Subject Name: Data Structure

Unit No:2 Unit Name: Stack and Queues

Faculty Name: Kausar Fakir

Index -

Lecture No: 4
Well form-ness of Parenthesis

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
- Tower of Hanoi

Parenthesis Checker(Parenthesis Balance)

- Stacks can be used to check the validity of parentheses in any algebraic expression. For example, an algebraic expression is valid if for every open bracket there is a corresponding closing bracket.
- Examples of balanced parenthesis.

  (a+b), (a/b+c), a/((b-c)*d)

  Open and closed parenthesis are properly paired.

- Examples of not balance parenthesis.

  ((a+b)*2 and m*(n+(k/2)))

  Open and closed parenthesis are not properly paired.

Parenthesis Checker(Parenthesis Balance)

Try this ➔ (a+b}, [a/b+c), a/((b-c)*d}

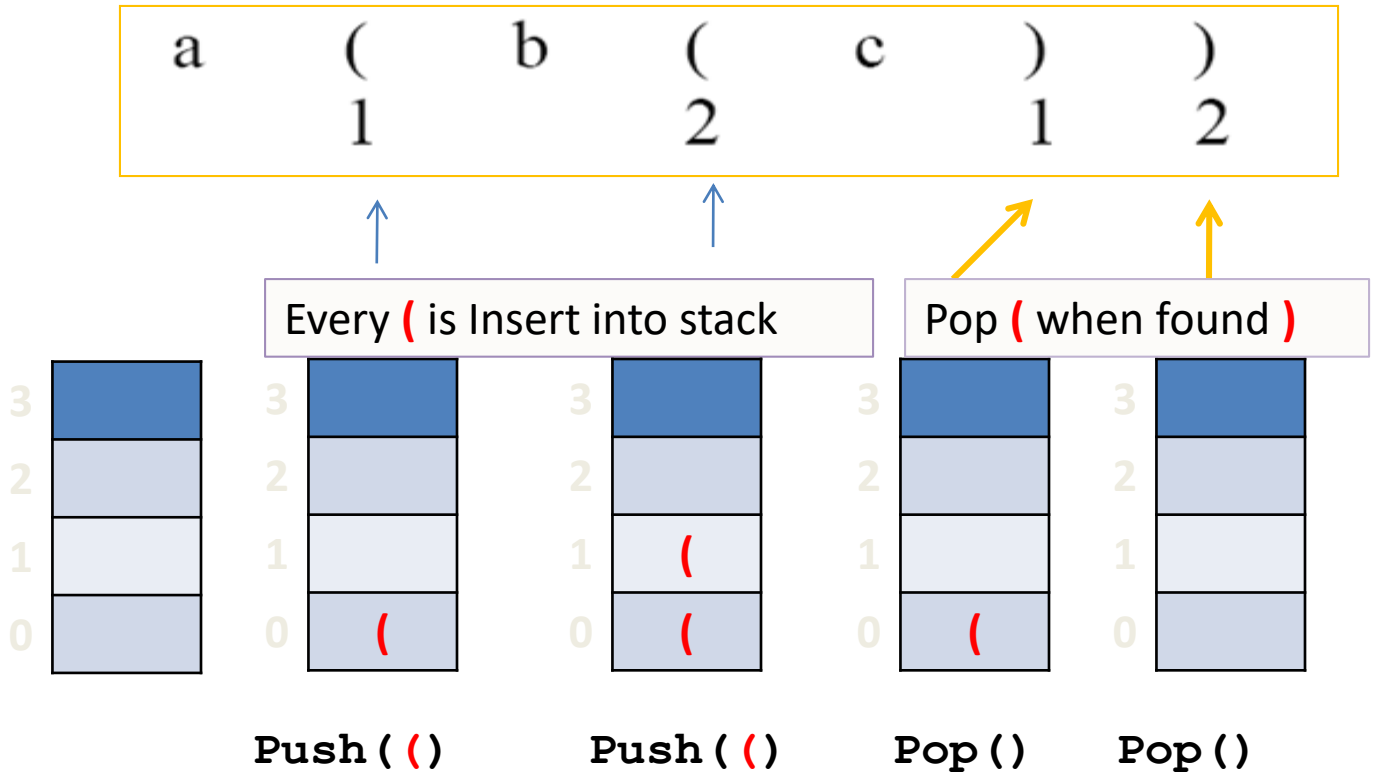
Check for Balanced Parenthesis

Expression	Balanced???
( )	
{( )}	
{( ) ( )	
[ ] ( )]	
{)	

Check for Balanced Parenthesis

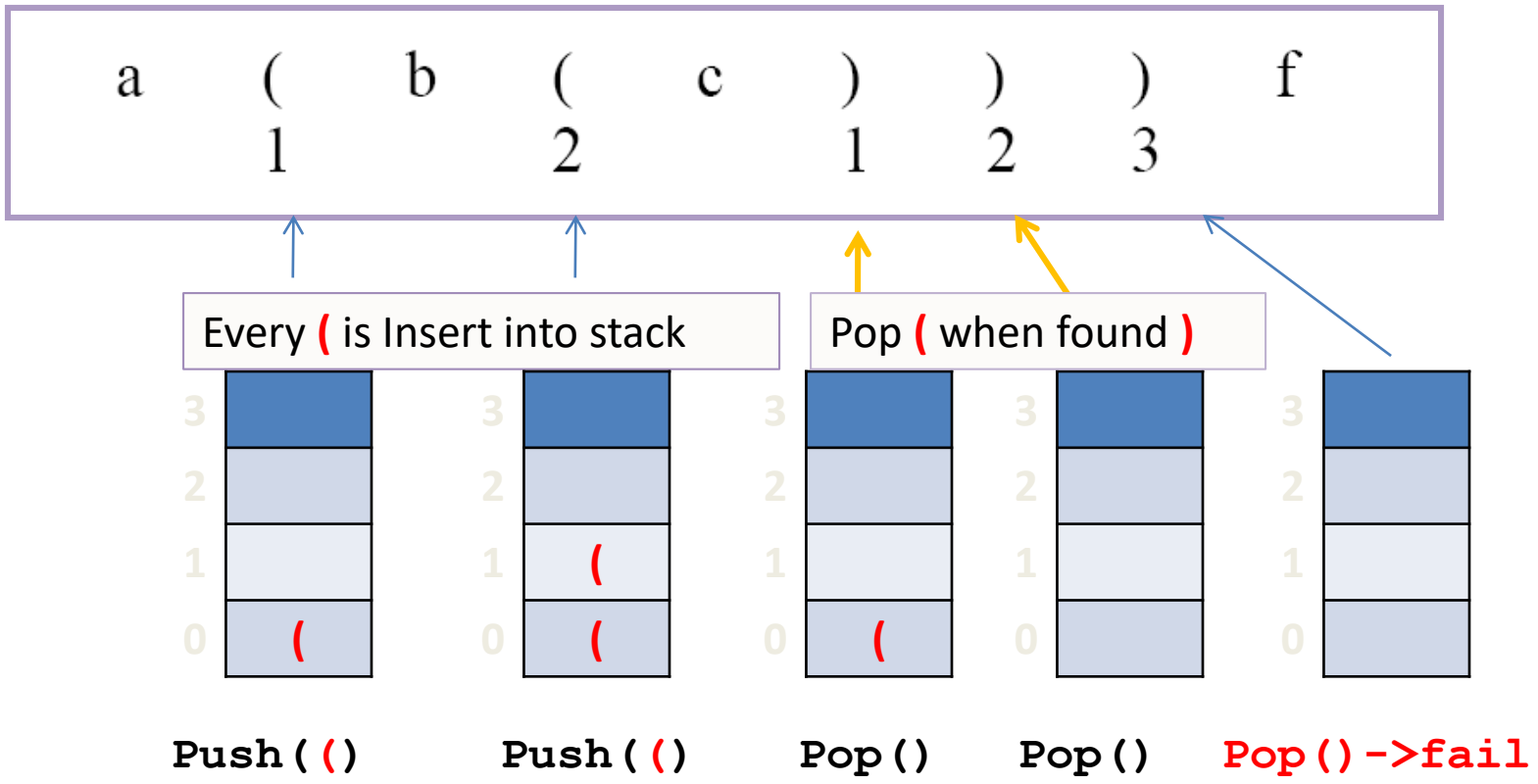
Expression	Balanced???
( )	Yes
{( )}	Yes
{( ) ( )	No
[ ] ( )]	No
{)	No

Example of Balance Parenthesis : Use of Stack



Expression **a(b(c))** have balance parenthesis since when end of string is found the stack is empty.

Example of ImBalance Parenthesis : Use of Stack



Expression **a(b(c))) f** does not have balance parentheses => the third) encountered does not has its match, the stack is empty.

Algorithm for Parenthesis Checker(Parenthesis Balance)

1.Create an empty stack

2. Scan expression from left to right

3. For Every opening bracket (**(**) is Insert that into stack.

4. For closing bracket

i) if stack is empty

invalid expression as closing brackets are more than opening brackets.

ii) Else Pop element (**(**) when found **)**

In this if Pop element does not match with opening bracket then print invalid expression

5. After scanning all elements

i)if stack is empty- It's a valid expression

ii else invalid expression left brackets are more than right brackets.

Program for Parenthesis Checker(Parenthesis Balance)

```
#include <stdio.h>
#include <conio.h>
#include <string.h>
#define MAX 10
int top = -1;
int stk[MAX];
void push(char);
```

```
char pop();
```

```
void main()
{
        char exp[MAX],temp;
        int i, flag=1;
        clrscr();
        printf("Enter an expression : ");
        gets(exp);
        for(i=0;i<strlen(exp);i++)
        {
                if(exp[i]=='(' || exp[i]=='{' || exp[i]=='[')
                        push(exp[i]);
                if(exp[i]==')' || exp[i]=='}' || exp[i]==']')
                        if(top == -1)
                                flag=0;
                        else
                        {
                                temp=pop();
                                if(exp[i]==')' && (temp=='{' || temp=='['))
                                        flag=0;
                                if(exp[i]=='}' && (temp=='(' || temp=='['))
                                        flag=0;
                                if(exp[i]==']' && (temp=='(' || temp=='{'))
                                        flag=0;
                        }
        }
        if(top>=0)
                flag=0;
        if(flag==1)
                printf("\n Valid expression");
        else
                printf("\n Invalid expression");
}
```

Program for Parenthesis Checker(Parenthesis Balance)

```
void push(char c)
{
        if(top == (MAX-1))
                printf("Stack Overflow\n");
        else
        {
                top=top+1;
                stk[top] = c;
        }
}
char pop()
{
        if(top == -1)
                printf("\n Stack Underflow");
        else
                return(stk[top--]);
}
```

Expression Validation Practice through Vlabs

http://ds1-iiith.vlabs.ac.in/data-structures-1/exp/infix-postfix/exp.html#Validation%20of%20Expressions

Questions

1. Write a program in ‘C’ to check for balanced parenthesis in an expression

using stack

2. Explain Linear and Non-Linear data structures.

Explain different types of data structures with example

3. Write a program in C to evaluate postfix equation using stack ADT

4. Define Data Structure. Differentiate linear and non-linear data structures with example.

5.What are various operations possible on data structures?

6. Use stack data structure to check well-formed ness of parentheses in an algebraic expression. Write C program for the same

Lecture No: 5
Infix to Postfix and Postfix Evaluation

Algebraic Expression

- One of the compiler's task is to evaluate algebraic expression.
- The way to write arithmetic expression is known as notation.
- An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of expression. These notations are :

1. Infix Notation
2. Prefix (Polish) Notation
3. Postfix (Reverse-Polish) Notation

Parsing Expressions

- Precedence and associativity, determines the order of evaluation of an expression. An operator precedence and associativity table is given below (highest to lowest) –

Sr No.	Operator	Precedence	Associativity
1	Exponentiation ^	Highest	Right Associative
2	Multiplication (*) & Division (/)	Second Highest	Left Associative
3	Addition (+) & Subtraction (−)	Lowest	Left Associative

Infix Expression

- The algebraic expression commonly used is infix.
- **The term infix indicates that every binary operators appears between its operands.**
- Example 1:

A	+	B
operand	operator	operand

- Example 2: A + B * C 

A + (B * C)
(a + b) * c

- To evaluate infix expression, the following rules were applied:
  1. Precedence rules.
  2. Association rules (associate from *left to right*)
  3. Parentheses rules

Prefix and Postfix Expressions

Alternatives to infix expression

Prefix : Operator appears before its operand.

Example:

+ a b

+ a * b c

* + a b c

Postfix : Operator appears after its operand.

Example:

a b +

a b c * +

a b + c *

Infix, Prefix and Postfix

Infix Notation	Prefix Notation	Postfix Notation
a+b	+ab	ab+
(a + b) * c	* + a b c	a b + c *
a * (b + c)	* a + b c	a b c + *
a / b + c / d	+ / a b / c d	a b / c d / +
(a + b) * (c + d)	* + a b + c d	a b + c d + *
((a + b) * c) - d	- * + a b c d	a b + c * d -

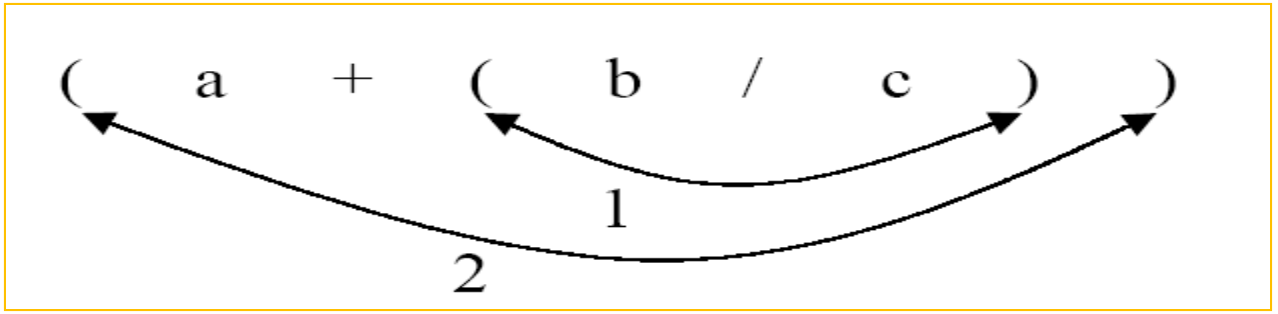
Infix, Prefix and Postfix

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + (b * c)	+ a * b c	a b c * +
(a + b) * c	* + a b c	a b + c *

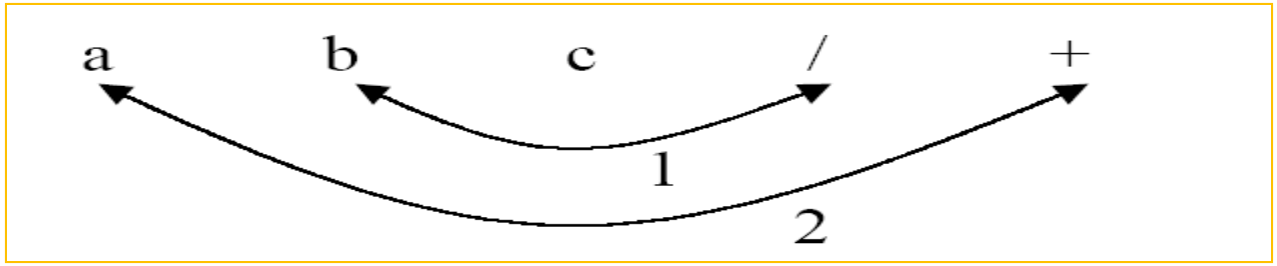
Converting Infix to Postfix

a + b / c

STEP 1



STEP 2



Examples

1. a + b

2. a + b * c

3. a + b * (c – d) / (p – r)

Rules for Infix to Postfix Conversion

1. Scan the expression from **left to right**
2. If element is operand then print **Operand in postfix** as it arrives
3. If symbol is " **(** " push it on stack.
4. If symbol is " **)** " then pop all the elements from stack and print to postfix string till "**(** " appears and then discard "**(** ".
5. If symbol **operator** arrives and stack is empty then **push** this operator onto the stack.

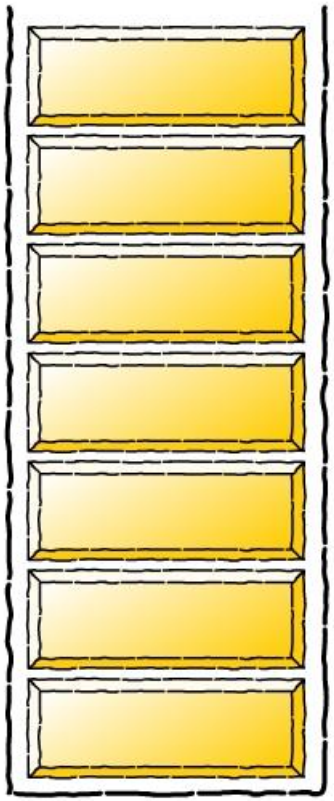
i) If **incoming operator** has **HIGHER precedence** than **TOP** of the stack operator then PUSH this operator onto the stack.

ii) If **incoming operator** has **LOWER or EQUAL precedence** than **TOP** of the stack then POP this operator and print in **POSTFIX** array. Then test the precedence of incoming operator with NEW TOP of the stack.

6. At the end of expression, POP and print all element of stack in postfix array.

Infix to Postfix Conversion

Stack



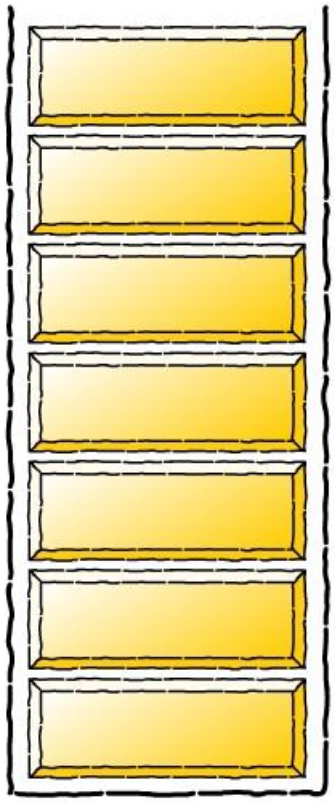
Infix Expression

A + B * C – D / E

Postfix Expression

Infix to Postfix Conversion

Stack



Infix Expression

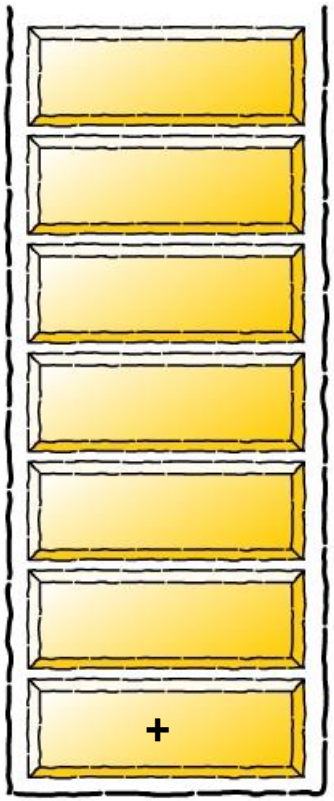
+ B * C – D / E

Postfix Expression

A

Infix to Postfix Conversion

Stack



Infix Expression

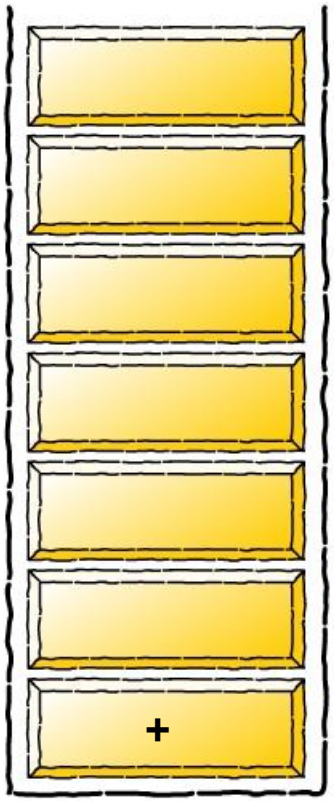
B * C – D / E

Postfix Expression

A

Infix to Postfix Conversion

Stack



Infix Expression

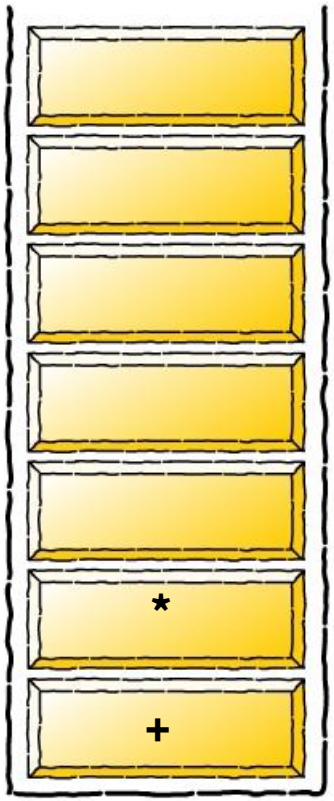
* C – D / E

Postfix Expression

AB

Infix to Postfix Conversion

Stack



Infix Expression

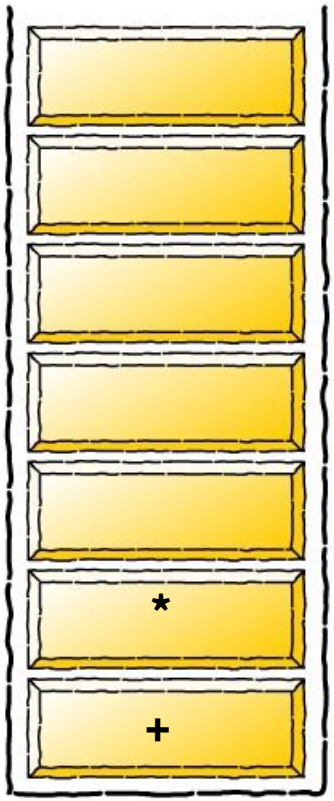
C – D / E

Postfix Expression

AB

Infix to Postfix Conversion

Stack



Infix Expression

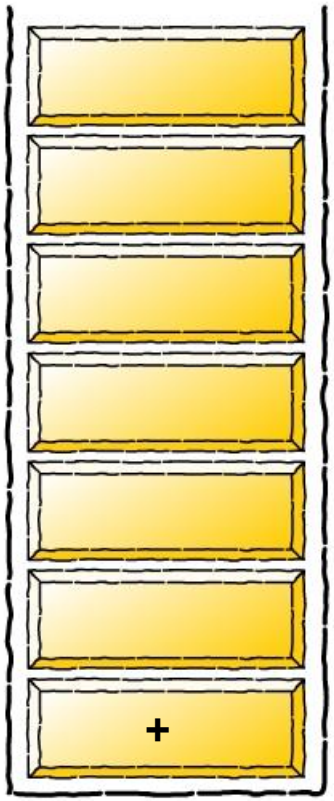
– D / E

Postfix Expression

ABC

Infix to Postfix Conversion

Stack



Infix Expression

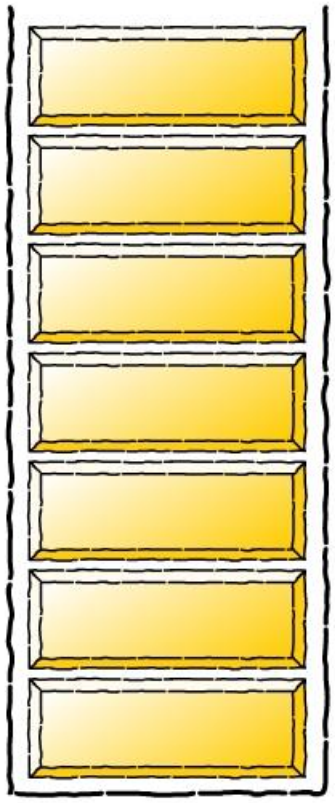
– D / E

Postfix Expression

A B C *

Infix to Postfix Conversion

Stack



Infix Expression

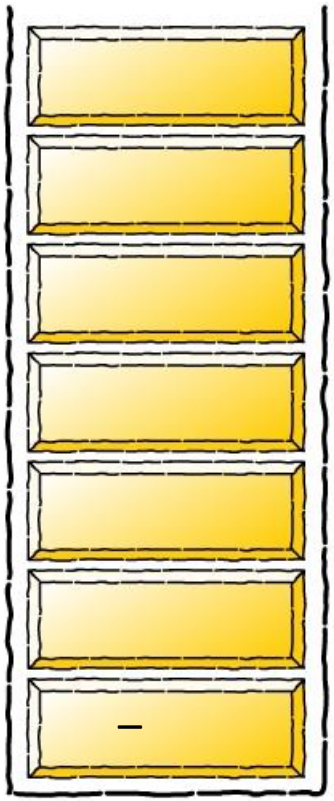
– D / E

Postfix Expression

A B C * +

Infix to Postfix Conversion

Stack



Infix Expression

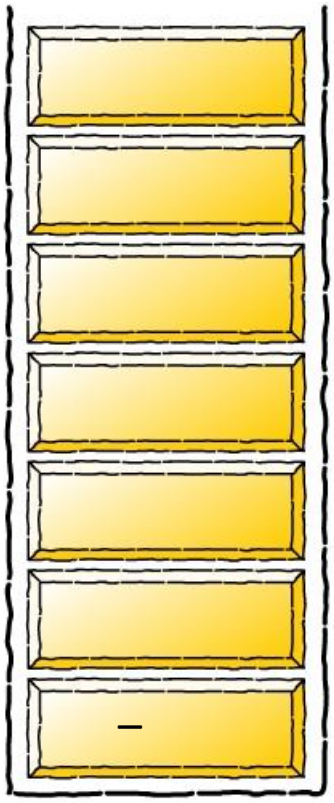
D / E

Postfix Expression

A B C * +

Infix to Postfix Conversion

Stack



Infix Expression

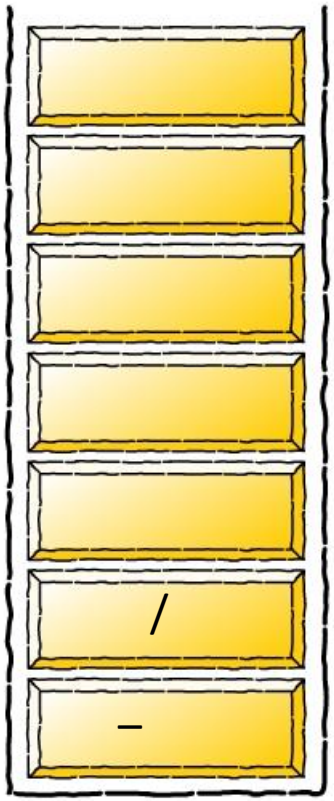
/ E

Postfix Expression

A B C * + D

Infix to Postfix Conversion

Stack



Infix Expression

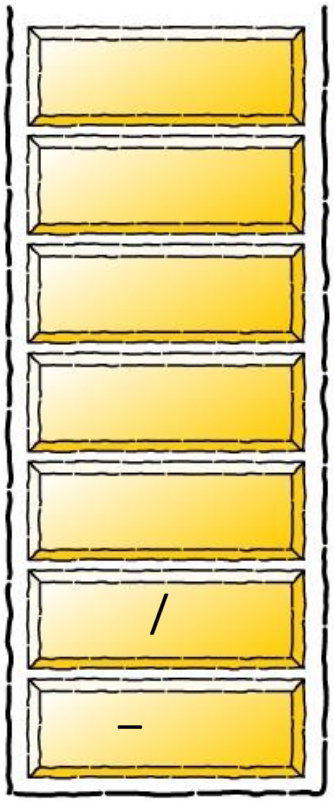
E

Postfix Expression

A B C * + D

Infix to Postfix Conversion

Stack



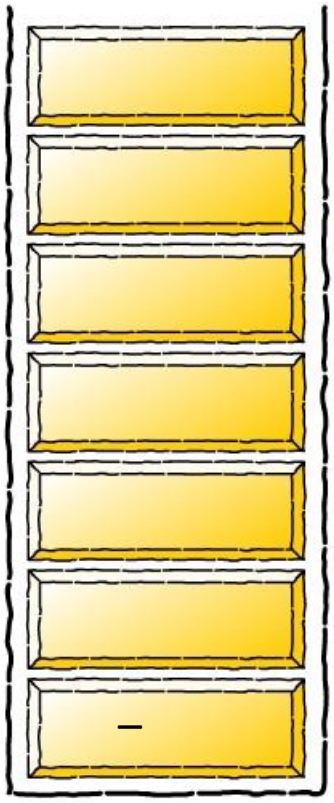
Infix Expression

Postfix Expression

A B C * + D E

Infix to Postfix Conversion

Stack



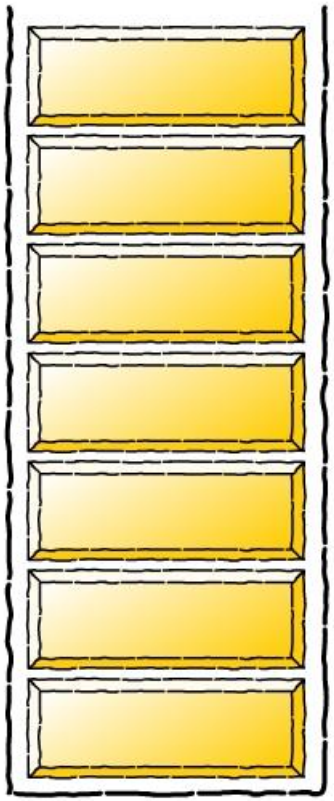
Infix Expression

Postfix Expression

A B C * + D E /

Infix to Postfix Conversion

Stack



Infix Expression

Postfix Expression

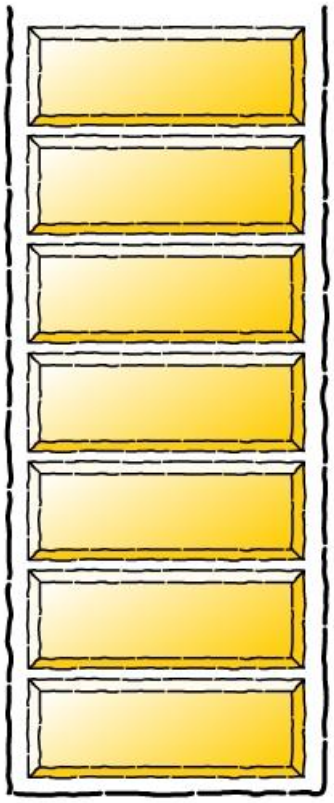
A B C * + D E / –

Converting Infix to Postfix using Stack :
A + B * C – D / E

infix	stack	postfix
A + B * C – D / E	#	
+ B * C – D / E	#	A
B * C – D / E	# +	A
* C – D / E	# +	A B
C – D / E	# + *	A B
– D / E	# + *	A B C
D / E	# -	A B C * +
/ E	# -	A B C * + D
E	# - /	A B C * + D
	# - /	A B C * + D E
	#	A B C * + D E / -

Infix to Postfix Conversion

Stack



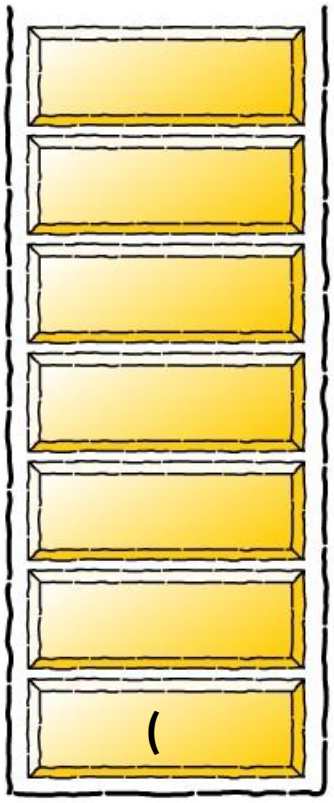
Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

Infix to Postfix Conversion

Stack



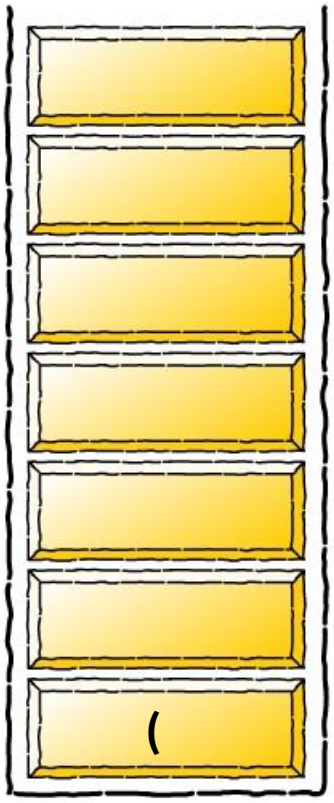
Infix Expression

a + b - c) * d – (e + f)

Postfix Expression

Infix to Postfix Conversion

Stack



Infix Expression

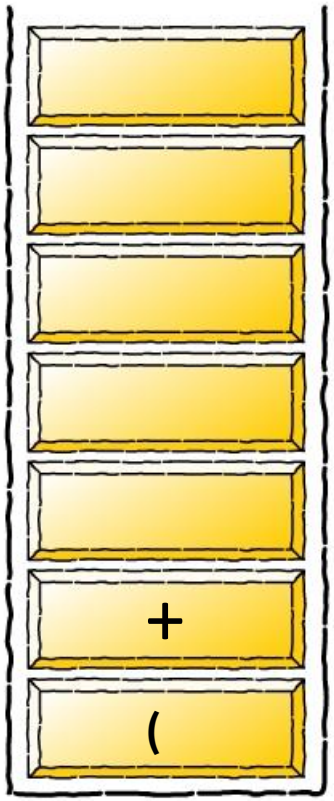
+ b - c) * d – (e + f)

Postfix Expression

a

Infix to Postfix Conversion

Stack



Infix Expression

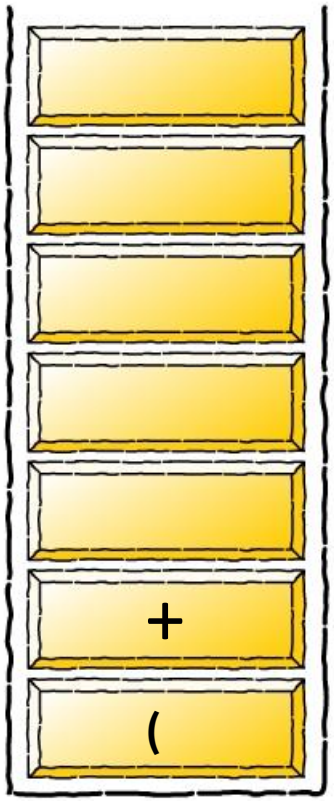
b - c) * d – (e + f)

Postfix Expression

a

Infix to Postfix Conversion

Stack



Infix Expression

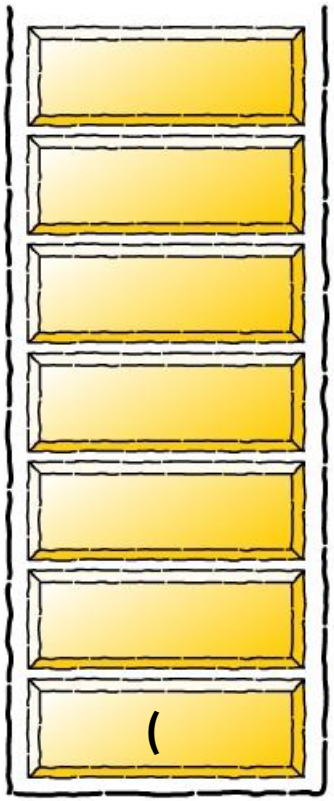
- c) * d – (e + f)

Postfix Expression

a b

Infix to Postfix Conversion

Stack



Infix Expression

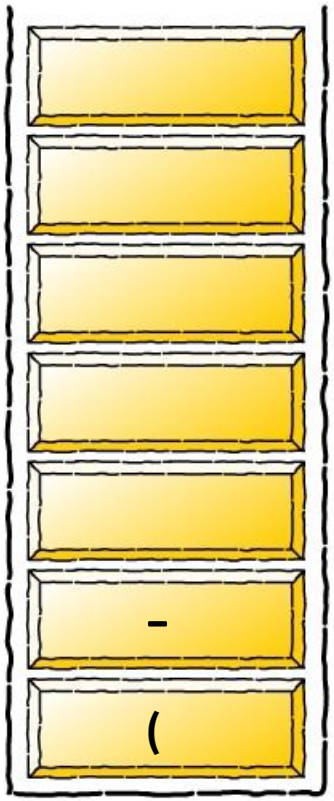
- c) * d – (e + f)

Postfix Expression

a b+

Infix to Postfix Conversion

Stack



Infix Expression

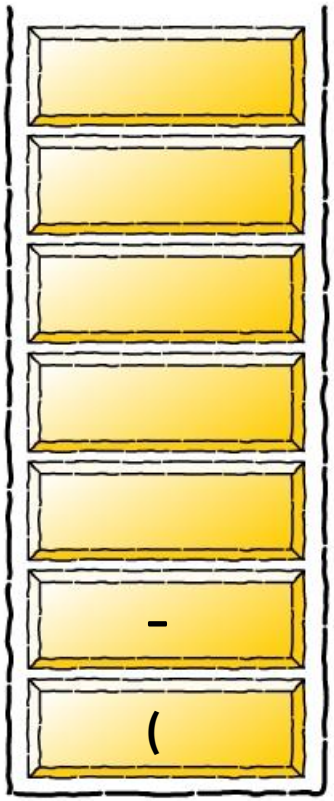
c) * d – (e + f)

Postfix Expression

a b +

Infix to Postfix Conversion

Stack



Infix Expression

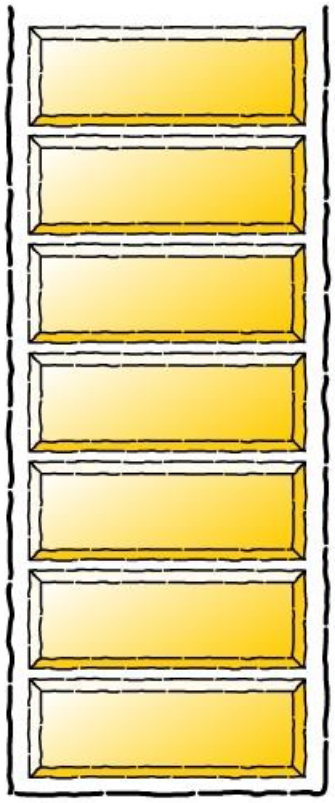
) * d – (e + f)

Postfix Expression

a b + c

Infix to Postfix Conversion

Stack



Infix Expression

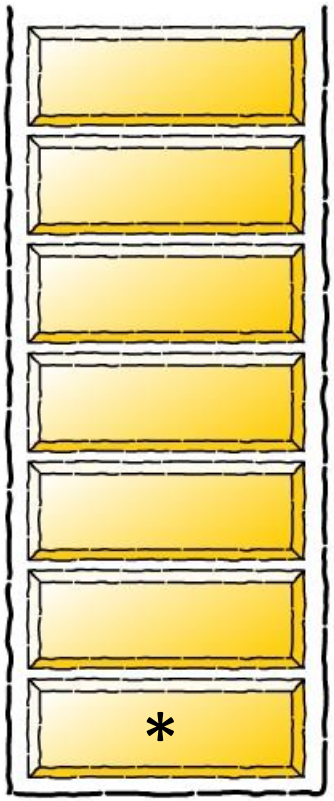
* d – (e + f)

Postfix Expression

a b + c -

Infix to Postfix Conversion

Stack



Infix Expression

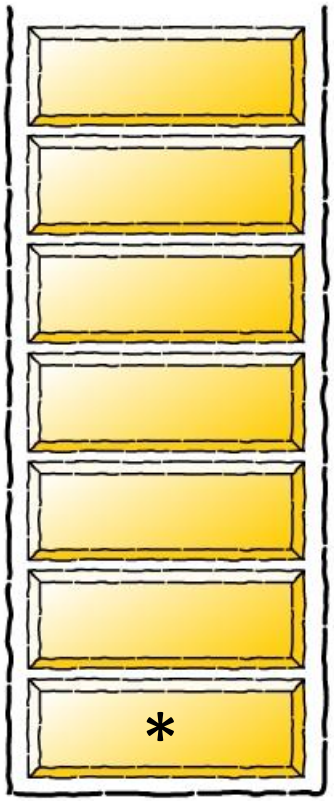
d – (e + f)

Postfix Expression

a b + c -

Infix to Postfix Conversion

Stack



Infix Expression

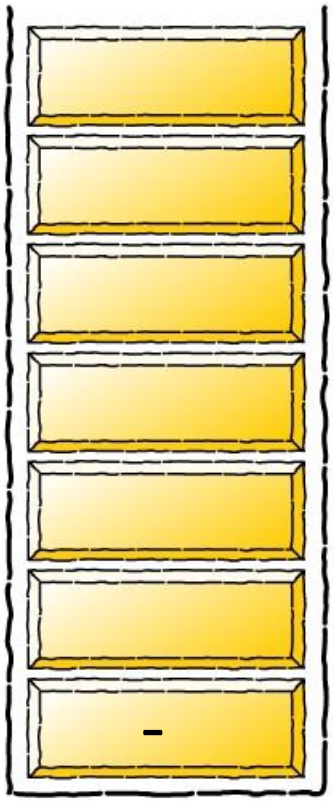
– (e + f)

Postfix Expression

a b + c - d

Infix to Postfix Conversion

Stack



Infix Expression

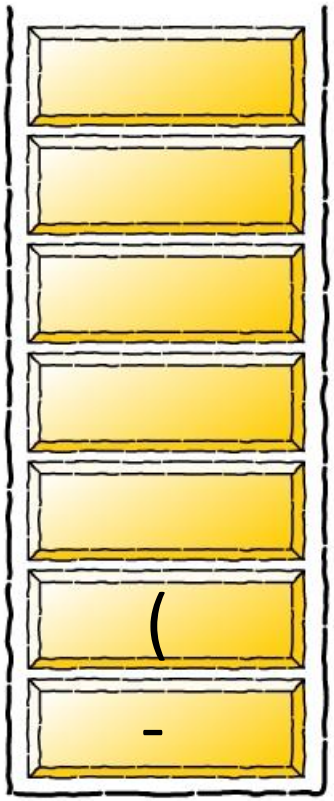
(e + f)

Postfix Expression

a b + c – d *

Infix to Postfix Conversion

Stack



Infix Expression

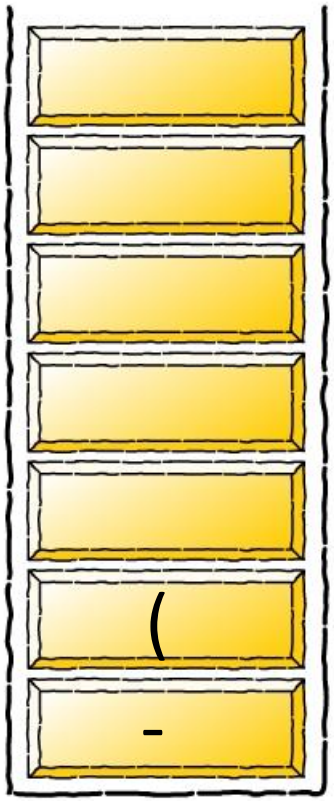
e + f)

Postfix Expression

a b + c – d *

Infix to Postfix Conversion

Stack



Infix Expression

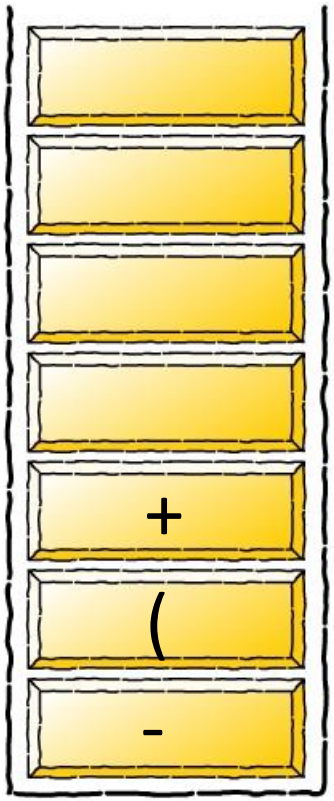
+ f)

Postfix Expression

a b + c – d * e

Infix to Postfix Conversion

Stack



Infix Expression

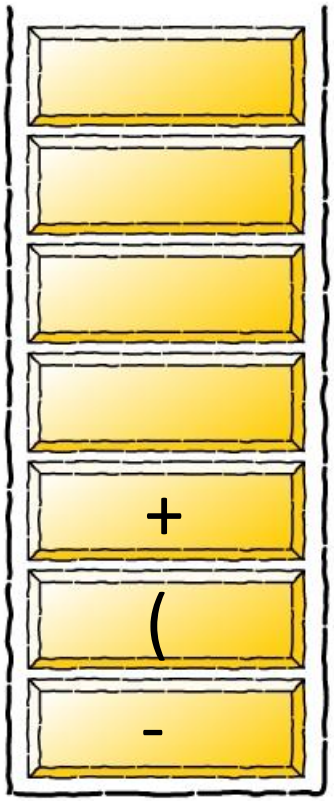
f)

Postfix Expression

a b + c – d * e

Infix to Postfix Conversion

Stack



Infix Expression

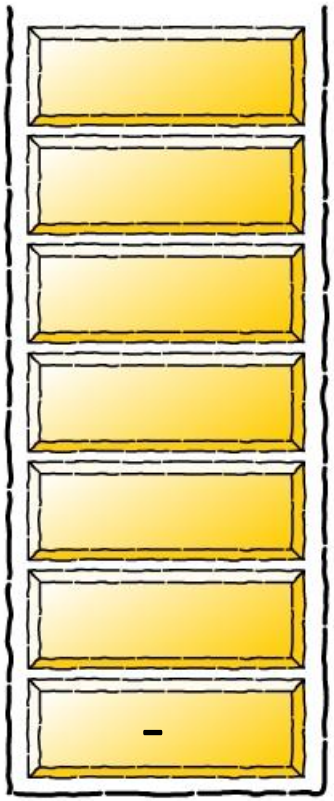
)

Postfix Expression

a b + c – d * e f

Infix to Postfix Conversion

Stack



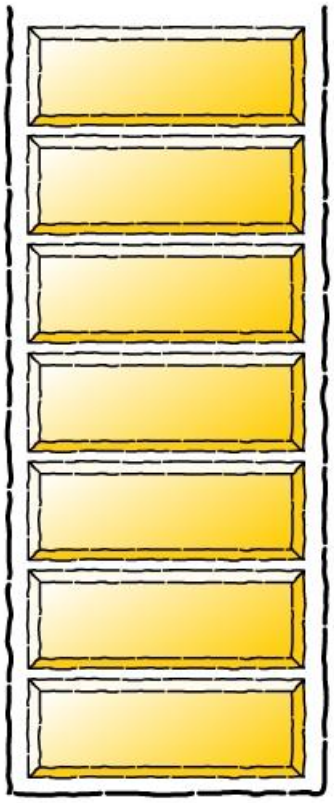
Infix Expression

Postfix Expression

a b + c – d * e +

Infix to Postfix Conversion

Stack

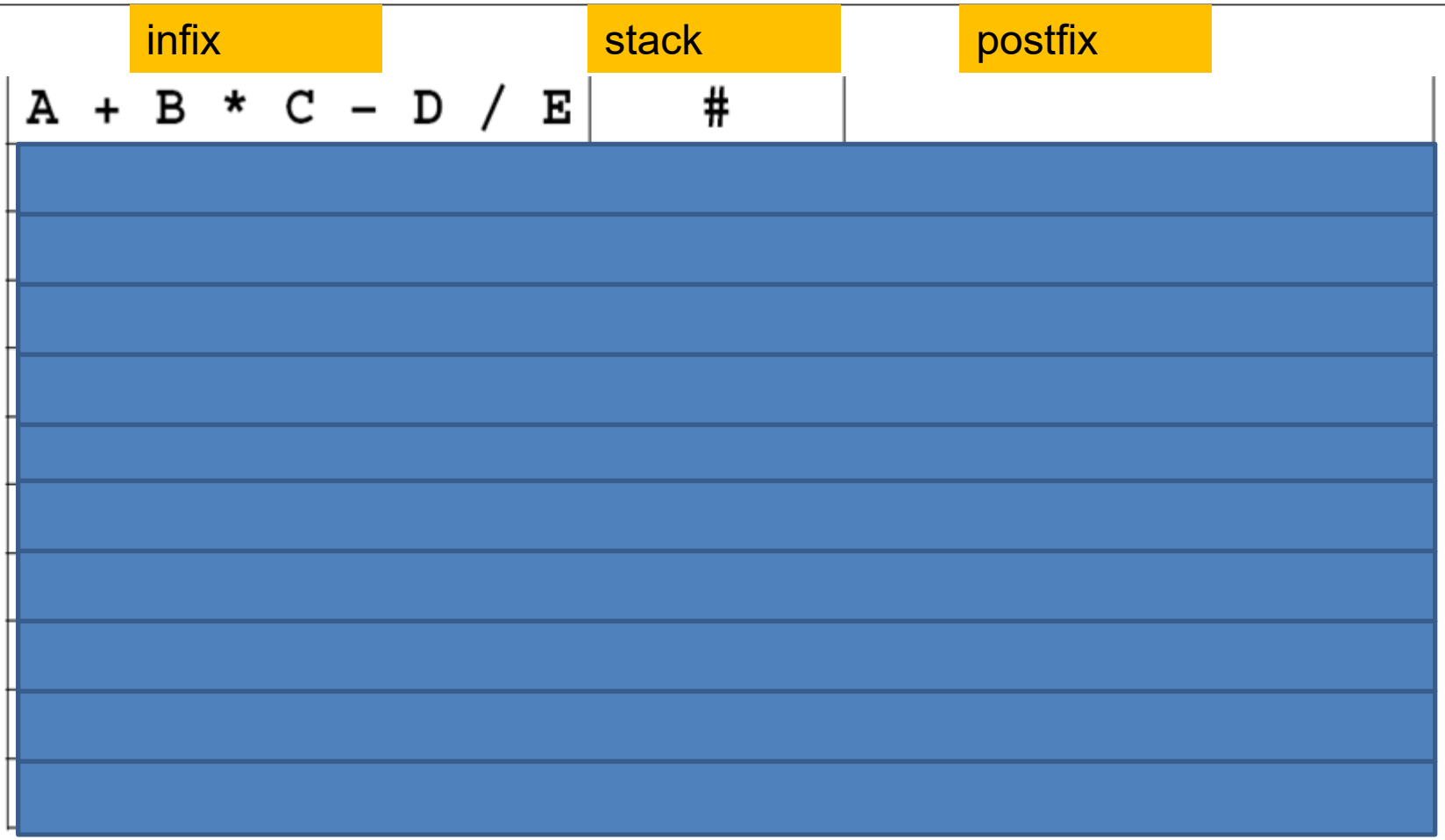


Infix Expression

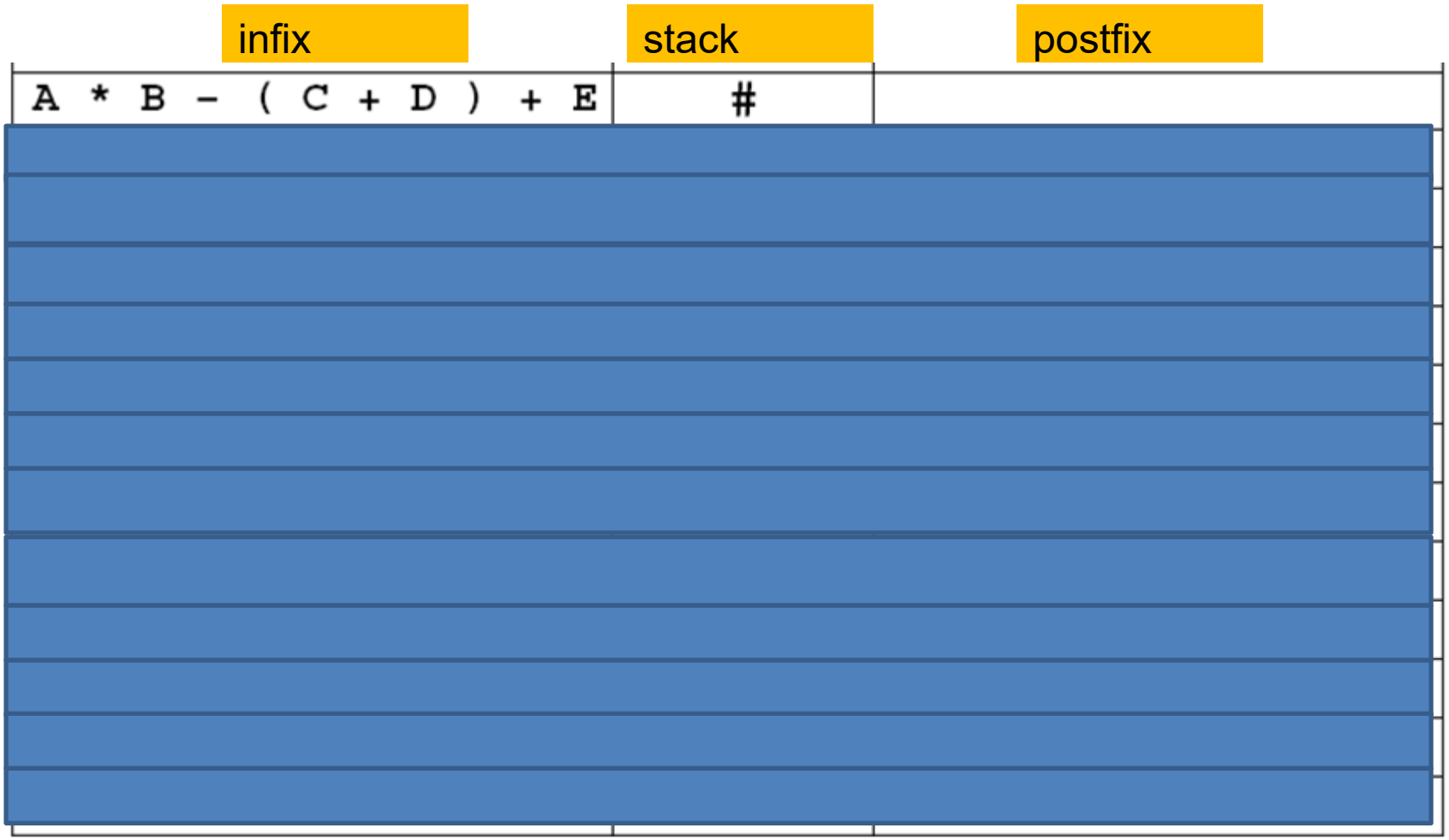
Postfix Expression

a b + c – d * e + -

Converting Infix to Postfix using Stack :
A + B * C – D / E



Converting Infix to Postfix using Stack :
A * B – (C + D)+ E



Converting Infix to Postfix using Stack :
A * B – (C + D)+ E

infix	stack	postfix
A * B – (C + D) + E	#	
* B – (C + D) + E	#	A
B – (C + D) + E	# *	A
– (C + D) + E	# *	A B
(C + D) + E	# -	A B *
C + D) + E	# - (	A B *
+ D) + E	# - (	A B * C
D) + E	# - (+	A B * C
) + E	# - (+	A B * C D
+ E	# -	A B * C D +
E	# +	A B * C D + -
	# +	A B * C D + - E
	#	A B * C D + - E +

Practice Problems

Convert A-(B/C+(D%E*F)/G)*H infix expression to postfix form:

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 * –

Infix to postfix

```
#include<stdio.h>
#include<ctype.h>

char stack[100];
int top = -1;

void push(char x)
{
   stack[++top] = x;
}

char pop()
{
   if(top == -1)
      printf("\n STACK UNDERFLOW");
   else
      return stack[top--];
}
```

Infix to postfix

```
int priority(char x)
{
   if(x == '(')
      return 0;
   if(x == '+' || x == '-' )
      return 1;
   if(x == '*' || x == '/' || x=='%')
      return 2;
   return 0;
}
```

Infix to postfix

```
int main()
{
   char exp[100];
   char c, x;
   int i=0;
   printf("Enter the expression : ");
   scanf("%s",exp);
   printf("\n");

   while(exp[i] != '\0')
   {
      if(isalnum(exp[i]))
         printf("%c ",exp[i]);
      else if(exp[i] == '(')
         push(exp[i]);
      else if(exp[i] == ')')
      {
         while((x = pop()) != '(')
            printf("%c ", x);
      }
      else
      {
         while(priority(stack[top]) >= priority(exp[i]))
            printf("%c ",pop());
         push(exp[i]);
      }
      i++;
   }

   while(top != -1)
   {
      printf("%c ",pop());
   }return 0;
}
```


Solve using vlabs

http://ds1-iiith.vlabs.ac.in/data-structures-1/exp/infix-postfix/exp.html#Whole%20Conversion%20Exercise

Postfix Expression Evaluation : Rules

1. If input read from postfix expression is an **operand**, push operand to stack.
2. If input read from postfix expression is an **operator**, pop the first 2 operand in stack and implement the expression using the following operations:

   op2= pop()

   op1= pop()

   result = op1 operator op2
3. Push the result of the evaluation to stack.
4. Repeat steps 1 to steps 3 until end of postfix expression

Finally, At the end of the operation, only one value left in the stack. The value is the result of postfix evaluation.

Evaluating Postfix Expression
2 4 6 + *

postfix | result | stack

	Ch	Opr	Opn1	Opn2		
2 4 6 + *						
4 6 + *	2					2
6 + *	4					2 4
+ *	6					2 4 6
*	+	+	4	6	10	2 10
	*	*	2	10	20	20

Evaluating Postfix Expression
2 7 * 18 - 6 +

postfix | result | stack

	Ch	Opr	Opn1	Opn2		
2 7 * 18 – 6 +						
7 * 18 – 6 +	2					2
* 18 – 6 +	7					2 7
18 – 6 +	*	*	7	2	14	14
– 6 +	18					14 18
6 +	–	–	14	18	-4	-4
+	6					-4 6
	+	+	-4	6	2	2

Practice Problems

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

Practice Problems

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

Evaluating Postfix Expression

Write a program to evaluate a postfix expression.

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAX 100
float st[MAX];
int top=-1;
void push(float st[], float val);
float pop(float st[]);
float evaluatePostfixExp(char exp[]);
int main()
{
        float val;
        char exp[100];
        clrscr();
        printf("\n Enter any postfix expression : ");
        gets(exp);
        val = evaluatePostfixExp(exp);
        printf("\n Value of the postfix expression = %.2f", val);
        getch();
        return 0;
}
```

Evaluating Postfix Expression

```
float evaluatePostfixExp(char exp[])
{
        int i=0;
        float op1, op2, value;
        while(exp[i] != '\0')
        {
                if(isdigit(exp[i]))
                        push(st, (float)(exp[i]-'0'));
                else
                {
                        op2 = pop(st);
                        op1 = pop(st);
                        switch(exp[i])
```

Evaluating Postfix Expression

```
                        {
                                case '+':
                                        value = op1 + op2;
                                        break;
                                case '-':
                                        value = op1 - op2;
                                        break;
                                case '/':
                                        value = op1 / op2;
                                        break;
                                case '*':
                                        value = op1 * op2;
                                        break;
                                case '%':
                                        value = (int)op1 % (int)op2;
                                        break;
                        }
                        push(st, value);
                }
                i++;
        }
        return(pop(st));
}
```

Thank You

D Y PATIL
DEEMED TO BE
UNIVERSITY
RAMRAO ADIK
INSTITUTE OF TECHNOLOGY
NAVI MUMBAI