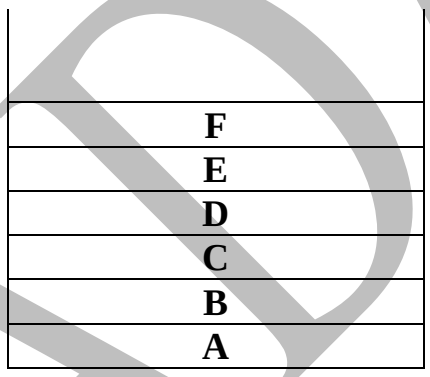
Stack

A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the **top** of the stack.

The definition of the stack provides for the insertion and deletion of items, so that a stack is a dynamic constantly changing object. The definition specifies that a single end of the stack is designated as the stack top. New items may be put on top of the stack, or items which are at the top of the stack may be removed.

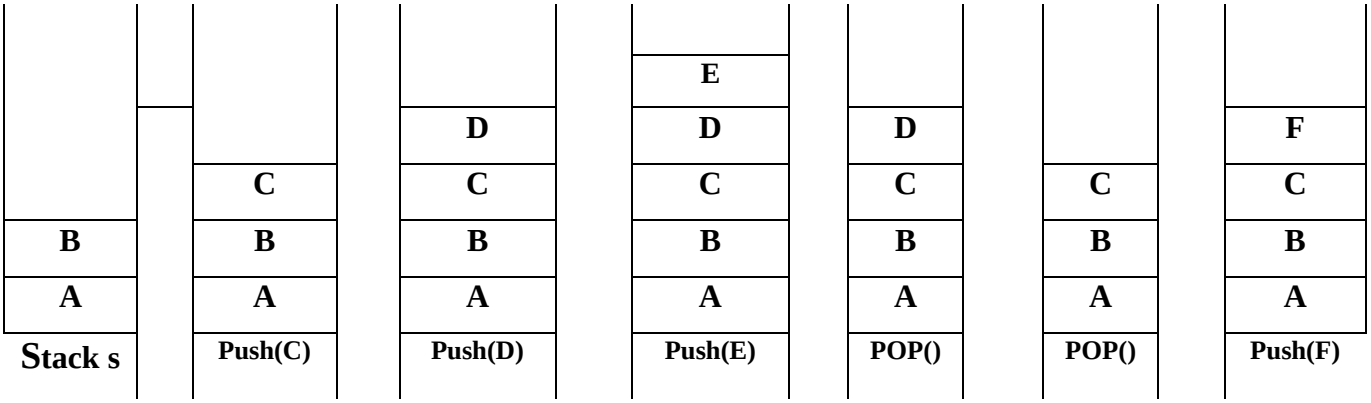
In the figure below, **F** is physically higher on the page then all the other items in the stack, so **F** is the current top element of the stack, If any new items are added to the stack they are placed on top of F, and if any items are deleted, F is the first to be deleted. Therefore, a stack, is collected **a Last – In – First – Out. (LIFO) list.**



Operations on a stack

The two main operations which can be applied to a stack are given spatial names, when an item is added to a stack, it is **push** onto the stack, and when an item is removed, it is **pop** from the stack.

Given a stack **S**, and an item **i**, performing the operation **Push (i)** adds the item **i** to the top of stack **S**. similarity, the operation pop() removes the top element and returns **i** as a function value. Thus the assignment operation **i = Pop();** removes the element at the top below is a motion picture of a stack **S** as it expands (items pushed) and shrinks (item poped) with the passage of item.



The **original idea** of the stack is that there is no upper limit on the number of items that may be kept in a stack. Pushing another item onto a stack produces a larger collection of items. When the stack been implemented in a program represented as an array, therefore we need to give the maximum size of this stack to be shrieked. So, we need as operation called **IsFull ( )** (which determines whether or not a stack is full (overflow) to be applied **before Push operation**). On the other hand, if a stack contains a single item and the stack is poped, the resulting stack contains no item and is called an **Empty stack**.

Therefore, before applying the pop operator to a stack, we must ensure that a stack is not empty. The operation **IsEmpty ()** determine whether or not a stack is empty.

Another operation that can be performed on a stack is to determine what the top item on a stack is without removing it. This operation is written **Peek ()** and return the top element of stack **s**.

i= Peek ();

The Stack Abstract Data Type

Class specification :

Stack
-top:int **- StackSize:int** **- items:object[];**
+Stack(int) **+Push (object):void** **+Pop():object** **+Peek():object** **+IsFull():bool** **+IsEmpty():bool** **+Size():int**

A stack **Stack** is an abstract data type (ADT) that supports the following three methods

ADT : Stack
{
Data: a non zero positive integer number representing StackSize and integer number representing the top of stack , and array of object elements represent the items .

Operations:
A constructor(Stack) :initialize the data to some Data object certain value.
Push(element) : insert object element at the top of the stack.

Input: object; Output: None.

Pop() : Remove from the stack and return the top object on the stack; an error occurs if the stack is empty .

Input: None ; Output: Object;

Peek() : Return the top object on the stack; without removing it; an error occurs if the stack is empty .

Input: None ; Output: Object;

Additionally, let us also define the following supporting methods :

IsEmpty() : Return a Boolean indicating if the stack is empty .

Input: None; Output : Boolean.

IsFull() : Return a Boolean indicating if the stack is full .

Input: None; Output : Boolean.

Size(): return the number of objects in the stack .

Input : None; Output: integer;

End ADT Stack

```
class Stack
{
// data member or data value
    private int top , StackSize;
    private object[] items;
// operations
// Constructer or default Constructer
      public stack(int n)
    {
      StackSize = n;
      items = new object[StackSize];
      top = -1;
    }
```

The pseudocode IsEmpty () is used to test whether a stack is empty, may be written as follow:

```
if (top equal  -1) return true
  else
  return false
```

The pseudocode IsFull () that is used to test whether a stack is full, may be written as fellow:

```
if (top equal  StackSize - 1) return true
  else
    return false;
```

The Algorithm Push ( ) is used to add a new elements to the top of the stack. This Algorithm must perform the following operations:

1- If the stack is full, print a warning message and halt execution.
2- Add a new element into the top of the stack.

The pseudocode Push that may be written as fellow:

```
if (IsFull())
   print ("Stack is full!")
else
   {
      top⟵top+1
   items[top] ⟵ element
   }
```

The Algorithm Pop that is used to remove the element on the top of the stack, must perform the following three actions:

1- If the stack is empty, print a warning message and halt execution.
2- Remove the top element from the stack.
3- Return this element to the calling program.

The pseudocode Pop that may be written as fellow:

```
if (IsEmpty())
        print  "Stack is empty!"
    else
        {
            ele←items[top]
            top← top-1
         }
```

The Algorithm Peek which returns the top element of a stack without removing it from the stack, must perform the following Two actions:

1- If the stack is empty, print a warning message and halt execution.

2- Return the top element from the stack to the calling program.

The pseudocode Peek may be written as follows:

```
if (IsEmpty())
        print  "Stack is empty!"
    else
         ele ←   items[top]
```

The Method Size() which returns the number of elements in the stack :

```
return top;
```

The Method Display() which Display all elements in the stack

```
    if (isEmpty())
        print "Stack is empty!"
    else
    {
      for (int i = top; i >  -1 ; i--)
       print  items[i]
  }
```

The table show the running times of methods in realization of a stack by an array

Method	Time
Size	O(1)
isEmpty	O(1)
isfull	O(1)
push	O(1)
pop	O(1)
peek	O(1)
Display	O(n)

Applications of stacks

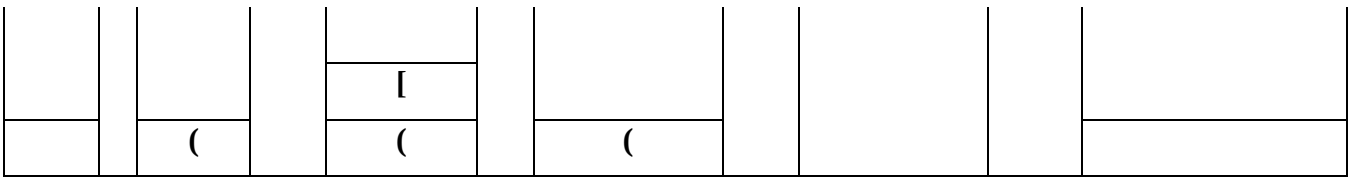
1- **Check for balancing of parenthesis.**
2- **Convert infix expression to postfix.**
3- **Evaluate postfix expression.**
4- **Use of Stack in Function calls.**
5- **Check for palindrome strings**

1-Check for balancing of parenthesis.

A stack is useful here because we know that when a closing symbol such as) is seen , it matches the most recently seen unclosed (. therefore , by placing an opening symbol on a stack, we can easily determine whether a closing symbol makes sense. Specifically , we have the following algorithm .

1- Make an empty stack.
2- Read symbols until the end of the file .
   a- If the symbol is an opening symbol , push it onto the stack.
   b- If it is a closing symbol, do the following:
      i- If the stack is empty , report an error
      ii- Otherwise ,pop the stack
         - if the symbol popped is not the corresponding opening symbol, report an error
         - else we continue.
3- At the end of the file , if the stack is not empty , report an error.

figure below , shows the state of the stack after reading in parts of the string :
{X+(Y-[a+b])*c}



{		{		{		{		{		
{… push		{x+(… push		{x+(y-[… push		{x+(y-[a+b]… pop		{x+(y-[a+b]… pop		{x+(y-[a+b]*c} pop

2-Convert infix expression to postfix expression

Infix, Postfix , and prefix Expressions

Consider the sum of A and B. we think of applying the operator + to the operands A and B and write the Sum as A+B . This particular representation is called **infix** . There are two alternate notations for expressing the Sum of A and B using the symbols A, B and + . These are

+ A B prefix , A B + postfix

In prefix notation the operator precedes the two operands , in postfix notation the operator follows the two operands , and in infix notation the operand is between the two operands. The evaluation of the expression A+B*c, as written in standard infix notation , require Knowledge of which of the two operations + or * is to be performed first. In the case of + and * we Know that mutilation is to be done before addition . Thus A+B*c is interpreter As a A+ (B*C) .

For the Same example , if we want need to write the expression as (A+B)*C .Therefore , Infix notation may require parentheses to specify a decimal or of operations. Using postfix and prefix notation , the notation, the need for parentheses is eliminated because the operator is placed directly after (before) the two operands to which it applied .

To convent infix expression to postfix forms ,we use the following algorithm :

1- Completely parenthesize the infix expression with order of priority.

2- Move each operator to the space hold by its corresponding right parenthesis.

3- Remove all parentheses.

We can apply this algorithm to the expression :

A / B*C + D*E – A*C

(((A / B) * C) + (D*E)) – (A * C))

AB/C* DE*+AC*-

The conversion algorithm for infix to prefix specify that , after compactly parenthesizing the infix expression with order of priority , we move each operator to its corresponding left parenthesis .

And after that , delimiting all parentheses For example :-

A / B*C + D*E – A*C

(((A / B) * C) + (D*E)) – (A * C))

-+ * /ABC*DE*AC

In the above example , we have consider four binary operations : addition , subtraction , multiplication and division. The available in C# and are dented by the usual operators + ,- , *. / .The following is the order of precedence (highest to lowest) :

Multiplication / division / And

Addition / Subtraction /OR

-Convert infix expression to postfix expression uses one stack

To correct an infix expression to postfix for, we will use an algorithm that **uses one stack** The description of the algorithm is as follows:

1-Read the character from the infix expression until the end of the file.

2- Test the character.:

- If the character is an operand, add it to the postfix string.
- If the character is an left parenthesis , push it to the stack1.
- If the character is a right parenthesis then pop entries from the stack add them to the postfix string until a left parenthesis is poped. Discovered both left and right parenthesis.
- If the character is a #(end of the expression) , pop all entries that remain in the stack add them to postfix string.
- Otherwise, pop from the stack and add to postfix the operator that have stack priority greater than or equal to the infix priority of the read character, then push the read character to the opstack.

Example

Convert the following infix expression to postfix using one stack:

A + (B /C) #

Ch	Opstack	Postfix (output string)	Commentary
A		**A**	**Add ch to postfix**
+	**+**	**A**	**Push ch to opstack**
(	**+(**	**A**	**Push ch to opstack**
B	**+(**	**AB**	**Add ch to postfix**
/	**+(/**	**AB**	**Push ch to opstack**
C	**+(/**	**ABC**	**Add ch to postfix**
)	**+**	**ABC/**	**Pop and add to postfix until (is reached.**

# or end of expression		**ABC/+**	**Pop and add to postfix until the opstack is empty.**

Example

Convert the following infix expression to postfix using one stack;

((A – (B + C)) * D) / (E +F) #

Ch	Opstack	Postfix Output string	Commentary
(	(		Push ch to opstack
(	((		Push ch to opstack
A	((	A	Add ch to postfix
_	((-	A	Push ch to opstack
(	((- (	A	Push ch to opstack
B	((- (	AB	Add ch to postfix
+	((- (+	AB	Push ch to opstack
C	((- (+	ABC	Add ch to postfix
)	((-	ABC+	Pop and add to postfix until (is reached
)	(	ABC+-	Pop and add to postfix until (is reached
*	(*	ABC+-	Push ch to opstack
D	(*	A B C + -D	Add ch to postfix
)	(*	ABC+ - D*	Pop and add to postfix until (is reached
/	/	A B C +-D *	Push ch to opstack
(	/(	A B C + -D *	Push ch to opstack
E	/ (	A B C+-D*E	Add ch to postfix
+	/ (+	ABC+-D*E	Push ch to opstack
F	/ (+	ABC+-D*E F	Add ch to postfix
)	/	A B C + - D * E F +	Pop and add to postfix until (is reached

# or end of expression		A B C + - D * E F +/	Pop and add to postfix until the opstack is empty.

-Convert infix expression to postfix expression using two stacks

To correct an infix expression to postfix for, we will use an algorithm that **using two stacks** The description of the algorithm is as follows:

1-Read the character from the infix expression until the end of the file.

2-Test the character.

- If the character is an operand, push it to the stack1.
- If the character is an left parenthesis , push it to the stack1.
- If the character is a right parenthesis then pop entries from the stack2 and push them to the stack1 until a left parenthesis is poped. Discovered both left and right parenthesis.
- If the character is a #(end of the expression) , pop all entries that remain in the stack2 and push them to stack1.
- Otherwise, pop from the stack2 and push to stack1 the operator that have stack priority greater than or equal to the infix priority of the read character, then push the read character to the stack2.

Example

Convert the following infix expression to postfix using two stack: a-b*(c+d)/(e-f) #

Ch	Stack2	Stack1	Commentary
A		**A**	**Push ch to stack1**
-	**-**	**A**	**Push ch to stack2**
B	**-**	**ab**	**Push ch to stack1**
*****	**-***	**ab**	**Push ch to stack2**
(	**-*(**	**ab**	**Push ch to stack2**
C	**-*(**	**abc**	**Push ch to stack1**
+	**-*(+**	**abc**	**Push ch to stack2**
D	**-*(+**	**abcd**	**Push ch to stack1**
)	**-***	**abcd+**	**Pop ch from stack2 and push them into stack1 until (is reached.**
/	**-/**	**abcd+***	**Push ch to stack2**
(	**-/(**	**abcd+***	**Push ch to stack2**
E	**-/(**	**abcd+*e**	**Push ch to stack1**

-	**-/(-**	**abcd+*e**	**Push ch to stack2**
F	**-/(-**	**abcd+* ef**	**Push ch to stack1**
)	**-/**	**abcd+*ef-**	**Pop ch from stack2 and push them into stack1 until (is reached.**
# or end of expression		**abcd+*ef-/-**	**Pop ch from stack2 and push them into stack1 until the stack2 is empty.**

3-Evaluate postfix expression(complier)

We consider evaluating postfix expression using one stack . we will use an algorithm that **using one stack** The description of the algorithm is as follows:

1-Read the character from the postfix expression until the end of the file.

2-Test the character.

- If the character is an operand, push the value associational with it onto the stack .
- If the character is an operator , pop two values from the stack, copy the operator to them, and push the result back onto the stack.

As an example , Execute the following postfix notation using one stack:

623 + - 382 / + * 2 * 3 +

				3												**8**			
		2		**2**		**5**				**3**		**3**							
6		**6**		**6**		**6**		**1**		**1**		**1**							
Read 6		**Read 2**		**Read 3**		**Read +**		**Read -**		**Read 3**		**Read 8**							

2																			
8		**4**																	

3		3		7				2				3
1		1		1		7		7		14		14
Read 2		Read /		Read +		Read *		Read 2		Read *		Read 3

17
Read +

Evaluate infix expression (interpreters using two stacks)

Example

Execute the following infix notation using the two stack : **3+7*2-6**

Ch	Stack1(operation)	Stack2(operant)	Commentary
3		3	Push ch to stack2
+	+	3	Push ch to stack1
7	+	7 3	push ch to stack2
*	* +	7 3	Push ch to stack1
2	* +	2 7 3	push ch to stack2
-	+	14 3	Pop stack1 (*) Pop stack2(2) Pop stack2 (7) 7*2= 14 push 14 to stack2
	-	17	Pop stack1 (+) Pop stack2 (14) Pop stack2 (3) 3+14 = 17 push 17 into stack2 Push ch to stack1 (-)
6	-	6 17	Push ch to stack2
# or end of expression	-	11	Pop stack1 (-) Pop stack2 (6) Pop stack2 (17)

			17-6= 11 push 11 to stack2

4-Use of Stack in Function calls

- Whenever a function begins execution, an **activation record** is created to store the **current environment** for that function
- Current environment includes the
  - values of its parameters,
  - contents of registers,
  - the function's return value,
  - local variables
  - address of the instruction to which execution is to **return** when the function finishes execution (If execution is interrupted by a call to another function)
- Functions may call other functions and thus interrupt their own execution, some data structure must be used to store these activation records so they can be recovered and the system can be reset when a function resumes execution.

- It is the fact that the last function interrupted is the first one reactivated
- It suggests that a stack can be used to store these activation records
- A stack is the appropriate structure, and since it is manipulated during execution, it is called the **run-time stack**

Consider the following program segment

```
static void Main(string[] args)
{
        int a=3;
     int s1= f1(a);
}
int  f1(int x)
{
        int s =( f2(x+1)) ;
}
int f2(int p)
{
        int q=f3(p/2);
        return 2*q;
}
int f3(int n)
{
        return n*n+1;
}
```

Run-time Stack

Parameters	function value	local variable	return address
		a **3**	OS

- OS denotes that when execution of main() is completed, it returns to the operating system

When a function is called …

- Copy of activation record pushed onto run-time stack
- Arguments copied into parameter spaces
- Control transferred to starting address of body of function

Function call f2 (x + 1)

top	P **4**		q	B	AR for f2( )
	x **3**			A	AR for f1( )
			a **3**	OS	AR for main()

The output

10

5-Check for palindrome strings

We have the following pseudocode.

```
Make an empty stack1 and stack2.
Initialize  palindrome=True
Read symbols  and push it into stack1 until the end of the string.
le= Size();     // find the number of item in the stack  (top)
Let   m= le divided by 2.
For (i=0, i< m ; i++)
{
   pop  character from  stack1
   push  character into  stack2
  }
If( m%2 == 1 ) pop character from  stack1
While (!emptystack1  and  palindrome)
                {
                        Pop  ch1 from stack1
                        Pop ch2 from  stack2
                    if(ch1!=ch2)
                    palindrome=False
                                        }

              if(palindrome)
                print "the string is palindrome";
              else
               print"the string is not palindrome;
```

```
or    if (  stack1  equal  stack 2 )
          print "the string is palindrome";
      else
          print "the string is  not   palindrome";
```

Exercises:

Q1:Let Stack be a stack of size (15) integer numbers , write C# program that push a list of integer numbers into stack and print this list in reverse order?

Q2: Let S for push an element in the stack, and U for pop element from the stack.If the order of stack input stream is 1 2 3 4 5

a- What is the output if we execute the following operations : SSUUSSSUUU

b- Which of the following permutations can be obtained as output stream (explain the reason for each case).

i- 51324

ii- 23514

iii- 32154

Q3: If the stack input stream is A B C D E F what is the sequence of operations to get the out put C B D E F A?

Q4- Write a function to check if the stack St is empty.

Q5- State the main applications of the stack?

Q6– Convert the following infix expressions into postfix notations using two stacks:

i- a+b*2/4-(c*5/8-f) +3

ii- m+3 or n-b/2 and (m+n)

Q7– Convert the following infix expressions into postfix notations using one stacks:

i- a+b+c*(-d/3+4)- f

ii- a+(b/2)or(x+y/3-w)and (c-2)*3

iii- x*n+(m-p*4)-f and (-b)+2/c+6

Q8- Execute the following postfix notation using the stack

ab*cde*/+ when a=5, b=6,c=8,d=2,e=2

Q9- Execute the following infix notation using the stack

a+(b+c/2)*4+m if a=10,b=8,c=4,m=20

Q10- Write a procedure that reads in a string and print its characters in reverse order (note: the string terminator is a "." . Which should not be printed as a part of the reversed string).

Q11- Consider the following psedocode:

```
Declare a stack of characters
        while(there are more characters in the word to read)
          {
               read  a character
               if non blank character
```

```
            push the character on the stack
    }
    while (the stack is not empty)
    {
        pop a character off the stack
        write the character to the screen
    }
```

What is written to the screen for the input "My Test"