Stack

Chapter 9

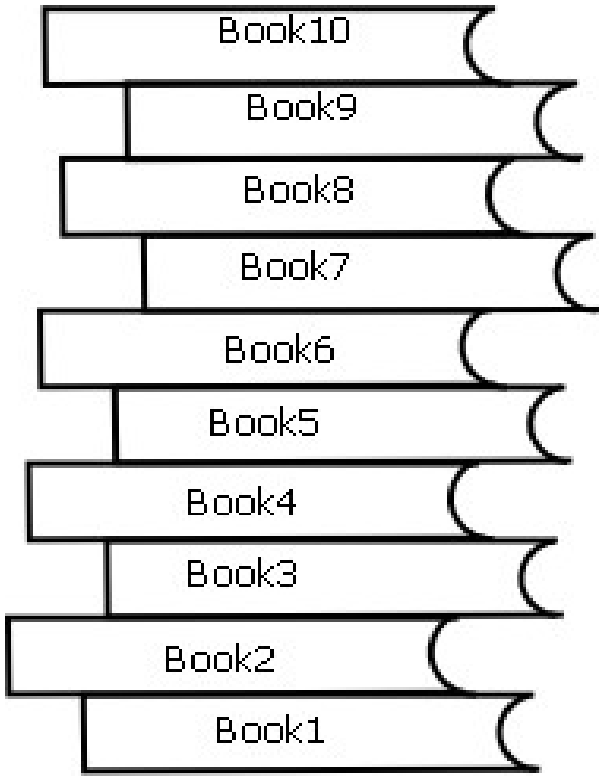
Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Outline

- Introduction
- Operations on stack
- Application of Stack
- Evaluating Postfix Expression
- Converting Infix Expression to Prefix
- Prefix to Infix Conversion
- Summary

Introduction

- Insertion and deletion of items are made through one end only.
- A book can be kept only on top of another book.
- A book can be removed only from the top



Contd…

- Because of the fact that only the elements added in the last can be removed first, stack is also known as LIFO structure.
- Further, because of the push operation, it is also called as pushdown list.
- All recursive programming languages use an implicit system-stack to allocate space for local variables of routines as they are called.

Uses of stacks in Computing

- **Calculators:**
  - To convert infix expression to postfix *to make evaluation easier and evaluating the postfix* expression.
- **Compilers:**
  - Convert infix expression to postfix to make translation to machine language easier.
- **Word processors, editors:**
  - To check expression or string of text to ensure that brackets are balanced
  - To implement undo operation

Program Runtime

- Extensively used in function call and return.
  - Runtime system uses a stack to keep track of function calls and return, which variables are currently accessible, etc.
- It is also key to recursion.
- Representing a function call in 'C' is like add (a, b), which happens to be a prefix notation.
- Postfix is universally accepted notation for designing ALU and CPU.
  - Expressions entered into the computer is first converted into postfix notation, stored in stack & then calculated.

The Stack Abstract Data Type

Easiest way to implement this ADT is by using 1-D array.

- The first or bottom element is stored at stack [0] position.
- The i^{th} element Is stored at stack[i - 1] position
- Initially the a variable top is initialized to -1.
- Elements are inserted and deleted using top.

Stack ADT

- ADT is independent of any implementation and exactly the same in every programming language.
- Thus understanding ADT shall enable us to write these operations in any programming language like C, C++ or java.
- **IsEmpty (stack s): Boolean**
  - Precondition: s is not null.
  - The primitive isempty is needed to avoid calling pop operation on an empty stack because this will result in an error.

Contd…

- **create( ): Stack**
  - Precondition: none
  - Postconditions: if s = create( ) then:
    - s is newly created stack
    - isEmpty(s) returns true
- **Top(stack s): Object**
  - Preconditions: isEmpty(s) returns false
  - Postcondition: positions the index at the top of stack

Contd…

- **Push(Stack s, Object o): Void**
  - Preconditions: s is not full
  - Postconditions:
    - top(stack) returns ”0”
    - isEmpty(s) returns false
- While implementing this stack, the push operation returns an error when the stack is ``full'' and thus no more items can be added.

Contd…

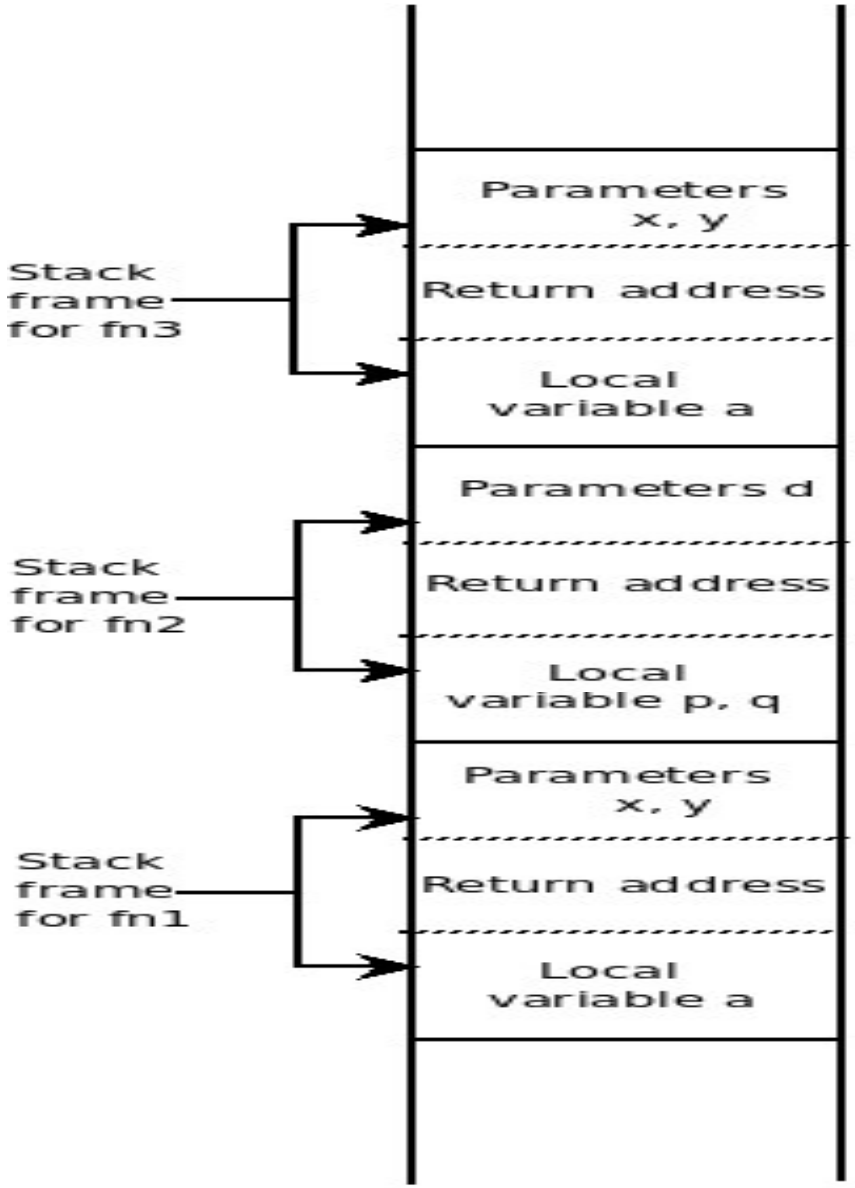
- **Pop(Stack s) : Void**
  - Preconditions: isEmpty(s) returns false
  - Postcondition: Returns the item from the top of stack (TOS)

- **Destroy(stack s) : void**
  - Preconditions: s is not null
  - Postconditions: s is null

Stack in function Call and Return

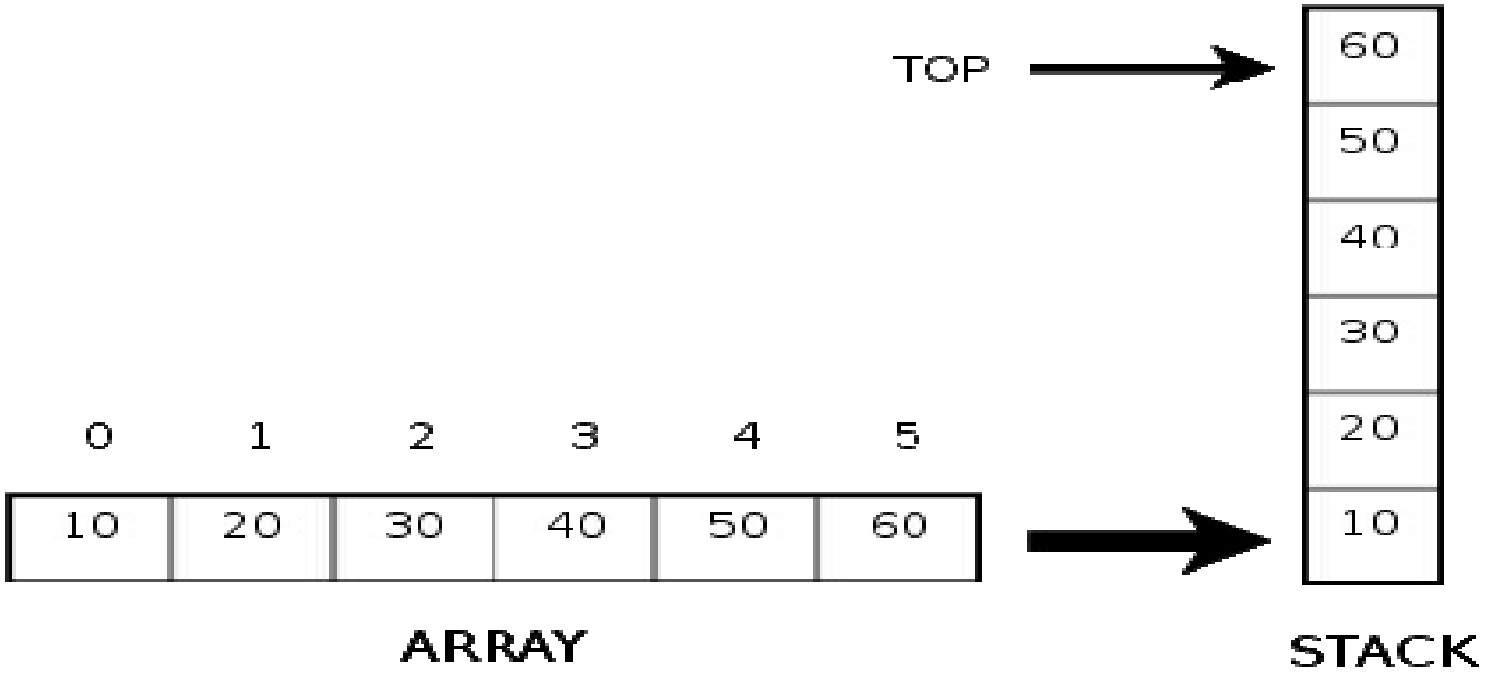
- **Constituents of Stack frame:**
  - Function argument
  - Return address
  - Local variables

```
function fn1(int x, int y)
    {
        int a;
        return fn2(a);
    }
function fn2(int d)
    {
        int p,q;
        return fn1(p,q);
    }
function fn3(int x, int y)
    {
        int q;
        q = x + y;
        return q;
    }
```



Introduction

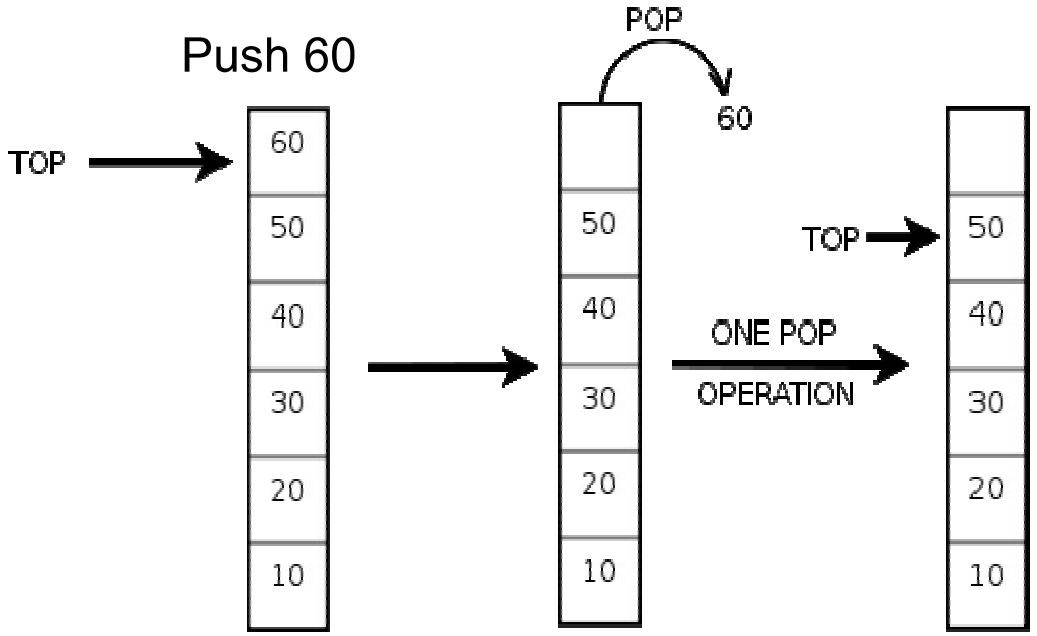
- A simple stack of integers.



Operations on Stack

Push and Pop

- Whenever an element is added to the stack, it is added to the top of the stack.



Data Structures: A Programming Approach with C, PHI
Dharmender Singh Kushwaha & A.K.Misra

Push Operation

Algorithm

push(stack[max _ size], element)

let stack[max _ size] be an array for implementing stack

[check for stack overflow ?]

if top = max_size – 1

then print “stack over flow”

set top = top + 1

set stack [top] = element

exit

POP Operation

POP: An element is deleted from the top of the stack.

```
pop(stack [max_size], element)
Check for stack under flow
      if top < 0
      then print ”stack under flow ” and exit
   else[remove the top element]
         set element = stack[top]
         decrement the stack top
         set top = top – 1
   return the deleted element
exit
```

Application of Stacks

Computation of Expression

- There are three types of notation for representing the expressions:
  - Infix notation (a+b)
  - Prefix notation (+ab)
  - Postfix notation (ab+)

Infix notation: A+B

- In this inorder notation, where operators are written in between the operands.
- This is the usual way of writing the expression.
- An expression such as A*(B+C) implies that first add B and C together, then multiply the result by A to give the final answer.
- Infix notation needs precedence of the operators and we sometimes use bracket () to override these rules.

Postfix notation: AB +

- This is also known as Reverse Polish notation.
- Operators are written after their operands thus called post order.
- The infix expression of A*(B+C) is equivalent to A B C + *.
- The order to evaluation of operators is always from left to right, and brackets cannot be used to change this order in postfix notation.
- In the given expression, “+” is to the left of the “*”, hence the addition must be performed before the multiplication.

Prefix Notation: + AB

- This is also known as Polish notation.
- Operators are written before their operands thus called preorder.
- The expression given above are equivalent to * A + B C.
- As in the case of postfix, operators are evaluated left to right and parenthesis (brackets) are not required.
- Operators act on the two nearest values on the right.
- The same expression can be supplemented with brackets for explicit representation as (* A (+ B C)).

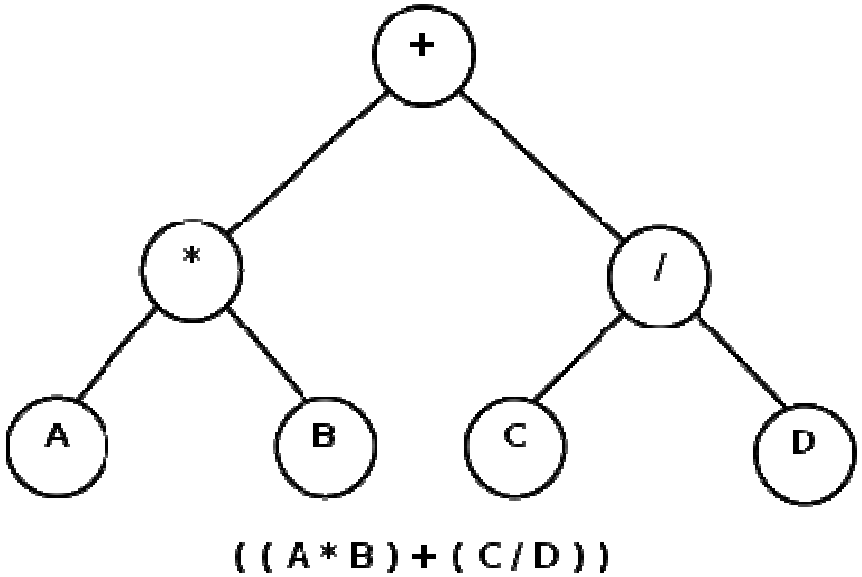
Examples

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the two result
A *(B + C) /D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A *(B + C /D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

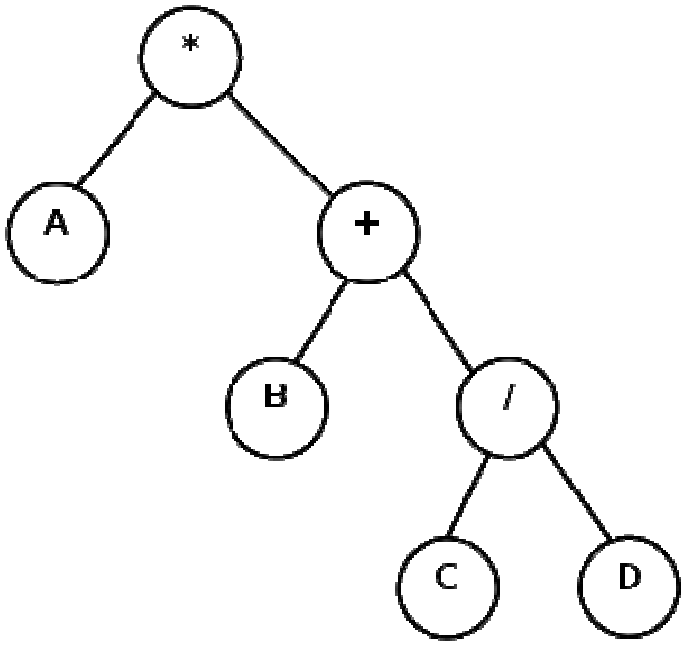
Parse trees

- An expression can be shown by a parse tree.

Example:



((A * B) + (C / D))



(A * (B + (C / D)))

Example

Convert the following expression into postfix notation.

a * b + c / d

- During left to right evaluation of expression, we find that the two operators * and / that have the same precedence that appear in the expression.
- So, the operator that is encountered first from left to right is first parenthesized.

(a * b) + c /d

(a b *) + c /d

Contd…

- For convenience, replace (a b *) with P, to obtain

  (P) + (c / d)

  P + (c d /)

- Representing (cd/) by Q, we have

  P + Q

  P Q +

- Now replace P and Q with the sub expressions to obtain the postfix notation a b * c d / + for the given expression.

Convert the Expression into Prefix notation

(a + b) * c /d + e ^ f

- (+ a b) * c / d + (e ^ f)
- Now replacing (+ a b) with P for convenience, we get

  (P * c) / d + (^ e f)
- Now replacing (^ef) with Q, we get

  (P * c) / d + Q

  (* P c) / d + Q

Contd…

- Replacing (* P c) with R, we have

  (R/d) + Q

  (/ R d) + Q

- Now replacing (/ R d) with S, we have

  S + Q

- Converting this to postfix we have

  + S Q

Contd…

- Having represented the expression in the simplest form, we can expand the resultant expression to obtain the result.
- Insert the sub expressions starting from the left.

  + (/ R d) Q

  + (/(* P c) d) Q

Contd…

+ (/ (* (+ a b) d) Q

+ (/ (* (+ a b) d) (^ef)

- Now we can remove the entire parenthesis to obtain the resultant postfix expression

+ / * + a b d ^ e f

Reverse Polish Notation

- Reverse Polish notation (RPN) a postfix notation was introduced in 1920 by the Polish mathematician Jan Lukasiewcz.
- Interpreters of reverse polish notation are often stack based.
- The operands are pushed onto a stack.
- When we encounter the operator, operands are popped from a stack, operation is performed its result pushed back again onto the stacks.
- Thus RPN has the advantage of being easy to implement and the procedure is very fast.

Algorithm for Evaluating Postfix (RPN) Expression

- A postfix expression can be evaluated using stack.
- Read the tokens of the expression from left to right and while input tokens left:
  - Read the next token from input.
  - If the token is a operand
    - Push it onto the stack.
  - If the token is a operator (It is known that the operator requires **n** operand)
    - If there are fewer than **n** operand on the stack

Algorithm for Evaluating Postfix Expression

- **Print (Error)** (The user has not input sufficient operand in the expression)

- Else, Pop the top n operands from the stack
  - Evaluate the expression, with the operand
  - Push the results, back onto the stack.
- If there is only one value in the stack, this is the result of the Evaluation of the Postfix Expression.

Example

- Evaluate expression 5 + ((1 + 2) * 4) (in RPN as 5 1 2 + 4) * +)

Input Tokens	Operation	Stack	Comments
5	Push operand	5	We read the tokens and push them on the stack until we encounter an operator
1	Push operand	5, 1	
2	Push operand	5, 1, 2	
+	Add	5, 3	Pop two values (1,2) and push result 3 back again onto the stack
4	Push operand	5, 3, 4	
*	Multiply	5, 12	Pop two values (3,4) and push result 12 back again onto the stack
+	Add	17	Pop two values (5,12) and push result 17 back again onto the stack

Algorithm to Convert an Expression from Infix Notation to RPN (postfix)

- Edsger Dijkstra proposed the Shunting Yard Algorithm to convert infix expression to postfix (RPN).
- Like the evaluation of RPN, the shunting yard algorithm is stack- based.

- While there are tokens to be read from the given expression, read the token.
- If the token is an operand, then insert it to the output queue.
- If the token is an operator O1 and is evaluated left to right then:
  - while there is an operator, O2 at the top of the stack (O2 is TOS), and
  - if precedence of O1 > O2,
  - we push O1 onto the stack (now O1 is TOS)

  else
  - if precedence of O1 <= O2, we pop O2 to the output queue and push O1 onto the stack

- If the token is a left parenthesis, then push it onto the stack.
- If the token is a right parenthesis:
  - Until the token at the top of the stack is a left parenthesis, pop operators off the stack onto the output queue.
  - Pop the left parenthesis from the stack, but don't insert it onto the output queue.
- If the stack runs out without finding a left parenthesis, then there are mismatched parentheses.

Infix to Postfix

- When there are no more tokens to read and there are still operator tokens in the stack:
  - If the operator token on the top of the stack is a parenthesis, then there is mismatched parenthesis.
  - else
  - Pop the operator onto the output queue.
- Exit.

Convert A*(B+C)/D to Postfix Expression

- Push A onto Queue.
- * into stack
- (into stack
- B onto Queue.
- + into stack
- C onto Queue.
-)➔ Pop everything from stack onto Queue and discard matching parenthesis.
- / ➔ / & * have same precedence, so pop * onto queue and push / into stack.
- D onto Queue.
- End of TOKEN encountered, so POP everything from stack onto queue.

➔**A B C + * D /**

Converting Infix Expression to Prefix

1. Reverse the input string i.e. the infix expression that has to be converted to prefix.
2. Examine next element / token in the input.
3. If it is operand, add it to the output string.
4. If it is closing parenthesis, PUSH it on the stack.
5. If it is an operator O, then:
   - if stack is empty, PUSH operator O on stack
   - if TOS is closing parenthesis, PUSH operator O on stack
   - if operator has same or higher priority than TOS, push operator on stack **else** POP TOS to o/p string and PUSH O into stack.

Converting Infix Expression to Prefix

6. If it is an opening parenthesis, POP operator from stack and add to output string until a closing parenthesis is encountered.
7. If there is no more input, unstack remaining operator and add it to output string i.e. POP.
8. Reverse the prefix string and display the result of the given infix expression.

Example

Convert the expression (a + b * c) into prefix

- Reverse the given expression to obtain (c * b + a)

Input Token	Stack	Postfix Expression	Comments
(	(	-	Push
C	(	C	Add to output string
*	(*	C	Push
B	(*	C B	Add to output string
+	(+	C B*	The input token i.e. the operator has lower precedence
A	(+	CB*A	Push
)	-	CB*A+	Pop the stack until opening parenthesis

Example

- Finally we obtain the expression CB*A+.
- We again reverse it to obtain the prefix notation **+ A * B C**.

Prefix to Infix Conversion

- Start with an initially empty stack and then loop through the following steps:
  - Reverse the given input string.
  - Read the input string character by character i.e. all the tokens in the expression.
  - If an operand is read, push it on the stack.
  - If an operator is read, pop the stack for the required number of operands and apply the operation on the two popped operands.
  - Push the result back onto the stack.
  - If the input is a proper prefix expression, the stack contains the corresponding infix expression after the entire input is read.

Convert the expression +/*+abcd*cd into infix

Input Token	Stack Operation	Infix Expression	Remarks
B	Push	b	
C	Push	c, b	
*	Pop, Pop, Push	(c*b)	Pop the two operands and Parenthesize the sub expression and push onto stack
D	Push	d, (c*b)	
C	Push	c, d, (c*b)	
B	Push	b, c, d,(c*b)	
A	Push	a, b, c, d, (c*b)	
+	Pop, Pop, Push	(a+b), c , d, (c*b)	Pop a and b, the two operands and Parenthesize the sub expression and push onto stack
*	Pop, Pop, Push	((a+b)*c), d, (c*b)	
/	Pop, Pop, Push	(((a+b)*c)/d),(c*b)	
+	Pop, Pop, Push	((((a+b)*c)/d)+(c*b))	

Summary

- Stack is a data structure used for storing and retrieving data elements.
- The stack provides temporary storage in such a way that the element stored last will be retrieved first.
- Stack uses two kinds of operations PUSH & POP.
- PUSH operation adds an element to the top of the stack.
- POP operation removes the element from the top of the stack.

Summary

- Stack can be implemented using arrays and linked list.
- Stack Pointer: Stack uses array and an integer index called the stack pointer.
- Stack pointer marks the current top of the stack.