CMR
INSTITUTE OF TECHNOLOGY

USN

Internal Assessment Test I – March 2023

Sub:	**Data Structures**								**Sub Code:**	**22MCA13**
Date:	**14/03/2023**	**Duration:**	**90 min's**	**Max Marks:**	**50**	**Sem:**	**I**	**Branch:**	**MCA**	

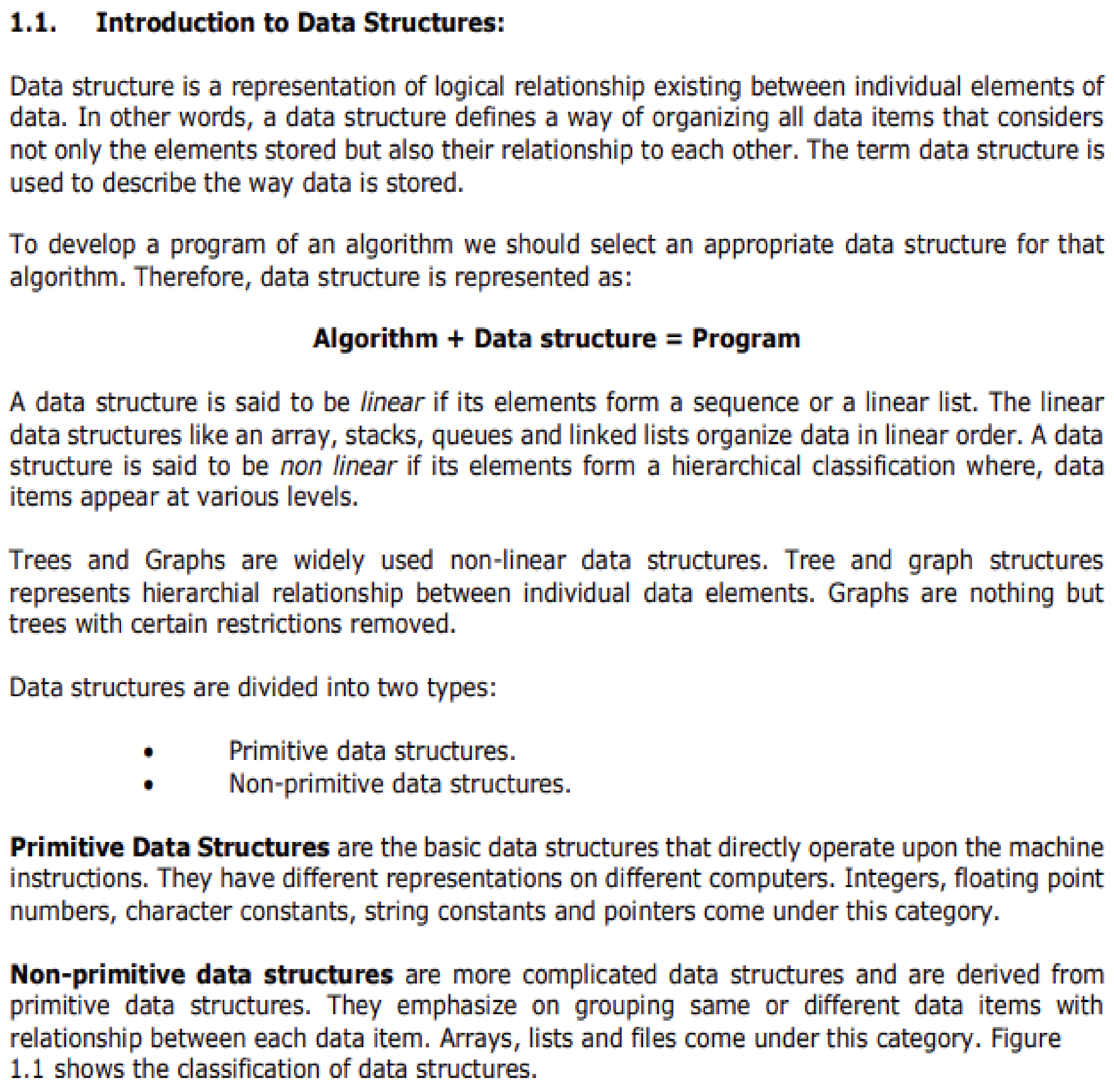
Note : Answer FIVE FULL Questions, choosing ONE full question from each Part.

	PART I	MARKS	OBE	
			CO	RBT
1	What are data structures? Explain the classification of data structures with a neat diagram. **OR**	[10]	CO1	L1
2	Write an Algorithm to convert postfix to infix expression. Trace the Algorithm for the expression ab+c*	[10]	CO1	L1
3	**PART II** Write a program to evaluate postfix expression. Evaluate the following expression **6 2 3 + - 3 8 2 / + * 2 ^ 3 +** **OR**	[5+5]	CO2	L2
4	Convert the following infix expression to postfix expression. 1. A*B/C+(B+C)*D **2.** (A+B^C)/D+E	[5+5]	CO2	L2

5	**PART III** With program explain the push and pop operations of Stack and list any five applications of stack. **OR**	[10]	CO2	L2
6	Write a C program to convert infix to postfix Expression with example.	[10]	CO2	L2
7	**PART IV** What are direct and indirect recursion? Write a C program to generate Fibonacci sequence using Recursion **OR**	[10]	CO2	L2
8	Explain the steps to convert infix to prefix expression. Convert the following infix expression to Prefix **K + L - M * N + (O^P) * W/U/V * T + Q**	[10]	CO2	L3
9	**PART V** Write an Algorithm to find factorial of a given number n. Trace the same for n=4 by showing Recursive tree. **OR**	[10]	CO2	L2
10	Define stack. Explain stack implementation using array	[5+5]	CO2	L3

1. What are data structures? Explain the classification of data structures with a neat diagram.

Data Structure is used for organizing the data in memory. There are various ways of organizing the data in the memory, for eg. array, list, stack, queue, and many more. A data structure is a collection of data values and the relationships between them. Data structures allow programs to store and process data effectively. There are many different data structures, each with its own advantages and disadvantages. Some of the most common data structures are arrays, lists, trees, and graphs. It is a set of algorithms that can be used in any programming language to organize the data in the memory.



1.1. Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.1 shows the classification of data structures.

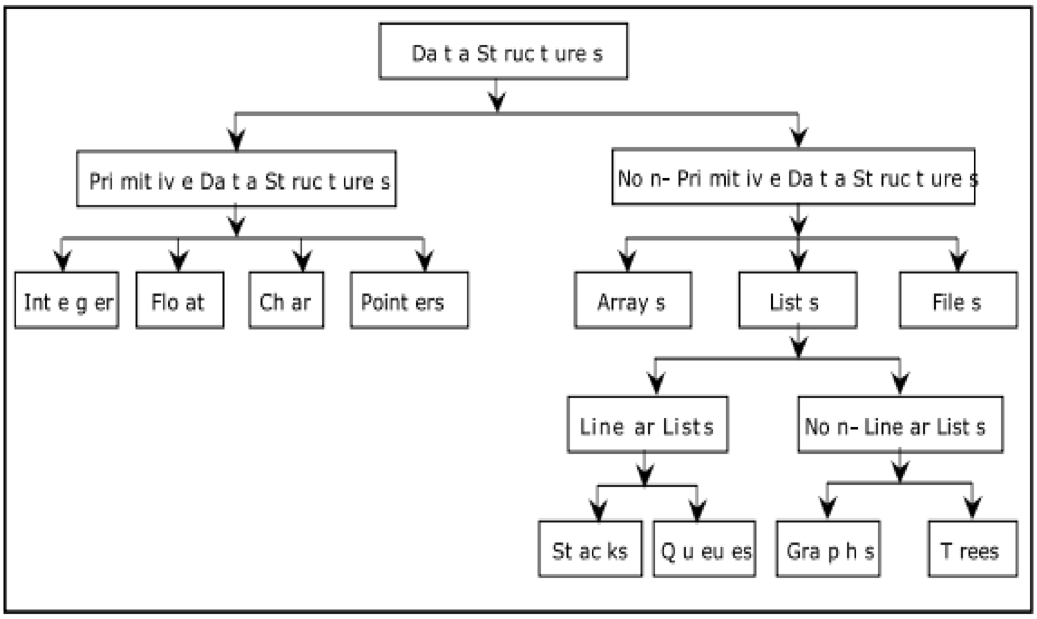


Fig ure 1. 1. C lass if icat io n of Da t a St ruct ures

1.2. Data structures: Organization of data

The collection of data you work with in a program have some kind of structure or organization. No matte how complex your data structures are they can be broken down into two fundamental types:

- Contiguous
- Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure and scattered in memory, but we linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1.2 below illustrates the difference between contiguous and non- contiguous structures.

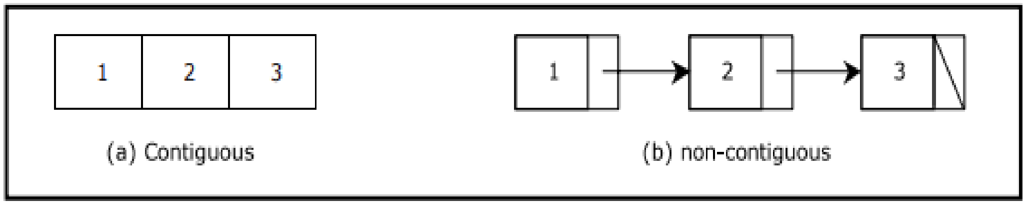


Figure 1.2 Contiguous and Non-contiguous structures compared

2. **Write an Algorithm to convert postfix to infix expression. Trace the Algorithm for the expression ab+c***

Steps to Convert Postfix to Infix :

Approach

For converting Postfix to infix we use a stack . The stack helps us store the operands. Whenever an operator is found , we pop two operands from the stack and push a new operand , which is the result of the current operator on the popped operands, into the stack with parenthesis . The final element at the top of the stack will be our infix expression .

Algorithm

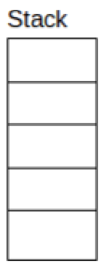
1. Scan all symbols one by one from left to right in the given postfix expression .
2. If the reading symbol is an operand , push it into the stack .
3. If the reading symbol is an operator , then a. Pop two expression from the stack , operand1 and operand2 , perform operation with respect to symbol scanned.(operand2

   b. Push "(operand2 operand1 ")" into the stack
4. If there is no symbol left then stop the process. Top of the stack will have the required infix expression .

   NOTE : '+' denotes the concatenation of strings .

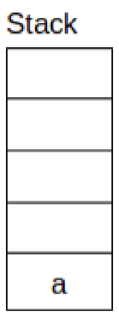
1. Scan the symbol from left to right, based on the input symbol go to step 2 or 3.
2. If symbol is operand then push it into stack.
3. If symbol is operator then pop top 2 values from the stack.
4. This 2 popped value is our operand .
5. Create a new string and put the operator between this operand in string.
6. Push this string into stack.
7. At the end only one value remain in stack which is our infix expression.

- **Example** :
- Input postfix expression is ab+c*

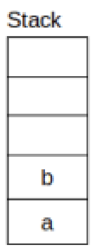


- Initial empty stack

- First input(character) is operand so pushed into stack.

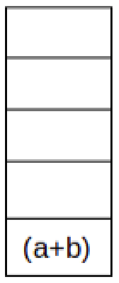


- Second character is also operand so pushed into stack.

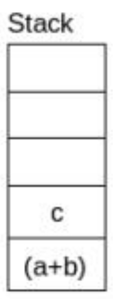


- Third character is operator so we need to popped 2 value from stack.then we need to add operator between this two operand.

Popped operand a,b .
New string =(a+b)
Push this string into stack

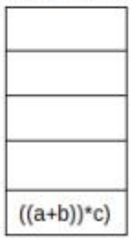


- Fourth character is operand so we need to push into stack.



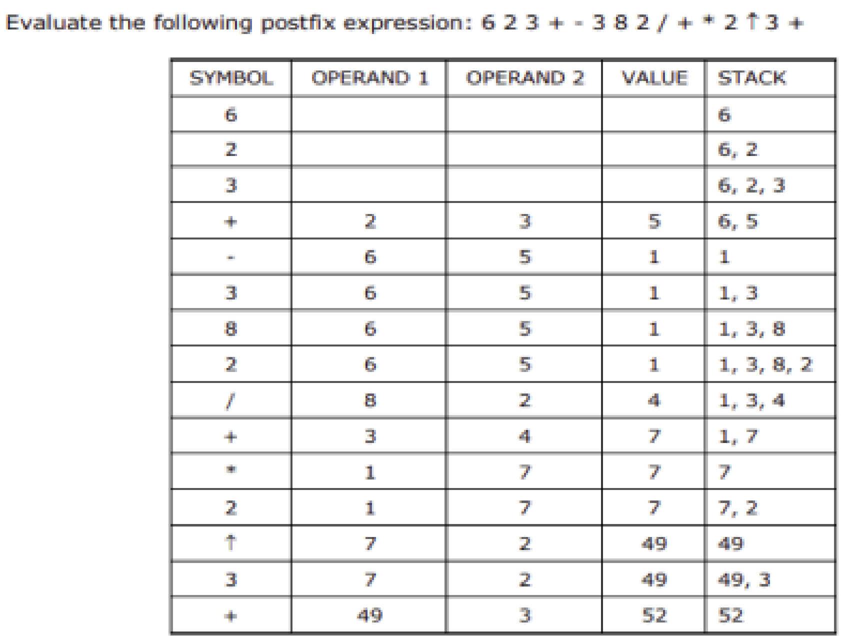
- Fifth character is operator so we need to popped 2 value from stack, then we need to add operator between this two operand.

Popped operand (a+b),c .
New string =((a+b)*c)
Push this string into stack



3. Write a program to evaluate postfix expression. Evaluate the following expression

6 2 3 + - 3 8 2 / + * 2 ^ 3 +



Evaluate the following postfix expression: 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

SYMBOL	OPERAND 1	OPERAND 2	VALUE	STACK
6				6
2				6, 2
3				6, 2, 3
+	2	3	5	6, 5
-	6	5	1	1
3	6	5	1	1, 3
8	6	5	1	1, 3, 8
2	6	5	1	1, 3, 8, 2
/	8	2	4	1, 3, 4
+	3	4	7	1, 7
*	1	7	7	7
2	1	7	7	7, 2
↑	7	2	49	49
3	7	2	49	49, 3
+	49	3	52	52

Implement a Program in C for evaluating a Postfix Expression.

```
#include<stdio.h>
int stack[20];
int top = -1;
void push(int x)
{
stack[++top] = x;
}
int pop()
{
return stack[top--];
}
int main()
{
char exp[20];
char *e;
int n1,n2,n3,num;
printf("Enter the expression :: ");
scanf("%s",exp);
e = exp;
while(*e != '\0')
{
if(isdigit(*e))
{
num = *e - 48;
push(num);
}
else
{
n1 = pop();
n2 = pop();
switch(*e)
{
case '+':
{
n3 = n1 + n2;
break;
}
Case ‘-‘
{
n3 = n2 - n1;
break;
}
case '*':
{
n3 = n1 * n2;
break;
}
case '/':
{
n3 = n2 / n1;
break;
}
}
push(n3);
}
e++;
}
printf("\nThe result of expression %s = %d\n\n",exp,pop());
return 0;
}
```

Output:

```
Enter the Expression: 234+*

The result of expression 234+* = 14
```

4. Convert the following infix expression to postfix expression.

1. A*B/C+(B+C)*D   **2.** (A+B^C)/D+E

Solution:

AB*C/BC+D*+   ABC^+D/E+

5. **With program explain the push and pop operations of Stack and list any five applications of stack.**

Stack:

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are: Stack and Queue

Stack Definition

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed. The two basic operations associated with stacks are:

Push: is the term used to insert an element into a stack.

- Pop: is the term used to delete an element from a stack.

- "Push" is the term used to insert an element into a stack.

"Pop" is the term used to delete an element from the stack. All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation and Operations:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a stack overflow condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a stack underflow condition. When an element is added to a stack, the operation is performed by push(). Figure 4.1 shows the creation of a stack and addition of elements using push()

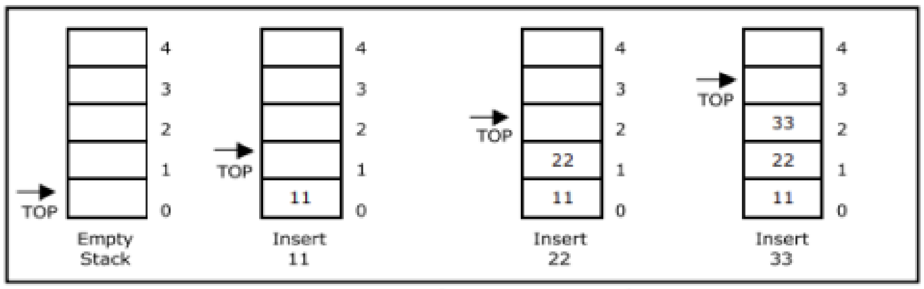


Figure 4.1. Push operations on stack

When an element is taken off from the stack, the operation is performed by pop(). Figure 4.2 shows a stack initially with three elements and shows the deletion of elements using pop().

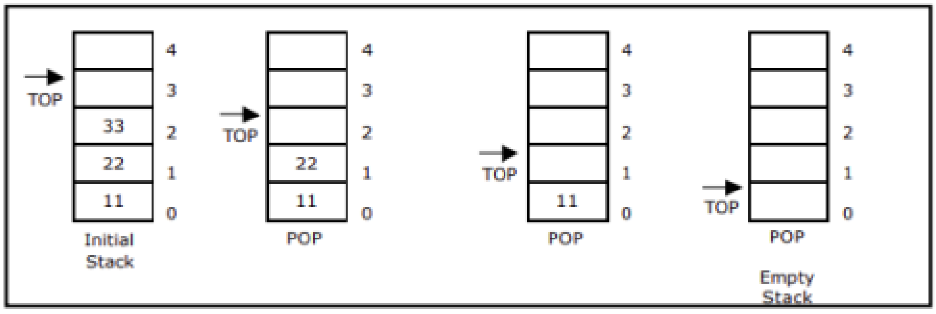
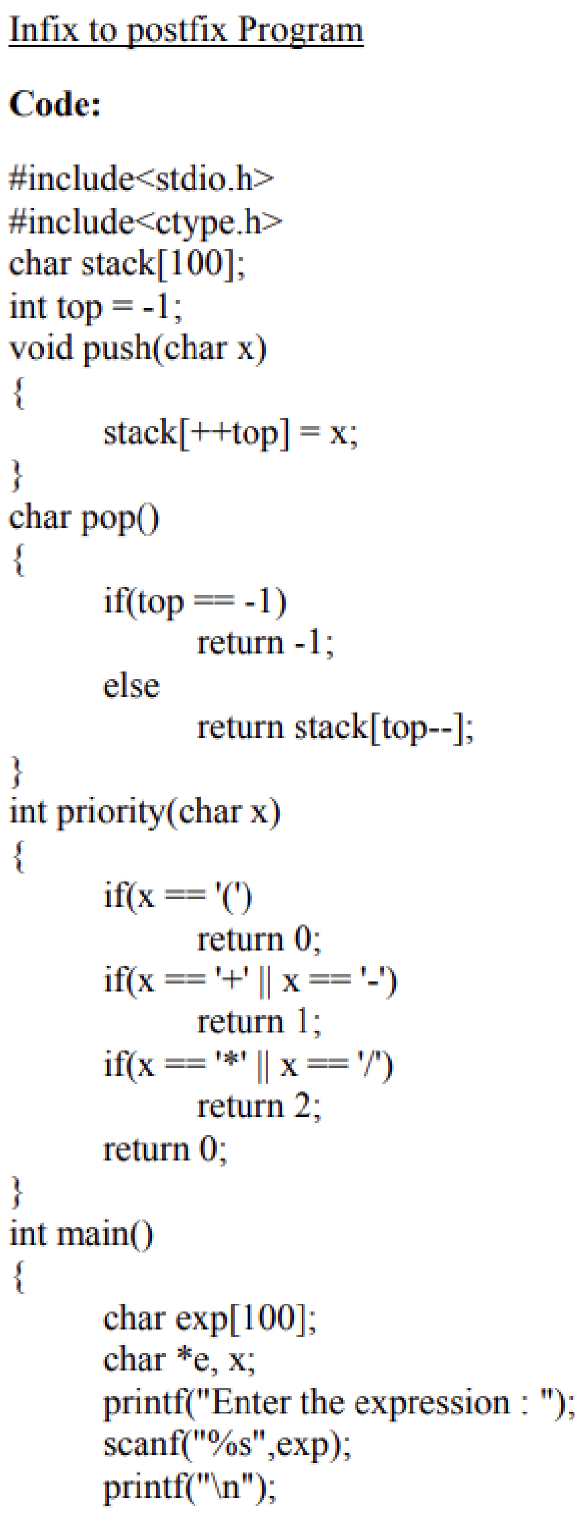


Figure 4.2. Pop operations on stack

Applications of Stack:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

6. Write a C program to convert infix to postfix Expression with example.



Infix to postfix Program

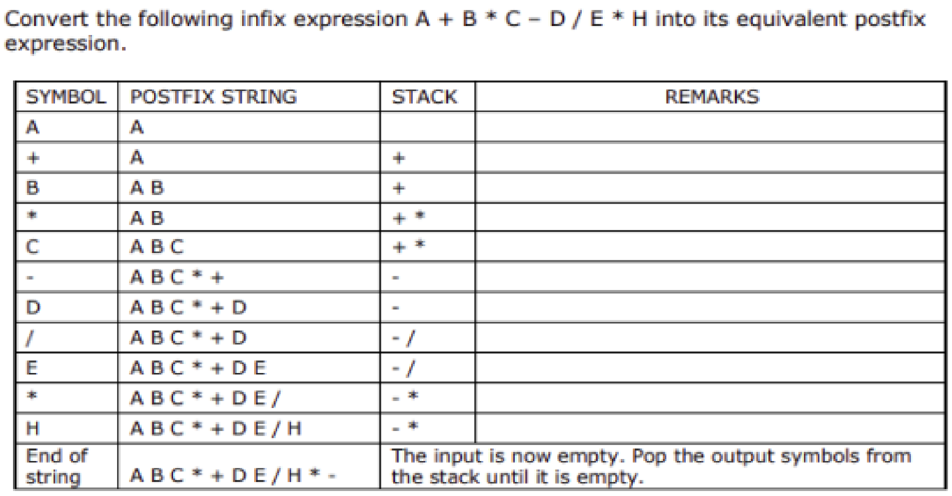
Code:

```
#include<stdio.h>
#include<ctype.h>
char stack[100];
int top = -1;
void push(char x)
{
        stack[++top] = x;
}
char pop()
{
        if(top == -1)
                return -1;
        else
                return stack[top--];
}
int priority(char x)
{
        if(x == '(')
                return 0;
        if(x == '+' || x == '-')
                return 1;
        if(x == '*' || x == '/')
                return 2;
        return 0;
}
int main()
{
        char exp[100];
        char *e, x;
        printf("Enter the expression : ");
        scanf("%s",exp);
        printf("\n");
        e = exp;
        while(*e != '\0')
        {
                if(isalnum(*e))
                        printf("%c ",*e);
                else if(*e == '(')
                        push(*e);
                else if(*e == ')')
                {
                        while((x = pop()) != '(')
                                printf("%c ", x);
                }
                else
                {
                        while(priority(stack[top]) >= priority(*e))
                        printf("%c ",pop());
                        push(*e);
                }
                e++;
        }
        while(top != -1)
        {
                printf("%c ",pop());
        }return 0;
}
```

Output:

```
Enter the Expression: a+b*c
abc*+
```

Example Expression:



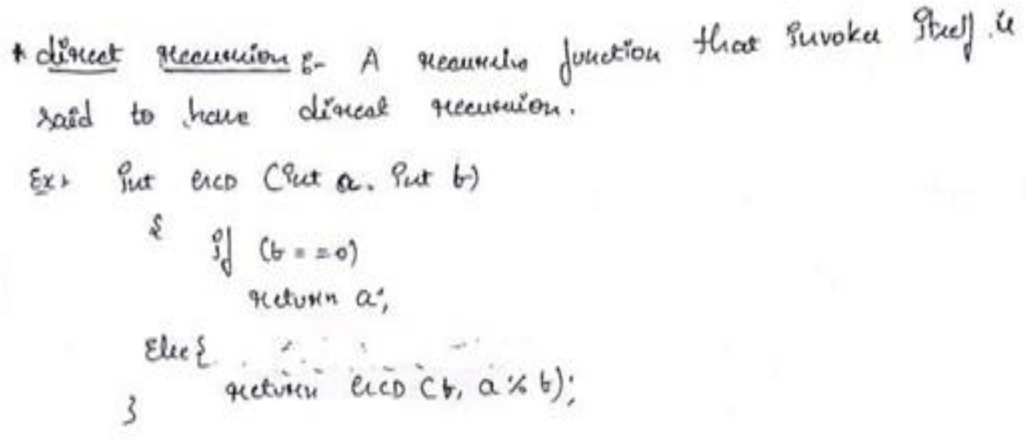
Convert the following infix expression A + B * C – D / E * H into its equivalent postfix expression.

SYMBOL	POSTFIX STRING	STACK	REMARKS
A	A		
+	A	+	
B	A B	+	
*	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 /	- *	
H	A B C * + D E / H	- *	
End of string	A B C * + D E / H * -	The input is now empty. Pop the output symbols from the stack until it is empty.	

7. **What are direct and indirect recursion? Write a C program to generate Fibonacci sequence using Recursion.**

Recursion:

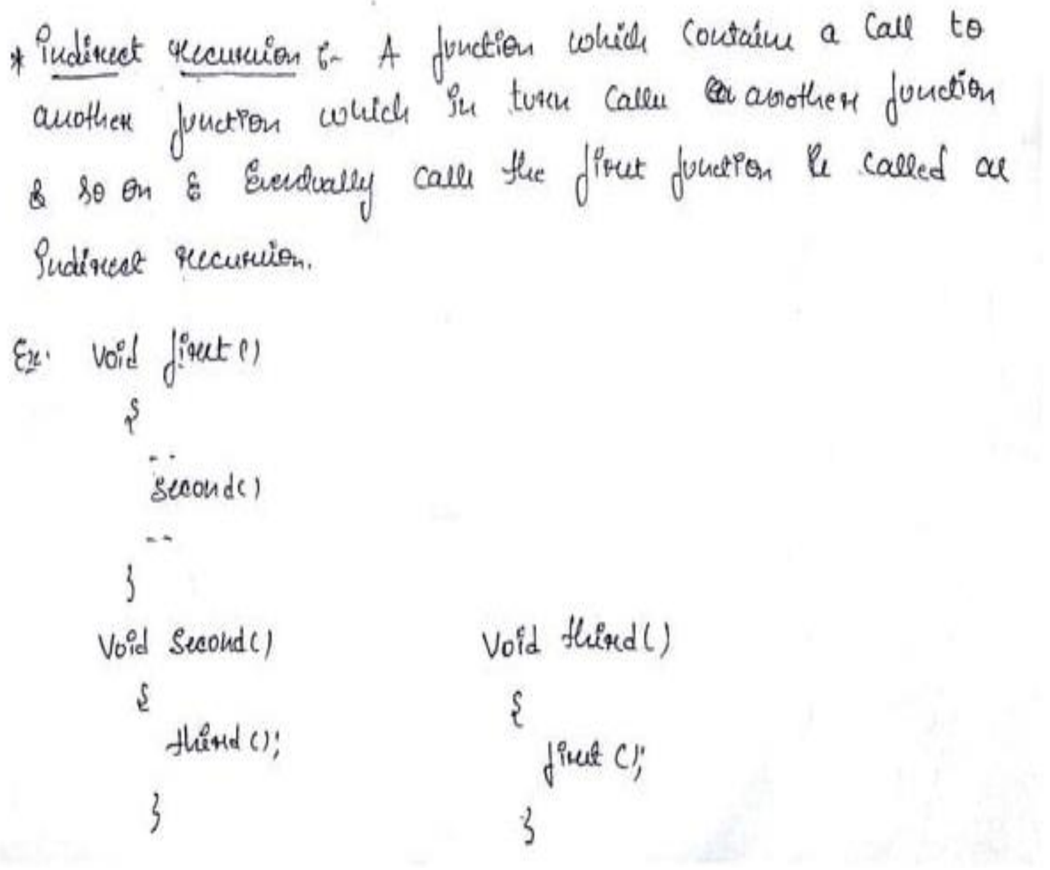
Introduction to Recursion: A function is recursive if a statement in the body of the function calls itself. Recursion is the process of defining something in terms of itself. For a computer language to be recursive, a function must be able to call itself.



* direct recursion :- A recursive function that invokes itself is said to have direct recursion.

Ex: int gcd (int a, int b)

```
    {
      if (b==0)
        return a;
    Else{
        return gcd (b, a%b);
    }
```

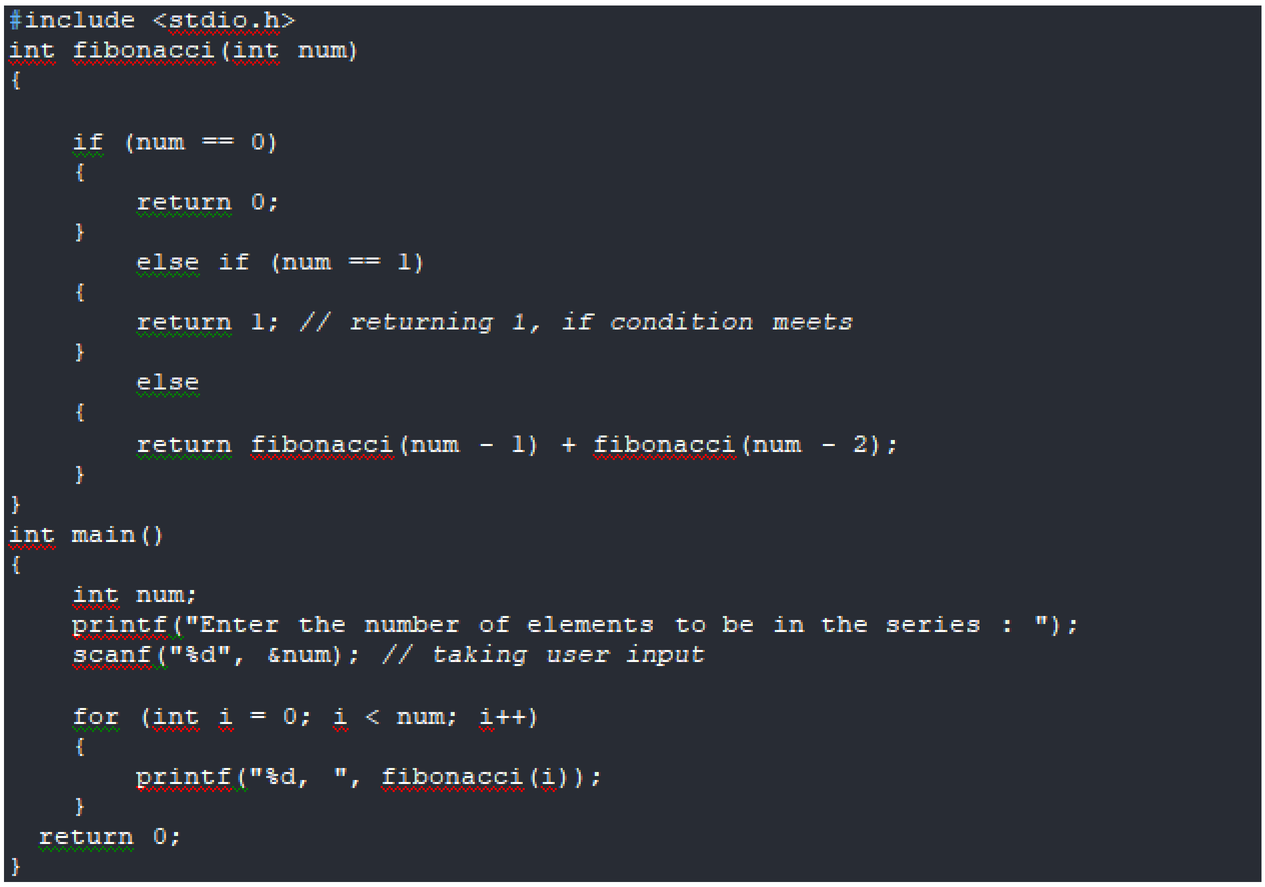


* indirect recursion :- A function which contains a call to another function which in turn calls another function & so on & eventually calls the first function is called as indirect recursion.

Ex: void first()

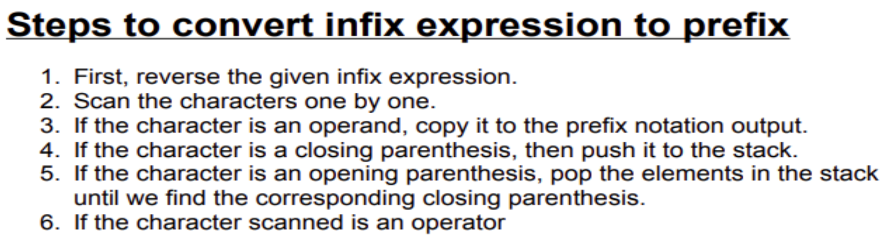
```
     {
       ..
       second()
       ..
     }
     void second()          void third()
      {                      {
        third();               first();
      }                      }
```

<u>C program to generate Fibonacci sequence using Recursion</u>



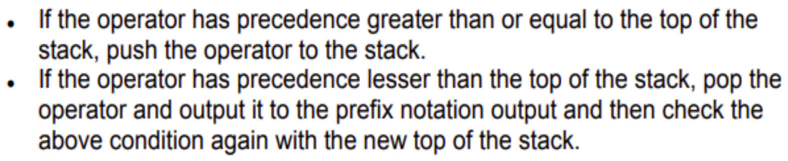
```c
#include <stdio.h>
int fibonacci(int num)
{

    if (num == 0)
    {
        return 0;
    }
        else if (num == 1)
    {
        return 1; // returning 1, if condition meets
    }
        else
    {
        return fibonacci(num - 1) + fibonacci(num - 2);
    }
}
int main()
{
    int num;
    printf("Enter the number of elements to be in the series : ");
    scanf("%d", &num); // taking user input

    for (int i = 0; i < num; i++)
    {
        printf("%d, ", fibonacci(i));
    }
  return 0;
}
```

8. Explain the steps to convert infix to prefix expression. Convert the following infix expression to Prefix K + L - M * N + (O^P) * W/U/V * T + Q

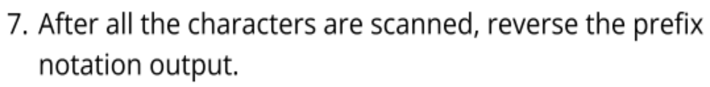


<u>Steps to convert infix expression to prefix</u>

1. First, reverse the given infix expression.
2. Scan the characters one by one.
3. If the character is an operand, copy it to the prefix notation output.
4. If the character is a closing parenthesis, then push it to the stack.
5. If the character is an opening parenthesis, pop the elements in the stack until we find the corresponding closing parenthesis.
6. If the character scanned is an operator



- If the operator has precedence greater than or equal to the top of the stack, push the operator to the stack.
- If the operator has precedence lesser than the top of the stack, pop the operator and output it to the prefix notation output and then check the above condition again with the new top of the stack.



7. After all the characters are scanned, reverse the prefix notation output.

Conversion of Infix to Prefix using Stack

K + L - M * N + (O^P) * W/U/V * T + Q

The Reverse expression would be:

Q + T * V/U/W *) P^O(+ N*M - L + K

To obtain the prefix expression, we have created a table that consists of three columns, i.e., input expression, stack, and prefix expression. When we encounter any symbol, we simply add it into the prefix expression. If we encounter the operator, we will push it into the stack.

Input expression	Stack	Prefix expression
Q		Q
+	+	Q
T	+	QT
*	+*	QT
V	+*	QTV
/	+*/	QTV
U	+*/	QTVU
/	+*//	QTVU
W	+*//	QTVUW
*	+*//*	QTVUW
)	+*//*)	QTVUW
P	+*//*)	QTVUWP
^	+*//*)^	QTVUWP
O	+*//*)^	QTVUWPO
(	+*//*	QTVUWPO^
+	++	QTVUWPO^*//*
N	++	QTVUWPO^*//*N
*	++*	QTVUWPO^*//*N
M	++*	QTVUWPO^*//*NM
-	++-	QTVUWPO^*//*NM*
L	++-	QTVUWPO^*//*NM*L
+	++-+	QTVUWPO^*//*NM*L
K	++-+	QTVUWPO^*//*NM*LK
		QTVUWPO^*//*NM*LK+-++

The above expression, i.e., QTVUWPO^*//*NM*LK+-++, is not a final expression. We need to reverse this expression to obtain the prefix expression. ++-+KL*MN*//*^OPWUVTQ

9. Write an Algorithm to find factorial of a given number n. Trace the same for n=4 by showing Recursive tree.

```
#include <stdio.h>
int factorial (int);
main()
{
        int num, fact;
        printf ("Enter a positive integer value: ");
        scanf ("%d", &num);
        fact = factorial (num);
        printf ("\n Factorial of %d =%5d\n", num, fact);
}

int factorial (int n)
{
        int result;
        if (n == 0)
                return (1);
        else
                result = n * factorial (n-1);

        return (result);
}
```

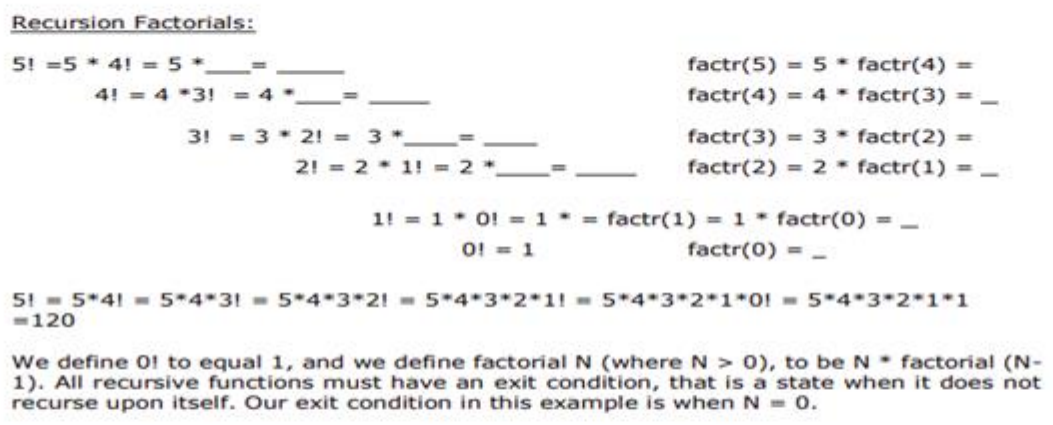
A non-recursive or iterative version for finding the factorial is as follows:

```
factorial (int n)
{
        int i, result = 1;
        if (n == 0)




                return (result);
        else
        {
                for (i=1; i<=n; i++)
                        result = result * i;
        }

        return (result);
}
```

The operation of the non-recursive version is clear as it uses a loop starting at 1 and ending at the target value and progressively multiplies each number by the moving product.

When a function calls itself, new local variables and parameters are allocated storage on the stack and the function code is executed with these new variables from the start. A recursive call does not make a new copy of the function. Only the arguments and variables are new. As each recursive call returns, the old local variables and parameters are removed from the stack and execution resumes at the point of the function call inside the function.

When writing recursive functions, you must have a exit condition somewhere to force the function to return without the recursive call being executed. If you do not have an exit condition, the recursive function will recurse forever until you run out of stack space and indicate error about lack of memory, or stack overflow.



Recursion Factorials:

```
5! =5 * 4! = 5 *____= ______                          factr(5) = 5 * factr(4) =
        4! = 4 *3!  = 4 *____= _____                  factr(4) = 4 * factr(3) = _
                3!  = 3 * 2! =  3 *____= ____         factr(3) = 3 * factr(2) =
                        2! = 2 * 1! = 2 *____= _____  factr(2) = 2 * factr(1) = _
                              1! = 1 * 0! = 1 * = factr(1) = 1 * factr(0) = _
                                     0! = 1           factr(0) = _
```

5! = 5*4! = 5*4*3! = 5*4*3*2! = 5*4*3*2*1! = 5*4*3*2*1*0! = 5*4*3*2*1*1 =120

We define 0! to equal 1, and we define factorial N (where N > 0), to be N * factorial (N-1). All recursive functions must have an exit condition, that is a state when it does not recurse upon itself. Our exit condition in this example is when N = 0.

10. Define Stack. Explain stack implementation using array.

Stack:

There are certain situations in computer science that one wants to restrict insertions and deletions so that they can take place only at the beginning or the end of the list, not in the middle. Two of such data structures that are useful are: Stack and Queue

Stack Definition

A stack is a list of elements in which an element may be inserted or deleted only at one end, called the top of the stack. Stacks are sometimes known as LIFO (last in, first out) lists.

As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed. The two basic operations associated with stacks are:

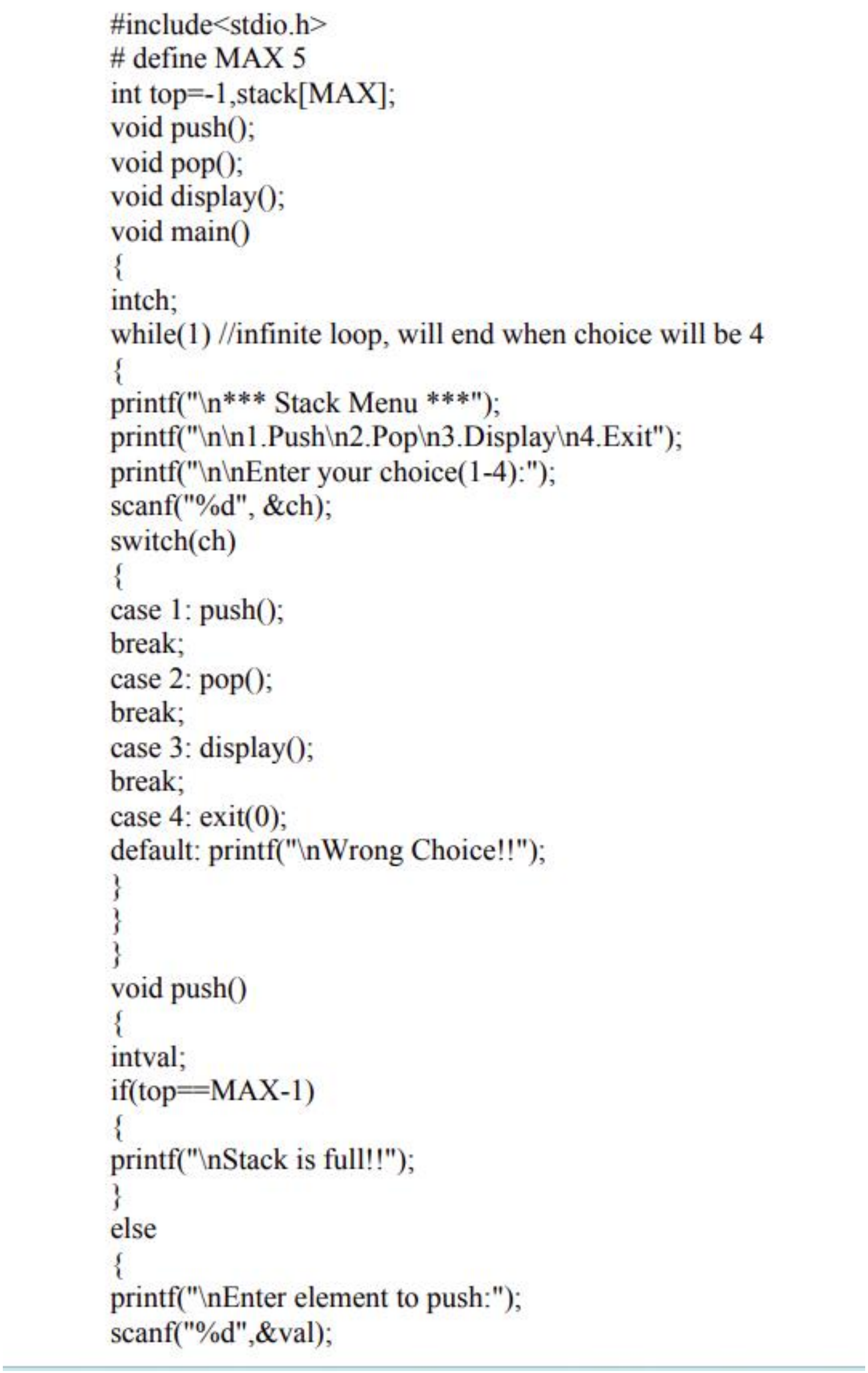
Push: is the term used to insert an element into a stack.

- Pop: is the term used to delete an element from a stack.

- "Push" is the term used to insert an element into a stack.

"Pop" is the term used to delete an element from the stack. All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Stack implementation using array.



```
#include<stdio.h>
# define MAX 5
int top=-1,stack[MAX];
void push();
void pop();
void display();
void main()
{
intch;
while(1) //infinite loop, will end when choice will be 4
{
printf("\n*** Stack Menu ***");
printf("\n\n1.Push\n2.Pop\n3.Display\n4.Exit");
printf("\n\nEnter your choice(1-4):");
scanf("%d", &ch);
switch(ch)
{
case 1: push();
break;
case 2: pop();
break;
case 3: display();
break;
case 4: exit(0);
default: printf("\nWrong Choice!!");
}
}
}
void push()
{
intval;
if(top==MAX-1)
{
printf("\nStack is full!!");
}
else
{
printf("\nEnter element to push:");
scanf("%d",&val);
top=top+1;
stack[top]=val;
}
}
void pop()
{
if(top==-1)
{
printf("\nStack is empty!!");
}
else
{ printf("\nDeleted element is %d",stack[top]);
top=top-1;
}
}
void display()
{ inti;
if(top==-1)
{ printf("\nStack is empty!!");
}
else
{ printf("\nStack is...\n");
for(i=top;i>=0;i--)
printf("%d\n",stack[i]);
}
}
```