# UNIT - 2

# STACKS AND QUEUES

# UNIT - 2

**Stacks:** Introduction to Stacks, Array Representation of Stacks, Operations on a Stack, Applications of Stacks: Implementing Parentheses Checker, Evaluation of Arithmetic Expressions, Recursion. **Queues:** Introduction to Queues, Array Representation of Queues, Types of Queues, Circular Queues, Basics of double ended Queues and Priority Queues, Applications of Queues..

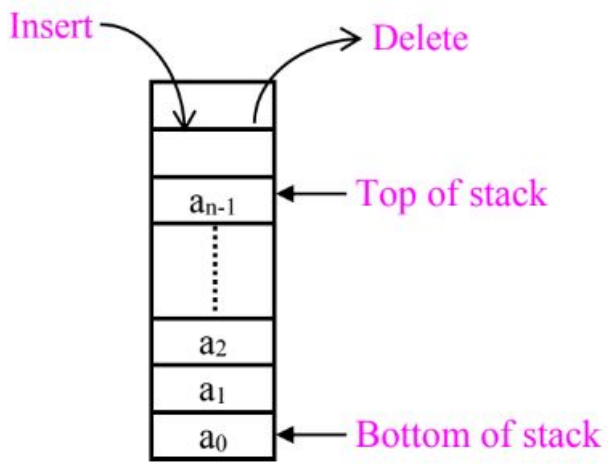
## Introduction to Stacks

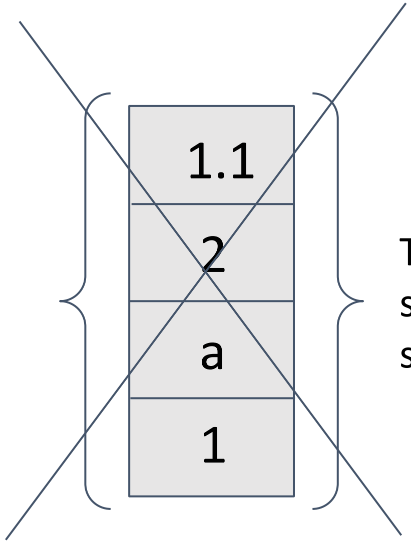
- A stack is an important data structure, where elements are inserted from one end and elements are deleted from the same end.
- Using this approach, the Last element Inserted is the First element to be deleted Out and hence stack is also called Last In First Out (LIFO) data structure

**Note :** Stack is a collection of similar data types

## Array Representation of Stacks

- In the computer’s memory, stacks can be represented as a linear array.
- Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from.
- There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold.





To store data in stack should use similar data type

The elements are inserted into the stack in the order a0, a1, a2,......an-1. That is, we insert a0 first, a1 next and so on. The item an-1 is inserted at the end. Since, it is on top of the stack, it is the first item to be deleted.

- If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full.
- The stack in Figure shows that TOP = 4, so insertions and deletions will be done at this position. In this stack, five more elements can still be stored.

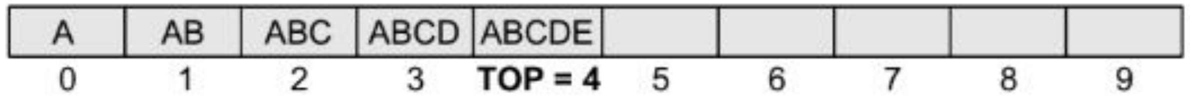


**Figure 7.4** Stack

**Overflow:** Check whether the stack is full or not.

**Underflow:** Check whether the stack is empty or not.

## Operations on a Stack

**A stack supports 3 basic operations:**

1. Push operation (Insertion operation): adds an element to the top of the stack
2. Pop operation (Deletion operation): removes the element from the top of the stack.
3. Peek operation: returns the value of the topmost element of the stack.

## Push Operation

- The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack.
- However, before inserting the value, we must first check if TOP=MAX–1, because if that is the case, then the stack is full and no more insertions can be done.
- If an attempt is made to insert a value in a stack that is already full, an OVERFLOW message is printed.

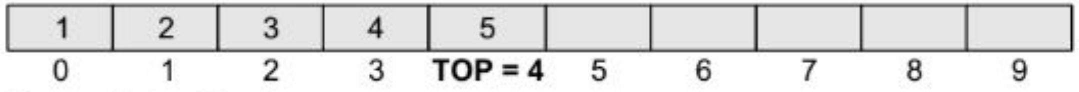


**Figure 7.5** Stack

- To insert an element with value 6, first check if TOP=MAX–1.
- If the condition is false, then we increment the value of TOP and store the new element at the position given by stack[TOP].

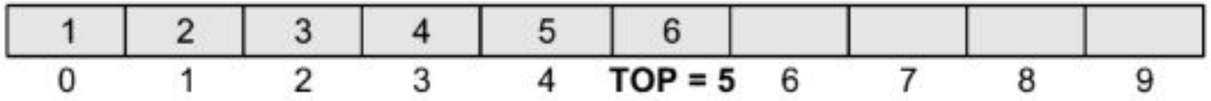
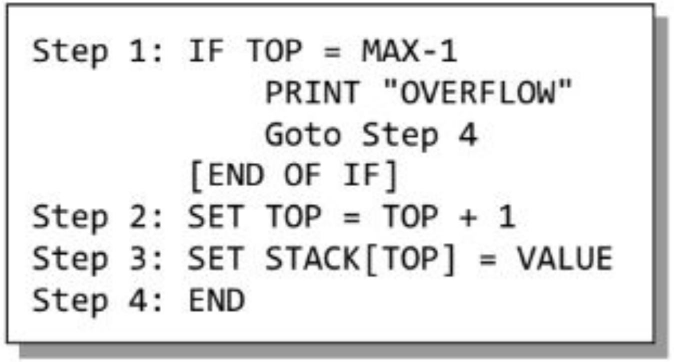


**Figure 7.6** Stack after insertion

## Algorithm to Insert an element in a stack



```
Step 1: IF TOP = MAX-1
            PRINT "OVERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET TOP = TOP + 1
Step 3: SET STACK[TOP] = VALUE
Step 4: END
```

**Step 1:** we first check for the OVERFLOW condition.

**Step 2:** TOP is incremented so that it points to the next location in the array.

**Step 3:** the value is stored in the stack at the location pointed by

## Pop Operation

- The pop operation is used to delete the topmost element from the stack.
- However, before deleting the value, we must first check if TOP=NULL because if that is the case, then it means the stack is empty and no more deletions can be done.
- If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.

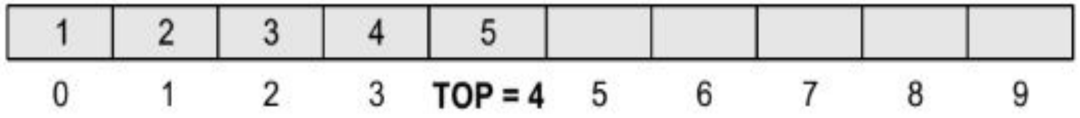


**Figure 7.8** Stack

- To delete the topmost element with a value 5, first check if TOP=NULL.
- If the condition is false, then we decrement the value pointed by TOP.

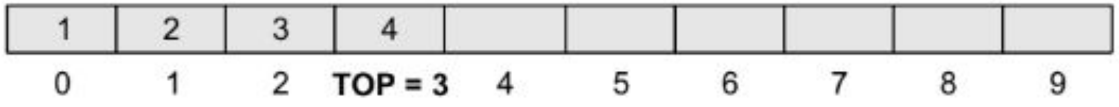
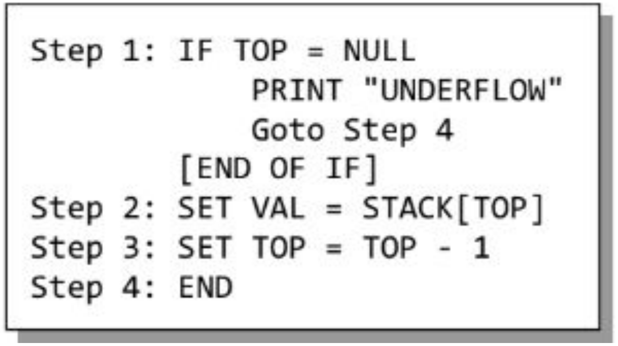


**Figure 7.9** Stack after deletion

## Algorithm to delete an element from stack



```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 4
        [END OF IF]
Step 2: SET VAL = STACK[TOP]
Step 3: SET TOP = TOP - 1
Step 4: END
```

**Step 1:** we first check for the UNDERFLOW condition.

**Step 2:** the value of the location in the stack pointed by TOP is stored in VAL.

**Step 3:** TOP is decremented.

## Peek Operation

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

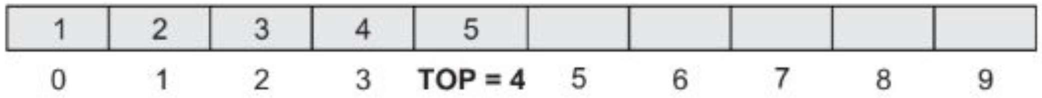
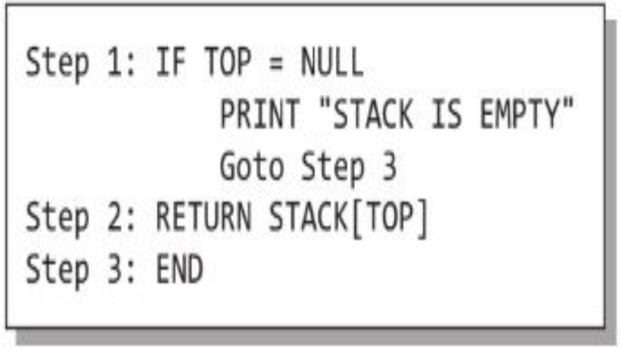


**Figure 7.12** Stack

- Peek operation first checks if the stack is empty,

  i.e., if TOP = NULL, then an appropriate message is printed, else the value is returned.

## Algorithm for Peek Operation



```
Step 1: IF TOP = NULL
            PRINT "STACK IS EMPTY"
            Goto Step 3
Step 2: RETURN STACK[TOP]
Step 3: END
```

Here, the Peek operation will return 5, as it is the value of the topmost element of the stack.

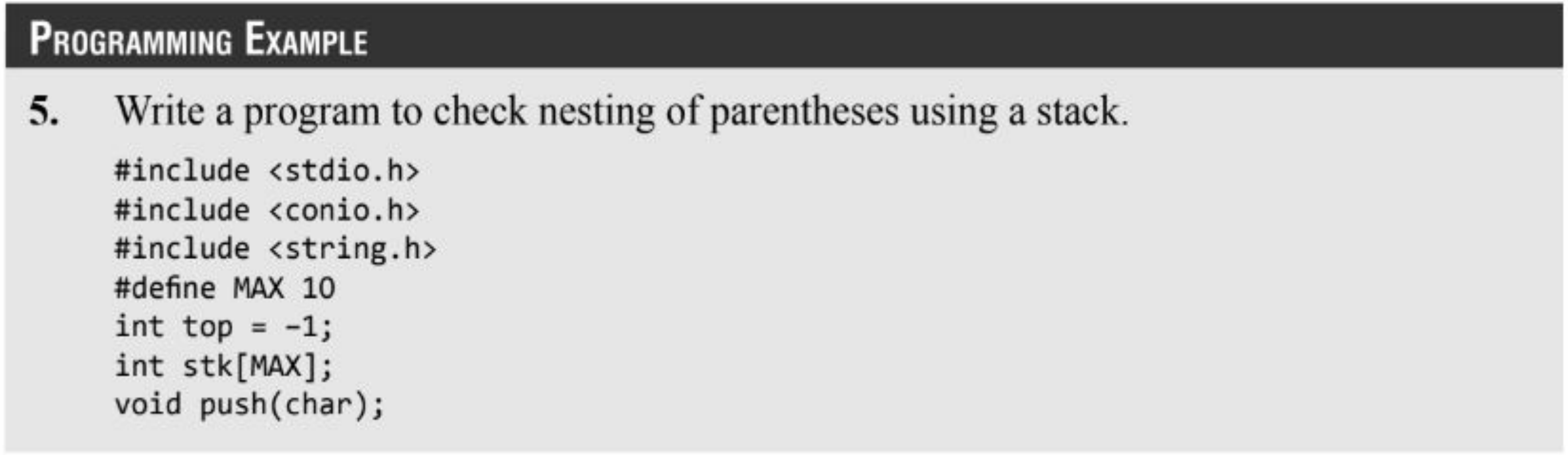
## Applications of Stack

- Reversing a list
- Parentheses checker
- Conversion of an infix expression into a postfix expression
- Evaluation of a postfix expression
- Conversion of an infix expression into a prefix expression
- Evaluation of a prefix expression
- Recursion
- Tower of Hanoi

## Implementing Parentheses Checker

- Stacks can be used to check the validity of parentheses in any algebraic expression.
- For example,an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.
- For example, the expression (A+B} is invalid but an expression {A + (B – C)} is valid. Look at the program below which traverses an algebraic expression to check for its validity.

- case 1: opening brackets (,[,{ - Push character
- case 2 : closing brackets ),],} - pop character and compare with character if character gets its matched one then its balanced if not its Unbalanced.



**PROGRAMMING EXAMPLE**

**5.** Write a program to check nesting of parentheses using a stack.

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 10
int top = -1;
int stk[MAX];
void push(char);
```

```
char pop();
void main()
{
          char exp[MAX],temp;
          int i, flag=1;
          clrscr();
          printf("Enter an expression : ");
          gets(exp);
          for(i=0;i<strlen(exp);i++)
          {
                    if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
                              push(exp[i]);
                    if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
                              if(top == -1)
                                        flag=0;
                              else
                              {
                                        temp=pop();
                                        if(exp[i]==')' && (temp=='{' || temp=='['))
                                                  flag=0;
                                        if(exp[i]=='}' && (temp=='(' || temp=='['))
                                                  flag=0;
                                        if(exp[i]==']' && (temp=='(' || temp=='{'))
                                                  flag=0;
                              }
          }
          if(top>=0)
                    flag=0;
          if(flag==1)
                    printf("\n Valid expression");
```

```
void push(char c)
{
        if(top == (MAX-1))
                printf("Stack Overflow\n");
        else
        {
                top=top+1;
                stk[top] = c;
        }
}
char pop()
{
        if(top == -1)
                printf("\n Stack Underflow");
        else
                return(stk[top--]);
}
```

**Output**

```
Enter an expression : (A + (B - C))
Valid Expression
```

## Evaluation of Arithmetic Expressions

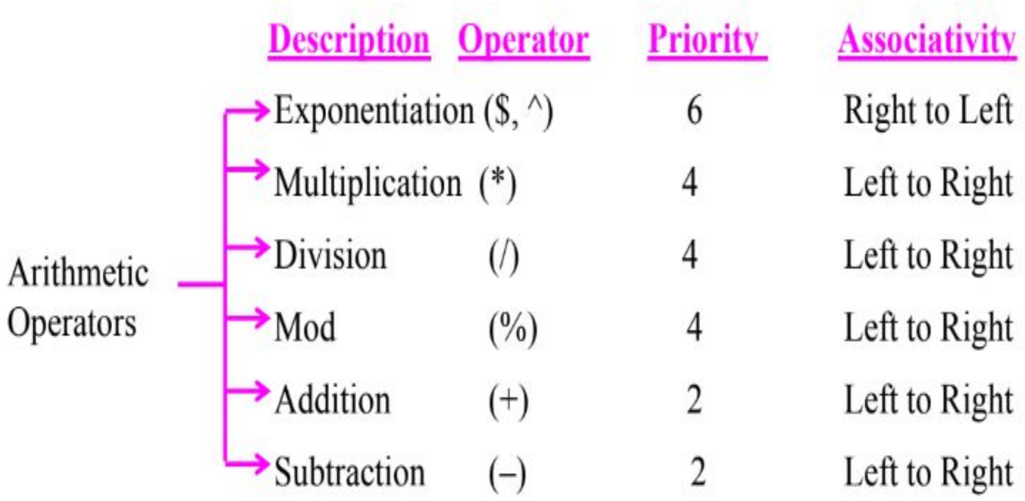
- Computers find it difficult to parse as the computer needs a lot of information to evaluate the expression.
- The sequence of operators and operands that reduces to a single value after evaluation is called an expression.
- Information is needed about operator precedence and associativity rules, and brackets which override these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

## Representation of expressions

1. **infix** - operator is placed in between the operands. Ex: A+B
2. **postfix (reverse polish notation)** - the operator is placed after the operands. Ex: AB+

   **The operator which occurs first in the expression is operated first on the operands.** Ex: AB+C*
3. **prefix (polish notation)** - the operator is placed before the operands. Ex: +AB

# Precedence of operators



| | Description | Operator | Priority | Associativity |
|---|---|---|---|---|
| Arithmetic Operators | Exponentiation | ($, ^) | 6 | Right to Left |
| | Multiplication | (*) | 4 | Left to Right |
| | Division | (/) | 4 | Left to Right |
| | Mod | (%) | 4 | Left to Right |
| | Addition | (+) | 2 | Left to Right |
| | Subtraction | (–) | 2 | Left to Right |

- Normally we associate values to determine the order in which the operators have to be evaluated.
- Highest precedence operators will have the **highest value** and lowest precedence operators will have the **least value**.

## “What if different operators have the same precedence?”

During evaluation, if two or more operators have the same precedence, then precedence rules are not applicable. Instead, we go for **associativity** of the operators.

## “What is associativity?”

The order in which the operators with same precedence are evaluated in an expression is called associativity of the operator.

i.e **Left to Right** and **Right to Left**

## Conversion of an Infix Expression into a Postfix Expression

- The precedence of these operators can be given as follows:

  **Higher priority : *, /, %     Lower priority : +, –**

- The order of evaluation of these operators can be changed by making use of parentheses.

  **Ex:** if we have an expression A + B * C, then first B * C will be done and the result will be added to A. But the same expression if written as, (A + B) * C, will evaluate A + B first and then the result will be multiplied with C.

## Rules to Convert infix to postfix expression

- Print Operands as they arrive.
- If stack is empty or contains a left Parenthesis on top, **push** the incoming operator onto the stack
- If incoming symbol is ' ( ', **push** it onto stack
- If incoming symbol is ')', **pop** the stack & **Print** the operators until left parenthesis in found
- If incoming symbol has higher precedence than the top of the stack, **push** it on the stack.

- If incoming symbol has lower precedence than the top of the stack, **pop & print** the top. Then test the incoming operator against the new top of the stack
- if incoming operator has equal precedence with the top of the stack, use **associativity rule.**
- At the end of the expression, **pop and print** all operator of stack.
- associativity L to R then **pop & print** the top of the stack & then **push** the incoming operator.
- R to L then **push** the incoming operator

**Example 7.1** Convert the following infix expressions into postfix expressions

***Solution***

```
(a) (A–B) * (C+D)
    [AB–] * [CD+]
    AB–CD+*
(b) (A + B) / (C + D) - (D * E)
    [AB+] / [CD+] - [DE*]
    [AB+CD+/] - [DE*]
    AB+CD+/DE*–
```

**Example 7.2** Convert the following infix expressions into prefix expressions.

***Solution***

```
(a) (A + B) * C
    (+AB)*C
    *+ABC
(b) (A–B) * (C+D)
    [–AB] * [+CD]
    *–AB+CD
(c) (A + B) / ( C + D) – ( D * E)
    [+AB] / [+CD] - [*DE]
    [/+AB+CD] - [*DE]
    –/+AB+CD*DE
```

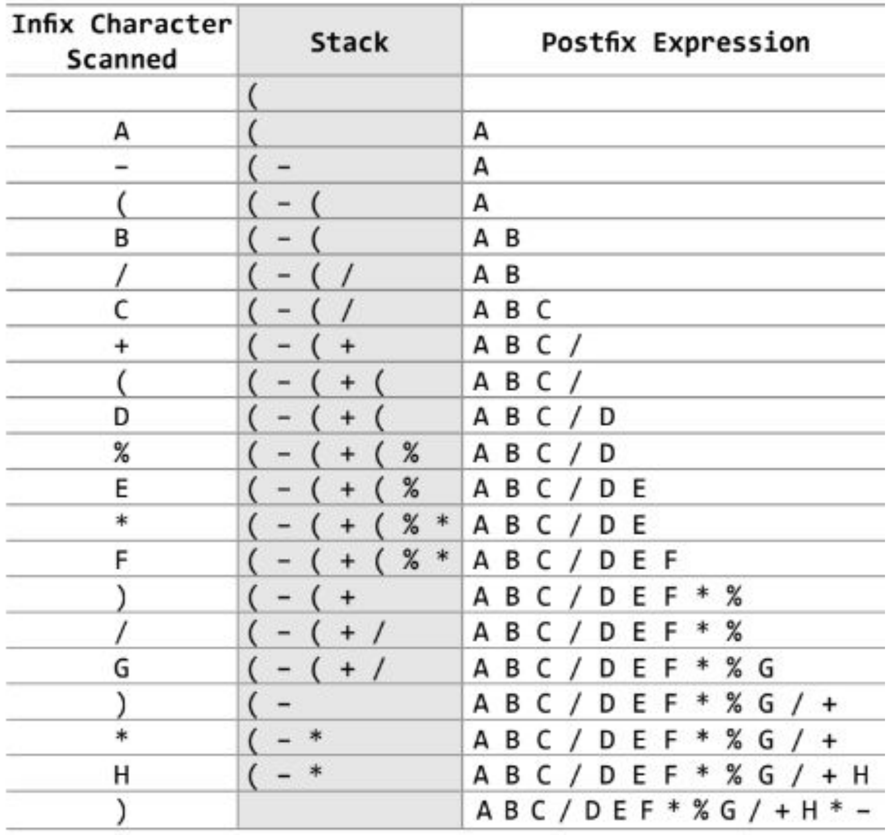
```
Step 1: Add ")" to the end of the infix expression
Step 2: Push "(" on to the stack
Step 3: Repeat until each character in the infix notation is scanned
        IF a "(" is encountered, push it on the stack
        IF an operand (whether a digit or a character) is encountered, add it to the
        postfix expression.
        IF a ")" is encountered, then
          a. Repeatedly pop from stack and add it to the postfix expression until a
             "(" is encountered.
          b. Discard the "(". That is, remove the "(" from stack and do not
             add it to the postfix expression
        IF an operator O is encountered, then
          a. Repeatedly pop from stack and add each operator (popped from the stack) to the
             postfix expression which has the same precedence or a higher precedence than O
          b. Push the operator O to the stack
        [END OF IF]
Step 4: Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
Step 5: EXIT
```

**Figure 7.22** Algorithm to convert an infix notation to postfix notation

Ex:Convert the following infix expression into postfix expression using the algorithm given bellow

(a) A – (B / C + (D % E * F) / G)* H

(b) A – (B / C + (D % E * F) / G)* H)



| Infix Character Scanned | Stack | Postfix Expression |
|---|---|---|
| | ( | |
| A | ( | A |
| – | ( - | A |
| ( | ( - ( | A |
| B | ( - ( | A B |
| / | ( - ( / | A B |
| C | ( - ( / | A B C |
| + | ( - ( + | A B C / |
| ( | ( - ( + ( | A B C / |
| D | ( - ( + ( | A B C / D |
| % | ( - ( + ( % | A B C / D |
| E | ( - ( + ( % | A B C / D E |
| * | ( - ( + ( % * | A B C / D E |
| F | ( - ( + ( % * | A B C / D E F |
| ) | ( - ( + | A B C / D E F * % |
| / | ( - ( + / | A B C / D E F * % |
| G | ( - ( + / | A B C / D E F * % G |
| ) | ( - | A B C / D E F * % G / + |
| * | ( - * | A B C / D E F * % G / + |
| H | ( - * | A B C / D E F * % G / + H |
| ) | | A B C / D E F * % G / + H * - |

## Evaluation of a Postfix Expression

- Given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression.
- Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool.

## “What is the need for evaluating postfix/prefix expressions?”

The evaluation of infix expression is not recommended because of the following reasons:

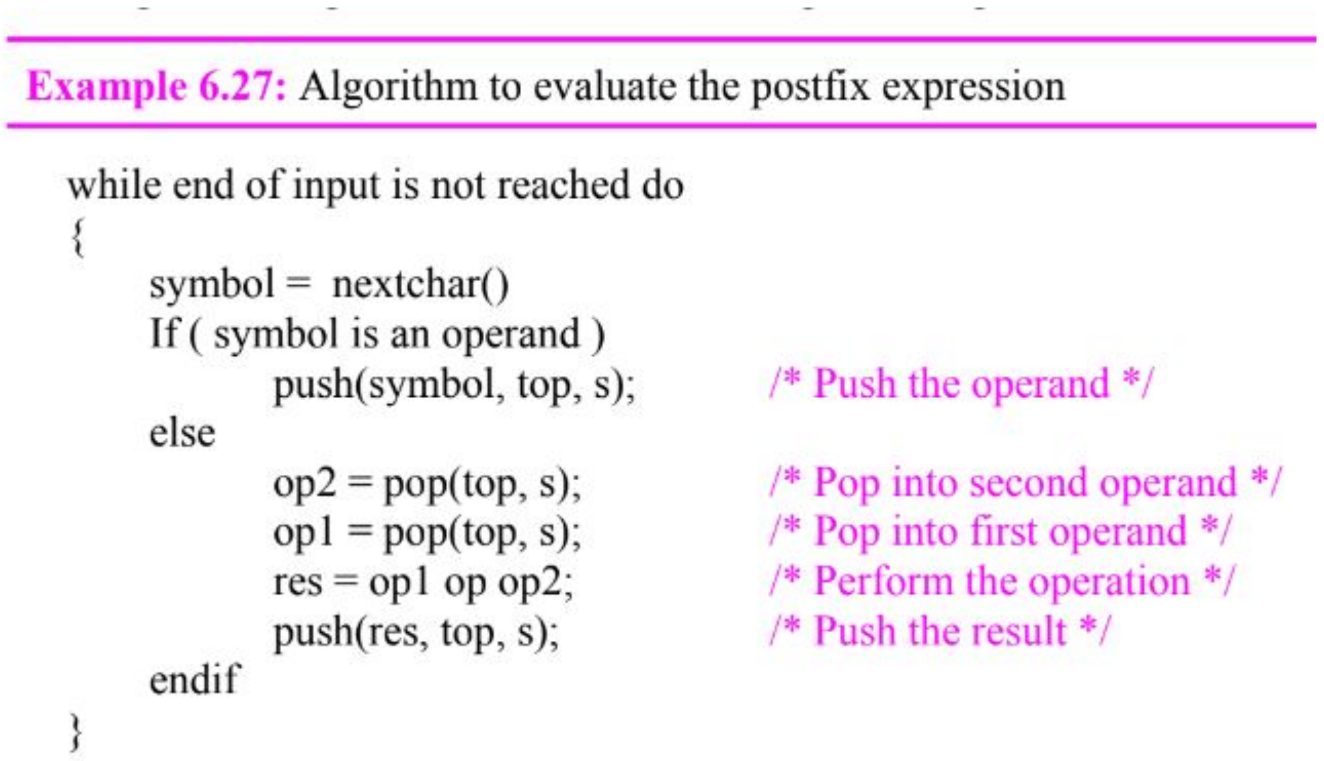
1. Evaluation of infix expression requires the knowledge of precedence of operators and the associativity of operators.
2. The problem becomes complex, if there are parentheses in the expression because they change the order of precedence.
3. During evaluation, we may have to scan from left to right and right to left repeatedly thereby complexity of the program increases.

## “How to evaluate the postfix expression?”

- Scan the **symbol** from left to right
- If the scanned symbol is an **operand**, push it onto the stack
- If the scanned symbol is an **operator**, pop two elements from the stack.
- The first popped element is **operand2** and the second popped element is **operand1**. **op2 = s[top--]; /* First popped element is operand2 */**

  **op1 = s[top--]; /* Second popped element is operand1 */**
- **res = op1 op op2 /* op is an operator such +, -, /, * etc. */**
- Push the result on to the stack.
- Repeat the above procedure till the end of input is encountered.

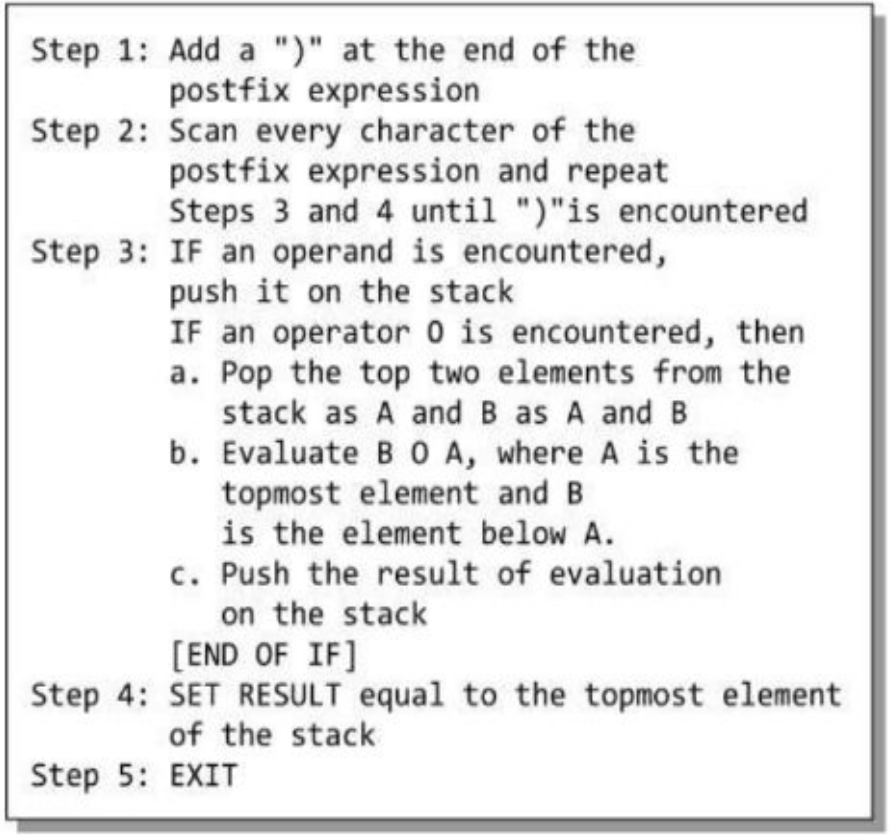
## The algorithm or pseudocode to evaluate the postfix expression is shown below:



**Example 6.27:** Algorithm to evaluate the postfix expression

```
while end of input is not reached do
{
    symbol =  nextchar()
    If ( symbol is an operand )
            push(symbol, top, s);          /* Push the operand */
    else
            op2 = pop(top, s);             /* Pop into second operand */
            op1 = pop(top, s);             /* Pop into first operand */
            res = op1 op op2;              /* Perform the operation */
            push(res, top, s);             /* Push the result */
    endif
}
```

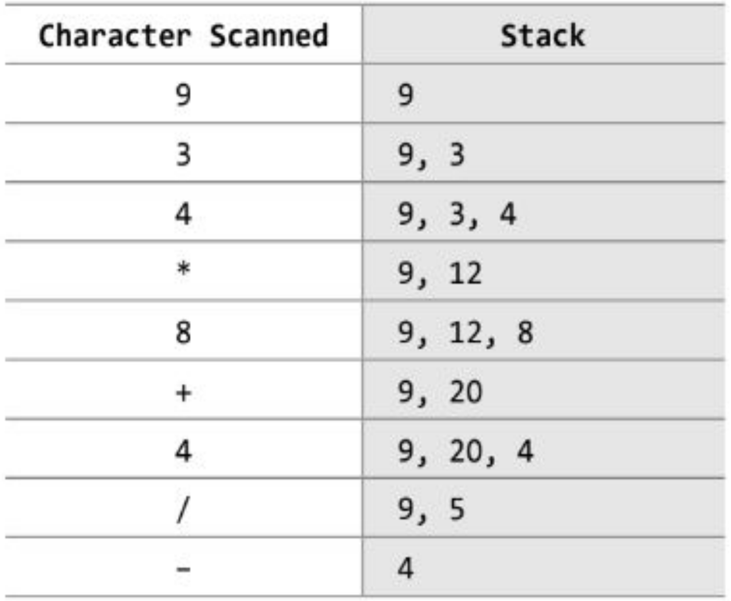
**Consider the infix expression given as 9 – ((3 * 4) + 8) / 4. Evaluation of Postfix expression.**



```
Step 1: Add a ")" at the end of the
        postfix expression
Step 2: Scan every character of the
        postfix expression and repeat
        Steps 3 and 4 until ")"is encountered
Step 3: IF an operand is encountered,
        push it on the stack
        IF an operator O is encountered, then
        a. Pop the top two elements from the
           stack as A and B as A and B
        b. Evaluate B O A, where A is the
           topmost element and B
           is the element below A.
        c. Push the result of evaluation
           on the stack
        [END OF IF]
Step 4: SET RESULT equal to the topmost element
        of the stack
Step 5: EXIT
```

**Figure 7.23** Algorithm to evaluate a postfix expression

**Table 7.1** Evaluation of a postfix expression



| Character Scanned | Stack |
|---|---|
| 9 | 9 |
| 3 | 9, 3 |
| 4 | 9, 3, 4 |
| * | 9, 12 |
| 8 | 9, 12, 8 |
| + | 9, 20 |
| 4 | 9, 20, 4 |
| / | 9, 5 |
| – | 4 |

## Rules to Convert infix to prefix expression

- Reverse the string in infix expression
- Print Operands as they arrive.
- If stack is empty or contains a left Parenthesis on top, **push** the incoming operator onto the stack
- If incoming symbol is ' ( ', **pop** & **print** the operators until left parenthesis in found
- If incoming symbol is ')', **push** onto stack
- If any operator comes after ‘)’, simply **push** onto stack

- If incoming symbol has **higher precedence** than the top of the stack, **push** it on the stack.
- If incoming symbol has **lower precedence** than the top of the stack, **pop & print** the top. Then test the incoming operator against the new top of the stack
- if incoming operator has **equal precedence** with the top of the stack, use **associativity rule.** associativity L to R then **push onto** stack
- R to L then **push** the incoming operator
- At the end of the expression, **pop and print** all operator of stack.

# Conversion of Infix Expression into a Prefix Expression

```
Step 1: Scan each character in the infix
        expression. For this, repeat Steps
        2-8 until the end of infix expression
Step 2: Push the operator into the operator stack,
        operand into the operand stack, and
        ignore all the left parentheses until
        a right parenthesis is encountered
Step 3: Pop operand 2 from operand stack
Step 4: Pop operand 1 from operand stack
Step 5: Pop operator from operator stack
Step 6: Concatenate operator and operand 1
Step 7: Concatenate result with operand 2
Step 8: Push result into the operand stack
Step 9: END
```

**Figure 7.24** Algorithm to convert an infix expression into prefix expression

```
Step 1: Reverse the infix string. Note that
        while reversing the string you must
        interchange left and right parentheses.
Step 2: Obtain the postfix expression of the
        infix expression obtained in Step 1.
Step 3: Reverse the postfix expression to get
        the prefix expression
```

**Figure 7.25** Algorithm to convert an infix expression into prefix expression

The corresponding prefix expression is obtained in the operand stack.

For example, given an infix expression (A – B / C) * (A / K – L)

*Step 1: Reverse the infix string. Note that while reversing the string you must interchange left and right parentheses.*

(L – K / A) * (C / B – A)

*Step 2: Obtain the corresponding postfix expression of the infix expression obtained as a result of Step 1.*

The expression is: (L – K / A) * (C / B – A)

Therefore, [L – (K A /)] * [(C B /) – A]

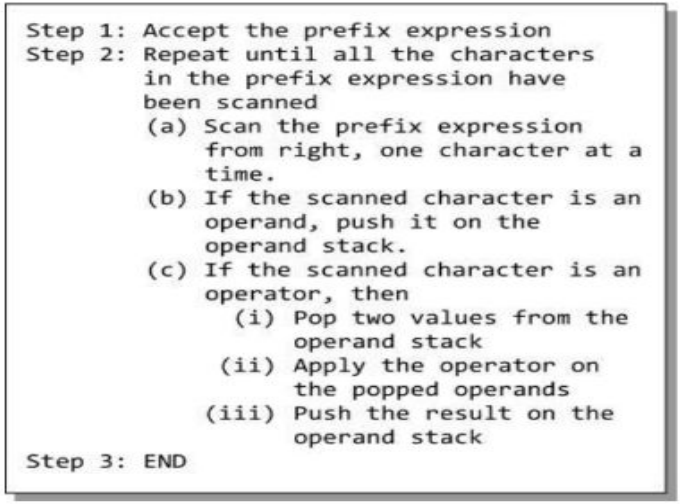
= [LKA/–] * [CB/A–]

= L K A / – C B / A – *

*Step 3: Reverse the postfix expression to get the prefix expression*

Therefore, the prefix expression is * – A / B C – /A K L

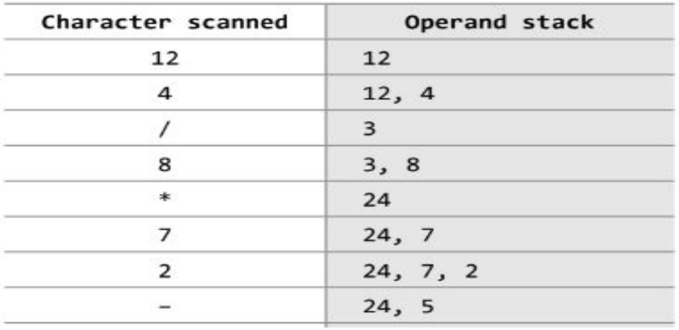
# Evaluation of a Prefix Expression



```
Step 1: Accept the prefix expression
Step 2: Repeat until all the characters
        in the prefix expression have
        been scanned
        (a) Scan the prefix expression
            from right, one character at a
            time.
        (b) If the scanned character is an
            operand, push it on the
            operand stack.
        (c) If the scanned character is an
            operator, then
               (i) Pop two values from the
                   operand stack
              (ii) Apply the operator on
                   the popped operands
             (iii) Push the result on the
                   operand stack
Step 3: END
```

**Figure 7.26** Algorithm for evaluation of a prefix expression

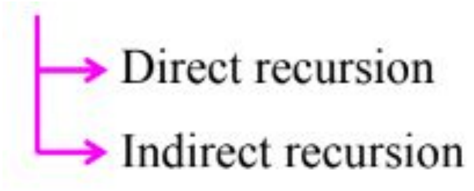
**+ – 9 2 7 * 8 / 4 12**



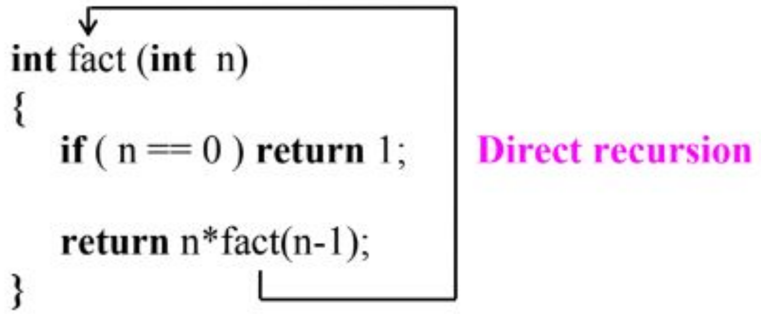
| Character scanned | Operand stack |
|---|---|
| 12 | 12 |
| 4 | 12, 4 |
| / | 3 |
| 8 | 3, 8 |
| * | 24 |
| 7 | 24, 7 |
| 2 | 24, 7, 2 |
| – | 24, 5 |

## Recursion

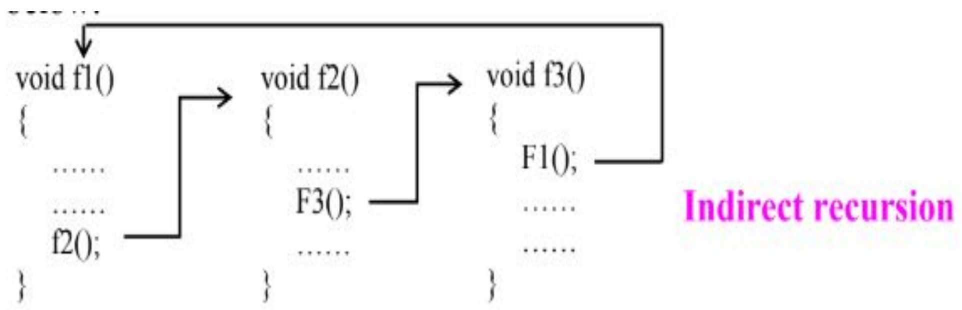
- A recursive function is defined as a function that calls itself to solve a smaller version of its task until a final call is made which does not require a call to itself.
- Since a recursive function repeatedly calls itself, it makes use of the system stack to temporarily store the return address and local variables of the calling function.



- **Base case**, in which the problem is simple enough to be solved directly without making any further calls to the same function.
- **Recursive case,** in which first the problem at hand is divided into simpler sub-parts. Second the function calls itself but with sub-parts of the problem obtained in the first step.  Third, the result is obtained by combining the solutions of simpler sub-parts.



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

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

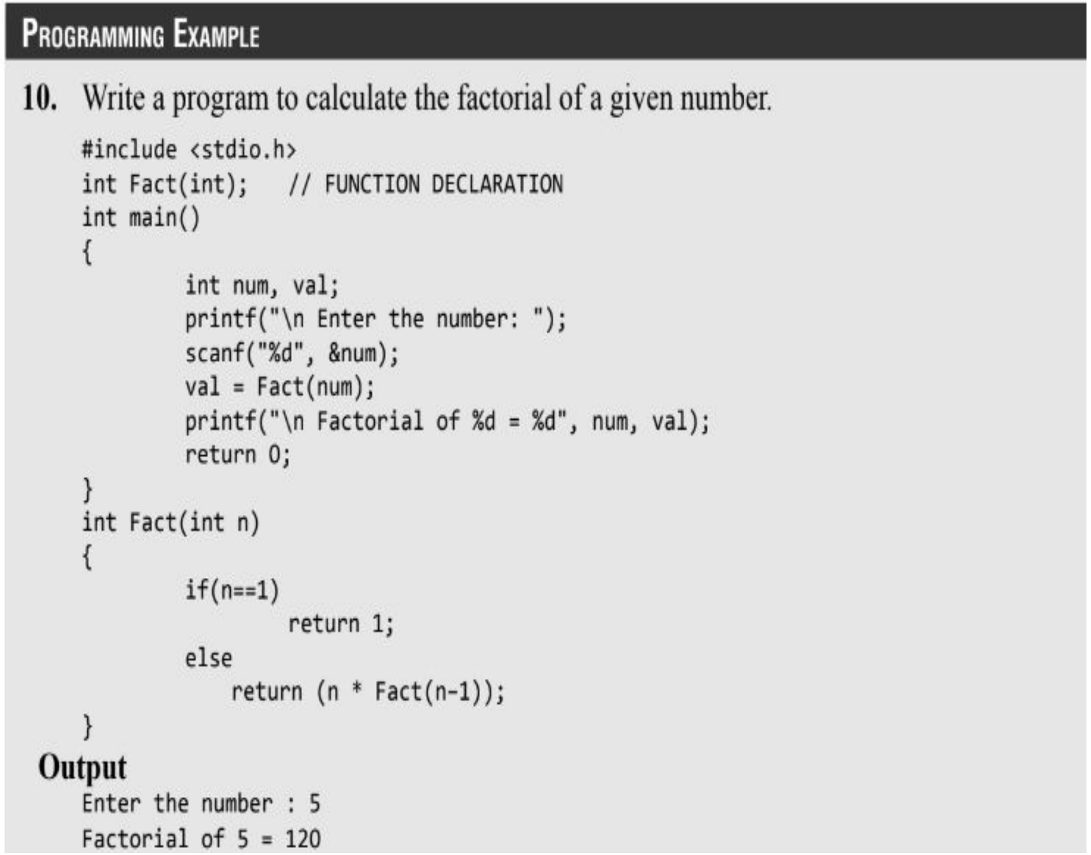


```
void f1()        void f2()        void f3()
{                {                {
  ......           ......           F1();
  ......           F3();            ......
  f2();            ......           ......
}                }                }
```

## write a recursive function to calculate the factorial of a number.

- **Base case** is when n = 1, because if n = 1, the result will be 1 as **1! = 1**.
- **Recursive case** of the factorial function will call itself but with a smaller value of n, this case can be given as

**factorial(n) = n × factorial (n–1)**



PROGRAMMING EXAMPLE

10. Write a program to calculate the factorial of a given number.

```
#include <stdio.h>
int Fact(int);    // FUNCTION DECLARATION
int main()
{
        int num, val;
        printf("\n Enter the number: ");
        scanf("%d", &num);
        val = Fact(num);
        printf("\n Factorial of %d = %d", num, val);
        return 0;
}
int Fact(int n)
{
        if(n==1)
                return 1;
        else
            return (n * Fact(n-1));
}
```

**Output**

```
Enter the number : 5
Factorial of 5 = 120
```

## Steps in executing a recursive function

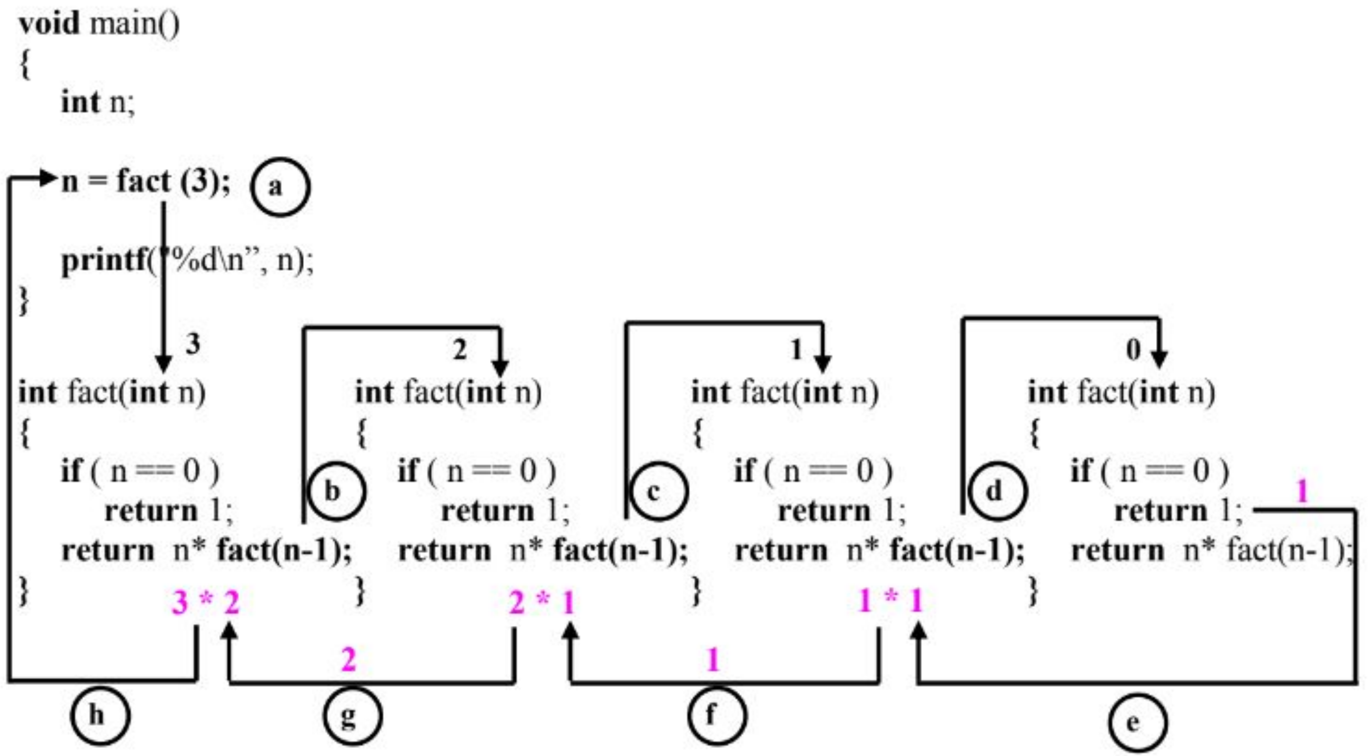
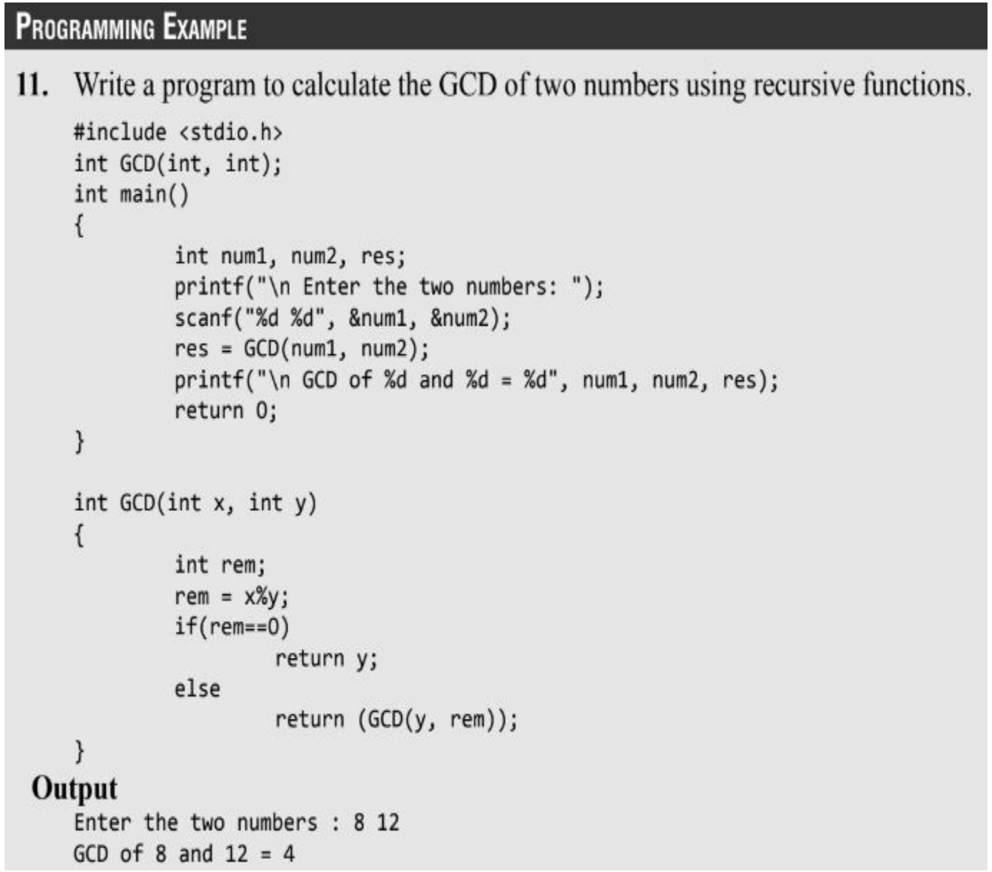

```
void main()
{
    int n;

    n = fact (3);

    printf("%d\n", n);
}
```
```
int fact(int n)
{
    if ( n == 0 )
        return 1;
    return  n* fact(n-1);
}
```

Fig 6.4 Steps in executing a recursive function

# Greatest Common Divisor (GCD)

- The GCD of two given numbers is the largest integer that divides both of them.
- GCD of two numbers is defined only for positive integers but, not defined for negative integers and floating point numbers.



**PROGRAMMING EXAMPLE**

11. Write a program to calculate the GCD of two numbers using recursive functions.

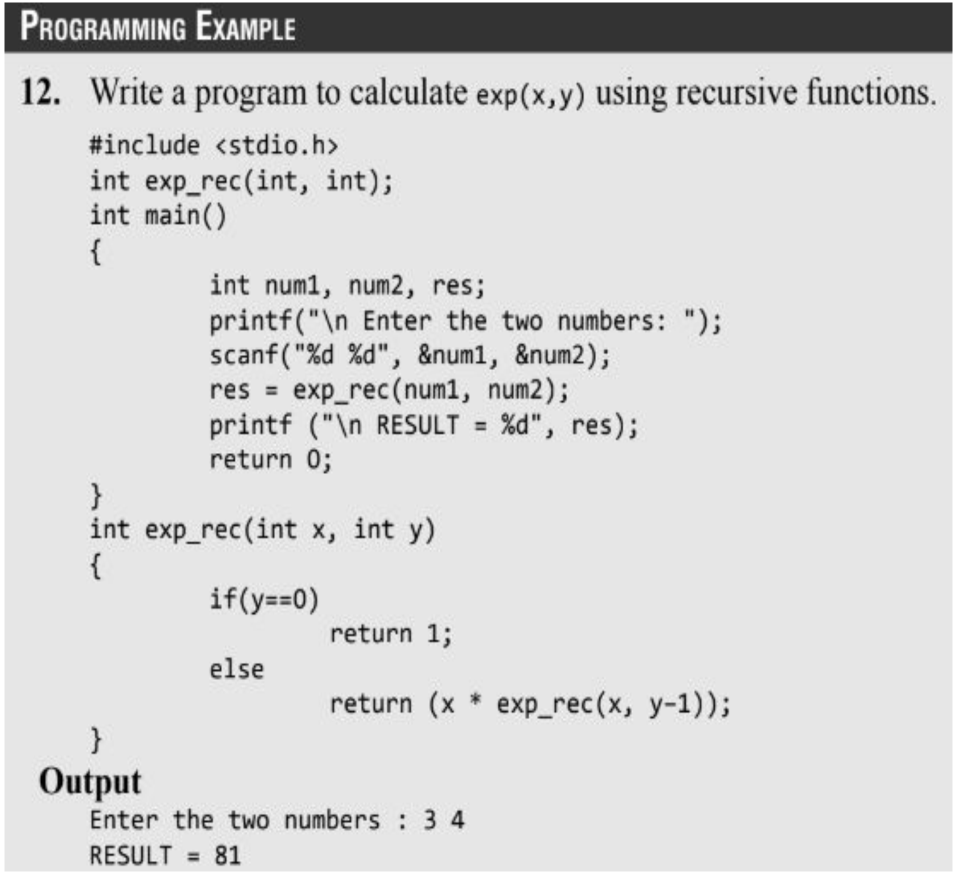
```
#include <stdio.h>
int GCD(int, int);
int main()
{
        int num1, num2, res;
        printf("\n Enter the two numbers: ");
        scanf("%d %d", &num1, &num2);
        res = GCD(num1, num2);
        printf("\n GCD of %d and %d = %d", num1, num2, res);
        return 0;
}

int GCD(int x, int y)
{
        int rem;
        rem = x%y;
        if(rem==0)
                return y;
        else
                return (GCD(y, rem));
}
```

**Output**

```
Enter the two numbers : 8 12
GCD of 8 and 12 = 4
```

# Finding Exponents

We can also find exponent of a number using recursion. To find xy, the base case would be when y=0, as we know that any number raised to the power 0 is 1.



**Programming Example**

**12.** Write a program to calculate `exp(x,y)` using recursive functions.

```
#include <stdio.h>
int exp_rec(int, int);
int main()
{
        int num1, num2, res;
        printf("\n Enter the two numbers: ");
        scanf("%d %d", &num1, &num2);
        res = exp_rec(num1, num2);
        printf ("\n RESULT = %d", res);
        return 0;
}
int exp_rec(int x, int y)
{
        if(y==0)
                return 1;
        else
                return (x * exp_rec(x, y-1));
}
```

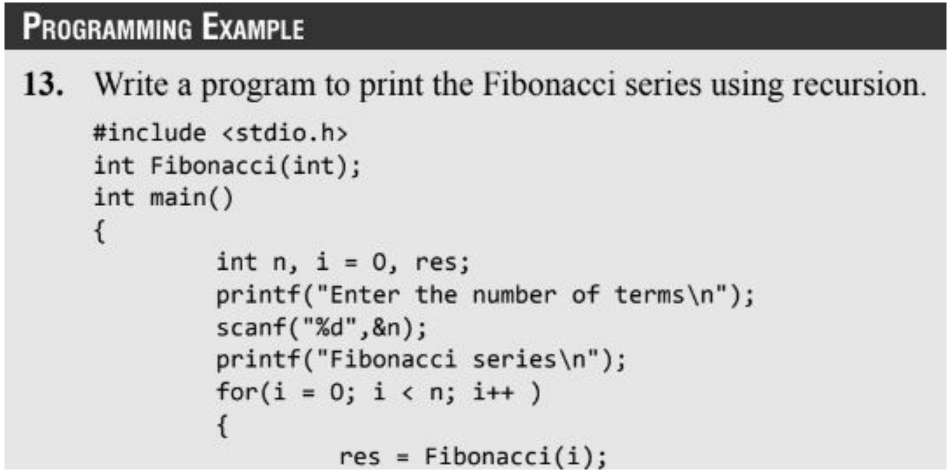
**Output**

```
Enter the two numbers : 3 4
RESULT = 81
```

# The Fibonacci series

The Fibonacci numbers are a series of numbers such that each number is the sum of the previous two numbers except the first and second number.

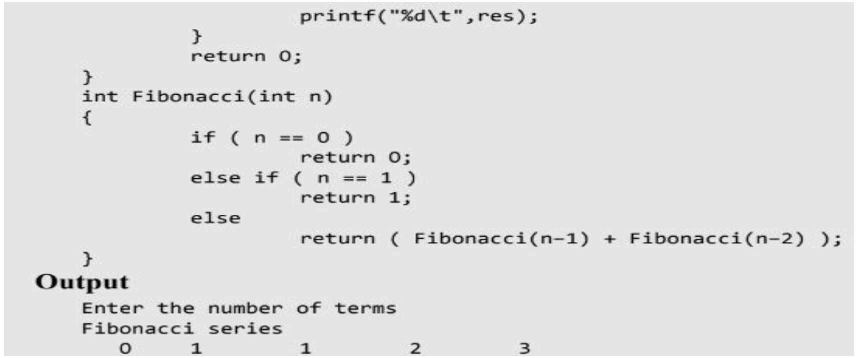
0 1 1 2 3 5 8 13 21 34 55 ......



**PROGRAMMING EXAMPLE**

13. Write a program to print the Fibonacci series using recursion.

```
#include <stdio.h>
int Fibonacci(int);
int main()
{
          int n, i = 0, res;
          printf("Enter the number of terms\n");
          scanf("%d",&n);
          printf("Fibonacci series\n");
          for(i = 0; i < n; i++ )
          {
                    res = Fibonacci(i);
```



```
                    printf("%d\t",res);
          }
          return 0;
}
int Fibonacci(int n)
{
          if ( n == 0 )
                    return 0;
          else if ( n == 1 )
                    return 1;
          else
                    return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
```

**Output**

```
Enter the number of terms
Fibonacci series
  0     1       1       2       3
```

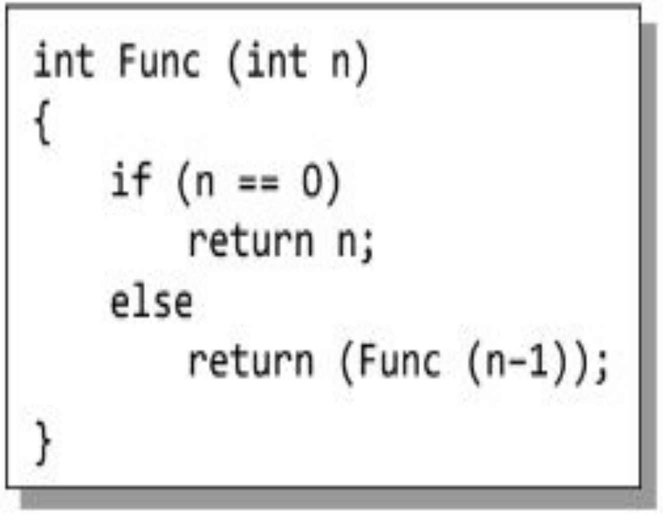
## Any recursive function can be characterized based on:

a) whether the function calls itself directly or indirectly (direct or indirect recursion)
b) whether any operation is pending at each recursive call (tail- recursive or not)
c) the structure of the calling pattern (linear or tree-recursive).

## Direct Recursion

A function is said to be directly recursive if it explicitly calls itself.

**Ex:** consider the code shown. Here, the function Func() calls itself for all positive values of n, so it is said to be a directly recursive function.



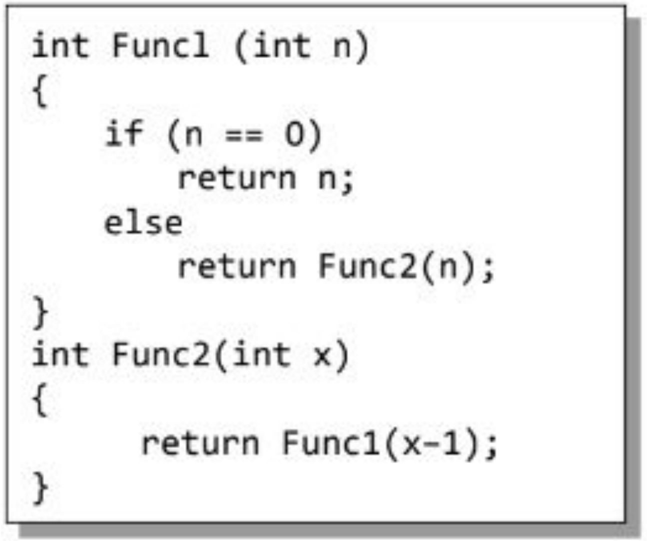
```
int Func (int n)
{
    if (n == 0)
        return n;
    else
        return (Func (n-1));
}
```

Figure 7.28 Direct recursion

## Indirect Recursion

A function is said to be **indirectly recursive** if it contains a call to another function which ultimately calls it.

**Ex:** Look at the functions given below. These two functions are i**ndirectly recursive** as they both call each other.

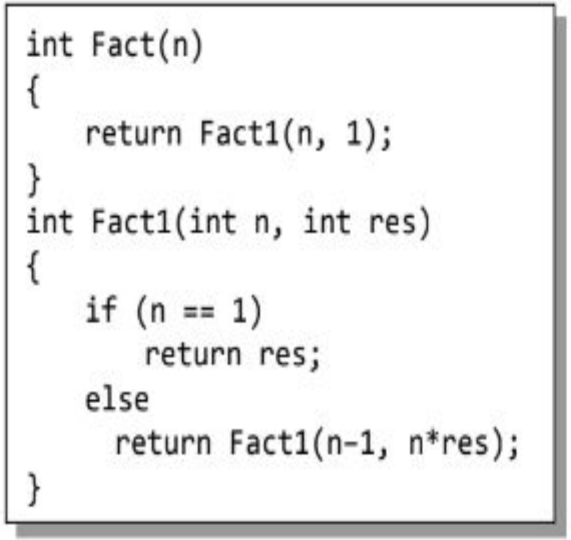


```
int Func1 (int n)
{
    if (n == 0)
        return n;
    else
        return Func2(n);
}
int Func2(int x)
{
      return Func1(x-1);
}
```

**Figure 7.29** Indirect recursion

## Tail Recursion

- A recursive function is said to be tail recursive if no operations are pending to be performed when recursive function returns to its caller.
- when the called function returns, the returned value is immediately returned from the calling function.
- Tail recursive functions are highly desirable because they are much more efficient to use as the amount of information that has to be stored on the system stack is independent of the number of recursive calls.

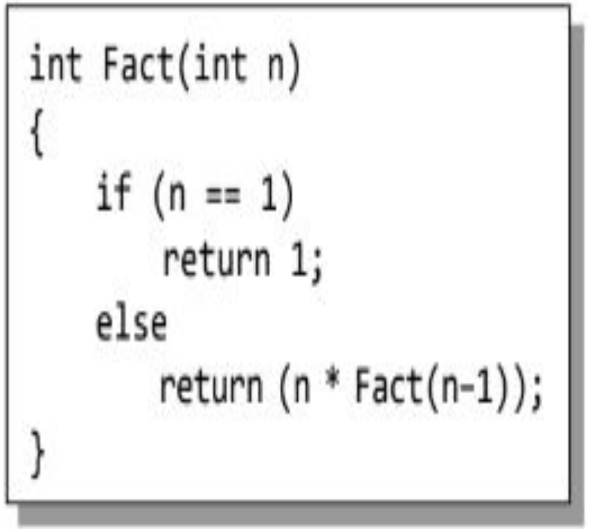


```
int Fact(n)
{
    return Fact1(n, 1);
}
int Fact1(int n, int res)
{
    if (n == 1)
        return res;
    else
      return Fact1(n-1, n*res);
}
```

**Figure 7.31** Tail recursion

## <u>Non-tail recursion</u>

factorial function that we have written is a non- tail-recursive function, because there is a **pending operation of multiplication** to be performed on return from **each recursive call.** Whenever there is a pending operation to be performed, the function becomes non-tail recursive.



```
int Fact(int n)
{
    if (n == 1)
        return 1;
    else
        return (n * Fact(n-1));
}
```

**Figure 7.30** Non-tail recursion

# Tower of Hanoi

```
                    #include <stdio.h>
                    int main()
                    {
    int n;
    printf("\n Enter the number of rings: ");
    scanf("%d", &n);
    move(n,'A', 'C', 'B');
    return 0;
}
void move(int n, char source, char dest, char spare)
{
    if (n==1)
        printf("\n Move from %c to %c",source,dest);
    else
    {
        move(n-1,source,spare,dest);
        move(1,source,dest,spare);
        move(n-1,spare,dest,source);
    }
}
```

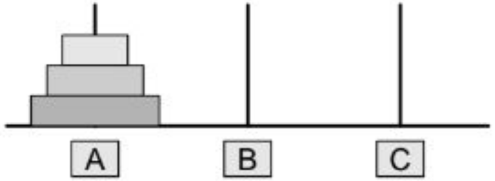


**Figure 7.33** Tower of Hanoi

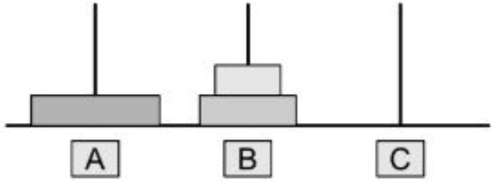


**Figure 7.34** Move rings from A to B

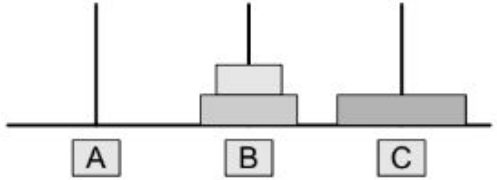


**Figure 7.35** Move ring from A to C

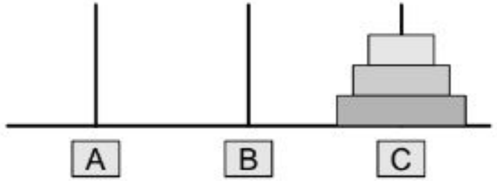


**Figure 7.36** Move ring from B to C

## Introduction to Queue

- A queue is a **FIFO** (First-In, First-Out) data structure in which the element that is inserted first is the first one to be taken out.
- The elements in a queue are added at one end called the **REAR** and removed from the other end called the **FRONT.**
- Queues can be implemented by using either **arrays** or **linked lists.**
- In this section, we will see how queues are implemented using each of these data structures.

## Array Representation Of Queues

- Queues can be easily represented using linear arrays.
- As stated earlier, every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

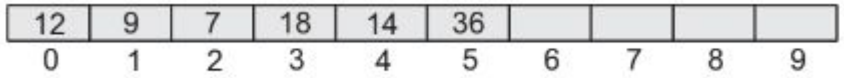


**Figure 8.1** Queue

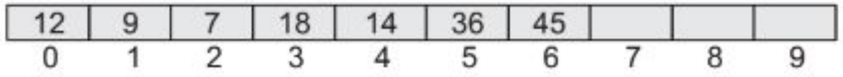


**Figure 8.2** Queue after insertion of a new element

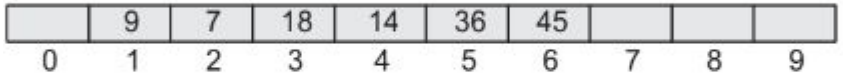


**Figure 8.3** Queue after deletion of an element

```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

**Figure 8.4** Algorithm to insert an element in a queue

```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

**Figure 8.5** Algorithm to delete an element from a queue

## Programming Example

1. Write a program to implement a linear queue.

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

```
#define MAX 10 // Changing this value will change length of array
int queue[MAX];
int front = -1, rear = -1;
void insert(void);
int delete_element(void);
int peek(void);
void display(void);
int main()
{
        int option, val;
        do
        {
                printf("\n\n ***** MAIN MENU *****");
                printf("\n 1. Insert an element");
                printf("\n 2. Delete an element");
                printf("\n 3. Peek");
                printf("\n 4. Display the queue");
                printf("\n 5. EXIT");
                printf("\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                case 1:
                     insert();
                     break;
                case 2:
                     val = delete_element();
                     if (val != -1)
                     printf("\n The number deleted is : %d", val);
                     break;
                case 3:
                     val = peek();
                     if (val != -1)
                     printf("\n The first value in queue is : %d", val);
                     break;
                case 4:
                     display();
```

```
            case 4:
                display();
                break;
            }
    }while(option != 5);
    getch();
    return 0;
}
void insert()
{
    int num;
    printf("\n Enter the number to be inserted in the queue : ");
    scanf("%d", &num);
    if(rear == MAX-1)
    printf("\n OVERFLOW");
    else if(front == -1 && rear == -1)
    front = rear = 0;
    else
    rear++;
    queue[rear] = num;
}
int delete_element()
{
    int val;
```

```
        if(front == -1 || front>rear)
        {
                printf("\n UNDERFLOW");
                return -1;
        }
        else
        {
                val = queue[front];
                front++;
                if(front > rear)
                front = rear = -1;
                return val;
        }
}
int peek()
{
        if(front==-1 || front>rear)
        {
                printf("\n QUEUE IS EMPTY")
                return -1;
        }
        else
        {
                return queue[front];
        }
}
void display()
{
        int i;
        printf("\n");
        if(front == -1 || front > rear)
        printf("\n QUEUE IS EMPTY");
        else
        {
                for(i = front;i <= rear;i++
                printf("\t %d", queue[i]);
        }
}
```

## Types of Queue

A queue data structure can be classified into the following types:

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

## 1. Circular Queue

In **linear queues**, we have discussed so far that insertions can be done only at one end called the **REAR** and deletions are always done from the other end called the **FRONT**.

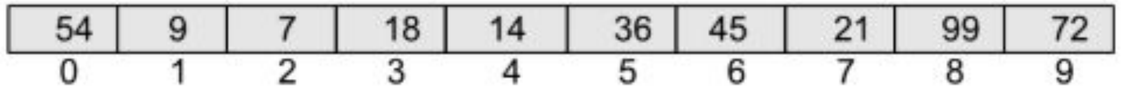


**Figure 8.13** Linear queue

Here, FRONT = 0 and REAR = 9.

Now, if you want to **insert another value**, it will not be possible because the queue is completely full. There is no empty space where the value can be inserted.

- Consider a scenario in which **two successive deletions** are made.

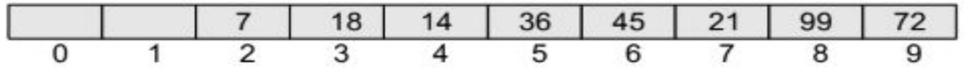


**Figure 8.14** Queue after two successive deletions

Here, FRONT = 2 and REAR = 9.

- Suppose, we want to insert a new element in the queue, Even though there is space available, the overflow condition still exists because the condition **rear = MAX – 1** still holds true. This is a major drawback of a linear queue.

<u>To resolve this problem, we have two solutions.</u>

1. First, shift the elements to the left so that the vacant space can be occupied and utilized efficiently.
2. The second option is to use a circular queue. In the circular queue, the first index comes right after the last index.

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

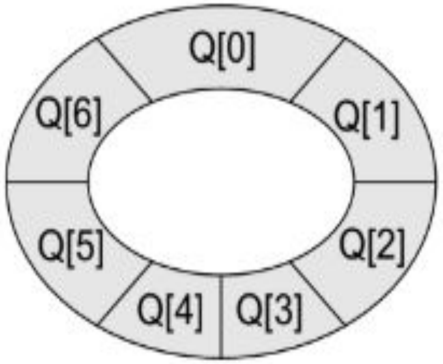


**Figure 8.15** Circular queue

## For insertion in circular queue, we have to check for the following:

- If front = 0 and rear = MAX-1, then the circular **queue is full**.

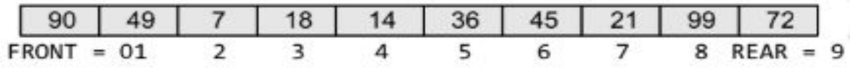
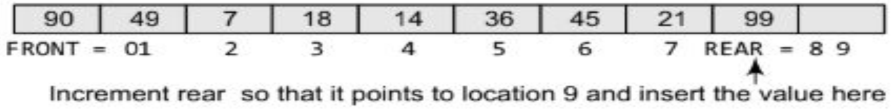


**Figure 8.16** Full queue

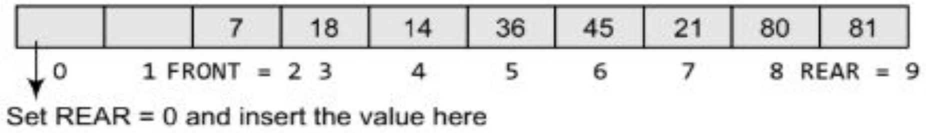
- If rear!= MAX – 1, then rear will be incremented value will be **inserted.**



90 49 7 18 14 36 45 21 99
FRONT = 01 2 3 4 5 6 7 REAR = 8 9
Increment rear so that it points to location 9 and insert the value here

**Figure 8.17** Queue with vacant locations

- If front!=0 and rear = MAX-1, that means queue is not full **set Rear =0** and insert new element there.



7 18 14 36 45 21 80 81
0 1 FRONT = 2 3 4 5 6 7 8 REAR = 9
Set REAR = 0 and insert the value here

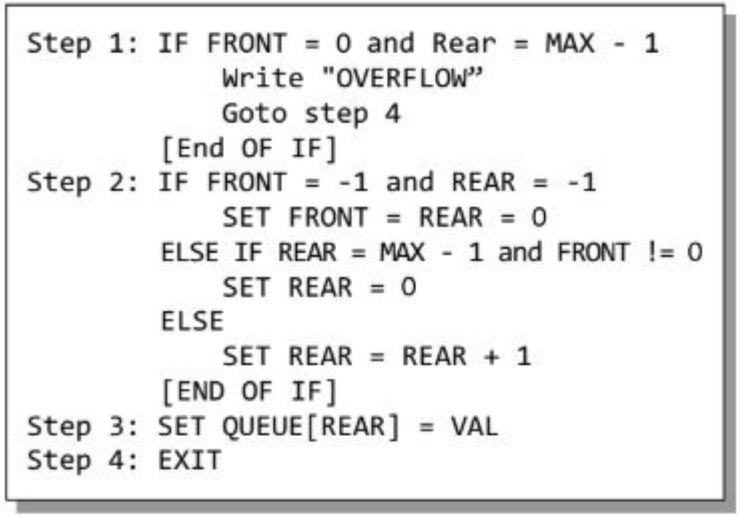
**Figure 8.18** Inserting an element in a circular queue

## <u>Algorithm to insert an element in a circular queue.</u>

**Step 1:** we check for the overflow condition.

**Step 2:** we make two checks. First to see if the queue is empty and second to see if the REAR end has already reached the maximum capacity while there are certain free locations before the FRONT end.

**Step 3:** the value is stored in the queue at the location pointed by REAR.



```
Step 1: IF FRONT = 0 and Rear = MAX - 1
             Write "OVERFLOW"
             Goto step 4
        [End OF IF]
Step 2: IF FRONT = -1 and REAR = -1
             SET FRONT = REAR = 0
        ELSE IF REAR = MAX - 1 and FRONT != 0
             SET REAR = 0
        ELSE
             SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = VAL
Step 4: EXIT
```

**Figure 8.19** Algorithm to insert an element in a circular queue

## For deletion in queue, we have to check for the following:

- If **front = –1**, then there are no elements in the queue. So, an underflow condition will be reported.

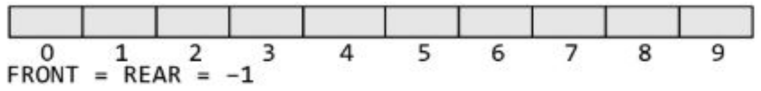


**Figure 8.20** Empty queue

- If the queue is **not empty and FRONT = REAR** then after deleting the element at the front the queue becomes empty and so **front and rear are set to –1.**

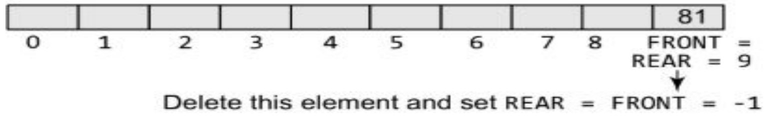
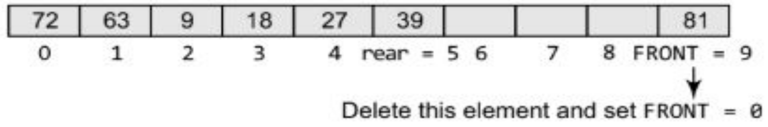


**Figure 8.21** Queue with a single element

- If the queue is **not empty and front = MAX–1,** then after deleting the element at the front, **front is set to 0.**



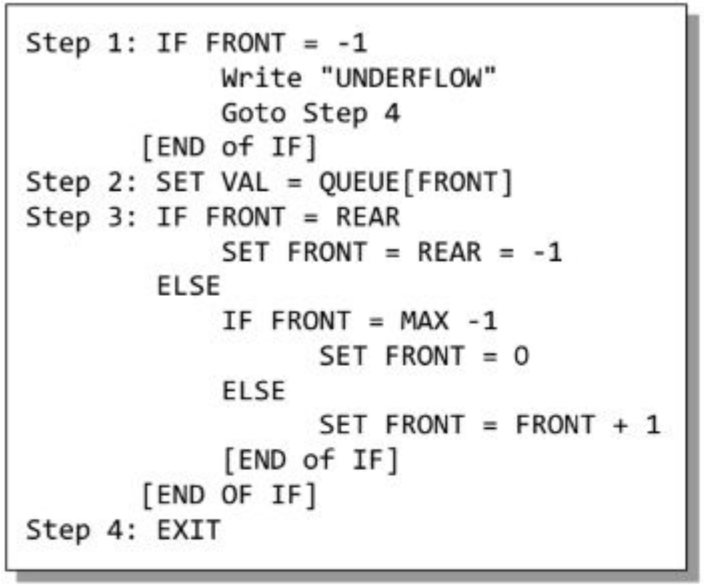
72 63 9 18 27 39 81
0 1 2 3 4 rear = 5 6 7 8 FRONT = 9
Delete this element and set FRONT = 0

**Figure 8.22** Queue where FRONT = MAX–1 before deletion

## Algorithm to delete an element in a circular queue.

**Step 1:** we check for the underflow condition.
**Step 2:** the value of the queue at the location pointed by FRONT is stored in VAL.

**Step 3:** we make two checks. First to see if the queue has become empty after deletion and second to see if FRONT has reached the maximum capacity of the queue. The value of FRONT is then updated based on the outcome of these checks.



```
Step 1: IF FRONT = -1
              Write "UNDERFLOW"
              Goto Step 4
        [END of IF]
Step 2: SET VAL = QUEUE[FRONT]
Step 3: IF FRONT = REAR
              SET FRONT = REAR = -1
        ELSE
              IF FRONT = MAX -1
                    SET FRONT = 0
              ELSE
                    SET FRONT = FRONT + 1
              [END of IF]
        [END OF IF]
Step 4: EXIT
```

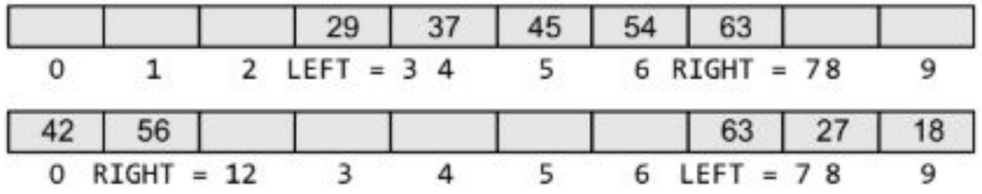
**Figure 8.23** Algorithm to delete an element from a circular queue

## 2. Deque (Double ended queue)

- A deque (pronounced as ‘deck’ or ‘dequeue’) is a list in which the elements can be inserted or deleted at either end.
- It is also known as a **head-tail linked list** because elements can be added to or removed from either the front (head) or the back (tail) end.
- However, no element can be added and deleted from the middle. In the computer’s memory, a deque is implemented using either a circular array or a circular doubly linked list.
- In a deque, two pointers are maintained, **LEFT and RIGHT,** which point to either end of the deque.
- The elements in a deque extend from the **LEFT end to the RIGHT end** and since it is circular, **Dequeue[N–1]** is followed by **Dequeue[0].**

## There are two variants of a double-ended queue.

- **Input restricted deque:** In this dequeue, insertions can be done only at one of the ends, while **deletions can be done from both ends.**
- **Output restricted deque:** In this dequeue, deletions can be done only at one of the ends, while **insertions can be done on both ends.**



29 37 45 54 63
0 1 2 LEFT = 3 4 5 6 RIGHT = 7 8 9
42 56 63 27 18
0 RIGHT = 1 2 3 4 5 6 LEFT = 7 8 9

**Figure 8.24** Double-ended queues

## 3. Priority Queues

A priority queue is a data structure in which each element is assigned a priority. The priority of the element will be used to determine the order in which the elements will be processed.

<u>The general rules of processing the elements of a priority queue are</u>

- An element with **higher priority** is processed before an element with a **lower priority.**
- Two elements with the same priority are processed on a **first-come-first-served (FCFS)** basis.

- Priority queues are widely used in operating systems to execute the **highest priority process first**. The priority of the process may be set based on the **CPU time** it requires to get executed completely.

**Ex 1:** if there are 3 processes, where the first process needs 5 ns to complete, the second process needs 4 ns, and the third process needs 7 ns, then the second process will have the highest priority and will thus be the first to be executed.

**Ex 2:** In case we have to run two processes at the same time, where one process is concerned with online order booking and the second with printing of stock details, then obviously the online booking is more important and must be executed first.

## Implementation of a Priority Queue

<u>There are 2 ways to implement a priority queue:</u>

- Use of **sorted list** to store the elements so that when an element has to be taken out, the queue will not have to be searched for the element with the highest priority
- Use of **unsorted list** so that insertions are always done at the end of the list. Every time when an element has to be removed from the list, the element with the highest priority will be searched and removed.

## Linked Representation of a Priority Queue

every node of the list will have 3 parts:

(a) the information or data part

(b) the priority number of the element

(c) the address of the next element.

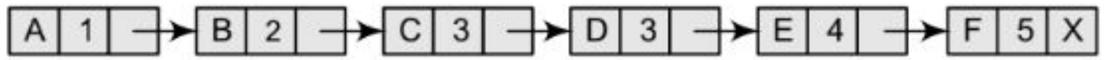


**Figure 8.25** Priority queue

**Ex:** Linked list is a sorted priority queue having 6 elements. The element with a higher priority comes before the element with a lower priority. However, we can definitely say that C was inserted in the queue before D because when two elements have the same priority the elements are arranged and processed on FCFS principle.

## Insertion in Priority Queue

- When a new element has to be inserted in a priority queue, we have to traverse the entire list until we find a node that has a **priority lower than that of the new element.**
- The new node is inserted before the node with the **lower priority**. However, if there exists an element that has the same priority as the new element, the new element is inserted after that element.

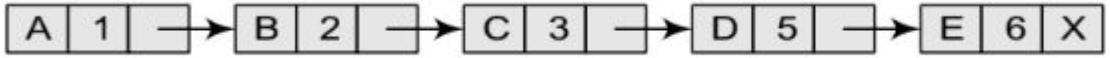


**Figure 8.26** Priority queue

- If we have to insert a new element with **data = F** and **priority number = 4,** then the element will be inserted before D that has priority number 5, which is lower priority than that of the new element.

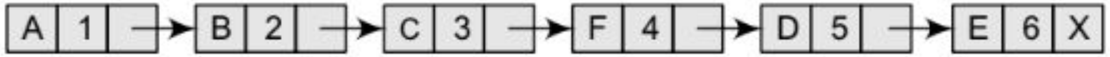


**Figure 8.27** Priority queue after insertion of a new node

- However, if we have a new element with **data = F** and **priority number = 2**, then the element will be inserted after B, as both these elements have the same priority but the insertions are done on FCFS basis

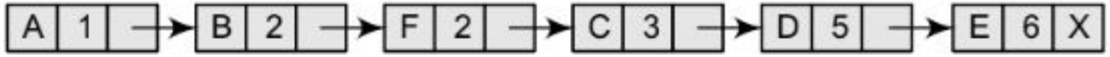


**Figure 8.28** Priority queue after insertion of a new node

## Deletion in Priority Queue

Deletion Deletion is a very simple process in this case. The first node of the list will be deleted and the data of that node will be processed first.

## Array Representation of a Priority Queue

Use a 2-D array for this purpose where each queue will be allocated the same amount of space. Look at the 2-D representation of a priority queue given below. FRONT[K] and REAR[K] contain the front and rear values of row K, where K is the priority number. Note that here we are assuming that the row and column indices start from 1, not 0.

**Insertion** To insert a new element with **priority K** in the priority queue, add the element at the rear end of row K, where K is the row number as well as the priority number of that element. For example, if we have to insert an **element R with priority number 3,** then the priority queue will be given as shown in Fig. 8.30.

**Deletion** To delete an element, we find the first non-empty queue and then process the front element of the first non-empty queue. In our priority queue, the first non-empty queue is the one with **priority number 1 and the front element is A**, so A will be deleted and processed first. In technical terms, find the element with the smallest K, such that FRONT[K] != NULL.

| FRONT | REAR |
|---|---|
| 3 | 3 |
| 1 | 3 |
| 4 | 5 |
| 4 | 1 |

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & & & & E & F \\ 4 & I & & & G & H \end{array}$$

**Figure 8.29** Priority queue matrix

| FRONT | REAR |
|---|---|
| 3 | 3 |
| 1 | 3 |
| 4 | 1 |
| 4 | 1 |

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & & & A & & \\ 2 & B & C & D & & \\ 3 & R & & & E & F \\ 4 & I & & & G & H \end{array}$$

**Figure 8.30** Priority queue matrix after insertion of a new element

## APPLICATIONS OF QUEUES

- Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
- Queues are used to transfer data asynchronously (data not necessarily received at same rate as sent) between two processes (IO buffers), e.g., pipes, file IO, sockets.
- Queues are used as buffers on MP3 players and portable CD players, iPod playlist.
- Queues are used in Playlist for jukebox to add songs to the end, play from the front of the list.

## APPLICATIONS OF QUEUES

- Queues are used in operating system for handling interrupts. When programming a real-time system that can be interrupted, for example, **by a mouse click**, it is necessary to process the interrupts immediately, before proceeding with the current job. If the interrupts have to be handled in the order of arrival, then a **FIFO queue** is the appropriate data structure.

**In Josephus problem**, n people stand in a circle waiting to be executed. The counting starts at some point in the circle and proceeds in a specific direction around the circle. In each step, a certain number of people are skipped and the next person is executed (or eliminated). The elimination of people makes the circle smaller and smaller. At the last step, only one person remains who is declared the ‘winner’.

Therefore, if there are n number of people and a number k which indicates that k–1 people are skipped and k–th person in the circle is eliminated, then the problem is to choose a position in the initial circle so that the given person becomes the winner.

**Ex:** if there are 5 (n) people and every second (k) person is eliminated, then first the person at position 2 is eliminated followed by the person at position 4 followed by person at position 1 and finally the person at position 5 is eliminated. Therefore, the person at position 3 becomes the winner.

Try the same process with n = 7 and k =3. You will find that person at position 4 is the winner. The elimination goes in the sequence of 3, 6, 2, 7, 5 and 1.