Dequeue

2. **Output Restricted Double Ended Queue**
   - In this type of queue, the deletion operation is allowed at only one end while insertion operation is allowed at both the ends.

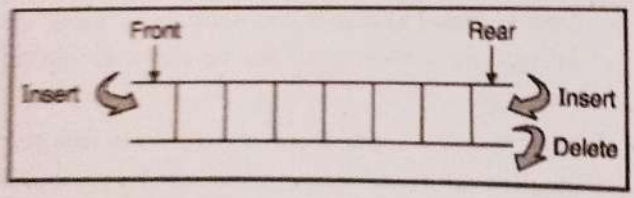


Fig. 2.16.4 : Output Restricted Dequeue

Algorithm to implement Double Ended Queue

Step 1 : Start

Step 2 : Show options: 1. Create 2. Insert Rear 3. Insert Front 4. Delete Rear 5. Delete Front

Step 3 : Accept choice

Step 4 : As per user's choice call a function out of enqueueRear(),enqueueFront(),dequeuerear(),dequeueFront()

Step 5 : Exit

Algorithm of EnqueueRear

Step 1 : If queue is full print the message and exit

Step 2 : Accept element

Step 3 : If queue is empty, set rear=front = 0

```
data[0] = element
    else
rear= (rear+1)%MAX
data[rear]= element
```

Algorithm of EnqueueFront

Step 1 : If queue is full print the message and exit

Step 2 : Accept element

Step 3 : If queue is empty, set rear=front = 0

```
data[0] = element
else
front=(front-1+MAX)%MAX
data[front]= element
```

Algorithm of DequeueFront

Step 1 : If queue is empty print the message and exit

Step 2 :

```
x=data[front]
if(rear=front)
  initialize()
else
  front=(front+1)%MAX
```

Step 3 : return(x)

Algorithm of DequeueRear

Step 1 : If queue is empty print the message and exit

Step 2 :

```
x=data[rear]
if(rear=front)
  initialize();
else
  rear=(rear-1+MAX)%MAX
```

Step 3 : return(x)

Algorithm of Display

Step 1 : If queue is empty print the message and exit

Step 2 : Print elements from front to rear

2.16.2 Program to implement Double Ended Queue (Dequeue)

UQ. 2.16.3 Write a C program to implement Double Ended Queue. **MU - Dec. 19, 8 Marks**

```
#include<stdio.h>
#include<process.h>
#define MAX 30
typedef struct dequeue
{
  int data[MAX];
  int rear,front;
}dequeue;
void initialize(dequeue *p);
int isEmpty(dequeue *p);
int isFull(dequeue *p);
void enqueueRear(dequeue *p,int x);
void enqueueFront(dequeue *p,int x);
int dequeueFront(dequeue *p);
int dequeueRear(dequeue *p);
void display(dequeue *p);

void main()
{
    int i,x,choice,n;
    dequeue q;
    initialize(&q);
    do
    {

printf("\n1.Create\n2.Insert(rear)\n3.Insert(front)\n4.Delete
(rear)\n5.Delete(front)");
    printf("\n6.Display\n7.Exit\n\nEnter your choice:");
    scanf("%d",&choice);

    switch(choice)
    {

case 1: printf("\nEnter number of elements:");
    scanf("%d",&n);
    initialize(&q);
    printf("\nEnter the elements :");

    for(i=0;i<n;i++)
    {
      scanf("%d",&x);
      if(isFull(&q))
      {
        printf("\nQueue is full!!");
        exit(0);
      }
      enqueueRear(&q,x);
    }
    break;

case 2: printf("\nEnter element to be inserted:");
    scanf("%d",&x);

    if(isFull(&q))
    {
      printf("\nQueue is full!!");
      exit(0);
    }

    enqueueRear(&q,x);
    break;

case 3: printf("\nEnter the element to be inserted:");
    scanf("%d",&x);

    if(isFull(&q))
    {
      printf("\nQueue is full!!");
      exit(0);
    }

    enqueueFront(&q,x);
```

Define typedef with members data, front and rear

Function prototypes

Menu, as per choice operation will be performed

Module 2

```
            break;
        case 4: if(isEmpty(&q))
                {
                  printf("\nQueue is empty!!");
                  exit(0);
                }
                x=dequeueRear(&q);
                printf("\nElement deleted is %d\n",x);
                break;
        case 5: if(isEmpty(&q))
                {
                  printf("\nQueue is empty!!");
                  exit(0);
                }
                x=dequeueFront(&q);
                printf("\nElement deleted is %d\n",x);
                break;
        case 6: display(&q);
                break;
        default: break;
      }
    }while(choice!=7);
}
void initialize(dequeue *P)
{
  P->rear=-1;
  P->front=-1;
}
```

Initially queue is empty

```
int isEmpty(dequeue *P)
{
  if(P->rear==-1)
    return(1);
  return(0);
}
int isFull(dequeue *P)
{
  if((P->rear+1)%MAX==P->front)
    return(1);
  return(0);
}
void enqueueRear(dequeue *P,int x)
{
  if(isEmpty(P))
  {
    P->rear=0;
    P->front=0;
    P->data[0]=x;
  }
  else
  {
    P->rear=(P->rear+1)%MAX;
    P->data[P->rear]=x;
  }
}
void enqueueFront(dequeue *P,int x)
{
  if(isEmpty(P))
  {
    P->rear=0;
    P->front=0;
    P->data[0]=x;
  }
  else
  {
    P->front=(P->front-1+MAX)%MAX;
    P->data[P->front]=x;
  }
}
int dequeueFront(dequeue *P)
{
  int x;
  x=P->data[P->front];
  if(P->rear==P->front)
    initialize(P);
  else
    P->front=(P->front+1)%MAX;
```

Adding first element at rear

Adding element at next positions

Adding first element

Deletes last element

```
    return(x);
}

int dequeueRear(dequeue *P)
{
    int x;
    x=P->data[P->rear];
    if(P->rear==P->front)
        initialize(P);
    else
        P->rear=(P->rear-1+MAX)%MAX;
    return(x);
}
void display(dequeue *P)
{
    if(isEmpty(P))
    {
        printf("\nQueue is empty!!");
        exit(0);
    }
    int i;
    i=P->front;

    while(i!=P->rear)
    {
        printf("\n%d",P->data[i]);
        i=(i+1)%MAX;
    }
        printf("\n%d\n",P->data[P->rear]);
}
```

When last element is deleted, queue become empty

Output

```
1.Create
2.Insert(rear)
3.Insert(front)
4.Delete(rear)
5.Delete(front)
6.Display
7.Exit

Enter your choice:1

Enter number of elements:3

Enter the elements :11 12 13

1.Create
2.Insert(rear)
3.Insert(front)
4.Delete(rear)
5.Delete(front)
6.Display
7.Exit

Enter your choice:2

Enter element to be inserted:14

1.Create
2.Insert(rear)
3.Insert(front)
4.Delete(rear)
5.Delete(front)
6.Display
7.Exit

Enter your choice:
```

Module 2

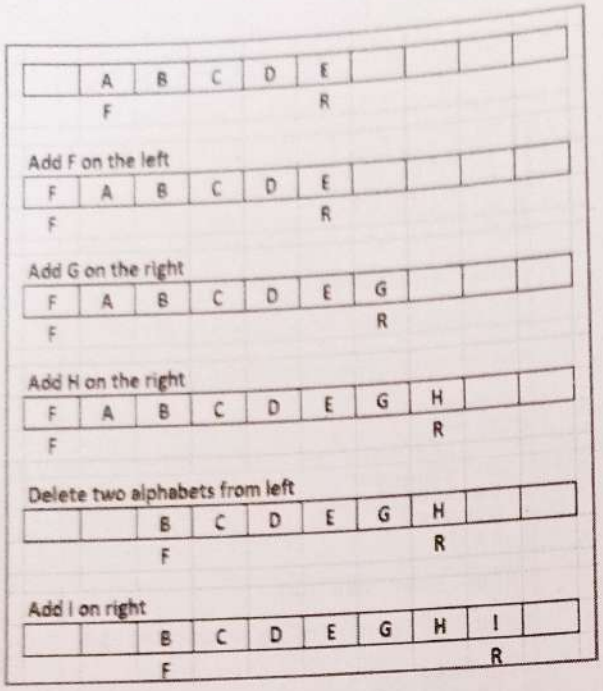
Example

GQ. 2.16.4 *Consider a dequeue given below which has LEFT=1, RIGHT=5*

_ A B C D E _ _ _ _

Now perform the following operations on the dequeue

1. *Add F on the left.*
2. *Add G on the right.*
3. *Add H on the right.*
4. *Delete two alphabets from left*
5. *Add I on the right* **(4 Marks)**



2.16.3 Difference between Circular Queue and Double-ended Queue

GQ. 2.16.5 *Compare circular queue and double-ended queue.* **(4 Marks)**

Parameter	Circular Queue	Double Ended Queue
Insertion	Insertion is allowed at one end	Insertion is allowed at both ends
Deletion	Deletion is allowed at one end	Deletion is allowed at both ends
Rear pointer	Rear pointer can be shifted to starting position from last position.	Rear pointer cannot be shifted to starting position from last position.
Position	Rear pointer can be before or after the front	Rear pointer is always ahead of the front

2.17 APPLICATIONS OF QUEUE

UQ. 2.17.1 *Discuss in brief any two applications of the queue data structure.* **MU - Dec. 14, 3 Marks**

- Queue, as the name suggests is basically useful when there is need to manage any group of elements in a sequence in which the first element coming in, also goes out first while all the other elements have to wait for their turn.
- It can be observed in following scenarios:
  1. Requests processing on a single resource which is of shared manner, such as a printer and task scheduling of CPU etc.
  2. In real life scenario, we can consider the Call Centre phone systems where there is use of queue concept. People are attended in an order. First call gets first priority and others are on hold until service representative becomes free.
  3. In real-time systems, the various types of interrupts which occurred are handled in the exact sequence in which they arrive i.e First come first serve.
  4. When there is data transferred in a asynchronous way between two different types of processes. For examples IO Buffers, pipes, file IO, etc.
- Computer systems must often provide a "holding area" for messages between two processes, two programs, or even two systems. This holding area is usually called a "buffer" and is often implemented as a queue.
  1. Simulation.
  2. Keyboard buffer.
  3. Round robin scheduling.