STACK

A *stack* is defined as a restricted list where all insertions and deletions are made only at one end, the *top*.

Each stack abstract data type (ADT) has a data member, commonly named as *top*, which points to the topmost element in the stack.

There are two basic operations push and pop that can be performed on a stack; insertion of an element in the stack is called push and deletion of an element from the stack is called *pop*.

In stacks, we cannot access data elements from any intermediate positions other than the top position.

Basic Operations of Stack

A stack is an object or more specifically an abstract data structure(ADT) that allows the following operations:

- Push: Add an element to the top of a stack
- Pop: Remove an element from the top of a stack
- IsEmpty: Check if the stack is empty
- IsFull: Check if the stack is full
- Peek: Get the value of the top element without removing it

Working of Stack Data Structure

The operations work as follows:

- A pointer called TOP is used to keep track of the top element in the stack.

- When initializing the stack, we set its value to -1 so that we can check if the stack is empty by comparing TOP == -1.

- On pushing an element, we increase the value of TOP and place the new element in the position pointed to by TOP.
- On popping an element, we return the element pointed to by TOP and reduce its value.
- Before pushing, we check if the stack is already full
- Before popping, we check if the stack is already empty

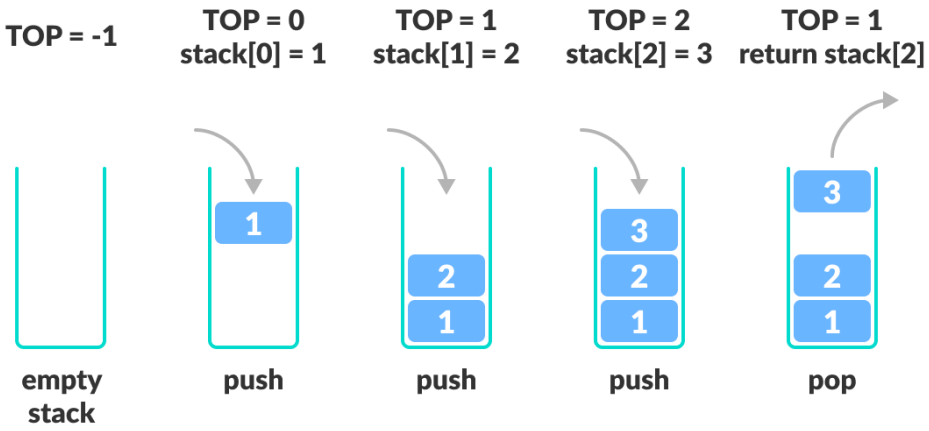


Fig. working of stack

Elements are taken out in the reverse order of the insertion sequence. So a stack is often called *last in first out* (*LIFO*) or *first in last out* (*FILO*) data structure.

Array Implementation of stack

Algorithm push operation on stack

push(Stack, Top)

1. If Top = MAX-1
   Print “overflow: stack is full!”
   END If
2. Read Item

3. Set Top = Top+1
4. Set Stack[Top] = item
5. END

Algorithm pop operation on stack

pop(Stack,Top)

1. If Top = -1
   Print “Underflow: stack is empty!”
   End If
2. Set item = Stack[Top]
3. Set Top = Top-1
4. Print “The popped element is: ”,item
5. END

C++ code to implement stack using array.

```
#include<iostream>
#define SIZE 5
using namespace std;

class Stack{
   private:
   int stck[SIZE];
   int top;
   public:
   Stack();
   void push(int);
   void pop();
   int isEmpty();
   int isFull();
   void peek();
};

Stack::Stack()
{
   this->top = -1;
}
int Stack:: isFull()
{
   if(this->top==SIZE-1)
   return 1;
   else
   return 0;
}
int Stack::isEmpty()
{
   if(this->top==-1)
   return 1;
   else
   return 0;
```

```
}
void Stack::push(int item)
{
   if(!this->isFull())
   {
      this->stck[++this->top]=item;
      cout <<"Successfully push in to the stack"<<endl;
   }
   else
   {
      cout <<"Overflow:stack is full!"<<endl;
   }

}
void Stack::pop()
{
   if(!this->isEmpty())
   {
      cout<<stck[this->top--]<<" is deleted from the stack"<<endl;
   }
   else
   {
      cout<<"Underflow: stack is empty!"<<endl;
   }

}
void Stack::peek()
{
   if(!this->isEmpty())
   {
      cout<<stck[this->top]<<" is at top of the stack"<<endl;
   }
   else
   {
      cout<<"stack is empty!"<<endl;
   }

}
```

```
int main()
{
   Stack s;
   char ch;
   int item;
   while(1)
   {
      cout<<"Do you want to add element into the stack(y/n): ";
      cin>>ch;
      if(ch=='n') break;
      else
      {
         cout<<"Enter the element to be pushed: ";
         cin>>item;
         s.push(item);
      }

   }
   s.peek();
   s.pop();
   s.peek();
}
```

Applications of STACK

The stack data structure is used in a wide range of applications. A few of them are the following:

1. Converting infix expression to postfix and prefix expressions 2. Evaluating the postfix expression

3. Checking well-formed (nested) parenthesis

4. Reversing a string

5. Processing function calls

6. Parsing (analyse the structure) of computer programs

7. Simulating recursion

8. In computations such as decimal to binary conversion

9. In backtracking algorithms (often used in optimizations and in games)

EXPRESSION EVALUATION AND CONVERSION

The most frequent application of stacks is in the evaluation of arithmetic expressions. An arithmetic expression is made of operands, operators, and delimiters. When high- level programming languages came into existence, one of the major difficulties faced by computer scientists was to generate machine language instructions that could properly evaluate any arithmetic expression.

A complex assignment statement such as X=(A/B+CxD -FxG/Q)

might have several meanings, and even if the meanings were uniquely defined, it is still difficult to generate a correct and reasonable instruction sequence. Fortunately, the solution we have today is both elegant and simple. Till date, this conversion is considered as one of the major aspects of compiler writing.

Let us see the difficulties in understanding the meaning of expressions. The first prob- lem in understanding the meaning of an expression is to decide the order in which the operations are to be carried out. This demands that every language must uniquely define such an order.

For instance, consider the following expression: X = a/b x c - d

Let a = 1, b = 2, c = 3, and d = 4.

One of the meanings that can be drawn from this expression could be

X = (1/2) x (3 - 4) = -1/2

Another way to evaluate the same expression could be X = (1/(2 x3)) - 4 = -23/6

To avoid more than one meaning being drawn out of an expression, we have to specify the order of operation by using parentheses. For instance,

X = (a/b) x (c - d)

To fix the order of evaluation, assign each operator a priority. Even though we write the expression in parentheses, we still query whether to evaluate (A/B) first or to evaluate (C - D) first. Once the priorities are assigned, then within any pairs of parentheses the operators with the highest priority are to be evaluated first. While evaluating an expres- sion, the following operation precedence is usually used:

The following operators are written in descending order of their precedence:

1. Exponentiation (^), Unary (+), Unary (-), and not (~)

2. Multiplication (x) and division (/)

3. Addition (+) and subtraction (-)

4. Relational operators <, <= , =, !=, >,>=

5. Logical AND

6. Logical OR

Operator and their priorities

Arithmetic, boolean, and relational operators	Priority
^, Unary +, Unary – , ~	1
×, /	2
+, –	3
<, ≤, =, ≠, ≥, >	4
AND	5
OR	6

Note that all the relational operators have the same priority. Exponentiation (^) and unary operators (+, -, and ~) have the highest priority. When there are two adjacent operators with the same priority, again the question arises as to which one to evaluate first. For example, the expression, A + B - C can be understood in two ways—(A + B) - C or A + (B - C).

This needs a decision on whether to evaluate the expression from right to left or left to right. Expressions such as A + B - C and A x B/C are to be evaluated from left to right. However, the expression A ^ B ^ C is to be evaluated from right to left as A ^ (B ^ C). For example, to compute 2 ^ 3 ^ 2, we need to represent and evaluate it as 2 ^ (3 ^ 2). When evaluated from left to right, the expression may be evaluated as ((2 ^ 3) ^ 2), which is wrong!

Hence, the operators need to decide on a rule for proceeding from left to right for all expressions except the operator exponential. This order of evaluation, from left to right or right to left, is called associativity. Exponentiation is right associative and all other operators are left associative. When we write a parenthesized expression, these rules can be overridden. In the parenthesized expressions, the innermost parenthesized expression is evaluated first.

Let us consider the expression

X = A/B ^ C + D x E - A x C

By using priorities and associativity rules, the expression X is rewritten as

X = A/(B ^ C) + (D x E) - (A x C)

For example, let X be an infix expression as = ((2 + 3) x 4)/2

We manually evaluate the innermost expression first as ((5) x 4)/2, followed by the next parenthesized inner expression (20)/2, which produces the result 10.

Still the question remains as to how a compiler can accept such an expression and produce the correct code. **The solution is to rework on the expression to a form called the postfix notation.**

Polish notation and Expression conversion

The Polish Mathematician Han Lukasiewicz suggested a notation called Polish notation, which gives two alternatives to represent an arithmetic expression, namely the postfix and prefix notations. The fundamental property of Polish notation is that the order in which the operations are to be performed is determined by the positions of the operators and operands in the expression. Hence, the advantage is that parentheses is not required while writing expressions in Polish notation. The conventional way of writing the expres- sion is called infix, because the binary operators occur between the operands, and unary operators precede their operand. For example, the expression ((A + B) x C)/D is an infix expression. In postfix notation, the operator is written after its operands, whereas in prefix notation, the operator precedes its operands. Table below shows one sample expression in all three notations.

Infix	Prefix	Postfix
(operand)(operator)(operand)	(operator)(operand)(operand)	(operand)(operand)(operator)
$(A + B) \times C$	$\times + ABC$	$AB + C\times$

Need for Prefix and Postfix Expressions

We just studied that evaluation of an infix expression using a computer needs proper code generation by the compiler without any ambiguity and is difficult because of various aspects such as the operator's priority and associativity. This problem can be overcome by writing or converting the infix expression to an alternate notation such as the prefix or the postfix. The postfix and prefix expressions possess many advantages as follows:

1. The need for parenthesis as in an infix expression is overcome in postfix and prefix notations.

2. The priority of operators is no longer relevant.

3. The order of evaluation depends on the position of the operator but not on priority and associativity.

4. The expression evaluation process is much simpler than attempting a direct evaluation from the infix notation.

Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push "(" onto Stack, and add ")" to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:

   I. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.

   II. Add operator to Stack.

[End of If]

6. If a right parenthesis is encountered ,then:
   I. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
   II. Remove the left Parenthesis.

[End of If]

END.

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

S.N.	SCANNED	STACK	POSTFIX	DESCRIPTION
1.		(		**START**
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	'*' has higher precedence than -
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	

12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from stack when right parentheses are encountered.
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from stack when right parentheses are encountered.
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	**EMPTY**	ABC*DEF^/G*-H*+	**END**

Q. Convert the following infix expression to its postfix form:

A ^ B * C - C + D/A/(E + F)

Follow the following steps:

S.N.	SCANNED	STACK	POSTFIX	DESCRIPTION

1.		(		START
2.	A	(	A	
3.	^	(^	A	
4.	B	(^	AB	
5.	*	(*	AB^	‘^’ has higher precedence
6.	C	(*	AB^C	
7.	-	(-	AB^C*	‘*’ has higher precedence
8.	C	(-	AB^C*C	
9.	+	(+	AB^C*C-	
10.	D	(+	AB^C*C-D	
11.	/	(+/	AB^C*C-D	
12.	A	(+/	AB^C*C-DA	
13.	/	(+/	AB^C*C-DA/	
14.	(	(+/(	AB^C*C-DA/	
15.	E	(+/(	AB^C*C-DA/E	
16.	+	(+/(+	AB^C*C-DA/E	
17.	F	(+/(+	AB^C*C-DA/EF	
18.	)	(+/	AB^C*C-DA/EF+	
19.	)	EMPTY	AB^C*C-DA/EF+/+	END

Q.Consider the following arithmetic infix expression P into postfix expression

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

S.N.	SCANNED	STACK	POSTFIX	DESCRIPTION
1.		(		START
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	/	(+(/	AB	
7.	C	(+(/	ABC	
8.	-	(+(-	ABC/	
9.	(	(+(-(	ABC/	
10.	D	(+(-(	ABC/D	
11.	*	(+(-(*	ABC/D	
12.	E	(+(-(*	ABC/DE	

13.	^	(+(-(*^	ABC/DE	
14.	F	(+(-(*^	ABC/DEF	
15.	)	(+(-	ABC/DEF^*	
16.	+	(+(+	ABC/DEF^*-	
17.	G	(+(+	ABC/DEF^*-G	
18.	)	(+	ABC/DEF^*-G+	
19.	*	(+*	ABC/DEF^*-G+	
20.	H	(+*	ABC/DEF^*-G+H	
21.	)	EMPTY	ABC/DEF^*-G+H*+	EMPTY

Evaluation of Postfix Expression

Algorithm for Evaluation of Postfix Expression

1. Create an empty stack and start scanning the postfix expression from left to right.
2. If the element is an operand, push it into the stack.
3. If the element is an operator O, pop twice and get A and B respectively. Calculate BOA and push it back to the stack.
4. When the expression is ended, the value in the stack is the final answer.

Q.Evaluate postfix expression: **456*+**

S.N.	**SCANNED**	**STACK**	**OPERATION**
1.	4	4	
2.	5	4,5	
3.	6	4,5,6	

4.	*	4,30	**5*6 = 30**
5	+	34	**4+30 = 34**
6.		EMPTY	**RESULT = 34**

Q. evaluate postfix expression: 2 10+9 6 -/

S.N.	SCANNED	STACK	OPS.
1.	2	2	
2.	10	2,10	
3.	+	12	2+10 = 12
4.	9	12,9	
5.	6	12,9,6	
6.	-	12,3	9-6=3
7.	/	4	12/3=4
8.		EMPTY	RESULT=4

Q. Evaluate POSTFIX EXPRESSION: 5 3 + 8 2 - *

S.N.	SCANNED	STACK	OPS.

1.	5	5	
2.	3	5,3	
3.	+	8	5+3=8
4.	8	8,8	
5.	2	8,8,2	
6.	-	8,6	8-2=6
7.	*	48	8*6 = 48
8.		EMPTY	Result=48

EVALUATION OF PREFIX EXPRESSION

EVALUATE_PREFIX(STRING)

Step 1: Put a pointer P at the end of the expression

Step 2: If character at P is an operand push it to Stack

Step 3: If the character at P is an operator pop two

elements from the Stack. Operate on these elements

according to the operator, and push the result

back to the Stack

Step 4: Decrement P by 1 and go to Step 2 as long as there

are characters left to be scanned in the expression.

Step 5: The Result is stored at the top of the Stack,

return it

Step 6: End

Q. Evaluate prefix expression:

Expression: +9*26

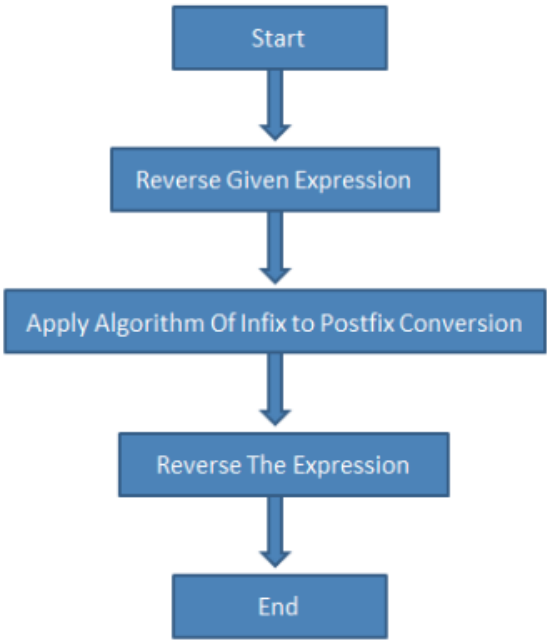
S.N.	SCANNED	STACK	OPS.
1.	6	6	
2.	2	6,2	
3.	*	12	6*2=12
4.	9	12,9	
5.	+	21	12+9=21
6		EMPTY	RESULT=21

Infix to Prefix Conversion Using Stack

Algorithm for infix to prefix

Step 1. Push “)” onto STACK, and add “(“ to end of the A
Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
Step 3. If an operand is encountered add it to B
Step 4. If a right parenthesis is encountered push it onto STACK
Step 5. If an operator is encountered then:
a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
b. Add operator to STACK
Step 6. If left parenthesis is encontered then
a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)
b. Remove the left parenthesis
Step 7. Exit

Flowchart


Start
Reverse Given Expression
Apply Algorithm Of Infix to Postfix Conversion
Reverse The Expression
End

Infix to prefix conversion

Expression = (A+B^C)*D+E^5

Step 1. Reverse the infix expression.

5^E+D*)C^B+A(

Step 2. Make Every '(' as ')' and every ')' as '('

5^E+D*(C^B+A)

Step 3. Convert expression to postfix form.

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

Expression	Stack	Output	Comment
5^E+D*(C^B+A)	Empty	-	Initial
^E+D*(C^B+A)	Empty	5	Print
E+D*(C^B+A)	^	5	Push
+D*(C^B+A)	^	5E	Push
D*(C^B+A)	+	5E^	Pop And Push
*(C^B+A)	+	5E^D	Print
(C^B+A)	+*	5E^D	Push
C^B+A)	+*(	5E^D	Push
^B+A)	+*(	5E^DC	Print
B+A)	+*(^	5E^DC	Push
+A)	+*(^	5E^DCB	Print
A)	+*(+	5E^DCB^	Pop And Push
)	+*(+	5E^DCB^A	Print
End	+*	5E^DCB^A+	Pop Until '('
End	Empty	5E^DCB^A+*+	Pop Every element

Step 4: reverse the expression

+*+A^BCD^E5 (**Result**)