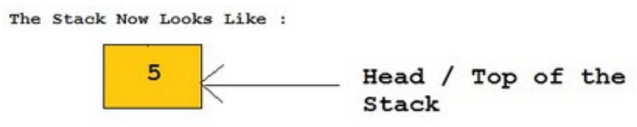
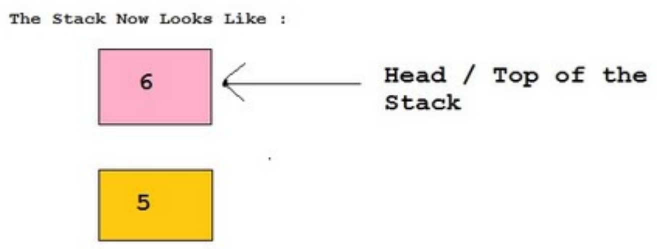
The Stack: Last In-First Out (LIFO)

The Empty Stack:
(null)

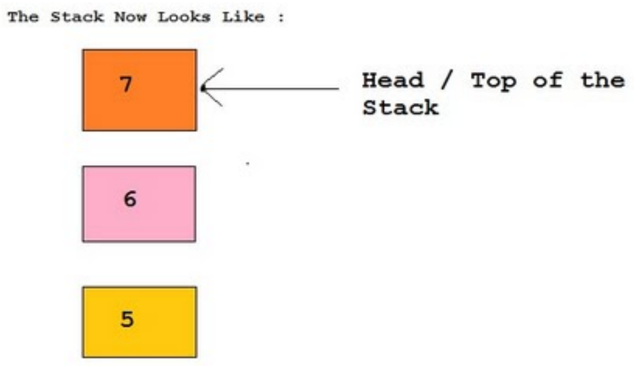
Push 5 onto the Stack:



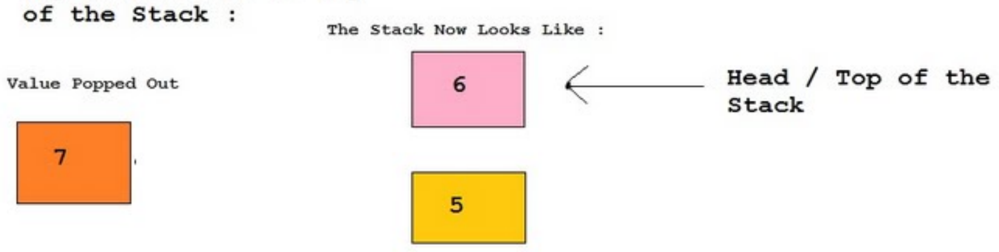
Push 6 onto the Stack:



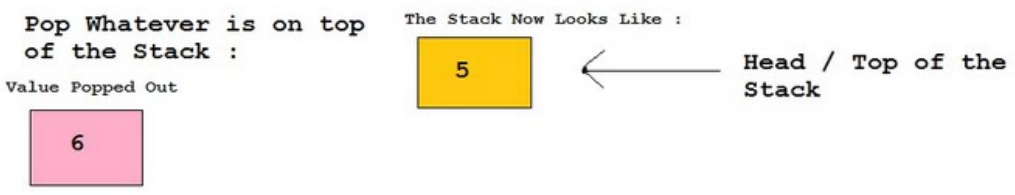
Push 7 onto the Stack:



Pop Whatever is on top of the Stack :



Pop Whatever is on top of the Stack :



Stack is a specialized data storage structure (Abstract data type). Unlike, arrays access of elements in a stack is restricted. It has two main functions push and pop. Insertion in a stack is done using push function and removal from a stack is done using pop function. Stack allows access to only the last element inserted hence, an item can be inserted or removed from the stack from one end called the top of the stack. It is therefore, also called Last-In-First-Out (LIFO) list. Stack has three properties: capacity stands for the maximum number of elements stack can hold, size stands for the current size of the stack and elements is the array of elements.

Algorithm:

Stack structure is defined with fields capacity, size and *elements (pointer to the array of elements).

Functions:

1. **createStack** function– This function takes the maximum number of elements (maxElements) the stack can hold as an argument, creates a stack according to it and returns a pointer to the stack. It initializes Stack S using malloc function and its properties.
    - elements = (int *)malloc(sizeof(int)*maxElements).
    - S->size = 0, current size of the stack S.
    - S->capacity = maxElements, maximum number of elements stack S can hold.
2. **push** function - This function takes the pointer to the top of the stack S and the item (element) to be inserted as arguments. Check for the emptiness of stack
    - If S->size is equal to S->capacity, we cannot push an element into S as there is no space for it.
    - Else, Push an element on the top of it and increase its size by one S->elements[S->size++] = element;
3. **pop** function - This function takes the pointer to the top of the stack S as an argument.
    - If S->size is equal to zero, then it is empty. So, we cannot pop.
    - Else, remove an element that is equivalent to reducing the size by 1.
4. **top** function – This function takes the pointer to the top of the stack S as an argument and returns the topmost element of the stack S. It first checks if the stack is empty (S->size is equal to zero). If it's not it returns the topmost element
    - S->elements[S->size-1]

Property:

1. Each function runs in *O*(1) time.
2. It has two basic implementations
    a. Array-based implementation – It's simple and efficient but the

maximum size of the stack is fixed.

b. Singly Linked List-based implementation – It’s complicated but there is no limit on the stack size, it is subjected to the available memory.

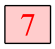
Example:

Create the stack S using createStack function, where S is the pointer to the structure Stack. The maximum number of elements (maxElements) = 5.

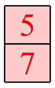
Initially

S->size = 0 and S->capacity = 5.

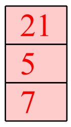
push(S,7): Since, S->size = S->capacity push 7 on the top of it and increase its size by one. Now, size = 1.



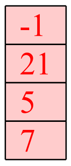
push(S,5): Since, S->size = S->capacity push 5 on the top of it and increase its size by one. Now, size = 2.



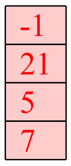
push(S,21): Since, S->size = S->capacity push 21 on the top of it and increase its size by one. Now, size = 3.



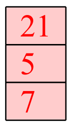
push(S,-1): Since, S->size = S->capacity push -1 on the top of it and increase its size by one. Now, size = 4.



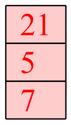
top(S): Since, S->size(=4) is not equal to zero. It returns the topmost element i.e. -1. Output = ‘-1’.



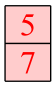
pop(S): Since, S->size(=4) is not equal to zero. It removes the topmost element i.e. -1 by simply reducing size of the stack S by 1. Now, size = 3.



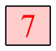
top(S): Since, S->size(=3) is not equal to zero. It returns the topmost element i.e. 21. Output = ‘21’.



pop(S): Since, S->size(=3) is not equal to zero. It removes the topmost element i.e. 21 by simply reducing size of the stack S by 1. Now, size = 2.



pop(S): Since, S->size(=2) is not equal to zero. It removes the topmost element i.e. 5 by simply reducing size of the stack S by 1. Now, size = 1.



pop(S): Since, S->size(=1) is not equal to zero. It removes the topmost element i.e. 7 by simply reducing size of the stack S by 1. Now, size = 0.

pop(S): Since, S->size(=0) is equal to zero. Output is ‘Stack is Empty’.

Stacks - C Program source code

```
#include<stdio.h>
#include<stdlib.h>
/* Stack has three properties. capacity stands for the maximum
number of elements stack can hold. Size stands for the current
size of the stack and elements is the array of elements */
typedef struct Stack
{
        int capacity;
        int size;
        int *elements;
}Stack;
```

```
/* crateStack function takes argument the maximum number of
elements the stack can hold, creates a stack according to it
and returns a pointer to the stack. */
Stack * createStack(int maxElements)
{
        /* Create a Stack */
        Stack *S;
        S = (Stack *)malloc(sizeof(Stack));
        /* Initialise its properties */
        S->elements = (int
*)malloc(sizeof(int)*maxElements);
        S->size = 0;
        S->capacity = maxElements;
        /* Return the pointer */
        return S;
}
void pop(Stack *S)
{
        /* If stack size is zero then it is empty. So we
cannot pop */
        if(S->size==0)
        {
                printf("Stack is Empty\n");
                return;
        }
        /* Removing an element is equivalent to reducing its
size by one */
        else
        {
                S->size--;
        }
        return;
}
int top(Stack *S)
{
        if(S->size==0)
        {
                printf("Stack is Empty\n");
                exit(0);
        }
        /* Return the topmost element */
        return S->elements[S->size-1];
}
void push(Stack *S,int element)
{
        /* If the stack is full, we cannot push an element
into it as there is no space for it.*/
```

```
        if(S->size == S->capacity)
        {
                printf("Stack is Full\n");
        }
        else
        {
                /* Push an element on the top of it and
increase its size by one*/
                S->elements[S->size++] = element;
        }
        return;
}
int main()
{
        Stack *S = createStack(5);
        push(S,7);
        push(S,5);
        push(S,21);
        push(S,-1);
        printf("Top element is %d\n",top(S));
        pop(S);
        printf("Top element is %d\n",top(S));
        pop(S);
        printf("Top element is %d\n",top(S));
        pop(S);
        printf("Top element is %d\n",top(S));

}
```

Applications of Stacks

There are a number of applications of stacks; three of them are discussed briefly in the preceding sections. Stack is internally used by compiler when we implement (or execute) any recursive function. If we want to implement a recursive function non-recursively, stack is programmed explicitly. Stack is also used to evaluate a mathematical expression and to check the parentheses in an expression.

Expression

One application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)

1. Infix notation
2. Prefix notation
3. Postfix notation

The infix notation is what we come across in our general mathematics, where the operator is written in-between the operands. For example: The expression to add two numbers A and B is written in infix notation as:

A+B

Note that the operator '+' is written in between the operands A and B. The prefix notation is a notation in which the operator(s) is written before the operands, it is also called polish notation in the honor of the polish mathematician Jan Lukasiewicz who developed this notation. The same expression when written in prefix notation looks like:

+AB

As the operator '+' is written before the operands A and B, this notation is called prefix.

In the *postfix* notation the operator(s) are written after the operands, so it is called the *postfix* notation, it is also known as *suffix* notation or reverse polish notation. The above expression if written in postfix expression looks like:

AB+

The *prefix* and *postfix* notations are not really as awkward to use as they might look. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction:

add(A , B)

Note that the operator **add** (name of the function) precedes the operands A and B. Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated.

Advantages of using postfix notation

Human beings are quite used to work with mathematical expressions in infix notation, which is rather complex. One has to remember a set of nontrivial rules while using this notation and it must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS, and associativity.

Using *infix* notation, one cannot tell the order in which operators should be applied. Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated first. But in a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the *stack*; likewise at the end of the whole operation the final result will be there in the stack.

Consider the infix expressions as '4+3*5' and '(4+3)*5'. The parentheses are not needed in the first but are necessary in the second expression. The postfix forms are:

Infix:	4+3*5	Postfix:	435*+
Infix:	(4+3)*5	Postfix:	43+5*

In case of not using the parenthesis in the infix form, you have to see the precedence rule before evaluating the expression. In the above example, if we want to add first then we have to use the parenthesis. In the postfix form, we do not need to use parenthesis. The position of operators and operands in the expression makes it clear in which order we have to do the multiplication and addition.

Notation Conversions

Let A + B * C be the given expression, which is an infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result. For example:

$$A + B * C = 4 + 3 * 7 = 7 * 7 = 49$$

The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like. The error in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus expression A + B * C can be interpreted as A + (B * C). Using this alternative method we can convey to the computer that multiplication has higher precedence over addition.

Operator precedence:

Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

Converting Infix to postfix expression:

The method of converting infix expression A + B * C to postfix form is:

A+B*C	Infix form
A + (B * C)	Parenthesized expression
A + (B C *)	Convert the multiplication
A (B C *) +	Convert the addition
ABC*+	Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Problem 1. Give postfix form for A+((B+C)+(D+E)*F) / G
Solution. Evaluation order is

A + (((BC +) + ((DE +) * F)) / G)
A + (((BC +) + (DE + F *) / G)
A + ((BC + DE + F * +) / G)
A+ (BC+DE+F*+G/)
ABC+DE+F *+G/+ Postfix Form

Problem 2. Give postfix form for (A+B)*C / D+E^A / B
Solution. Evaluation order is

((A+B) * C / D)+((E^A) / B)
((AB +) * C / D) + ((EA ^) / B)
((AB +) * C / D) + ((EA ^) B /)
((AB +) C * D /) + ((EA ^) B /)
(AB +) C * D / (EA ^) B / +
AB + C * D / EA ^ B / + Postfix Form

Algorithm

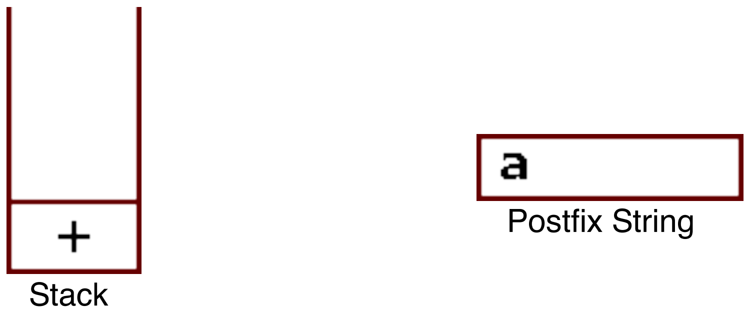
Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-). The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

1. Push “(” onto stack, and add “)” to the end of P.
2. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
3. If an operand is encountered, add it to Q.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator **X** is encountered, then:
    a. Repeatedly pop from stack and add P each operator (on the top of stack), which has the same precedence as, or higher precedence than **X**.
    b. Add **X** to stack.
6. If a right parenthesis is encountered, then:
    a. Repeatedly pop from stack and add to P (on the top of stack until a left parenthesis is encountered.
    b. Remove the left parenthesis. [Do not add the left parenthesis to P.]
7. Exit.

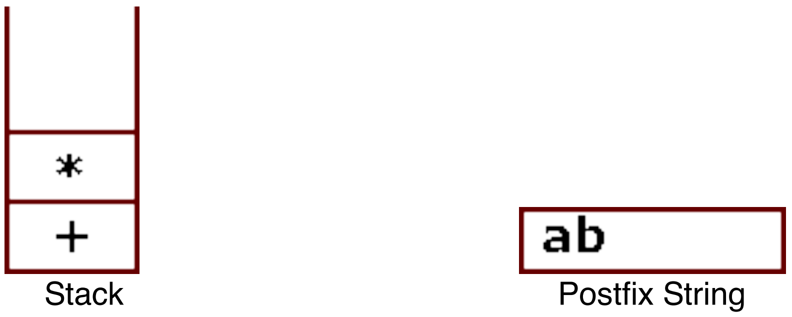
Example 1: Any expression in the standard form like "2*3-4/5" is an Infix (Inorder) expression. The Postfix (Postorder) form of the above expression is "23*45/-". Let us see how the above algorithm will be implemented using an example.

Infix String: a+b*c-d

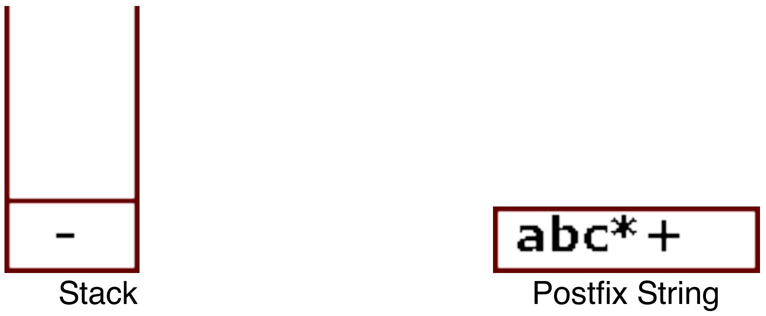
Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.



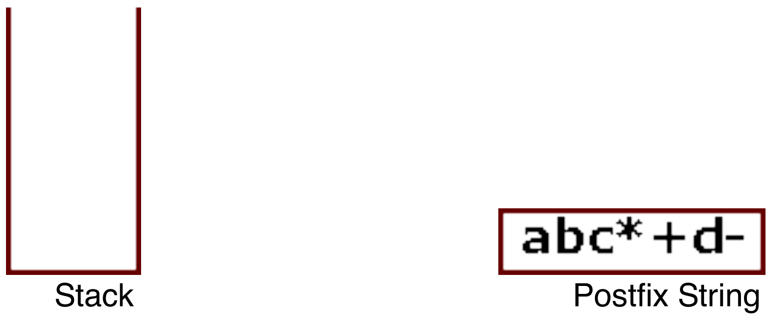
Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.



The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.



Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be :



End result:

Infix String: a+b*c-d

Postfix String: abc*+d-

Example 2: Consider the following arithmetic infix expression P

P=A+(B/C-(D*E^F)+G)*H

Following figure shows the character (operator, operand or parenthesis) scanned, status of the stack and postfix expression Q of the infix expression P.

Character scanned	*Stack*	*Postfix Expression (Q)*
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 * +

Evaluating Postfix Expression

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

```
1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of
   P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator X is encountered, then:
    a. Remove the two top elements of STACK, where A is the top
       element and B is the next-to-top element.
    b. Evaluate B X A.
    c. Place the result on to the STACK.
5. Result equal to the top element on STACK.
6. Exit.
```

SELF REVIEW QUESTIONS

1. Convert the following infix expression into postfix form (A + B)* (C + B)* (E-F).
2. Write procedure to convert infix to postfix expressions.
3. Write the prefix and postfix form for: A+B*(C-D) / (E-F).
4. Explain how a postfix expression is evaluated using stack with suitable example?