Stacks

Stack is an important data structure which stores its elements in an ordered manner.

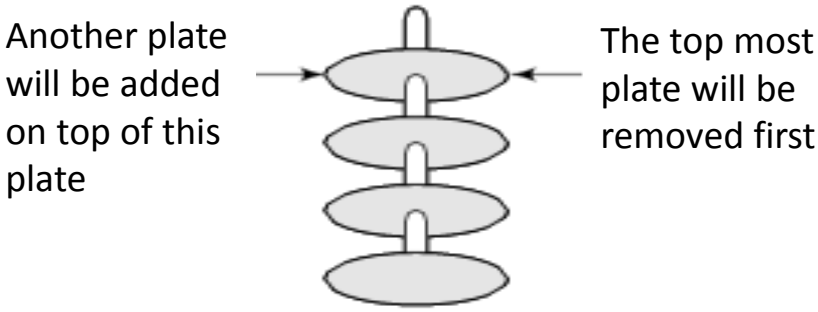


Fig. Stack of plates

The elements in a stack are added and removed only from one end, which is called the TOP. Hence, a stack is called a LIFO (Last-In-First-Out) data structure, as the element that was inserted last is the first one to be taken out.

Contd.

Now the question is where do we need stacks in computer science?

The answer is in function calls. Consider an example, where we are executing function A. In the course of its execution, function A calls another function B. Function B in turn calls another function C, which calls function D.

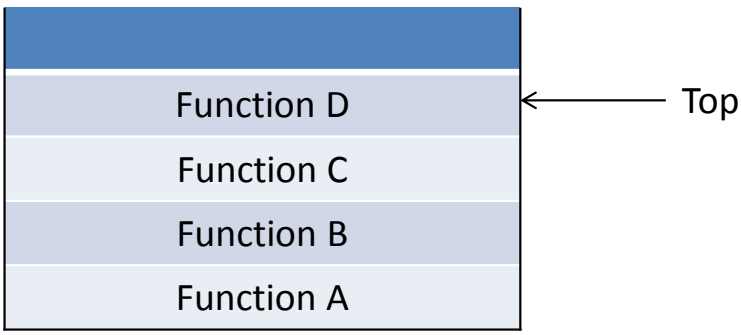


Fig. System stack in the case of function calls

Array representation of stacks

In the computer's memory, stacks can be represented as a linear array. Every stack has a variable called TOP associated with it, which is used to store the address of the topmost element of the stack. It is this position where the element will be added to or deleted from. There is another variable called MAX, which is used to store the maximum number of elements that the stack can hold. If TOP = NULL, then it indicates that the stack is empty and if TOP = MAX–1, then the stack is full.

A	**AB**	**ABC**	**ABCD**	**ABCDE**					
0	1	2	3	Top=4	5	6	7	8	9

Fig. Stack

Operation on Stack

A stack supports three basic operations: push, pop, and peek. The push operation adds an element to the top of the stack and the pop operation removes the element from the top of the stack. The peek operation returns the value of the topmost element of the stack.

PUSH operation:

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

Algorithm to insert(PUSH) an element in the Stack

1. IF TOP = MAX-1
   PRINT "OVERFLOW“ & Go to Step 4
2. SET TOP = TOP + 1
3. SET STACK[TOP] = VALUE
4. END

POP operation:

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

Algorithm to delete(POP) an element from the Stack

1.IF TOP = NULL

PRINT "UNDERFLOW & Goto Step 4

2. SET VAL = STACK[TOP]

3. SET TOP = TOP - 1

4. END

Peek Operation:

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

Algorithm for Peek operation:

1. IF TOP = NULL

   PRINT "STACK IS EMPTY“ & Goto Step 3

2. RETURN STACK[TOP]

3. END

LINKED REPRESENTATION OF STACKS

Array representation of stack

Adv: easy representation/implementation

Disadv: size can not be determined in advance

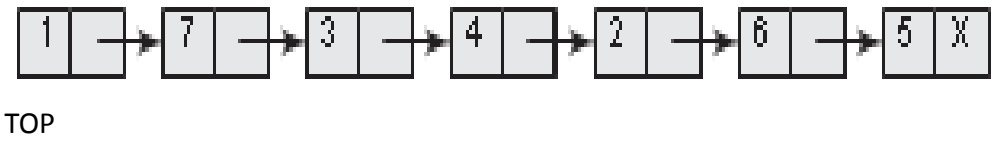
In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

Fig. Stack represented by Linked list

OPERATIONS ON A LINKED STACK

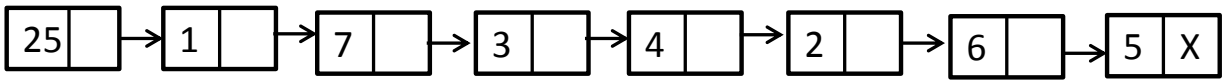
A linked stack supports all the three stack operations, that is, push, pop, and peek

Push Operation



TOP

Fig. Linked stack



TOP

Fig. Linked stack after PUSH operation

Algorithm to insert an element in a linked stack

1. Allocate memory for the new node and name it as NEW_NODE
2. SET NEW_NODE -> DATA = VAL
3. IF TOP = NULL
   SET NEW_NODE -> NEXT = NULL
   SET TOP = NEW_NODE
   ELSE
   SET NEW_NODE -> NEXT = TOP
   SET TOP = NEW_NODE
   [END OF IF]
4. END

Pop Operation

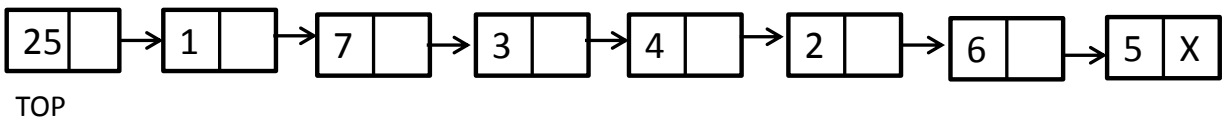


Fig. Linked stack

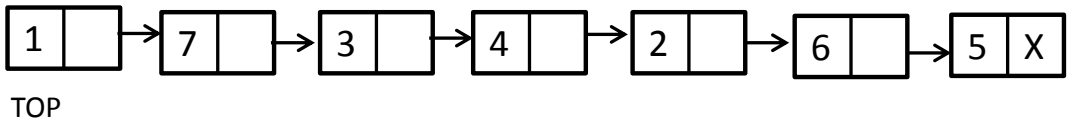


Fig. Linked stack after deletion of the topmost element

Algorithm to delete an element from a linked stack

1. IF TOP = NULL PRINT "UNDERFLOW“ & Goto Step 5
2. SET PTR = TOP
3. SET TOP = TOP -> NEXT
4. FREE PTR
5. END

Applications of stacks

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

Evaluation of Arithmetic Expressions

Infix, postfix, and prefix notations are three different but equivalent notations of writing algebraic expressions. But before learning about prefix and postfix notations, let us first see what an infix notation is. We all are familiar with the infix notation of writing algebraic expressions. While writing an arithmetic expression using infix notation, the operator is placed in between the operands. For example, A+B; here + operator is placed between the two operands A and B. Although it is easy for us to write expressions using infix notation, computers find it difficult to parse as the computer needs a lot of information to evaluate the expression. Information is needed about operator precedence and associativity rules, and brackets which override these rules. So, computers work more efficiently with expressions written using prefix and postfix notations.

Contd.

Postfix notation was developed by Jan Łukasiewicz who was a Polish logician, mathematician, and philosopher. His aim was to develop a parenthesis-free prefix notation (also known as Polish notation) and a postfix notation, which is better known as Reverse Polish Notation or RPN.

In postfix notation, as the name suggests, the operator is placed after the operands. For example, if an expression is written as A+B in infix notation, the same expression can be written as AB+ in postfix notation. The order of evaluation of a postfix expression is always from left to right. Even brackets cannot alter the order of evaluation.

The expression (A + B) * C can be written as:

[AB+]*C

AB+C* in the postfix notation

Contd.

A postfix operation does not even follow the rules of operator precedence. The operator which occurs first in the expression is operated first on the operands. For example, given a postfix notation AB+C*. While evaluation, addition will be performed prior to multiplication. Thus we see that in a postfix notation, operators are applied to the operands that are immediately left to them. In the example, AB+C*, + is applied on A and B, then * is applied on the result of addition and C.

Contd.

Although a prefix notation is also evaluated from left to right, the only difference between a postfix notation and a prefix notation is that in a prefix notation, the operator is placed before the operands. For example, if A+B is an expression in infix notation, then the corresponding expression in prefix notation is given by +AB.

Conversion of an Infix Expression into a Postfix Expression

Let I be an algebraic expression written in infix notation. I may contain parentheses, operands, and operators. For simplicity of the algorithm we will use only +, –, *, /, % operators. The precedence of these operators can be given as follows:

Higher priority *, /, %

Lower priority +, –

Convert the following infix expressions into postfix expressions.

Solution

(a) (A–B) * (C+D)

[AB–] * [CD+]

AB–CD+*

(b) (A + B) / (C + D) – (D * E)

[AB+] / [CD+] – [DE*]

[AB+CD+/] – [DE*]

AB+CD+/DE*–

Algorithm to convert an infix notation to postfix notation

The algorithm accepts an infix expression that may contain operators, operands, and parentheses. For simplicity, we assume that the infix operation contains only modulus (%), multiplication (*), division (/), addition (+), and subtraction (—) operators and that operators with same precedence are performed from left-to-right.

The algorithm uses a stack to temporarily hold operators. The postfix expression is obtained from left-to-right using the operands from the infix expression and the operators which are removed from the stack. The first step in this algorithm is to push a left parenthesis on the stack and to add a corresponding right parenthesis at the end of the infix expression. The algorithm is repeated until the stack is empty.

Contd.

1. Add ")" to the end of the infix expression
2. Push "(" on to the stack
3. Repeat until each character in the infix notation is scanned.

IF a "(" is encountered, push it on the stack

IF an operand (whether a digit or a character) is encountered, add it to the postfix expression.

IF a ")" is encountered, then

   a. Repeatedly pop from stack and add it to the postfix expression until a "(" is encountered.
   b. Discard the "(". That is, remove the "(" from stack and do not add it to the postfix expression

IF an operator is encountered, then

   a. Repeatedly pop from stack and add each operator (popped from the stack) to the postfix expression which has the same precedence or a higher precedence than operator
   b. Push the operator to the stack

[END OF IF]

4. Repeatedly pop from the stack and add it to the postfix expression until the stack is empty
5. EXIT

Contd.

Convert the following infix expression into postfix expression using the algorithm given

A – (B / C + (D % E * F) / G)* H ≡ (A – (B / C + (D % E * F) / G)* H)

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

(A+B)*C-(D-E)*(F+G) ≡ ((A+B)*C-(D-E)*(F+G))

Infix character scanned	Stack	Postfix expression
	(	
(	((	
A	((	A
+	((+	A
B	((+	AB
)	(	AB+
*	(*	AB+
C	(*	AB+C
-	(-	AB+C*
(	(-(	AB+C*
D	(-(	AB+C*D
-	(-(-	AB+C*D
E	(-(-	AB+C*DE
)	(-	AB+C*DE-
*	(-*	AB+C*DE-
(	(-*(	AB+C*DE-
F	(-*(	AB+C*DE-F
+	(-*(+	AB+C*DE-F
G	(-*(+	AB+C*DE-FG
)	(-*	AB+C*DE-FG+
)		AB+C*DE-FG+*-

Evaluation of a Postfix Expression

Given an algebraic expression written in infix notation, the computer first converts the expression into the equivalent postfix notation and then evaluates the postfix expression. Both these tasks—converting the infix notation into postfix notation and evaluating the postfix expression—make extensive use of stacks as the primary tool. Using stacks, any postfix expression can be evaluated very easily. Every character of the postfix expression is scanned from left to right. If the character encountered is an operand, it is pushed on to the stack. However, if an operator is encountered, then the top two values are popped from the stack and the operator is applied on these values. The result is then pushed on to the stack.

Algo: Evaluating Postfix expression

1. Add a “)” at the end of the postfix expression
2. Scan every character of the postfix expression and repeat Steps 3 and 4 until “)” is encountered
3. IF an operand is encountered, push it on the stack
   IF an operator (say θ) is encountered then
      a) pop the top two elements from the stack as A and B
      b) evaluate B θ A, where A is the topmost element and B is the element below A
      c) push the result of evaluation on the stack
   [END OF IF]
4. Set result equal to the topmost element of the stack
5. Exit.

Contd.

Consider the infix expression given as

9 – ((3 * 4) + 8) / 4. Evaluate the expression.

The infix expression 9 – ((3 * 4) + 8) / 4 can be written as 9 3 4 * 8 + 4 / – using postfix notation.

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

Fig. Evaluation of postfix notation

Conversion of an Infix Expression into a Prefix Expression

Algorithm to convert an infix expression into prefix expression

a)

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

Contd.

b)

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

Contd.

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

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

(L – K / A) * (C / B – A)

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

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

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

= [LKA/–] * [CB/A–]

= L K A / – C B / A – *

3. Reverse the postfix expression to get the prefix expression

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

Evaluation of a Prefix Expression

Algorithm for evaluation of a prefix expression

1. Accept the prefix expression
2. Repeat until all the characters in the prefix expression have been scanned

    a) scan the prefix expression from right one character at a time

    b) if the scanned character is an operand, push it on the operand stack

    c) if the scanned character is an operator, then

        i) pop two values from the operand stack

        ii) apply the operator on the popped operands

        iii) push the result on the operand stack

3. Exit.

Contd.

For example, consider the prefix expression

+ – 2 7 * 8 / 4 12

Let us now apply the algorithm to evaluate this expression.

Character scanned	Operand stack
12	12
4	12, 4
/	3
8	3, 8
*	24
7	24, 7
2	24, 7, 2
-	24, 5
+	29