Unit-I Topic-III Stack

Introduction:

DATA: Data is collection of unprocessed things or information that we provide to the computer for processing.

e.g. – Numbers, String, character etc.

Data Object: Data object is a set of elements or items that may be finite or infinite.

e.g. – 1) set of integers (Infinite set)

Z= { 0, 1, 2, 3, ---------, ∞}

2) Set of Alphabets (Finite set)

P= {A, B, C, ---------, Z}

Data Structure: "Data Structure consists of three things viz. Data objects, its properties and set of legal operations that can be operates on elements of data objects."

OR

"Data structure is logical relationship between elements of data objects and its operation"

e.g: Array, Stack, Queue, Linked List, Tree, Graph etc.

These Data Structures are implemented with the help of some programming languages such C, C++ etc. but Data structure is not programming language.

Types of Data structure:

Depending upon how the elements are stored in memory, there are two types of data structure-

I) Linear data structure
II) Non-linear data structure (Hierarchical Data structure)

Let's see all these types in details-

1) Linear data structure:

The data structure whose elements are stored in memory in sequential (linear) manner is called as "Linear data structure"

e.g. Array, Stack, Queue, Linked list.

Drawback-

In linear data structure, searching process is slow. Because searching also happen in sequential manner i.e. searching starts from first element to the last element in sequential manner and it requires maximum comparisons & hence searching is slow.

2) Non-Linear data structure: (Hierarchical Data structure)

The data structure whose elements are stored in memory in non-sequential (random) manner is called as "Non-Linear data structure"

e.g. Tree, Graph etc.

Advantages-

In non-linear data structure, searching process is fast. Because searching is happen in non-sequential (random) manner i.e. there is no need to compare search element with every element and hence it requires minimum comparisons therefore searching is fast.

STACK

Definition: "Stack is the linear data structure which is an ordered collection of elements or items into which elements or items are inserted (push) and removed (pop) from only one end that end is called top of Stack."

The Stack operates in **LIFO** (Last In First Out) manner i.e. the element which is lastly inserted in stack is the element to come out first from stack. Because stack has only one end to insert and remove element.

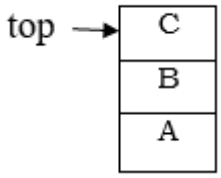
e.g.- 1) Stack of CD plates-

In Stack of CD plate only topmost CD can be taken out and any new CD can be put at the top.

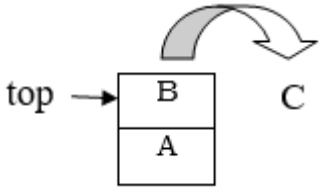
2) Stack of boxes.

Working of Stack is shown in following figures:

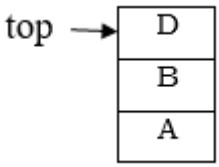
1) After pushing (Adding) elements A, B and C into Stack then it look like as-



2) After popping (Removing) element from above Stack then it look like as-



3) After pushing (Adding) element 'D' into above Stack then it look like as-



Note: For implementation of Stack (by using array i.e. static implementation), we have to define one class as fallow:

```
      # define      max   5
    class       stack
    {
                int   item[max] , top ;
                public:
                      void  create(stack*);
                      void  isempty(stack*);
                      void  isfull(stack*);
                      void  push(stack*,int);
                      int    pop(stack*);
                      void  display(stack*);
    };
```

Here, 'item' is an array used to store elements.
'top' is used to hold index value.

STACK OPERATIONS:

There are some following operations that can be performed on stack.

1) Create( ) operation:

This operation creates the new stack. We can say that stack is created if we assign stack 'top' to -1.

Operation:

```
void      create ( stack   *sp)
{
  sp→top = -1;
}
```

2) isempty( ) operation:

This operation checks whether stack is empty or not. To check stack is empty or not we have to check its *top*. If the value of *'top'* is -1 then we can say that stack is empty otherwise not.

Operation:

```
void   isempty (stack  *sp )
{
  if (sp→ top = =  -1)
  {
    cout<<"\n Stack Is Empty ... ";
  }
  else
  {
    cout<<"\n Stack Is Not Empty....";
  }
}
```

3) isfull( ) operation:

This operation checks whether stack is full or not. To check stack is full or not, we have to check its *top*. If the value of *'top'* is max - 1 then we can say that stack is full otherwise Not.

Operation:

```
void   isfull (stack  *sp )
{
  if (sp→top = = max - 1)
  {
         cout<<"\n Stack Is Full ... ";
  }
  else
  {
             cout<<"\n Stack Is Not Full....";
  }
}
```

4) push( ) operation:

This operation is used to add the new element into stack. The new element is always pushed at the top of stack.

Precondition- While performing push operation, first we have to check **stack is Full or Not** and this is precondition for push operation.

If stack is full and we are going to perform push operation at that time element cannot push into stack i.e. push operation failed such condition is called as "Stack Overflow" condition.

Operation:

```
void  push(stack   *sp, int   x)
{
  if(sp→top = = max-1)
  {
       cout<<"\n Stack Overflows…..";
  }
  else
  {
       ++ sp →top;
       sp → item[sp→top] = x;
  }
}
```

5) pop( ) operation:

This operation is used to remove the element from the stack. Always, top most element is removed from the stack.

Precondition- While performing pop operation, first we have to check **stack is Empty or Not** and this is precondition for pop operation.

If stack is Empty and we are going to perform pop operation at that time element cannot pop from stack i.e. pop operation failed & such condition is called "Stack Underflow" condition.

Operation:

```
int     pop(stack    *sp )
{
   if (sp → top = = -1)
   {
         cout<<"\n Stack Underflows…";
   }
   else
   {
         return(sp → item[sp → top - - ]);
   }
}
```

6) Status( ) operation: (Display( ) operation)

This operation is used to display the elements of stack. The elements are displayed from *top to 0 index*.

Operation:

```
void    status( stack     *sp)
{
   int    i;
   for(i= sp → top; i>=0; i - -)
   {
         cout<<"\n %d "<< sp→item[i];
   }
}
```

Implementation of Stack by using array (Static Implementation of Stack)

```
#include<process.h>
#include<conio.h>
#include<iostream.h>
#define max   100
class      stack
{
   int      item[max],top;
public:
        void  create(stack *);
        void  push(stack *, int);
        int    pop(stack *);
        void  isempty(stack *);
        void  isfull(stack  *);
        void  status(stack *);
};
void  stack::create( stack  *sp)
{
   sp→top = -1;
   cout<<"\n Stack is Created.....";
}
void    stack::isempty( stack    *sp)
{
   if (sp → top = =  -1)
   {
       cout<<"\n Stack Is Empty … ";
   }
  else
 {
     cout<<"\n Stack Is Not Empty....";
   }
}
```

```
int   stack::pop( stack      *sp)
{
       if (sp → top = = -1)
    {
      cout<<"\n Stack Underflows....";
     }
   else
   {
      return(sp → item[sp →top - - ] );
   }
}
void    stack::status( stack   *sp)
{
     int    i;
     for(i= sp → top; i>=0; i - -)
   {
        cout<<"\n<<sp → item[i];
   }

}
void main( )
{
  stack   obj;
  stack  *p , q;
  p = & q ;     // initialization of pointer
  int    n, r, ch ;
  clrscr( );
  do
  {
   cout<<"\n Enter Your choice :   ";
   cout<<"\n 1:Create \n 2:Check for
   Empty \n 3:Check for Full \n 4:Push
```

```
void   stack::isfull (stack   *sp )
{
    if (sp →top = = max - 1)
   {
       cout<<"\n Stack Is Full ... ";
    }
   else
   {
      cout<<"\n Stack Is Not Full....";
   }
}

void    stack::push(stack   *sp, int     x)
{
     if(sp→ top = = max-1)
      {
       cout<<"\n Stack Overflows.....";
       }
  else
   {
      + + sp →top;
      sp → item[sp →top] = x;
      cout<<"\n Element is Pushed....";
   }
}
```

```
\n5:Pop \n 6:Status \n7:Exit ";
cin>>ch;
switch(ch)
{
  case     1:
           obj.create(p);        // call
           break;

    case      2:
              obj.isempty(p);    //call
              break;
    case      3:
              obj.isfull(p);      // call
              break;

    case      4:
      cout<<"\n Enter any number ";
      cin>>n;
      obj.push(p, n);          //call
      break;
   case      5:
     r = obj.pop( p);
  cout<<"\n Poped Element : ="<<r;
    break;
  case     6:
           obj.status( p);
           break;
   case      7:
                exit(0);
   }
 }while(ch != 7);
 getch( );
}   //end of main( )
```

Basic Definitions:

1) Expression: "Expression is the collection of parenthesis, operators and operands."
e.g.- { [A+B]*[X*(M+N)] }
2) Infix Expression: Incase of Infix expression Operator is in between operands.
e.g.- X+Y
3) Prefix Expression: Incase of Prefix expression Operator is before the operands.
e.g- +XY
4) Postfix Expression: Incase of Postfix expression Operator is after the operands.
e.g.- XY+

-- APPLICATIONS OF STACK --

1) Inter conversion between Infix expression, Prefix expression and Postfix expression:

I) Conversion of Infix expression to postfix expression:

To convert infix expression into postfix expression we require following things.
Requirements:

1) Infix expression string which is parenthesized.
2) Stack to store opening parenthesis and operator.
3) Postfix expression string to store result.

Algorithm:

Step 1) Start
Step 2) Input infix expression string which is parenthesized.
Step 3) Read character by character of the infix expression string.
Step 4)

* If Reading character is *Opening parenthesis or operator* then push that character into stack.
* If Reading character is *Operand* then add it into postfix expression string.
* If Reading character is *closing parenthesis* then *repeatedly pop the stack* and add the popped character into postfix expression string if it is not parenthesis, until the corresponding opening parenthesis is encountered.

Step 5) Repeat the Step 3 and Step 4 till to the end of infix expression string.
Step 6) After reading entire infix expression string,
Check stack is empty or not. If Stack is Not empty then perform pop operation repeatedly until stack becomes empty and add popped character into postfix expression string if it not parenthesis.
Step 7) Display postfix expression string.
Step 8) Stop
e.g. - Infix Expression: ((X + Y) * Z)

Reading Character	**Stack**	**Postfix String**
(	(	
(	((	
X	((	X
+	+ ((	X
Y	+ ((	XY
)	(	XY+
*	* (	XY+
Z	* (	XY+Z
)	EMPTY	**XY+Z***

Postfix expression is: XY+Z*

Question: Convert following infix expressions into postfix expression:

1) ((A+B) * (C – D)) / E **ANWER:** AB + CD - * E /

2) (A+ (B*C)-(D/E $ F)*G) **ANSWER:** ABC * DEF $ / G * - +

// Program Which Convert Infix Expression to Postfix Expression

```
# include<iostream.h>
 #include<process.h>
 #include<conio.h>
 #define  max  100
 class  stack
 {
  public:
    int  item[max],top;
    void  create(stack*);
    void  push(stack *, int);
    int   pop(stack *);
  };
void   stack::create(stack  *sp)
{
    sp→top=-1;
```

```
void   main( )
{
   stack    obj;
   stack    *p, q;
   char     infix[50], post[50], r;
   int      i = 0, j = 0;
   clrscr( );
   p = &q ;    // initialization of pointer
   obj.create(p);    //stack creation
   cout<<"\n Enter infix expression =  ";
   cin>>infix;
   while(infix[i] != '\0')
   {
   if(infix[i] = = '(' || infix[i] = = '*' || infix[i]
= = '+' || infix[i] = = '-' || infix[i] = = '/')
```

```
}
void   stack::push(stack   *sp, int   x)
 {
       if(sp → top = = max-1)
        {
             cout<<"\n Stack Overflows…..";
        }
    else
   {
       + + sp  → top;
       sp → item[sp →top] = x;
   }
 }
 int   stack::pop( stack   *sp)
 {
       if (sp → top = = -1)
      {
         cout<<"\n Stack Underflows….";
      }
     else
     {
       return(sp → item[sp →top - - ] );
     }
 }
```

```
  {
      obj.push(p, infix[i] );
  }
  else if( infix[i]==')' )
  {
     while(p → item[p → top] != '('  )
      {
        r = obj.pop( p
       if(r != '(' )
       {
          post[j]=r
          j++;
       }
     }
     obj.pop( p );
  }
 else              // if reading char is operand
 {
    post[j] = infix[i];
    j++;
 }
 i++;
}
if(p → top != -1)
{
    while(p → top !=  -1)
    {
          r = obj.pop(p);
          if(r !=  '(')
          {
               post[j] = r;
               j++;
          }
   }
}
post[j]= '\0';      // set last character as NULL
cout<<"\n Postfix Expression="<<post;
getch( );
}
```

II) Conversion of Infix expression to Prefix expression:

To convert infix expression into prefix expression we require following things.

1) Infix expression string which is parenthesized.
2) One Stack say 'Opstack' to store closing parenthesis and operator.
3) One Stack say 'Oprndstack' to store operands.
4) Prefix expression string to store result.

Algorithm:

Step 1) Start
Step 2) Input infix expression string which is parenthesized.
Step 3) Reverse the infix expression String.
Step 4) Read character by character of the reverse infix expression string.
Step 5) * If Reading character is *closing parenthesis or operator* then push that character into *Opstack*.
* If Reading character is *Operand* then push it into *Oprndstack*.
* If Reading character is *opening parenthesis* then *repeatedly pop* the *Opstack* and *push the popped character into Oprndstack* if it is not parenthesis, until the *Opstack becomes empty*.
Step 6) Repeat the Step 4 and Step 5 till to the end of reverse infix expression string.
Step 7) After reading entire reverse infix string,

Check *Opstack* is empty or not. If *Opstack* is Not empty then perform pop operation repeatedly until *Opstack* becomes empty and push popped character into *Oprndstack* if it not parenthesis.

Step 8) Lastly, perform pop operation repeatedly onto *Oprndstack* until it becomes empty and add the popped character into prefix expression string if it is not parenthesis.

Step 9) Display *prefix* expression string. Step 10) Stop.

e.g. - 1) Infix expression: (X + (Y * Z))

→ Reverse Infix expression string:)) Z * Y (+ X (

Reading character	Opstack	Oprndstack	Prefix Expression
)	)		
)	))		
Z	))	Z	
*	*))	Z	
Y	*))	Y Z	
(	EMPTY	* Y Z	
+	+	* Y Z	
X	+	X * Y Z	
(	EMPTY	+ X * Y Z	
	EMPTY	EMPTY	**+X*YZ**

Prefix Expression= +X*YZ

Que: Convert Following Infix Expressions in to Prefix expression:

1) (A+B) * C Answer: *+ABC
2) ((A+B) * (C-D)) / E Answer: *+AB/-CDE

// Program Which Convert Infix Expression to Prefix Expression

```
#include<process.h>
#include<conio.h>
# include<string.h>
# include<iostream.h>
#define max 100
class stack
{
public:
    int item[max],top;
    void create(stack*);
    void push(stack *, int);
    int pop(stack *);
};
```

```
void main( )
{  char infix[50], pre[50] , r;
   int i=0 , j=0;
  stack obj;
  stack *opstack, p;
  stack *oprndstack, q;
  opstack = &p ;
  oprndstack = &q ;
  obj.create(opstack);
  obj.create(oprndstack);
  clrscr( );
  cout<<"\n Enter infix expression= ";
  cin>>infix;
  strrev(infix);   //  reverse the string
```

```
void  stack::create(stack    *p)
{
    p→top=-1;
}
 void   stack::push(stack   *sp, int   x)
 {
       if(sp → top = = max-1)
        {
             cout<<"\n Stack Overflows.....";
        }
    else
    {
        + + sp  → top;
        sp → item[sp →top] = x;
    }
 }
int   stack::pop( stack   *sp)
 {
       if (sp → top = = -1)
      {
         cout<<"\n Stack Underflows....";
      }
      else
      {
        return(sp → item[sp →top - - ] );
      }
 }
```

```
  while(infix[i] != '\0')
  {
  if(infix[i] = = ')' || infix[i ] = ='*' || infix[i]
= = '+' || infix[i] = = '-' || infix[i] = = '/')
    {
        obj.push(opstack , infix[i]);
    }
   else   if( infix[i] = = '(' )
  {
      while(opstack →top !=  -1)
      {
                r = obj.pop(opstack);
                if(r != ')' )
                {
                   obj.push(oprndstack , r);
                }
      }
  }
  else
  {
      obj.push( oprndstack , infix[i] );
  }
  i++;
 }
if ( opstack→top!=-1)
{
          while(opstack → top !=  -1)
      {
                r = obj.pop(opstack);
                if(r != ')' )
                {
                   obj.push(oprndstack , r);
                }
      }
}

  while(oprndstack → top !=  -1)
 {
   r = obj.pop(oprndstack);
   pre[j] = r;
   j++;
 }
pre[j]= ' \0';
cout<<"\n Prefix expression="<<pre;
 getch( );
}
```

III) Matching parenthesis in an expression:

Basic Definitions:

❖ *Balanced Expression (Valid Expression):*

The expression is said to be balanced or valid if there is closing parenthesis related with opening parenthesis at <u>proper position.</u>

e.g.- 1) (A+B)*[C-D] 2) { [X+Y]*[M*(A+B)] }

❖ *Imbalanced Expression (Invalid Expression):*

The expression is said to be imbalanced or invalid if there is no closing parenthesis related with opening parenthesis at proper position.

e.g. - 1) (A+B] 2) { [X+Y) }

To check expression is balanced (Valid) or Not, we require following Things:

1) An expression which is parenthesized.
2) Stack to store opening parenthesis.

Algorithm:

Step 1) Start

Step 2) Input an infix expression which is parenthesized.

Step 3) Read character by character of an infix expression

Step 4) If reading character is

* *Opening parenthesis* then push it into stack.
* *closing parenthesis* then perform pop operation one time onto stack and compare popped opening parenthesis with reading closing parenthesis,

If popped opening parenthesis is not matches with reading closing parenthesis then display "Expression is Invalid" and goto step 7.

Else if popped opening parenthesis is matches with reading closing parenthesis then it is OK, read next character.

Step 5) Repeat the step 3 and 4 till to the end of input expression.

Step 6) After reading entire expression, check stack is empty or not.

If Stack is *not empty* then display "Expression is Invalid"

Else display "Expression is Valid". Step 7) Stop.

// Program which check entered expression is valid or not.

```
#include<conio.h>
#include<iostream.h>
#define max  50
class   stack
{
public:
      int item[max],top;
      void create(stack*);
      void push(stack*,int);
      int  pop(stack*);
};
void  stack::create(stack *p)
{
   p->top=-1;
}
void stack::push(stack *p,int x)
{
   if(p->top==max-1)
     cout<<"Overflow...";
   else
   {
     ++p->top;
     p->item[p->top]=x;
   }

}
int stack::pop(stack *p)
{
  if(p->top==-1)
   return(0);
   else
return(p->item[p->top--]);
}
```

```
void main()
{
   clrscr();
   stack *p,q,obj;
   p=&q;
   char   exp[50],ch;
   int i=0,t=0;
   obj.create(p);
   cin>>exp;
   while(exp[i]!='\0')
   {
if(exp[i]=='('||exp[i]=='['||exp[i]=='{')
       {
         obj.push(p,exp[i]);
       }
       else
if(exp[i]==')'||exp[i]==']'||exp[i]=='}')
       {
         ch=obj.pop(p);

if((ch=='(')!=(exp[i]==')')||(ch=='[')!=(
exp[i]==']')||(ch=='{')!=(exp[i]=='}'))
         {
            t=1;
            break;
         }
       }
       i++;
   }
   if(p->top!=-1 || t==1)
cout<<"\nInvalid or Imbalance
        Expression";
   else
cout<<"\nValid or Balanced Expression ";
 getch();
}
```

IV) Evaluation (Calculating the value) of postfix expression:

For evaluation of postfix expression we require:

1) Any postfix expression string. 2) One Stack to store operand.

Algorithm:

Step 1) Start
Step 2) Input any postfix expression string.
Step 3) Read character by character of postfix expression string
Step 4) If reading character is -
* Operand then push it into stack.
* Operator then-
I) Perform pop operation two times on to the stack.
II) Calculate the value by using popped operands and reading operator.
III) Push the calculated value into stack.
Step 5) Repeat the step 3 and step 4 till to the end of postfix expression.
Step 6) Perform pop operation one time onto stack and display popped value which is value of postfix expression. Step 7) Stop

e.g.- 1) Evaluate the postfix expression: **345*+**

Reading Char.	Stack	Calculations
3	3	
4	4	
	3	
5	5	
	4	
	3	
*	20	Val= 4 * 5
	3	Val= 20
+	23	Val= 3 + 20
		Val= 23

Here, we read entire postfix expression. Now we perform pop operation one time and display popped value. Which is value of given postfix expression.

➔ *Thus, value of given postfix expression is **23***

Que. Evaluate following postfix expression (Home Work)

I) AB+CD-*E/ If A=4, B=5, C=6, D=3, E=9 *ANS: 3*

II) 45*9-3$ *ANS:1331*

// Program for evaluation of Postfix expression.

```
#include<conio.h>
# include<iostream.h>
# include<math.h>
#define max 100
class stack
{
   public
   int item[max],top;
void create(stack*);
void push(stack *, int);
int pop(stack *);
};
void stack::create(stack *sp)
{
  sp→top=-1;
}
```

```
void main( )
{
   stack obj;
   stack *p, q;
   p = &q ;  //  initialization of pointer
  char exp[50];
   int i =0, oprnd1 ,oprnd2 , val, z;
   clrscr();
   obj.create(p);  //stack creates
   cout<<"\n Enter postfix expression: ";
   cin>>exp;
   while(exp[i] != '\0')
   {
      if(exp[i] = = '+' || exp[i] ='-' || exp[i] = =
'/' || exp[i]= = '*' || exp[i] = = '$')
      {
```

```
void  stack::push(stack   *sp, int   x)
 {
       if(sp → top = = max-1)
        {
             cout<<"\n Stack
    Overflows…..");
        }
     else
    {
        + + sp  → top;
        sp → item[sp →top] = x;
    }
}

int   stack::pop( stack   *sp)
 {
 if (sp → top = = -1)
 {
   cout<<"\n Stack Underflows….";
 }
 else
 {
    return(sp → item[sp →top - - ] );
  }
 }
```

```
          oprnd2 = obj.pop(p);
          oprnd1 = obj.pop(p);
          switch(exp[i])
          {
                case   '+':
                       val = oprnd1+oprnd2;
                       obj.push(p , val);
                       break;
                case   '-':
                       val = oprnd1- oprnd2;
                       obj.push(p,val);
                       break;
                case   '/':
                       val = oprnd1/oprnd2;
                       obj.push(p,val);
                       break;
                case   '*':
                       val = oprnd1* oprnd2;
                       obj.push(p,val);
                       break;
                case   '$':
                val = pow(oprnd1,oprnd2);
                       obj.push(p,val);
                       break;
            }
          }
          else
          {
             obj.push(p , exp[i]- 48);
          }
          i++;
    }
z=obj.pop(p);
cout<<"\n Value of Postfix expression"<<z;
getch( );
 }
```

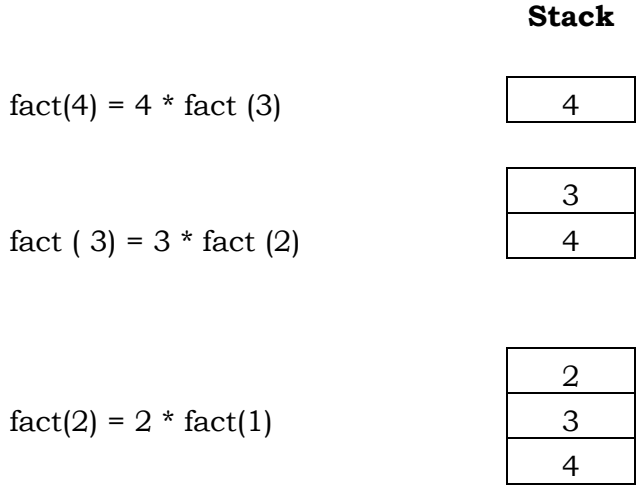
V) Stack in Recursion (Subroutine call):

- We know that recursion is the process of calling the function to itself.
- In recursion, compiler uses system stack to store the remaining values before the next function call is given.
- After the last function call is made then the compiler pops all the values from stack and resume the execution of previous function call.

  Consider the following recursive function to calculate factorial of number:

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

Sequence of function call for n = 4 are shown in following fig:



In above fig. the values 4, 3 and 2 are pushed into stack by the compiler. When value of n becomes to 1 then fact( ) function get last call. At that time all the pushed values i.e. 2, 3 and 4 are popped from stack. And they are multiplied with each other and only one value. i.e. 24 is returned from the fact( ) function.
Thus, for recursion process Stack is used.

Theory Assignment No. 2

Note: Student must submit this assignment in theory note book within seven days from assignment issue date.

1) Define "Data Structure" and explain its types.
2) What is Stack? Explain different operations performed onto stack.
3) Write an algorithm for Push and Pop operation.
4) Why stack is called LIFO data structure?
5) Write preconditions for PUSH and POP operations.
6) What is "stack overflow" and "stack underflow"?
7) Write an algorithm to convert infix expression into postfix form with example.
8) Write an algorithm to convert infix expression into prefix form with example.
9) Write an algorithm to check entered expression is valid or not.
10) Write an algorithm for evaluation of postfix expression with example.
11) Explain the use of Stack in Recursion.
12) Explain peak ( ) operation that returns topmost element of stack.

Practical Assignment No. 2

Note: Student must submit this assignment in practical journal within seven days from assignment issue date.

1) Write a program to implement stack by using array. (Static Implementation of stack)

2) Write a program, which reverses the entered string by using stack.

3) Write a program to check entered string is palindrome or not by using stack.

4) Write a program to convert decimal number into binary number by using stack.

5) Write a program that counts total number of vowels present in string by using stack.

6) Write a program which converts infix expression into prefix expression.

7) Write a program which converts infix expression into Postfix expression.

8) Write a program which check entered expression is valid or not.

9) Write a program that evaluates entered postfix expression.

10) Write a program to calculate factorial of entered number by using recursion.

11) Write a program to calculate digit sum of entered number by using recursion.

12) Write a program to find face value of entered number by using recursion.

13) Write a program that demonstrate implementation of multiple stacks. *(Hint: Take one stack to store even numbers and other stack to store odd numbers.)*