CLASSIFICATION OF DATA STRUCTURES, STACK, RECURSION AND QUEUE

NOTE: The classification of data structures has been covered in Module 2. The same topics are mentioned in Module 3 of the syllabus as well (may be by mistake). Hence, the topics from Stacks are covered here.

3.1 STACKS

Stack is non-primitive linear data structure into which, new items may be inserted and from which items may be deleted. In this data structure, the insertion and deletion of the elements are carried out at one end and is called as 'top' of the stack. In stack, the element last inserted will be the first to be deleted. Hence, stack is known as Last In First Out (LIFO) structure.

For example, consider the task of keeping books on a table one above the other. When a person wants to take the book, he has to take the book which was kept last. Thus, the first book kept on the table will be the last book to be taken out.

In the study of data structures, insertion of new item into the stack is called as ***push*** operation; whereas, deletion of an item at the top of the stack is ***pop*** operation. When the stack is full, we can't insert any more elements. This situation is called as ***stack overflow***. Similarly, when stack is empty, we can't delete element from it. This condition is known as ***stack underflow.***

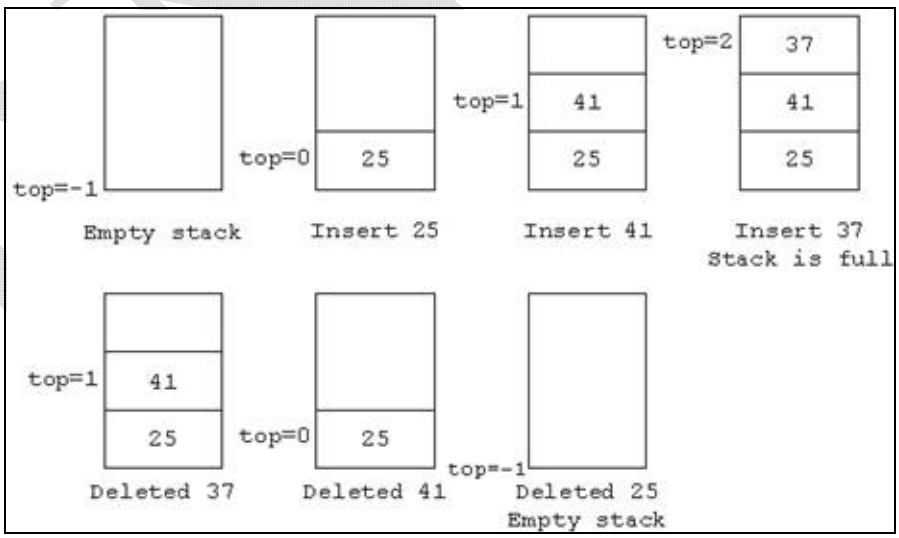


Figure 3.1 Demo of stack operations

An integer variable ***top*** is normally used for denoting current status of the stack – that is, the value of ***top*** gives the number of items of the stack. For programming purpose, an empty stack is denoted by setting the value of ***top*** as -1. Each time the ***push*** operation is

encountered, the ***top*** will be incremented. When the ***pop*** operation is done, ***top*** will be decremented. Usually, we define the size of the stack at the beginning. For example, stack of 5 elements, stack of 10 elements etc. When the value of ***top*** becomes -1 during deletion, it is stack underflow. When the ***top*** reaches the predefined size, it is stack overflow.

Figure 3.1 depicts the example of primitive operation on an integer stack of size 3.

Program 3.1 Primitive operations on stack

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void push(int st[], int item, int *t)
{
        if(*t==MAX-1)
        {
                printf("Stack full!!");
                return;
        }
        st[++(*t)]=item;
}

int pop(int st[], int *t)
{
        int item;

        if(*t== -1)
        {
            printf("\nStack Empty!!");
            return ;
        }
        item=st[(*t) - -];
        return item;
}

void disp(int st[], int *t)
{
        int i;

        if(*t== -1)
        {
                printf("\nStack Empty!!");
                return;
        }
```

```
        printf("\nStack contents:\n");
        for(i=*t ; i>=0; i- -)
                printf("\n%d",st[i]);
}

void main()
{
        int st[MAX], top= -1, opt, item;

        for(;;)
        {
                printf("\n*****Stack Operations****\n");
                printf("\n1. Push \n 2. Pop \n 3. Display \n 4. Exit\n");
                printf("Enter your option:");
                scanf("%d", &opt);

                switch(opt)
                {
                        case 1: printf("\nEnter item:");
                                scanf("%d",&item);
                                push(st, item, &top);
                                break;
                        case 2: item=pop(st, &top);
                                 printf("\n Deleted item is %d", item);
                                  break;
                        case 3: disp(st, &top);
                                  break;
                        case 4:
                        default:exit(0);
                }
        }
}
```

3.1.1 Application of Stacks

The concept of stacks has various applications in the field of computer science. Some of them includes conversion of arithmetic expressions, evaluation of expressions, recursion etc. Few of these applications are discussed in the following sections.

3.1.2 Conversion of Expressions

Let us consider an arithmetic expression

x op *y*

Here, x and y are two arithmetic expressions or operands and 'op' is the arithmetic operator like +, -, *, / etc. For example, consider a simple arithmetic expression, (x+y). Here x and y

are operands and + is an operator. Representing the equation like this is known as 'infix' expression. There are two more representations for denoting an arithmetic expression: xy+,
known as postfix expression
+xy, known as prefix expression

The words 'pre', 'post' and 'in' specifies the relative position of the operator in the expression.

Thus, any expression having an operator in between two operands is called as an ***infix expression***. Any expression having an operator followed by two operands is ***postfix expression***. An expression, in which the operator precedes the two operands, is a ***prefix expression***.

Note that an infix expression may have parentheses in-between. But, postfix and prefix expressions will not be having any parentheses. While converting a particular infix expression into either prefix or postfix expression, one should consider precedence of operators, as given below:

Precedence	Operator
1	(or) -> parentheses
2	^ or $ -> exponentiation
3	* or / -> Multiplication or division
4	+ or - -> Addition or subtraction

Example:
Convert the following infix expression into postfix and prefix expression:
((A+B)$C-(D/E)*F)

Solution: ***Conversion into Postfix:***

- The given expression is ((A+B) $ C – (D/E) * F)
- We have to resolve inner brackets first.
- Consider, (A+B). This will be AB+ in postfix. Let P = AB+
- Consider, (D/E). This is DE/ in postfix. Let Q= DE/
- Now, the expression will be (P$C – Q * F)
- Now resolve operators with high precedence. That is $ (power symbol) and * (multiplication)
- That is, P$C = PC$ Let R = PC$
- And, Q * F = QF* Let S= QF*
- Then, expression is (R-S), which when converted into postfix, gives RS –
- Now, by replacing the values of R and S and in turn, the values of P and Q we will get –
    - PC$QF* –
    - AB+C$DE/F*–
- The required postfix expression is **AB+C$DE/F*–**

Conversion into Prefix:

- Consider, ((A+B)$C – (D/E)*F)
- As we did in conversion into postfix notation, here also, we make use of some temporary variables like P, Q etc.
- Consider (A+B). In prefix notation, this will be +AB. Let P = +AB.
- Let (D/E) = /DE = Q.

- Now, the expression is (P$C – Q*F)
- Again, P$C=$PC = R, say and, Q*F = *QF = S
- Then, expression is (R-S) = –RS
- Now, putting the values of P, Q, R and S, we will get –
  - ⇨ –$PC*QF
  - ⇨ **–$+ABC*/DEF**, is the required prefix expression.

Procedure for converting an Infix expression to Postfix expression programmatically:

- For converting an infix expression to postfix expression, one has to analyze the precedence value of a symbol or element both in input and in stack.
- If an operator is left associative, then the input precedence value is less than the stack precedence value.
- If the operator is right associative, then, the input precedence value is greater than the stack precedence value.

The following table is used for writing an algorithm and program for conversion of infix to postfix.

Symbols	Input precedence (IP)	Stack precedence (SP)
+ or –	1	2
* or /	3	4
$ or ^	6	5
Operands	7	8
(	9	0
)	0	-
#	-	–1

Algorithm/Pseudo code for converting infix expression into postfix expression:

1. Initialize empty stack with symbol '#'.
   i.e. st[0]='#'
2. Read the character from infix expression.
   i.e. symbol=infix[i]

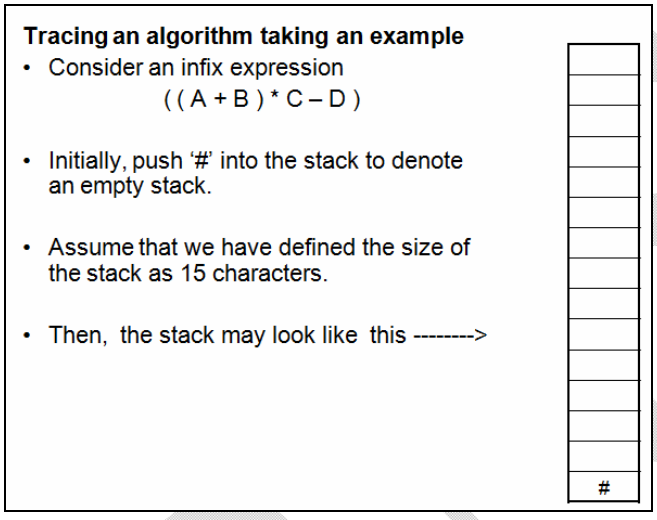
3. while SP(st[top]) > IP(symbol)
   postfix [j]=pop(st[top])

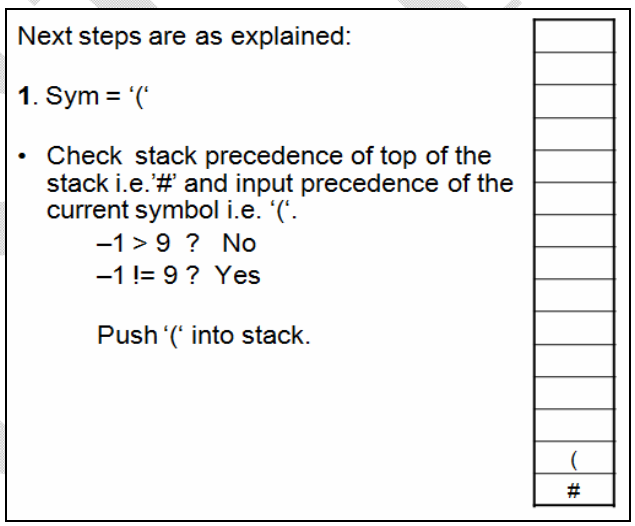
4. if SP(st[top]) != IP(symbol)
   push(symbol)
   else
   pop(st[top])

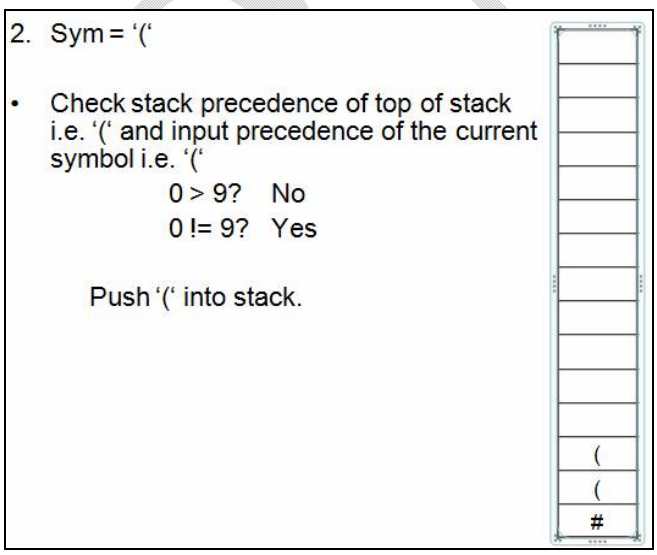
5. Repeat the steps (2) to (4) till the last character of infix expression.

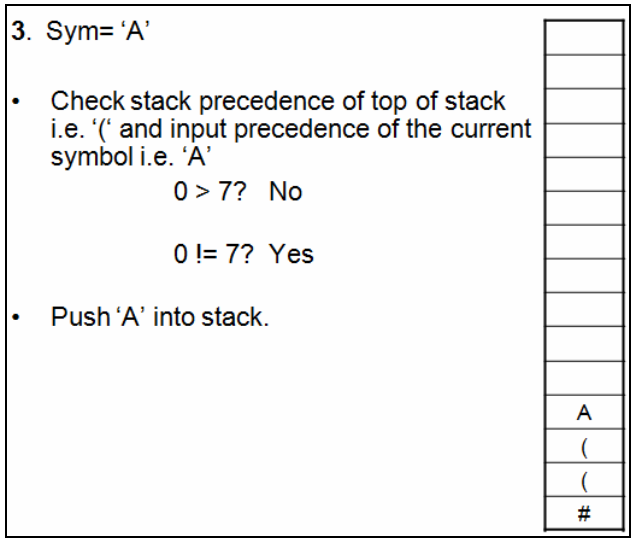
6. While stack becomes empty, //only for partially parenthesized expression
   postfix [j]=pop(st[top])

Observe the following figures to understand the tracing of the above algorithm.









4. Sym = '+'

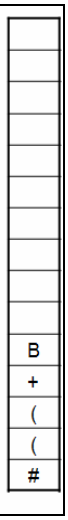
- Check stack precedence of top of stack i.e. 'A' and input precedence of current symbol i.e. '+'
  8 >1 ? Yes
- Pop A and put into postfix expression.
- Since, we are in a while loop, we have to check the condition once again.
- So, check the stack precedence of top of stack i.e. '(' and the input precedence of current input symbol i.e. '+'
  0>1? No
  0=1? No
- Push '+' into the stack

Postfix Expression: A

5. Sym= 'B'

- Check stack precedence of top of stack i.e. '+' and the input precedence of current input symbol i.e. 'B'
  2>8? No
  2=8? No
- Push 'B' into the stack

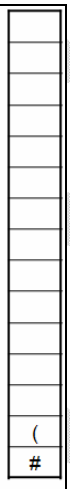
Postfix Expression: A



6. Sym = ')'

- Check stack precedence of top of stack i.e. 'B' and the input precedence of current input symbol i.e. ')'
  8>0? Yes
- Pop 'B' and place it into the postfix expression.
- Again, check the stack precedence of top of stack i.e. '+' and the input precedence of current input symbol i.e. ')'
  2>0? Yes
- Pop '+' and put it into the postfix expression.
- Again, check the stack precedence of top of the stack i.e. '(' and the input precedence of current input symbol i.e. ')'
  0>0? No
  0!=0? No
  so, *else* block of algorithm will be executed.
- So, pop '(' from the stack, but ***do not*** put into the postfix expression.

Postfix Expression: A B +



7. Sym= '*'

- Check the stack precedence of top of the stack i.e. '(' and the input precedence of current input symbol i.e. '*'
  0>3? No
  0=3? No
- So, push '*' into the stack

8. Sym= 'C'

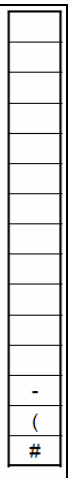
- Check the stack precedence of top of the stack i.e. '*' and the input precedence of current input symbol i.e. 'C
  4>7? No
  4=7? No
- So, push 'C' into the stack

Postfix Expression: A B +

9. Sym= '-'

- Check the stack precedence of top of the stack i.e. 'C' and the input precedence of current input symbol i.e. '-'
  8>1? Yes
- Pop 'C' and put into postfix expression.
- Again, check the stack precedence of top of the stack i.e. '*' and the input precedence of current input symbol i.e. '-'
  4>1? Yes
- Pop '*' and put into postfix expression.
- Again, check the stack precedence of top of the stack i.e. '(' and the input precedence of current input symbol i.e. '-'
  0>1? No
  0=1? No
- Push '-' into the stack.

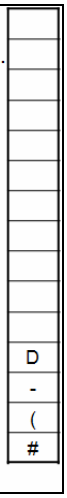
Postfix Expression: A B + C *

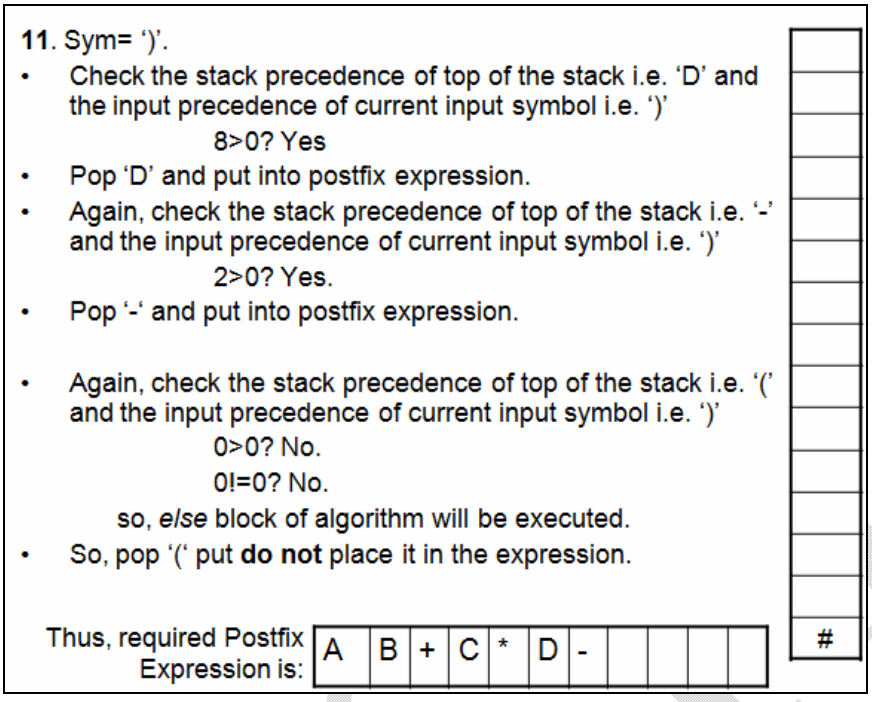


10. Sym= 'D'

- Check the stack precedence of top of the stack i.e. '-' and the input precedence of current input symbol i.e. 'D'
  2>7? No
  2=7? No
- Push 'D' into the stack.

Postfix Expression: A B + C *





Program 3.2 Converting a valid infix expression into the postfix expression:

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

int inputpre(char sym)        //Function for input precedence
{
        switch(sym)
        {
                case '+' :
                case '-' : return 1;
                case '*' :
                case '/' : return 3;
                case '^' :
                case '$' : return 6;
                case '(' : return 9;
                case ')' : return 0;
                default : return 7;
        }
}

int stackpre(char sym)        //Function for stack precedence
{
        switch(sym)
        {
                case '+' :
```

```
                case '-' : return 2;
                case '*' :
                case '/' : return 4;
                case '^' :
                case '$' : return 5;
                case '(' : return 0;
                case '#' : return -1;
                default : return 8;
        }
}

void push (char item, int *top, char s[])
{
        s[++(*top)] = item;
}

char pop(int *top, char s[])
{
          return s[(*top)--];
}

void infix_to_postfix (char ifix[], char pfix[])
{
        int top = -1, i, j = 0;
        char s[30] , sym;

        push('#',&top,s);

        for(i=0;i < strlen(ifix);i++)
        {
                sym = ifix[i];
                while (stackpre(s[top]) > inputpre(sym))
                        pfix[j++] = pop(&top,s);

                if(stackpre(s[top]) != inputpre(sym))
                        push(sym,&top,s);
                else
                        pop(&top,s);
        }

        while(s[top] != '#')
                pfix[j++] = pop(&top,s);

        pfix[j] = '\0';
}
```

```
void main()
{
        char ifix[20], pfix[20];
        clrscr();

        printf("Enter valid infix expression\n");
        scanf("%s", ifix);
        infix_to_postfix (ifix,pfix);
        printf("The postfix expression is = %s", pfix);
}
```

Procedure for converting an Infix expression to Prefix expression:
To convert an infix expression into prefix expression, we have to use the following precedence table.

Symbols	Input precedence (IP)	Stack precedence (SP)
+ or -	2	1
* or /	4	3
$ or ^	5	6
Operands	7	8
(	0	--
)	9	0
#	-	-1

Algorithm: The algorithm to convert an infix expression into prefix expression is given below:

- Reverse the given infix expression.
- Follow the steps used for obtaining postfix expression, but use the above precedence table.
- Reverse the result obtained.

Program 3.3 Converting a valid infix expression into the prefix expression:

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

int inputpre(char sym)
{
        switch(sym)
        {
                case '+' :
                case '-' : return 2;
                case '*' :
                case '/' : return 4;
                case '^' :
                case '$' : return 5;
```

```
                case '(' : return 0;
                case ')' : return 9;
                default : return 7;
        }
}

int stackpre(char sym)
{
        switch(sym)
        {
                case '+' :
                case '-' : return 1;
                case '*' :
                case '/' : return 3;
                case '^' :
                case '$' : return 6;
                case ‘)' : return 0;
                case '#' : return -1;
                default : return 8;
        }
}

void push (char item, int *top, char s[])
{
        s[++(*top)] = item;
}

char pop(int *top, char s[])
{
          return s[(*top)--];
}

void infix_to_prefix (char ifix[], char pfix[])
{
        int top = -1, i, j = 0;
        char s[30] , sym;

        push('#',&top,s);
        strrev(ifix);

        for(i=0;i < strlen(ifix);i++)
        {
                sym = ifix[i];
                while (stackpre(s[top]) > inputpre(sym))
                        pfix[j++] = pop(&top,s);
```

```
                if(stackpre(s[top]) != inputpre(sym))
                        push(sym,&top,s);
                else
                        pop(&top,s);
        }

        while(s[top] != '#')
                pfix[j++] = pop(&top,s);

        pfix[j] = '\0';
        strrev(pfix);
}

void main()
{
        char ifix[20], pfix[20];
        clrscr();

        printf("Enter valid infix expression\n");
        scanf("%s", ifix);
        infix_to_prefix (ifix,pfix);
        printf("The prefix expression is = %s", pfix);
}
```

Evaluation of Postfix expression:

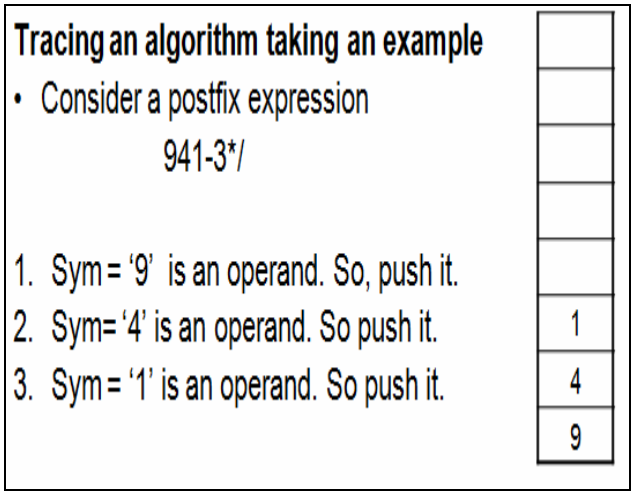
To evaluate an infix expression, we will scan from left to right repeatedly. But if the expression contains parentheses, evaluation becomes complex as the parentheses changes the order of precedence. So, it is always easy to evaluate an infix expression by converting it into either prefix or postfix expression.

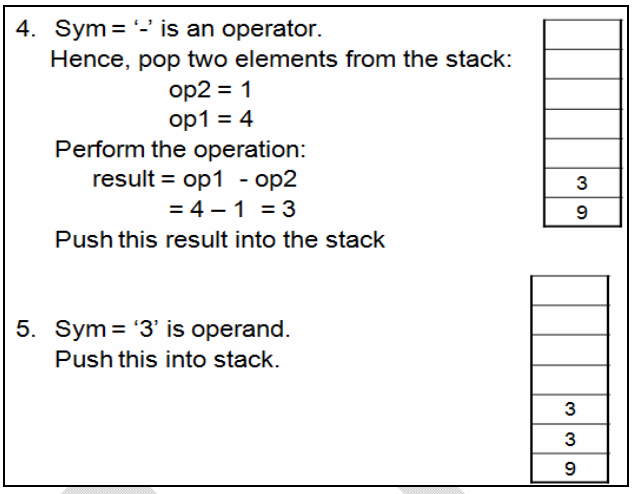
Algorithm to evaluate postfix expression:

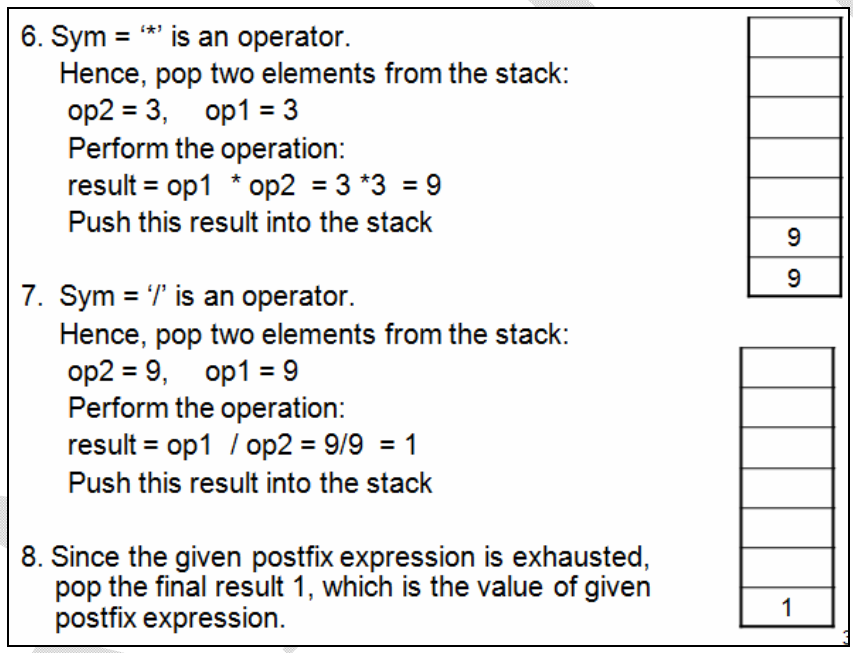
1. Scan a symbol in postfix expression from left to right.
2. If the symbol is operand, push it into stack.
3. If the symbol is an operator, pop two elements from the stack and perform the operation indicated.
4. Push the result of step (3) into the stack.
5. Repeat the above steps till all the symbols get exhausted in the given postfix expression.
6. Now, pop the element from the stack, which will be the result of entire postfix expression.

Note that, here, a single digit is treated as an operand and scanned.

Observe the following figures to understand the working of above algorithm.







Program 3.4 Evaluating a valid postfix expression:

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

float oper(char sym, float op1, float op2)
{
        switch(sym)
        {
                case '+': return op1 + op2;
```

```
                case '-': return op1 - op2;
                case '*': return op1 * op2;
                case '/': if(op2== 0)
                        {
                          printf("Can't evalueate");
                          exit(0);
                        }
                        return op1 / op2;
                case '^':
                case '$': return pow(op1,op2);
        }
}

void push(float item, int *top, float s[ ])
{
        s[++(*top)] = item;
}

float pop(int *top, float s[ ])
{
        return s[(*top)--];
}

void main()
{
        float s[20], result, op1, op2, x;
        int top = -1, i;
        char postfix[20], sym;

        printf("Enter valid postfix expression\n");
        scanf("%s",postfix);

        for(i=0;i<strlen(postfix);i++)
        {
                sym = postfix[i];

                if(isdigit(sym))
                        push(sym-'0', &top, s);   // character to digit conversion
                else if (isalpha(sym))
                {
                        printf("Enter the value of %c: ", sym);
                        scanf("%f",&x);
                        push(x,&top,s);
                }
```

```
                else
                {
                        op2 = pop(&top,s);
                        op1 = pop(&top,s);
                        result = oper(sym,op1,op2);
                        push(result,&top,s);
                }
        }
        result = pop(&top,s);
        printf("Result =%.4f",result);
}
```

Sample Output 1:

Enter valid postfix expression: 941-3*/
Result = 1.0000

Sample Output 2:

Enter valid postfix expression: abc-d*/
Enter value of a: -54
Enter value of b: 23
Enter value of c: 5
Enter value of d: 8
Result = -0.3750

NOTE: The **sample output 1** takes the postfix expression with digits. Hence, each digit has to be treated as one operand. Whereas, in the **Sample Output 2,** the postfix expression is a series of alphabets (variables). So, the program will allow you to read the values for each of these variables. And hence, the user can give an operand containing more than one digit, a floating point number, a negative number etc. In the **Sample Output2,** the meaning of expression (in infix format) will be –

- 54 / ((23-5)*8) = - 0.3750.

3.2 RECURSION

Recursion is a technique of defining something in terms of itself. In the field of mathematics and computer science, many concepts can be explained using recursion. In computer terminology, if a function calls itself then it is known as recursive function. If the function calls itself directly, then it is known as *direct recursion.* For Example:

```
void myfun()
{       -----
    -----
    myfun();
    -----
}
```

If the function calls itself through another function, then it is known as *indirect recursion.* For example:

```
void myfun()
{
    -----
    -----
    fun();
    -----
    -----
}
void fun()
{
    ------
    ------
    myfun();
    -------
    -------
}
```

Here, the function myfun() is calling the function fun(), which in turn calls myfun().

3.2.1 Factorial of a Number

Factorial of a non-negative integer n is defined as the product of all integers from 1 to n. That is:

n! = n(n-1)(n-2)…3.2.1 , for all n>=1 and
0! = 1.

By definition we have, n!= n(n-1)!

Thus, factorial is a recursive function.

Consider one example of finding 5! –

```
5! = 5*(5-1)!
   = 5*4!
   = 5*4*3!
   = 5*4*3*2!
   = 5*4*3*2*1!
   = 5*4*3*2*1*0!
   = 5*4*3*2*1*1
   = 120
```

This procedure can be implemented through program given below:

```
#include<stdio.h>

int fact(int n)
{
        if (n==0)
                return 1;
        return n* fact(n-1);
}
```

```
void main()
{
        int n;

        printf("Enter the value of n");
        scanf("%d", &n);

        if(n<0)
                printf("Invalid input");
        else
                printf("\nFactorial of %d is %d ", n, fact(n)) ;
}
```

The output would be:

Enter the value of n: 6
The factorial of 6 is 720

3.2.2 Fibonacci Sequence

A sequence of integers is called Fibonacci sequence if an element of a sequence is the sum of its immediate two predecessors. That is:

f(x)=0 for x=1
f(x)=1 for x=2
f(x)=f(x-2)+f(x-1) for x>2

Thus the Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, ….

Consider an example of finding 6th Fibonacci number.

f(6) = f(4) + f(5)
= f(2) + f(3) + f(3) + f(4)
= 1 + f(1) + f(2) + f(1) + f(2) + f(2)+ f(3)
= 1 + 0 + 1 + 0 + 1 + 1 + f(1) +(2)
= 4 + 0 + 1
= 5

The above procedure can be illustrated through the program given below:

Finding nth Fibonacci number

```
#include<stdio.h>

int fibo(int n)
{
        if (n==1)
                return 0;
        else if (n==2)
```

```
            return 1;
        return fibo(n-1) + fibo (n-2);
}

void main()
{
        int n;

        printf("Enter value of n");
        scanf("%d", &n);

        if(n<=0)
                printf("invalid input");
        else
                printf("%d th fibonacci number is %d ", n, fibo(n));
}
```

The output would be:

Enter the value of n : 10
10th fibonacci number is 34

3.2.3 Greatest Common Divisor (GCD)

The GCD of two numbers (at least one of them should be non-zero) can be computed using Euclid's algorithm as –

$$GCD(m,n) = \begin{cases} GCD(n, m\%n), & \text{if } n!=0 \\ m & \text{if } n=0 \end{cases}$$

Consider an example of finding GCD of 75 and 20.

```
GCD(75, 20) = GCD(20, 75%20)
            = GCD(20, 5)
            = GCD(5, 20%5)
            = GCD(5, 0)
            = 5
```

Program for finding GCD of two numbers:

```
#include<stdio.h>

int GCD(int m, int n)
{
        if (n==0)
                return m;
        return GCD(n, m%n);
}
```

```
void main()
{
        int m, n, gcd;

        printf("Enter two positive integers:");
        scanf("%d%d", &m, &n);

        if(m==0 && n==0)
        {
                printf("At least one of the numbers should not be zero!!");
                return;
        }

        gcd=GCD(m, n);
        printf("\nGCD=%d ", gcd);
}
```

3.2.4 Binary Search

This method is applied on the sorted array. Initially, the key element is compared with the middle element of the array. If they are equal then the search is successful. Otherwise, the array is divided into two parts viz. from the first element to middle element and from middle to last element. If the key element is greater than the middle element then the second sub-array is searched for. If the key is less than the middle then the first sub-array is searched. The procedure is repeated till the key is found or till the sub-array contains a non-matching single element.

Program for Binary Search:

```
#include<stdio.h>

int binary(int item, int a[],int low,int high)
{
        int mid;
        if(low<=high)
        {
                mid=(low+high)/2;

                if(a[mid]==item)
                        return mid+1;
                if(a[mid]>item)
                        return binary(item,a,low,mid-1);

                return binary(item,a,mid+1,high);
        }
        return -1;
}
```

```
void main()
{
        int a[10],n,pos,item, i;

        printf("Enter the array size:");
        scanf("%d",&n);
        printf("Enter the elements in ascending order\n");
        for(i=0;i<n;i++)
                scanf("%d",&a[i]);
        printf("Enter the key element:");
        scanf("%d",&item);

        pos=binary(item,a,0,n-1);

        if(pos==-1)
                printf("\nItem not found");
        else
                printf("\nItem found at position %d",pos);
}
```

3.2.5 Tower of Hanoi Problem

In the problem of Tower of Hanoi, there will be three poles, viz. A, B and C. Pole A (source pole) contains 'n' discs of different diameters and are placed one above the other such that larger disc is placed below the smaller disc. Now, all the discs from source (A) must be transferred to destination (C) using the pole B as temporary storage.

Conditions are:

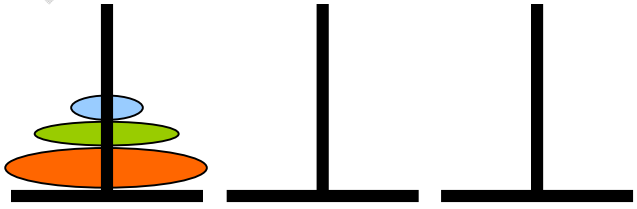
- Only one disc must be moved at a time
- smaller disc is on the top of larger disc at every step

Algorithm to move n discs from source to destination is as follows –

(i) move n-1 discs from source to temporary
(ii) move n^{th} disc from source to destination
(iii) move n-1 disc from temporary to destination

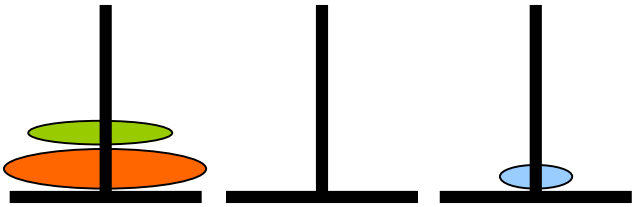
For example, consider the number of discs = 3. Then, various steps involved in transferring 3 discs from source to destination are shown below –

Initial Stage

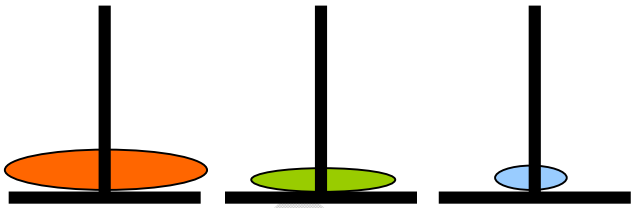


Source Temp Destination

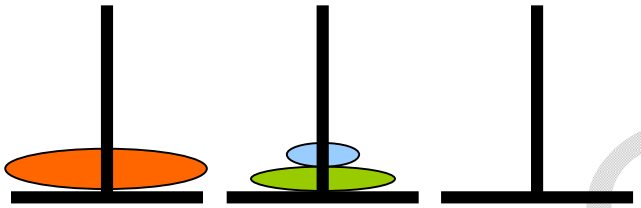
Step 1:



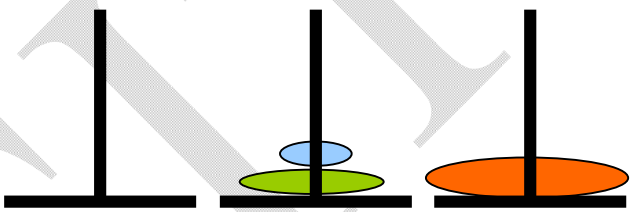
Step 2:



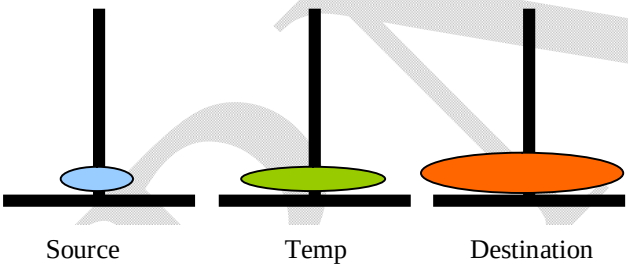
Step 3:



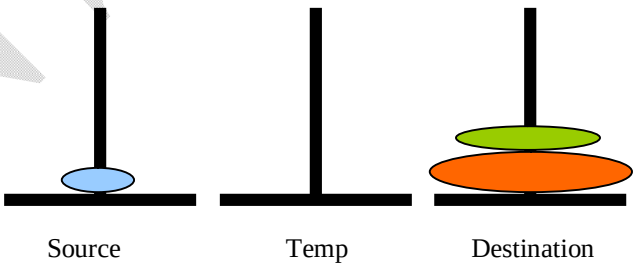
Step 4:



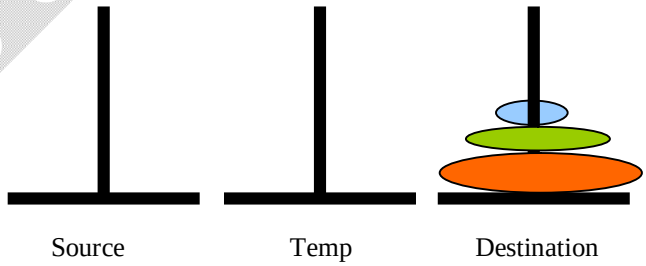
Step 5



Step 6



Step 7



Program:

```
#include<stdio.h>
int count=0;

void tower(int n, char s, char t, char d)
{
        if(n==1)
        {
                printf("Move disc 1 from %c to %c ", s, d);
                count ++;
                return;
        }
        tower(n-1, s, d, t);
        printf("Move disc %d from %c to %c", n, s,d);
        count ++;
        tower(n-1, t, s, d);
}
void main()
{
        int n;
        printf("Enter the number of discs");
        scanf("%d", &n);
        tower(n, 'A','B','C');
        printf("Total number of moves =%d", count);
}
```

Output:

```
Enter the number of discs: 3
Move disc 1 from A to C
Move disc 2 from A to B
Move disc 1 from C to B
Move disc 3 from A to C
Move disc 1 from B to A
Move disc 2 from B to C
Move disc 1 from A to C
Total number of disc moves = 7
```

3.2.6 Properties of Recursive Algorithms

When a function calls itself, a new set of local variables and parameters are being allocated memory in the stack. Then the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function, only the values being operated upon are new. When each recursive call returns, the old local variables and parameter are removed from the stack and the execution resumes at the point of the function call inside the function.

Some facts about recursive functions are:

- Recursive functions do not usually reduce the code size.
- They do not improve memory utilization compared to iterative functions.
- Many of the recursive functions may execute a bit slower compared to iterative functions because of the overhead of repeated function calls.
- They may cause a stack overrun, as each new call to the recursive function creates a new copy of variables and parameters and puts them into the stack.
- The advantage of using recursive functions is to create clearer and simpler programs.
- Moreover, some of the algorithms do require recursive functions as the iterative versions of them are quite difficult to write and implement.
- While writing a recursive function, a conditional statement has to be included which will terminate the recursive function without using a recursive function call.
- Otherwise, the recursive function will enter into an infinite loop and will not terminate at all.

Thus, any recursive function should satisfy the following conditions:

- In each and every call, the function must be nearer to the solution. (In other words, at every step, the problem size must reduce)
- There should be at least one non-recursive exit condition.

3.3 QUEUES

In our day-to-day life, we come across many situations in which we have to stand in a queue like at a bank counter, at a cinema hall etc. In the field of computer science also we can give the examples for queue system such as printing a several files using only one printer, Processes waiting to be executing by CPU etc. Using this analogy, the data structure 'queue' is defined.

Queue is a non-primitive linear data structure, where the elements are inserted at rear end and deleted from the front end.

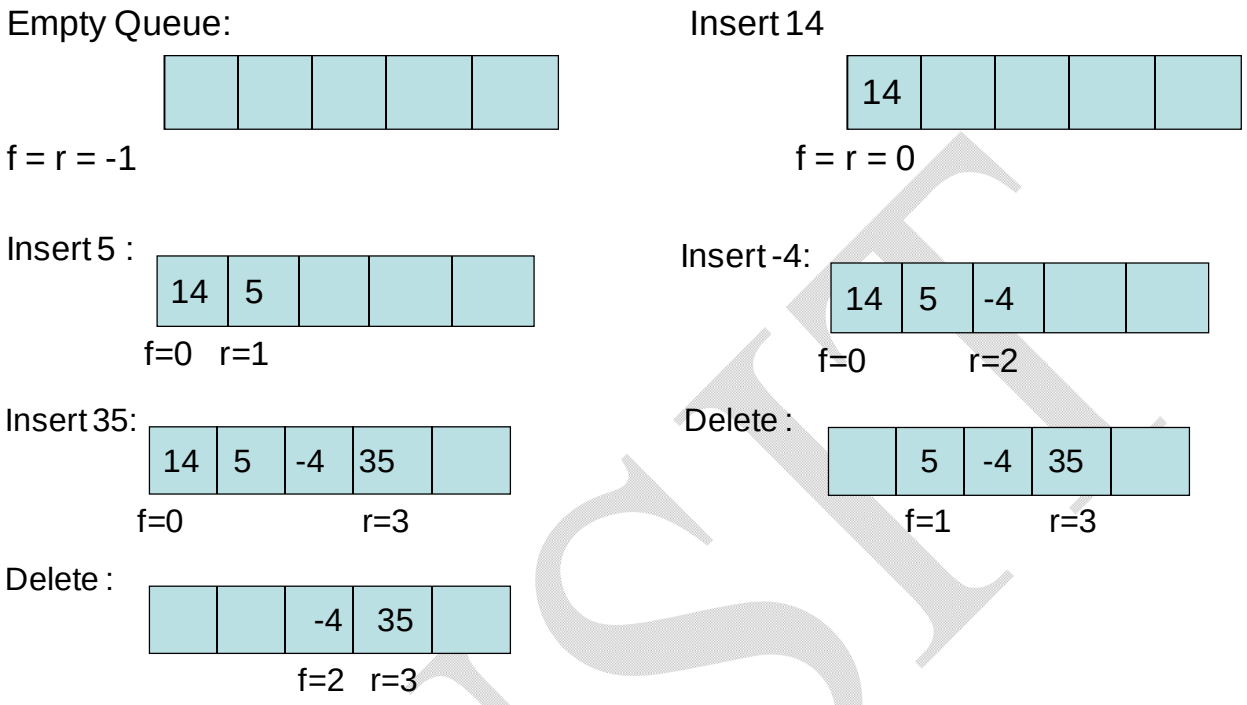
The item inserted first will be the first to be deleted. Hence it is known as ***First In First Out (FIFO)*** data structure.

The primitive operations of a queue structure include:

- Inserting an element into queue
- Deleting element from queue
- Displaying the contents of a queue

An attempt to insert an element into a full queue is called as ***Queue overflow***. Trying to delete an element from an empty queue is known as ***queue underflow.*** The status of the queue is maintained by two variables viz. ***front*** and ***rear.*** Initially, both *front* and *rear* are set to -1 to indicate empty queue. While inserting the very first element into queue, both *front* and *rear* are incremented. After that, in every insertion, *rear* will be incremented. The *front* will be incremented at every deletion.

Consider the following illustration to understand the working of queue.



Note that, after every insertaion, ***rear*** is incremented and after every deletion, ***front*** is incremented.

Program for Primitive operations on ordinary queue

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void insert(int q[ ], int *f, int *r, int item)
{
        if(*r==MAX-1)
        {
                printf("\nQueue overflow");
                return;
        }
        q[++(*r)]=item;

        if(*f== -1)
                (*f)++;
}
```

```
void del(int q[], int *f, int *r)
{
        if(*f==-1 || *f>*r)
        {
                printf("\nQueue underflow");
                return;
        }
        printf("\nDeleted item is %d", q[(*f)++]);
}

void disp(int q[], int *f, int *r)
{
        int i;

        if(*f== -1 || *f>*r)
        {
                printf("\nNo elements to display!!");
                return;
        }
        printf("\n Contents of queue:\n");

        for(i=*f;i<=*r;i++)
                printf("%d\t",q[i]);
}

void main()
{
        int q[MAX], f=-1, r=-1,item, opt;

        for(;;)
        {
                printf("\n*****Queue operations*****");
                printf("\n1.Insert\n 2.Delete\n 3.Display \n 4.Exit");
                printf("\nEnter your option: ");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nEnter item to be inserted:");
                                scanf("%d",&item);
                                insert(q,&f,&r,item);
                                break;
                        case 2: del(q, &f, &r);
                                break;
```

```
                    case 3: disp(q, &f,&r);
                            break;
                    case 4:
                    default:exit(0);
                }
            }
        }
```

3.3.1 Circular Queue

We have seen in the previous program that when an element is deleted from the queue, the **front** is incremented and when the element is inserted, **rear** is incremented. Such implementation of a queue has a drawback. To understand the drawback, consider the following illustration:

Assume, we have inserted the elements into queue up to its full capacity. That is, *rear* reached maximum size. Then, if we delete some elements from front end, there will be empty spaces at the beginning of a queue. But still, we can't insert elements into queue, as *rear* is already at *MAX.*

Let *MAX=3*

Now, if we try to insert an item, there will be queue overflow, as the condition (r==MAX-1) is true. But, we can see that, two positions in the beginning are actually free to receive the elements. To overcome this problem, we adopt ***circular queue.***

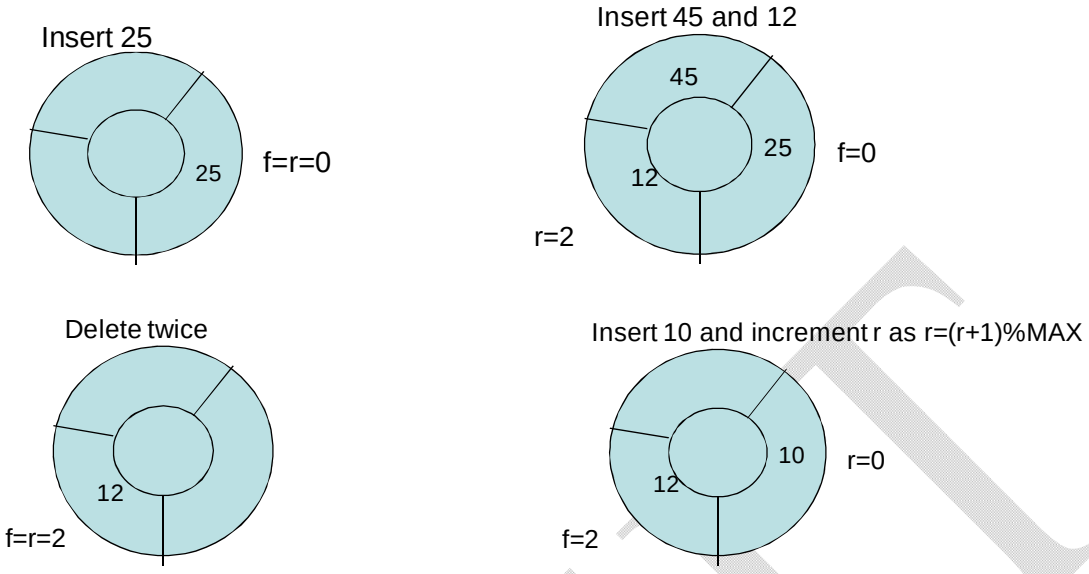
In ordinary queue, we will just increment ***front*** and ***rear***. Hence, there is chance that ***rear*** will go beyond the range of the size of the queue. Hence, in circular queue, instead of using the statements like

f= f+1 and r=r+1,

we use

f=(f+1)%MAX and r=(r+1)%MAX

This will ensure that both ***front*** and ***rear*** will fall within the range of MAX always. This can be illustrated as below –



Thus, we can overcome the problem with ordinary queue using circular queue.

Program for Implementation of Circular Queue

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void insert(int q[], int *f, int *r, int item)
{
        if(*f==(*r+1)%MAX)
        {
                printf("\nQueue overflow");
                return;
        }

        *r=(*r+1)%MAX;
        q[*r]=item;

        if(*f== -1)
                (*f)++;
}

void del(int q[], int *f, int *r)
{
        if(*f==-1 )
        {
                printf("\nQueue underflow");
```

```
                return;
        }

        printf("\nDeleted element is %d", q[*f]);

        if(*f==*r)
                *f=*r=-1;
        else
                *f=(*f+1)%MAX;
}

void disp(int q[], int *f, int *r)
{
        int i;

        if(*f== -1)
        {
                printf("\nNo elements to display!!");
                return;
        }
        printf("\n Contents of queue:\n");

        if(*f>*r)
        {
                for(i=*f;i<MAX;i++)
                        printf("%d\t",q[i]);
                for(i=0;i<=*r;i++)
                        printf("%d\t", q[i]);
        }
        else
        {
                for(i=*f;i<=*r;i++)
                        printf("%d\t",q[i]);
        }
}

void main()
{
        int q[10], f=-1, r=-1,item, opt;

        for(;;)
        {
                printf("\n*****Circular Queue operations*****");
                printf("\n1.Insert\n 2.Delete\n 3.Display \n 4.Exit");
                printf("\nEnter your option: ");
```

```
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nEnter item to be inserted:");
                                scanf("%d",&item);
                                insert(q,&f,&r,item);
                                break;
                        case 2: del(q, &f, &r);
                                break;
                        case 3: disp(q, &f,&r);
                                break;
                        case 4:
                        default:exit(0);
                }
        }
}
```

3.3.2 Priority Queue

Priority Queue is a data structure in which the items are served (deleted) based on their priority levels. The insertion and deletion operations of priority queue are based on the priority of the elements. The element with highest priority is processed first and the element with second highest priority is processed next and so on. The use of this data structure is in job scheduling algorithms in the design of operating system.

These are two different types of priority queues viz.

- Ascending priority queue.
- Descending priority queue.

In both the methods the items are inserted in any order. But in ascending priority queue the smallest element is deleted first, while in the descending priority queue, the largest element is deleted first.

However, the term *smallest* in the above statement not necessarily mean the value of the element, but it may be any quantity associated with the element. For example, we can think of stack as a priority queue, in which the elements are deleted based on the time of insertion as a priority level. That is, the item inserted most recently (i.e. least time) will be deleted first. Similarly, ordinary queue can be thought of as a priority queue, where deletion is based on maximum time spent by the element in a queue. That is, the item which spent maximum time (inserted at the beginning of the process) will be deleted first.

One should note that, the meaning of deletion here (in the study of queue data structures, as well) indicates providing the service for which the item/element is waiting in a queue.

Priority queue can be implemented using arrays. In further discussion with priority queue, we assume a queue of integers where, the value of item itself indicates its priority. That is, smallest item has higher priority.

As discussed earlier, in priority queue, the insertion of elements can be in any order, but the deletion is based on priority. That is, in ascending priority queue, the smallest item must be searched for, and then it should be deleted. Hence, element present at any in between position of the array must be deleted and rest of elements must be re-adjusted. Thus, the deletion process becomes difficult.

So to avoid this problem and for the sake of simplicity, we assume that the elements are inserted in an ascending order, so that the deletion requires only deleting the element at front end. Thus, in the programmatic implementation of priority queue, during insertion, we should see that items are inserted in a proper position to maintain ascending order.

Program for Implementation of Priority Queue

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void insert(int PQ[], int *f, int *r, int item)
{
        int j;

        if (*r==MAX-1)
        {
                printf("\nQueue overflow");
                return;
        }

        j=*r;
        while( j>=0 && item<PQ[ j ])
        {
                PQ[ j+1]=PQ[ j ];
                j - -;
        }

        PQ[ j+1]=item;
        (*r)++;

        if(*f== -1)
                (*f)++;
}
```

```
void del(int PQ[ ], int *f, int *r)
{
        if(*f==-1 || *f>*r)
        {
                printf("\nQueue underflow");
                return;
        }

        printf("\nDeleted item is %d", PQ[(*f)++]);
}

void disp(int PQ[], int *f, int *r)
{
        int i;

        if(*f== -1 || *f>*r)
        {
                printf("\nQueue underflow");
                return;
        }

        printf("\nContents of priority queue:");
        for(i=*f;i<=*r;i++)
                printf("%d\t", PQ[i]);
}

void main()
{
        int PQ[MAX], f=-1,r=-1, item, opt;

        for(;;)
        {
                printf("\n****Priority Queue****\n");
                printf("1.Insert\n2.Delete\n3.Display\n4.Exit\n");
                printf("\nEnter your option:");
                scanf("%d",&opt);

                switch(opt)
                {
                        case 1: printf("\nEnter the item to be inserted:");
                                scanf("%d",&item);
                                insert(PQ,&f,&r,item);
                                break;
                        case 2: del(PQ, &f, &r);
                                break;
```

```
                    case 3: disp(PQ, &f, &r);
                            break;
                    case 4:
                    default:
                            exit(0);
                }
            }
        }
```

3.3.3 Double-Ended Queue

A double ended queue or deque (pronounced as deck) is a set of items from which items may be deleted from either end and items may be inserted at either end. A deque can have some sub-types:

- **Input restricted deque**: deletion can be made at both the ends, but insertion can be made at only one end.
- **Output restricted deque**: deletion is at only one end and insertion can be made at both the ends.

Considering the definition of deque, the stack and queue data structures can be thought of as special cases of deque. Deque can be implemented either using arrays or linked lists. The implementation of deque using linked list will be discussed along with linked list in Module 4.