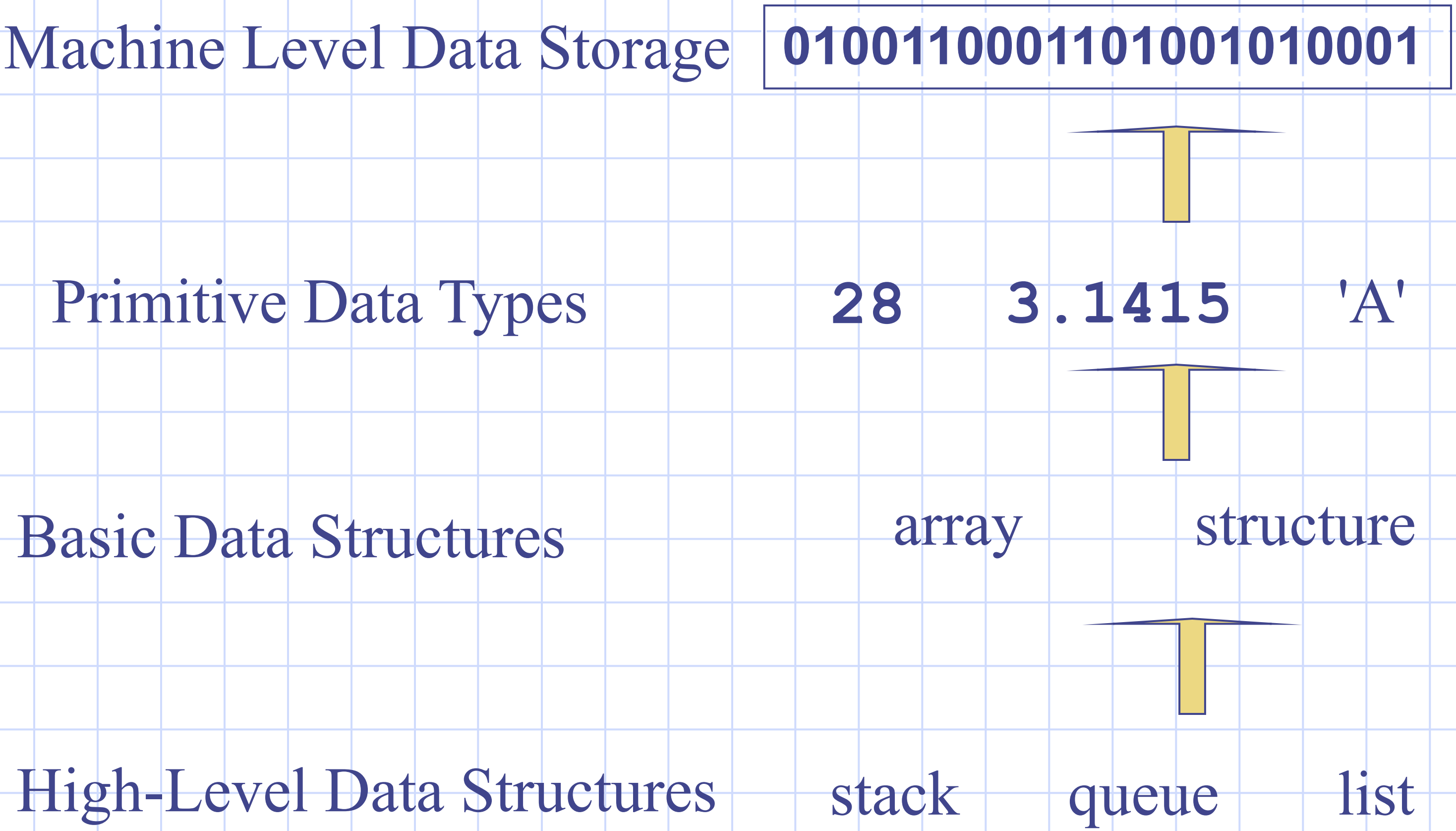
From Data to Data Structures



Machine Level Data Storage 0100110001101001010001

Primitive Data Types 28 3.1415 'A'

Basic Data Structures array structure

High-Level Data Structures stack queue list

Important Data Structures

- *Linear structures*
  - Array: Fixed-size
  - Linked-list: Variable-size
  - Stack: Add to top and remove from top
  - Queue: Add to back/end and remove from front
  - Priority queue: Add anywhere, remove the highest priority

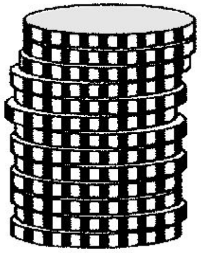
Important Data Structures

- *Hash tables*: Unordered lists which use a ‘hash function’ to insert and search
- *Tree*: A branching structure with no loops
- *Graph*: A more general branching structure, with less stringent connection conditions than for a tree

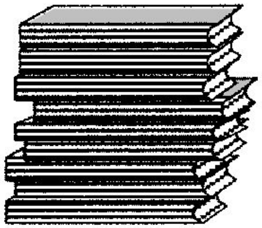
Stack

STACK

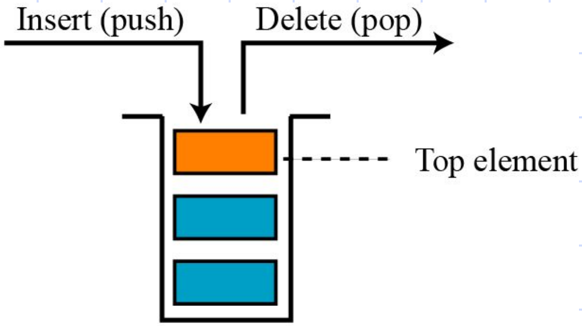
- A stack is a linear data structure in which addition and deletion of an existing element always takes place at the same end.
- A stack is a data structure of *ordered* entries such that entries can be inserted and removed at only one end (call the top)
- This end is known as top of the stack.



Stack of coins



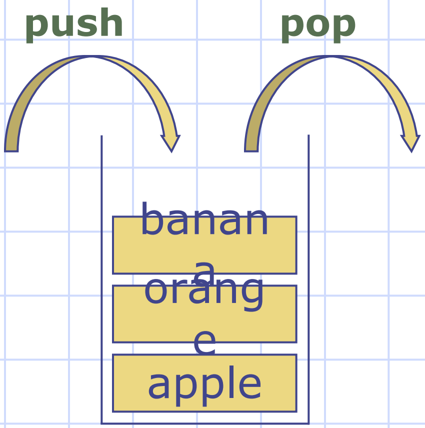
Stack of books



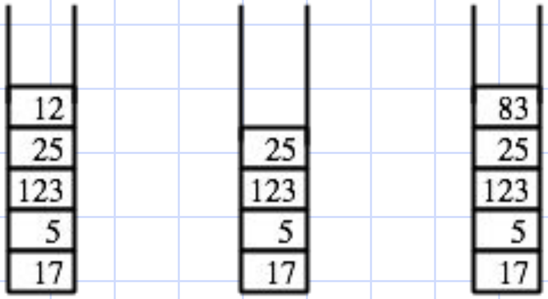
Computer stack

STACK

- Addition of an element in the stack is called as push operation.
- Deletion of an element from stack is known as pop operation.
- Stack is also called as last-in first-out (LIFO) list.
- Example: Cafeteria Trays



In a stack, all operations take place at the "top" of the stack. The "push" operation adds an item to the top of the stack. The "pop" operation removes the item on the top of the stack and returns it.



Original stack. After pop(). After push(83).

Terminology for stacks

- **Stack** = a list in which entries are removed and inserted only at the head
- **LIFO** = last-in-first-out
- **Top** = head of list
- **Bottom** or **base** = tail of list
- **Pop** = remove entry from the top
- **Push** = insert entry at the top

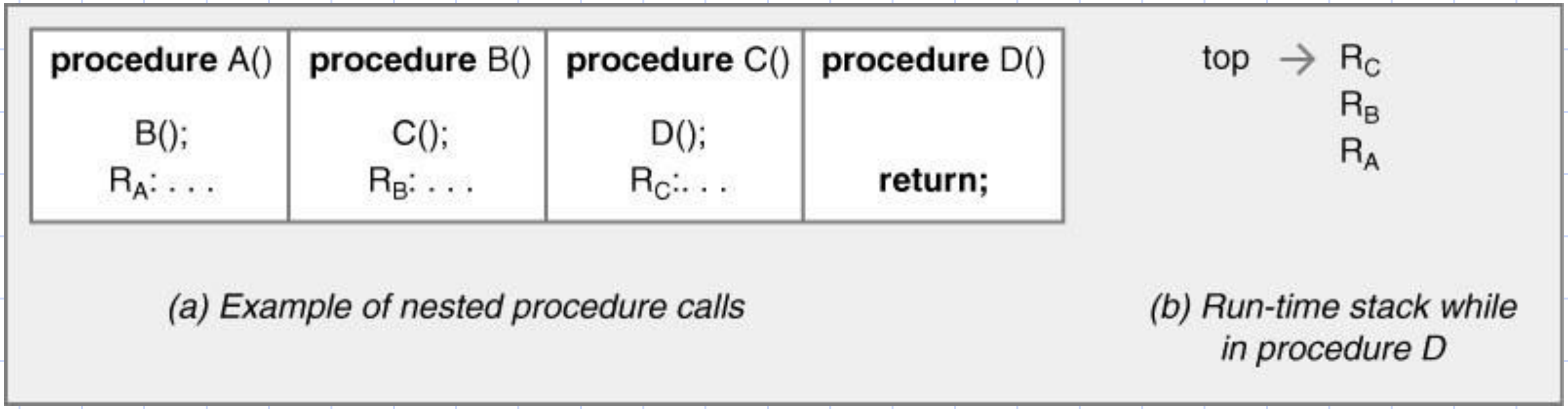
Applications of Stacks

- Direct applications
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine or C++ runtime environment (Program execution)
  - Parsing
  - Reversing a string
  - postponing data usage and backtracking steps.
  - Evaluating an expression
  - Call stack (recursion).
  - Searching networks, traversing trees (keeping a track where we are).

    Examples:
    - Checking balanced expressions
    - Recognizing palindromes
    - Evaluating algebraic expressions
- Indirect applications
  - Component of other data structures

Stack Applications

- Run-time procedure information



procedure A()	**procedure** B()	**procedure** C()	**procedure** D()
B(); R_A: . . .	C(); R_B: . . .	D(); R_C:. . .	**return;**

(a) Example of nested procedure calls

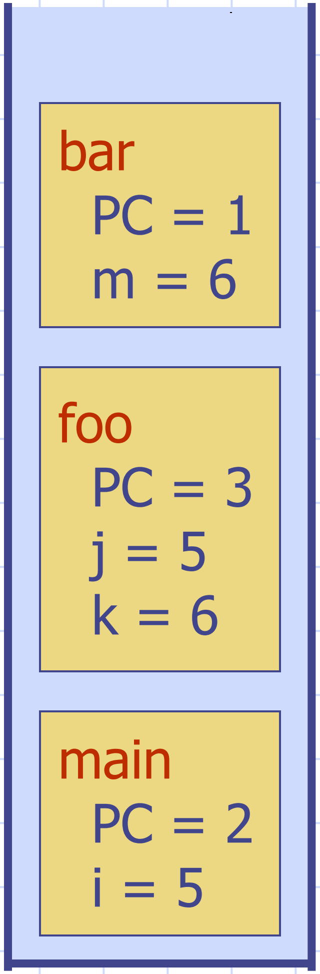
(b) Run-time stack while in procedure D

- Arithmetic computations
  - Postfix notation

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack
- When a method is called, the JVM pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
      int i = 5;
      foo(i);
      }

foo(int j) {
      int k;
      k = j+1;
      bar(k);
      }

bar(int m) {
      …
      }
```



bar
PC = 1
m = 6

foo
PC = 3
j = 5
k = 6

main
PC = 2
i = 5

Stack Operation

- All the data item is accessed at one end and it is called top of the stack

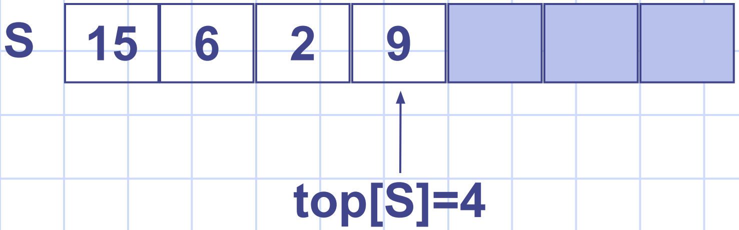
- Operations

1. **Stack: create an empty stack**
2. **Push : insert the element at the top of the stack**
3. **Pop : Delete the element from the top of the stack**
4. Peep : Function returns the value of the ith element from top of the stack
5. Change: It changes the value of the ith element from the top of the stack to new value.
6. **Empty : Check whether stack is empty or not**

Stack example: matching braces and parenthesis

- Goal: make sure left and right braces and parentheses match
  - This can't be solved with simple counting
  - { (x) } is OK, but { (x}) isn't
- Rule: **{ ok string }** is OK
- Rule: **(ok string)** is OK
- Use a stack
  - Place left braces and parentheses on stack
  - When a right brace / paren is read, pop the left off stack
  - If none there, report an error (no match)

Illustration of PUSH

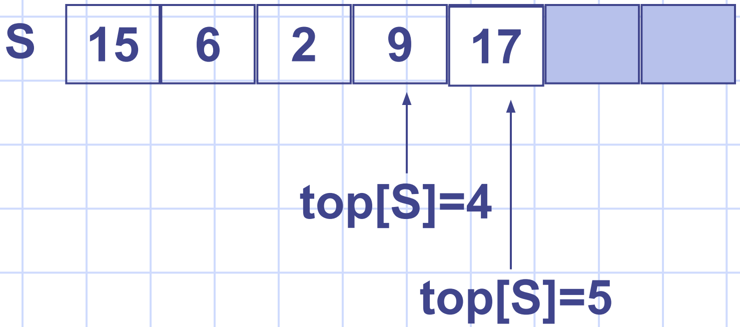


S

15	6	2	9			

top[S]=4

PUSH(S, 17)



S

15	6	2	9	17		

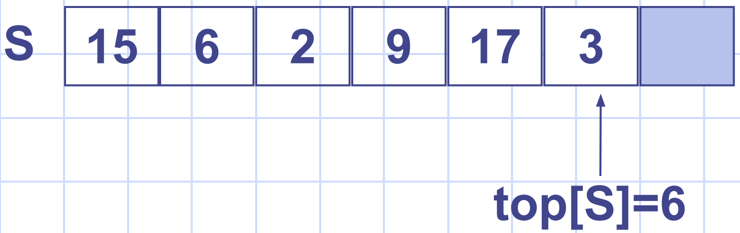
top[S]=4

top[S]=5

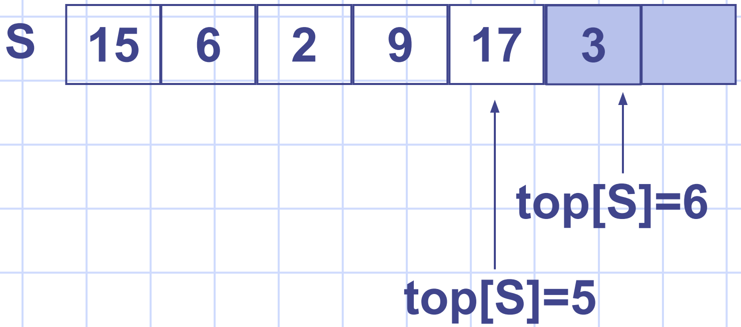
top[S] ← top[S] + 1 (top[S] = 5)

S[top[S]] ← x (S[5] = 17)

Illustration of POP



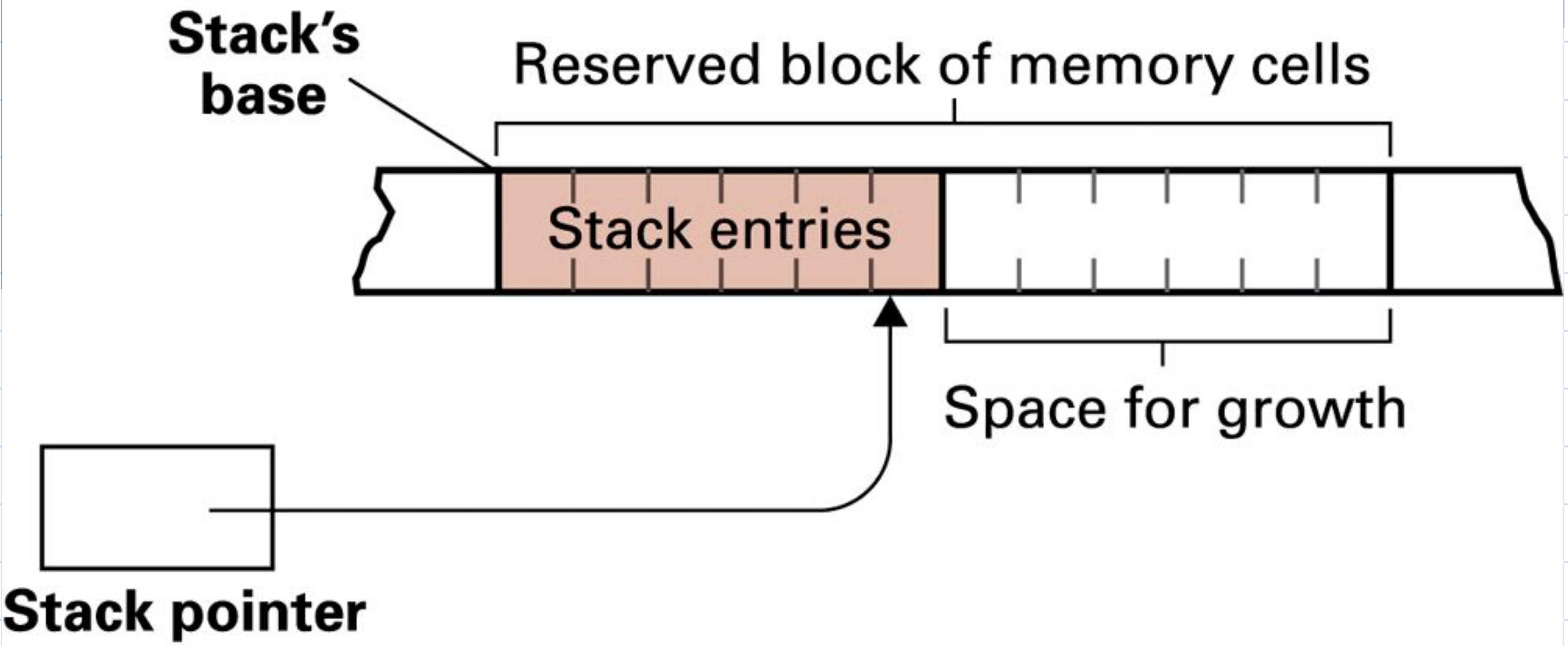
POP(S)



top[S] ← top[S] - 1 (top[S] = 5)

return S[top[S]+1] (return S[6])

A stack in memory



Implementing a Stack

- There are two ways we can implement a stack:
  - Using an array
  - Using a linked list

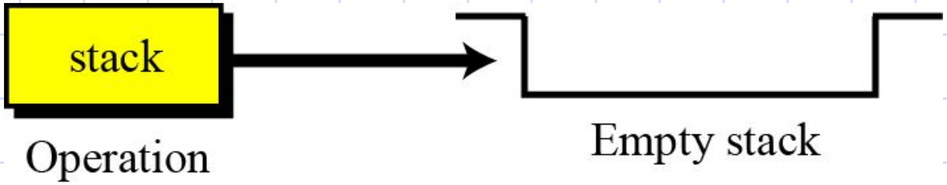
Operations on stacks

There are four basic operations, stack, push, pop and empty.

The **stack** operation

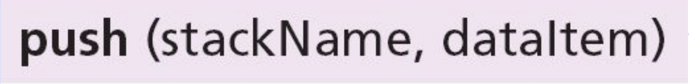
The stack operation creates an empty stack. The following shows the format.

```
stack (stackName)
```

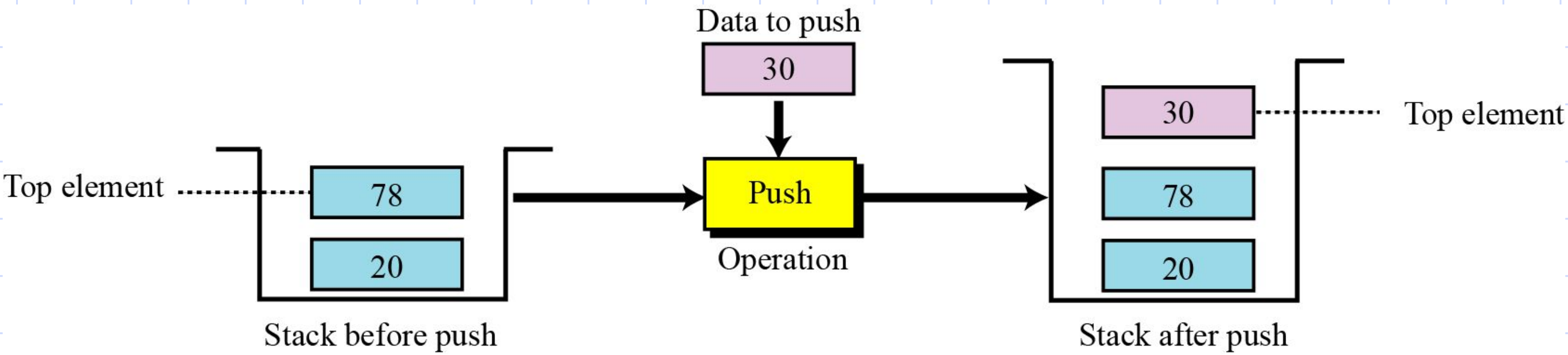


The *push* operation

The *push* operation inserts an item at the top of the stack. The following shows the format.



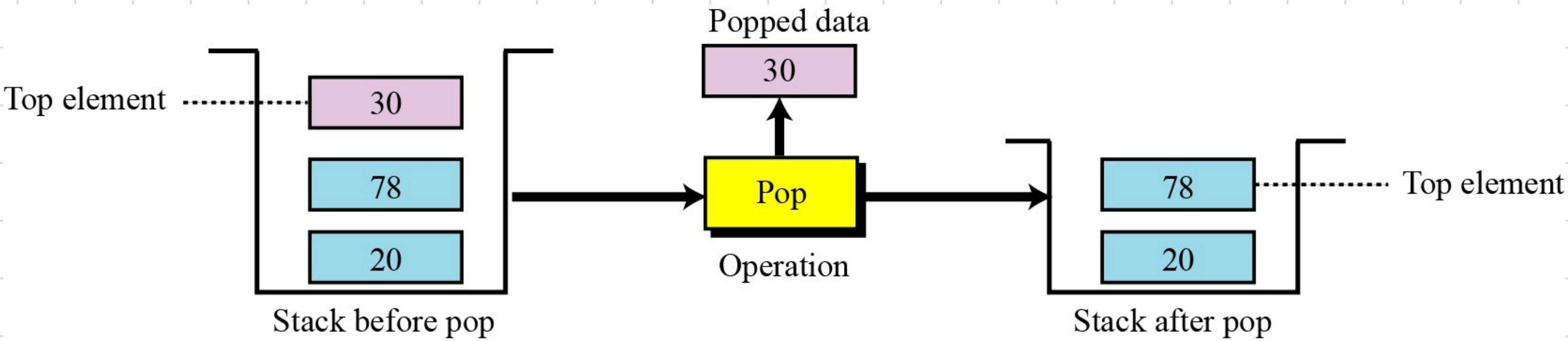
```
push (stackName, dataItem)
```



The *pop* operation

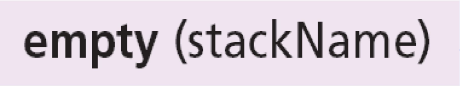
The *pop* operation deletes the item at the top of the stack. The following shows the format.

pop (stackName, dataItem)



The *empty* operation

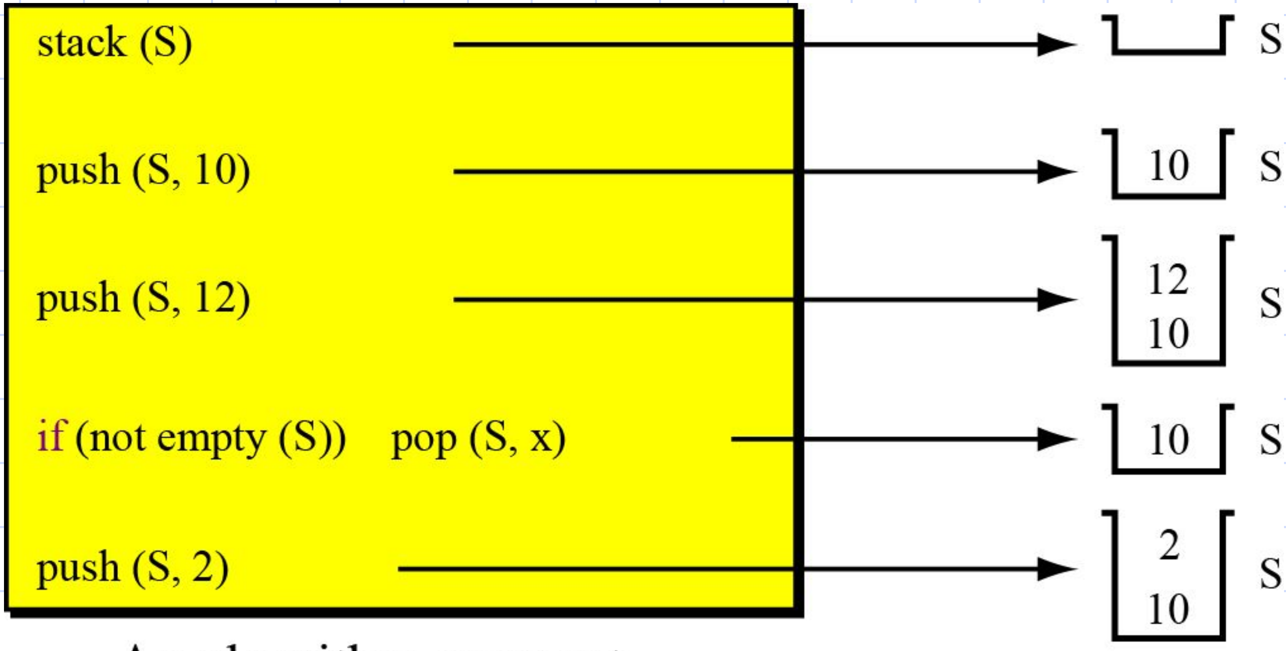
The *empty* operation checks the status of the stack. The following shows the format.



```
empty (stackName)
```

This operation returns true if the stack is empty and false if the stack is not empty.

Stack Operations



An algorithm segment

Stack Operations

- The Push (S, TOP, X) operation(top=0)

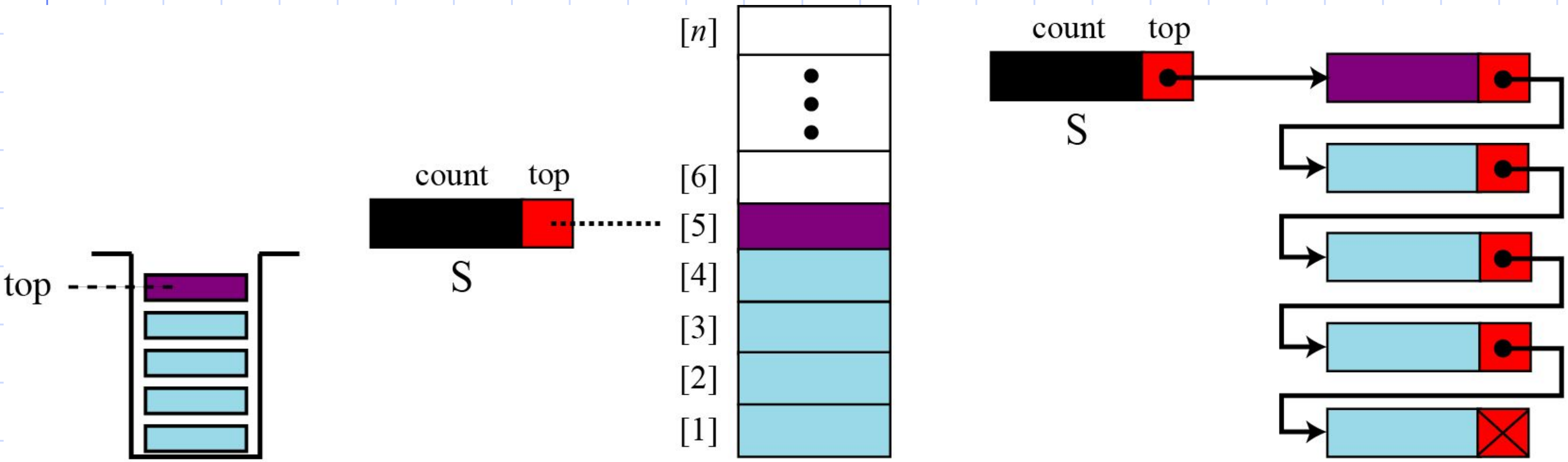
```
1.   [Check for Stack overflow]
   if TOP >= N
          then Write (‘Stack Overflow’)
          Return
2. [Increment TOP]
     TOP  TOP + 1
3.[Insert Element]
     S[TOP]  x
4. [Finished]
     Return
```

Stack Operations

- The POP (S, TOP) operation(top=0)

```
1.   [Check for underflow on Stack]
     if TOP = 0
             then Write (‘Stack Underflow on POP’)
             take action in response to underflow
             Exit
2. [Decrement pointer]
      TOP  TOP - 1
3.[Return former top element of stack]
      Return (S[TOP+1])
```

Stack Implementation



a. ADT

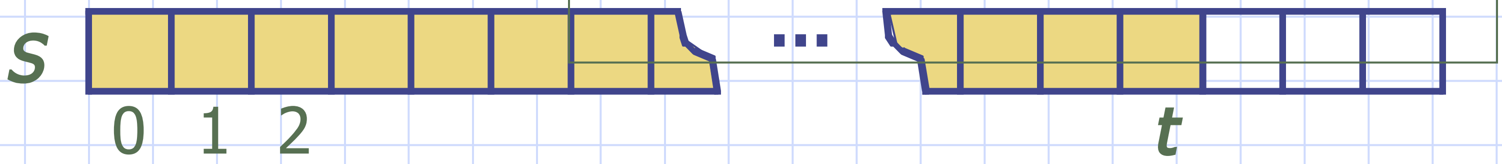
b. Array implementation

c. Linked list implemenation

Array-based Stack

- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable t keeps track of the index of the top element (size is t+1)

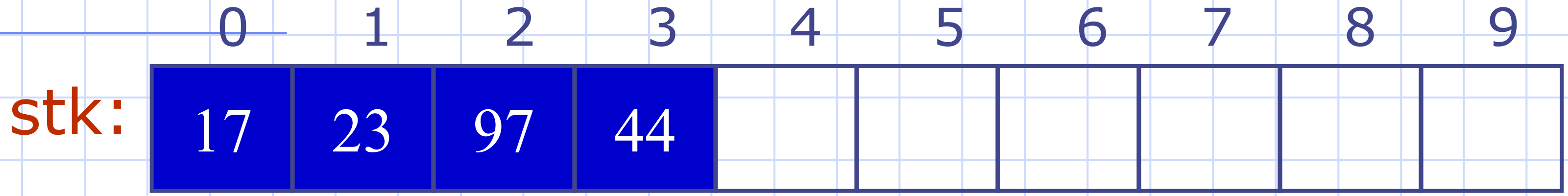
```
Algorithm pop():
    if isEmpty() then
        Error Underflow
    else
        $t \leftarrow t - 1$
        return $S[t + 1]$

Algorithm push(o)
    if $t = S.length - 1$ then
        Error OverFlow
    else
        $t \leftarrow t + 1$
        $S[t] \leftarrow o$
```



Array implementation of stacks

- To implement a stack, items are inserted and removed at the same end (called the top)
- To use an array to implement a stack, you need both the array itself and an integer
- The integer tells you either:
  - Which location is currently the top of the stack, or
  - How many elements are in the stack

Pushing and popping

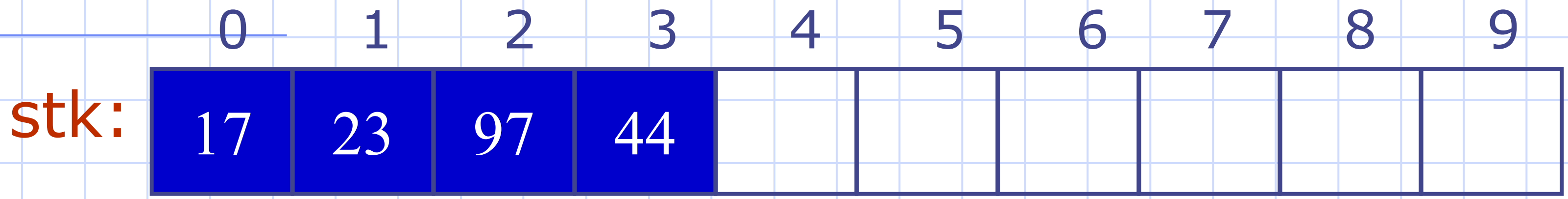


	0	1	2	3	4	5	6	7	8	9
stk:	17	23	97	44						

top = 3 or count = 4

- If the bottom of the stack is at location 0, then an empty stack is represented by top = -1 or count = 0
- To add (push) an element, either:
  - Increment top and store the element in stk[top], or
  - //Store the element in stk[top] and increment count
- To remove (pop) an element, either:
  - Get the element from stk[top] and decrement top, or

After popping



	0	1	2	3	4	5	6	7	8	9
stk:	17	23	97	44						

top = 2 or count = 3

- When you pop an element, do you just leave the "deleted" element sitting in the array?
- The surprising answer is, "*it depends*"
  - If this is an array of primitives, *or* if you are programming in C or C++, *then* doing anything more is just a waste of time
  - If you are programming in Java, and the array contains objects, you should set the "deleted" array element to null
  - Why? To allow it to be garbage collected!

Sharing space

- Of course, the bottom of the stack could be at the *other* end

	0	1	2	3	4	5	6	7	8	9
stk:							44	97	23	17

top = 6 or count = 4

- Sometimes this is done to allow two stacks to share the *same storage area*

	0	1	2	3	4	5	6	7	8	9
stks:	49	57	3				44	97	23	17

topStk1 = 2 topStk2 = 6

Error checking

- There are two stack errors that can occur:
  - Underflow: trying to pop (or peek at) an empty stack
  - Overflow: trying to push onto an already full stack

Stack Operations

- The PEEP (S, TOP, I) operation(top=0)

```
1.  [Check for underflow on Stack]
    if TOP-I+1 <= 0
            then Write ('Stack Underflow on PEEP')
            take action in response to underflow
            Exit
2. [Return Ith element from top of stack
    Return (S[TOP-I+1])
```

Stack Operations

- The CHANGE (S, TOP,X, I) operation

  (top=0)

```
1.    [Check for underflow on Stack]
   if TOP-I+1 <= 0
          then Write ('Stack Underflow on CHANGE')
          Return
 2. [Change the Ith element from top of stack]
      S[TOP-I+1] ← X
 3. [Finished]
      Return
```

Stack_Arr.c

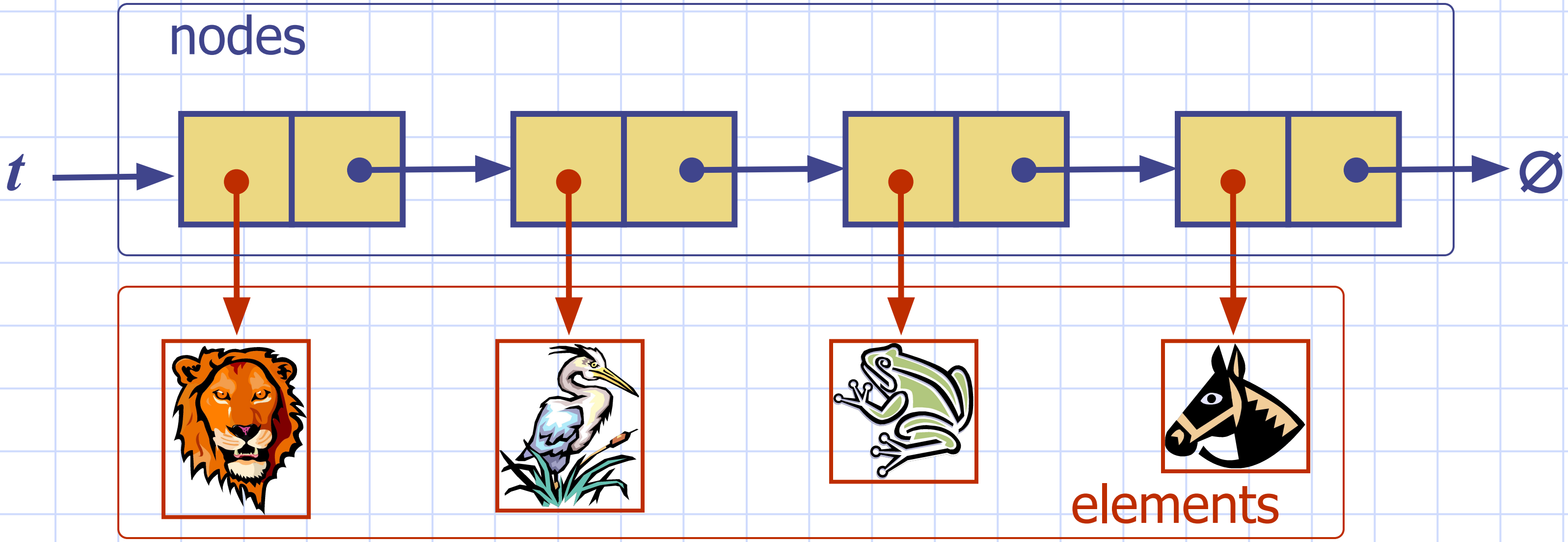
Structure of stack

```
struct stack
 { int top;
    item[Size];
  };
```

Stack.c

Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Evaluating Expressions

Infix, Prefix and Postfix Notations

Mathematical Calculations

- What is 3 + 2 * 4?   2 * 4 + 3?   3 * 2 + 4?

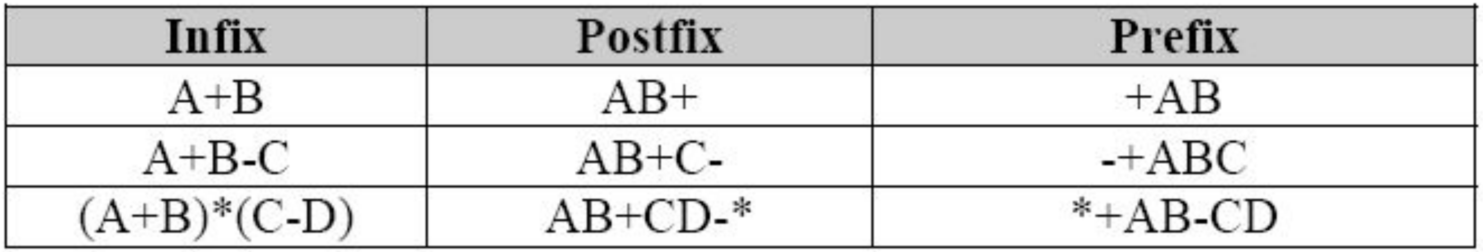
  The precedence of operators affects the order of operations. A mathematical expression cannot simply be evaluated left to right.

- A challenge when evaluating a program.

- *Lexical analysis* is the process of interpreting a program.

- In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it.
- A common technique is to convert a infix notation into postfix notation, then evaluating it.

INFIX, POSTFIX AND PREFIX NOTATIONS



Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

Its denote relative position of the operators(+,*,^,-,/) with respect to two operands.

Infix notation is known as Polish notation. , operator symbol is used in-between of operand.
Post fix notation :- operator is placed after operands.
Pre fix notation : operator is placed before operands

- Fundamental property of Polish notation is that order in which operation are to be performed is completely determined by the position of operators.
- Postfix notation is knows as Reverse Polish notation.
- Infix, Postfix and Prefix notations are used in many calculators.
- The easiest way to implement the Postfix and Prefix operations is to use stack.
- Infix and prefix notations can be converted to postfix notation using stack.
- The reason why postfix notation is preferred is that you don’t need any parenthesis and there is no precedence problem.

Arithmetic Priority

Parentheses() -- 1
Exponent(^,$,|) – 2
Multiplication/ division – 3
Addition/Substraction -- 4

In case of equal precedence/priority of operators ,expression solved from left to right.

<table>
<tr><th>Infix</th><th>Postfix</th><th>Prefix</th></tr>
<tr><td>A+B*C</td><td>ABC*+</td><td>+A*BC</td></tr>
<tr><td>((A-(B+C))*D)|(E+F)</td><td>ABC+-D*EF+|</td><td>|*-A+BCD+EF</td></tr>
<tr><td>(A+B)*(C-D)$E*F</td><td>AB+CD-E$*F*</td><td>**+AB$-CDEF</td></tr>
</table>

Example(((A-(B+C))*D)|(E+F)

- Post fix

((A-BC+)*D)|(E+F)

(ABC+- * D)|(E+F)

ABC+-D* | (E+F)

ABC+-D* | EF+

ABC+-D*EF+|

- Prefix

((A - +BC)*D)|(E+F)

(-A+BC * D) |(EF)

*-A+BCD |(E+F)

*-A+BCD | +EF

|*-A+BCD+EF

POSTFIX Evaluation

- In ***Postfix*** notation the expression is scanned from left to right.
- When a number is seen it is pushed onto the stack;
- when an operator is seen the operator is applied to the two numbers popped from the stack and the result is pushed back to the stack.

PostFix Examples

Infix	Postfix	Evaluation
2 - 3 * 4 + 5	2 3 4 * - 5 +	-5
(2 - 3) * (4 + 5)	2 3 - 4 5 + *	-9
2- (3 * 4 +5)	2 3 4 * 5 + -	-15

Stack example: postfix notation

- Postfix notation
  - Operations are done by specifying operands, then the operator
  - Example: 2 3 4 + * results in 14
    - Calculates 2 * (3 + 4)
- Postfix evalution implemented with a stack
  - When we see a operand (number), push it on the stack
  - When we see an operator
    - Pop the appropriate number of operands off the stack
    - Do the calculation
    - Push the result back onto the stack
  - At the end, the stack should have the (one) result of the calculation

Postfix Evaluation

Operand: push

Operator: pop 2 operands, do the math, pop result back onto stack

1 2 3 + *

	Postfix	Stack(bot -> top)
a)	1 2 3 + *	
b)	2 3 + *	1
c)	3 + *	1 2
d)	+ *	1 2 3
e)	*	1 5 // 5 from 2 + 3
f)		5 // 5 from 1 * 5

Evaluation of postfix expressions

Steps

given the postfix expression, get a token and:

1) If the token is an operand, push its value on the (value) stack.
2) If the token is an operator, pop two values from the stack and apply that operator to them ,perform calculation; then push the result back on the stack.

Example 1: A B * C D E / - +

Let A = 5, B = 3, C = 6, D = 8, E = 2.

	value stack
push (5)	5
push (3)	5 3
push (pop * pop)	15
push (6)	15 6
push (8)	15 6 8
push (2)	15 6 8 2
push (pop / pop)	15 6 4

Infix to Postfix conversion

Infix Expression: 3 + 2 * 4

Postfix Expression:

Operator Stack:

Simple Example

Infix Expression: + 2 * 4

PostFix Expression: 3

Operator Stack:

Simple Example

Infix Expression: 2 * 4

PostFix Expression: 3

Operator Stack: +

Simple Example

Infix Expression: * 4

PostFix Expression: 3 2

Operator Stack: +

Simple Example

Infix Expression: 4

PostFix Expression: 3 2

Operator Stack: + *

Simple Example

Infix Expression:

PostFix Expression: 3 2 4

Operator Stack: + *

Simple Example

Infix Expression:

PostFix Expression: 3 2 4 *

Operator Stack: +

Simple Example

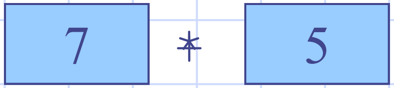
Infix Expression:

PostFix Expression: 3 2 4 * +

Operator Stack:

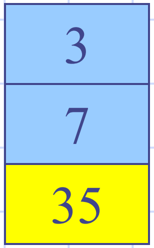
More on postfix notation

- Calculate 5 * (4 + 3)
- Numbers orderer 5 4 3
- Operands ordered + *
  - Note reverse order!
  - Must compute + first!
- See example at right



7 * 5

5 4 3 + *



3

7

35

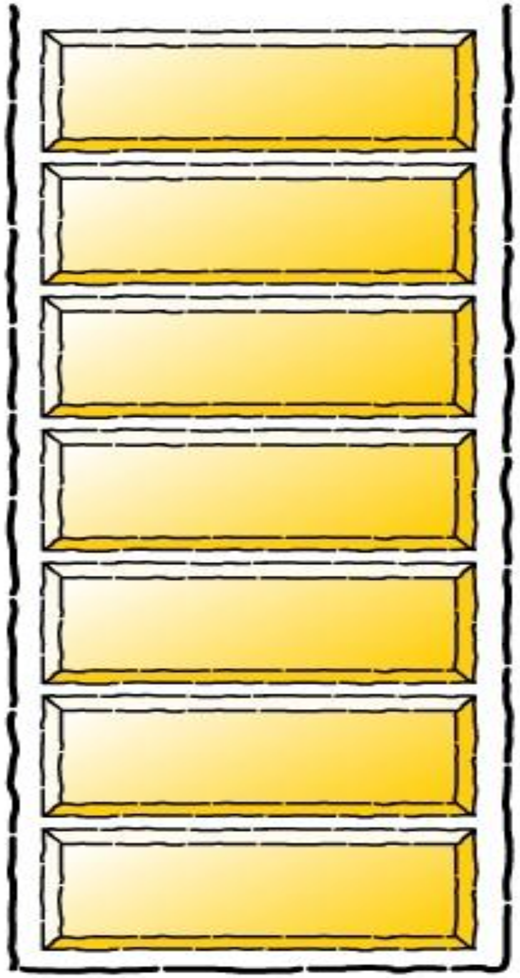
Infix to postfix Conversion

- Rule

1. Push “(“onto Stack, and add “)” to the end of infix expression.
2. Scan infix expression 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 postfix expression.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to postfix expression each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
       [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to postfix expression each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
       [End of If]
7. END

Infix to postfix conversion

Stack

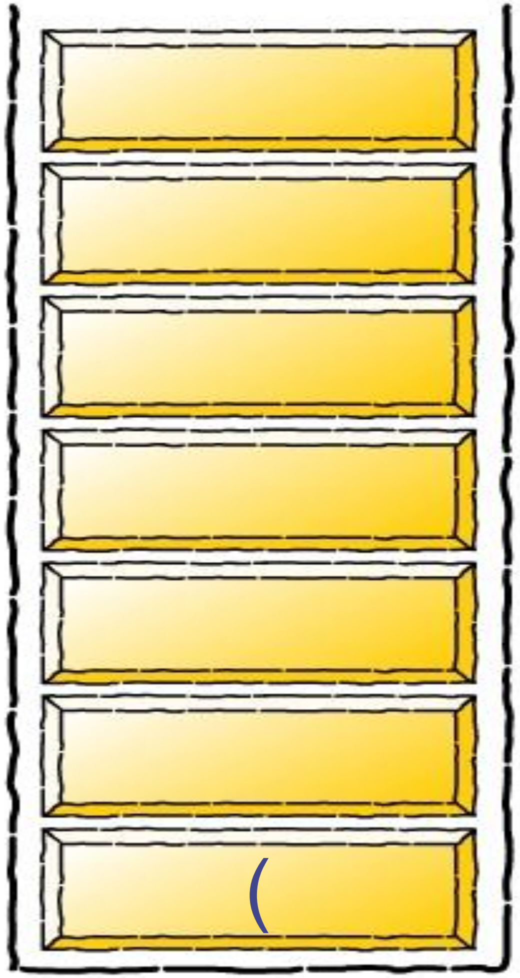


Infix Expression

(a + b - c) * d – (e + f)

Postfix Expression

Stack

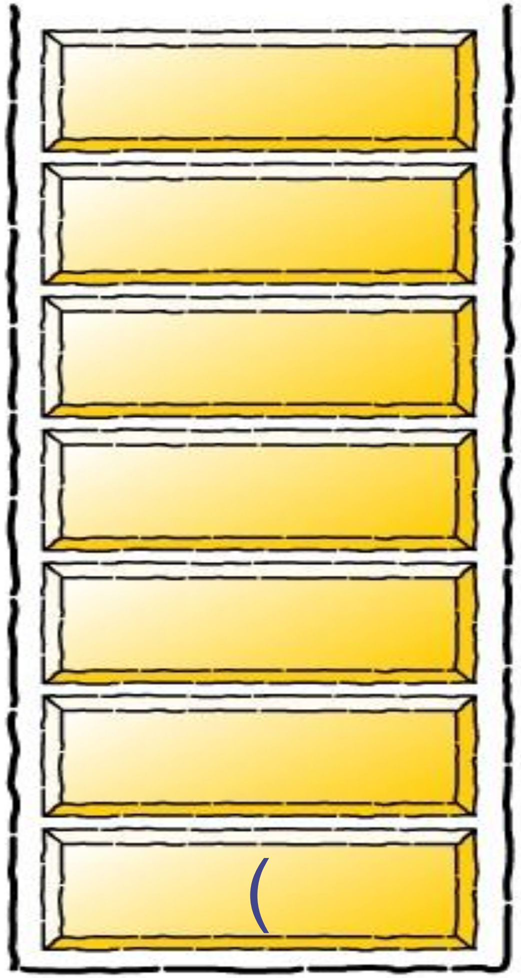


Infix Expression

a + b - c) * d – (e + f)

Postfix Expression

Stack



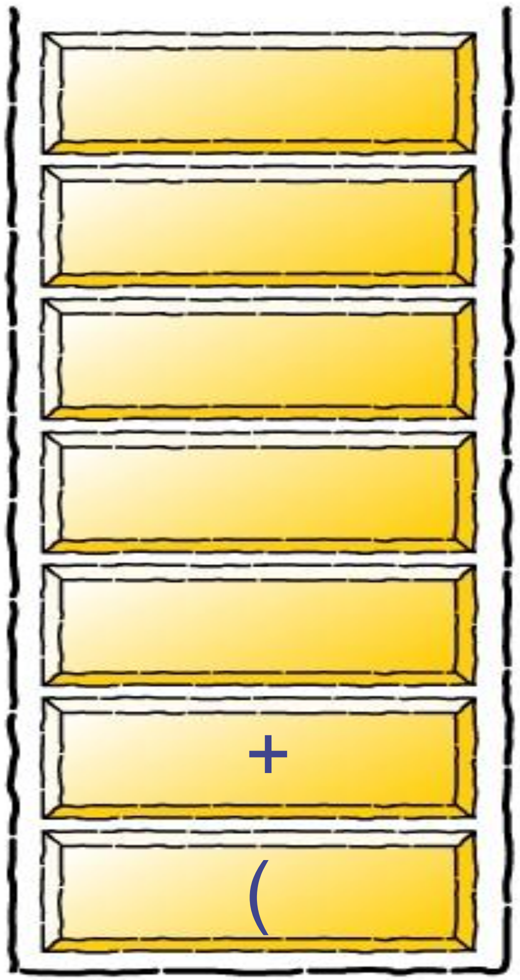
Infix Expression

+ b - c) * d – (e + f)

Postfix Expression

a

Stack



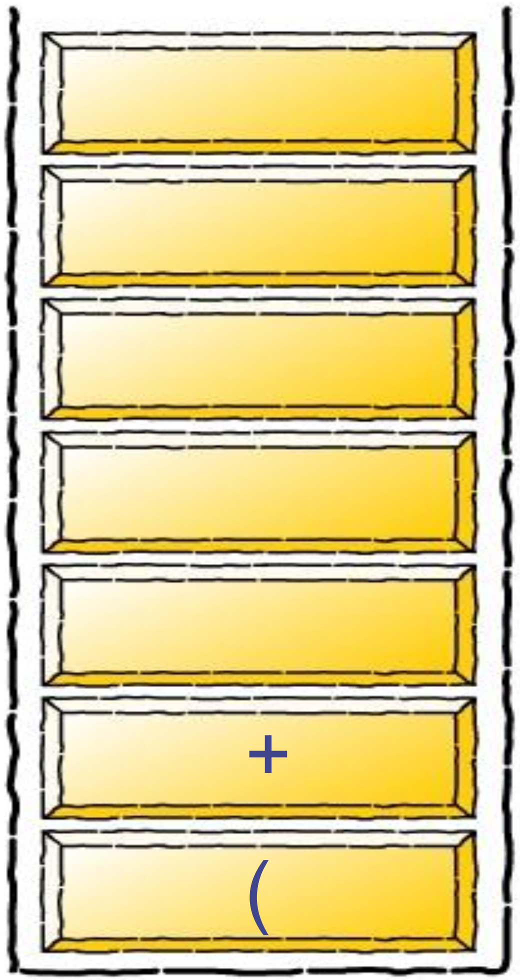
Infix Expression

b - c) * d – (e + f)

Postfix Expression

a

Stack



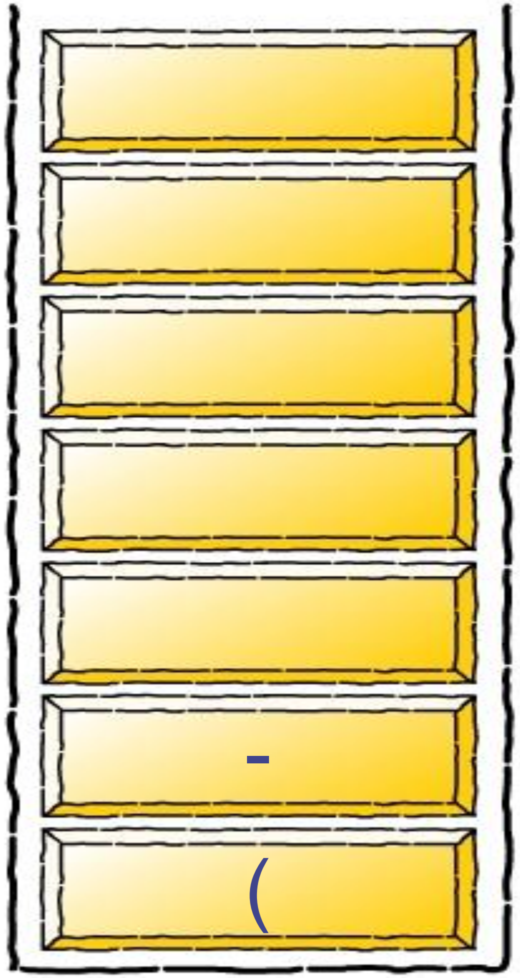
Infix Expression

- c) * d – (e + f)

Postfix Expression

a b

Stack



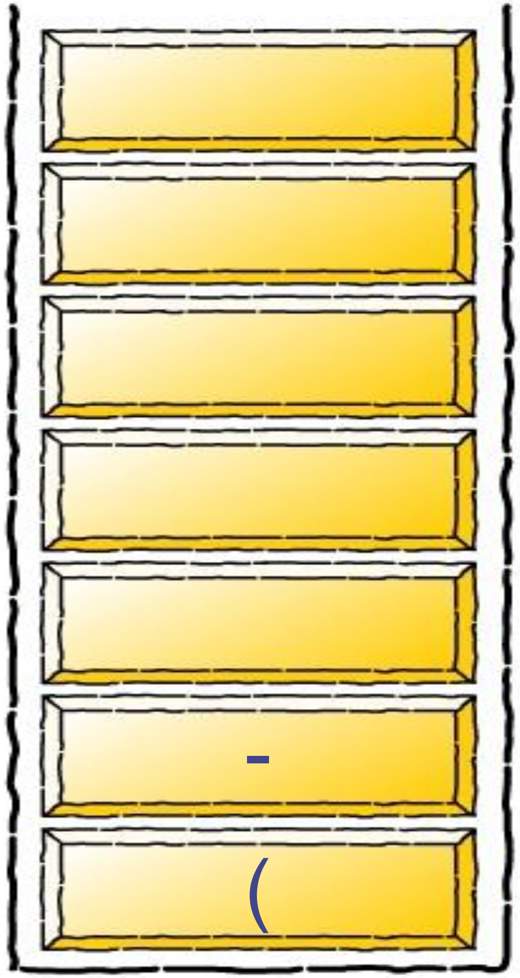
Infix Expression

c) * d – (e + f)

Postfix Expression

a b +

Stack



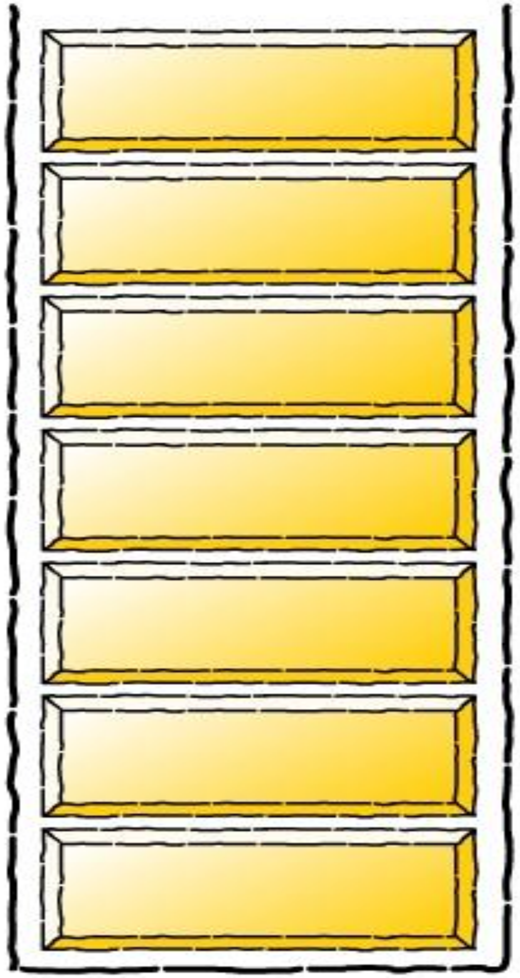
Infix Expression

) * d – (e + f)

Postfix Expression

a b + c

Stack



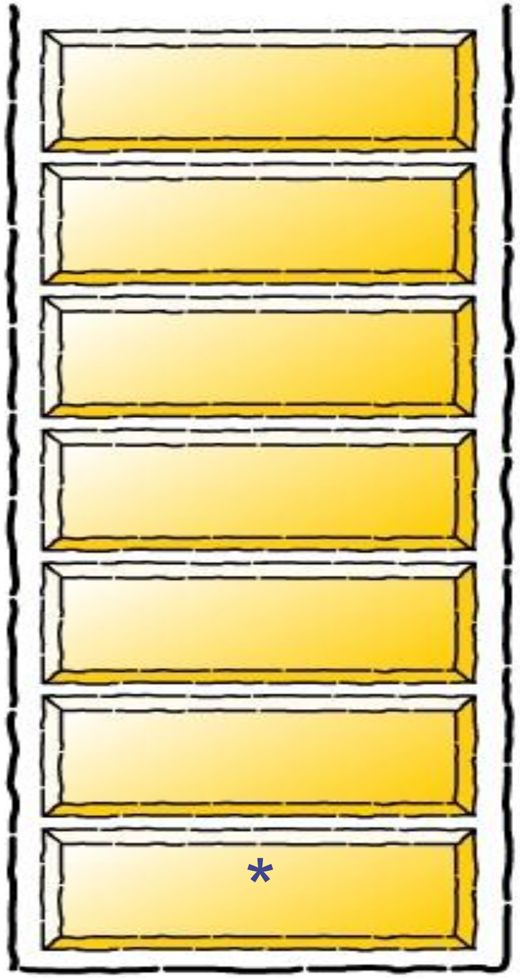
Infix Expression

* d – (e + f)

Postfix Expression

a b + c -

Stack



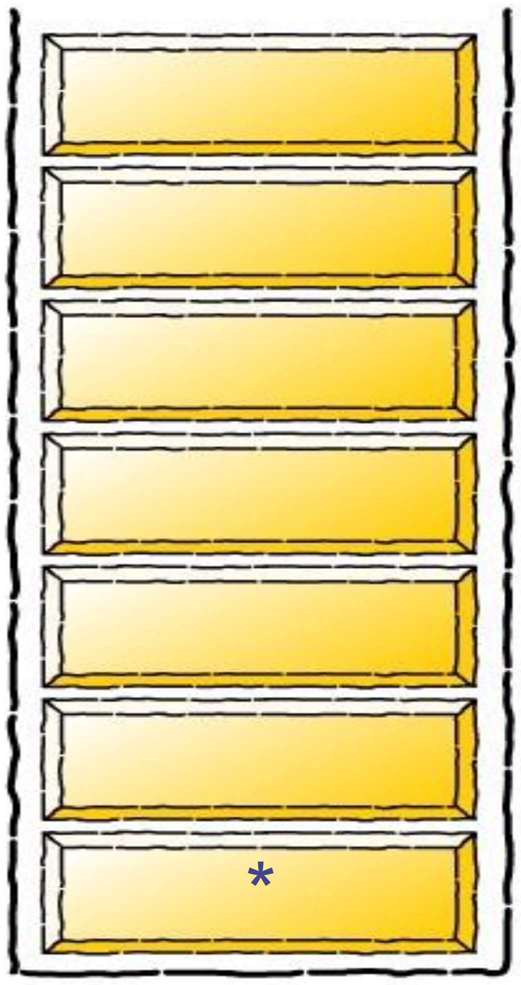
Infix Expression

d – (e + f)

Postfix Expression

a b + c -

Stack



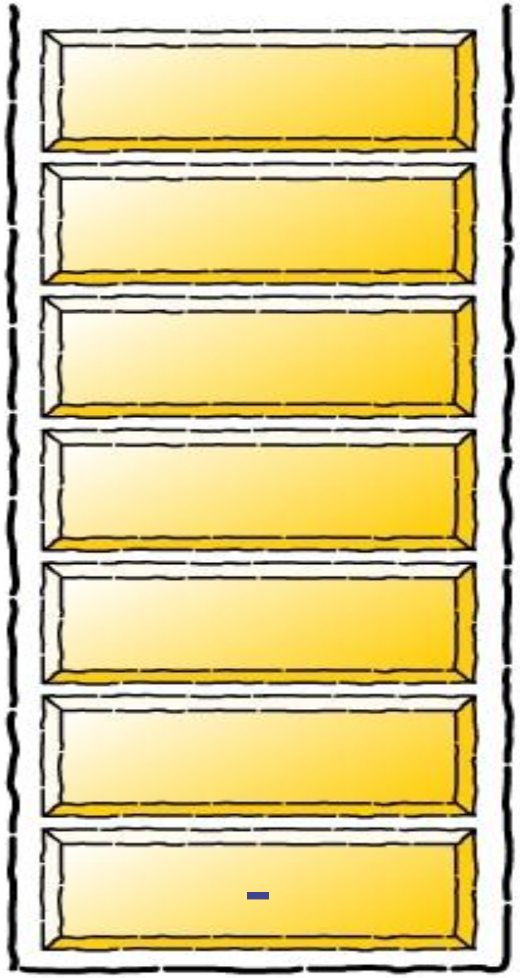
Infix Expression

– (e + f)

Postfix Expression

a b + c - d

Stack



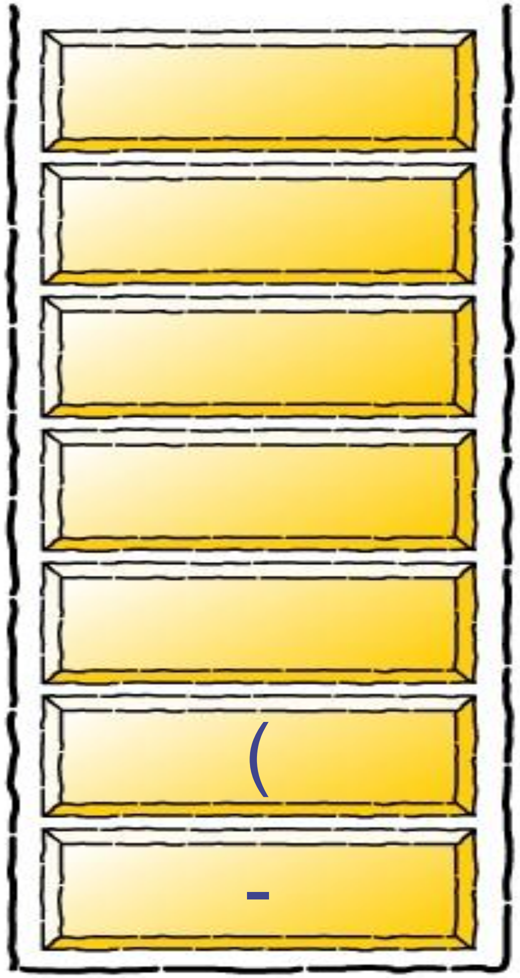
Infix Expression

(e + f)

Postfix Expression

a b + c – d *

Stack



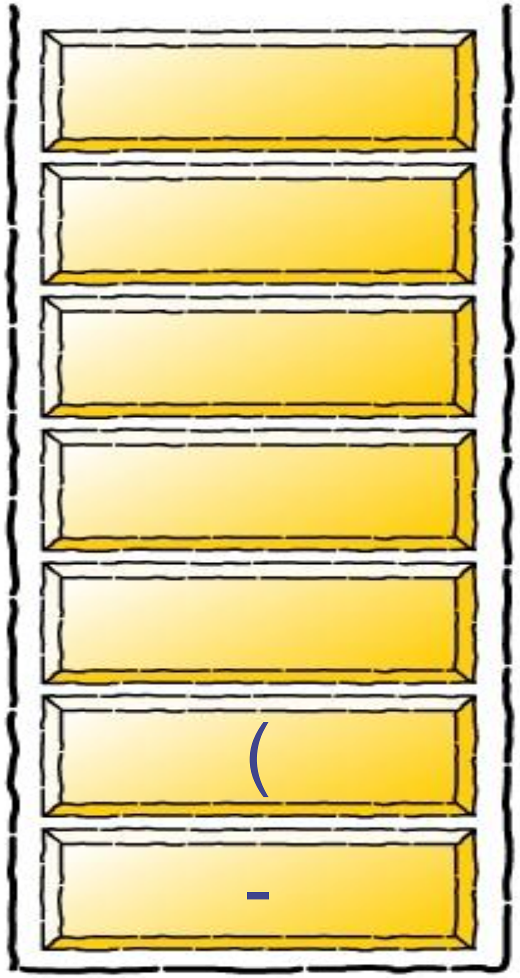
Infix Expression

e + f)

Postfix Expression

a b + c – d *

Stack



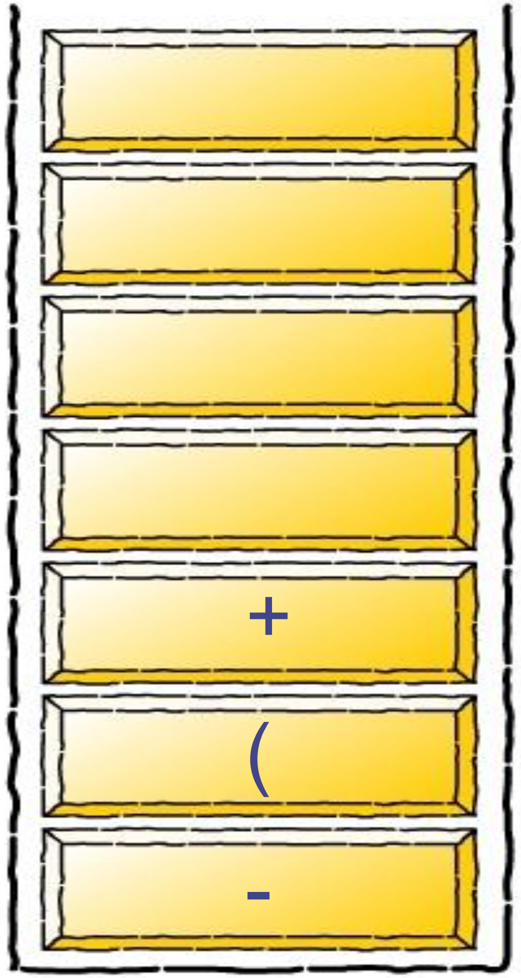
Infix Expression

+ f)

Postfix Expression

a b + c – d * e

Stack



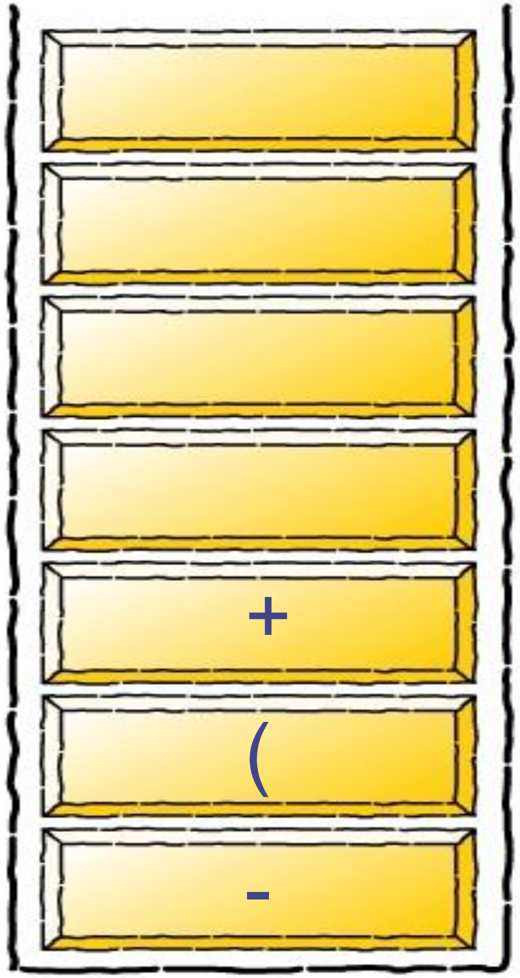
Infix Expression

f)

Postfix Expression

a b + c – d * e

Stack



+

(

-

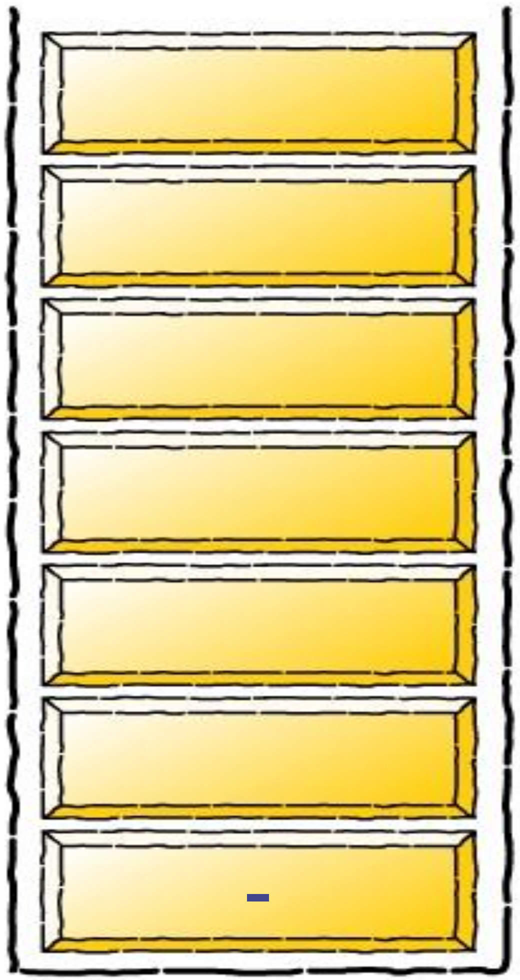
Infix Expression

)

Postfix Expression

a b + c – d * e f

Stack

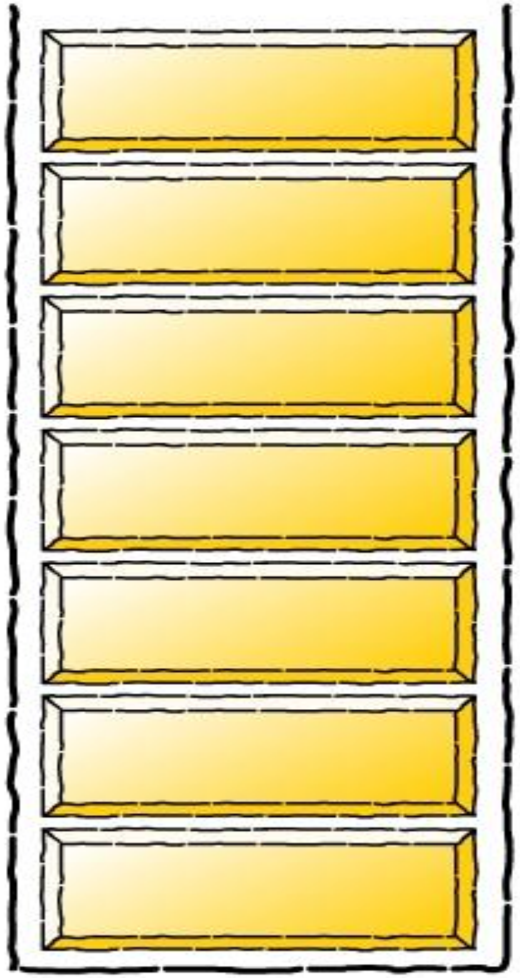


Infix Expression

Postfix Expression

a b + c – d * e f +

Stack



Infix Expression

Postfix Expression

a b + c – d * e f + -

Inf_pst.c

Example : Translate the following infix form to a postfix form: A * B + (C - D / E)

Infix queue	Operator stack	Postfix queue
A		A
*	*	A
B	*	A B
+	+	A B *
(	(+	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 / -
		A B * C D E / - +

Converting Expression from Infix to Prefix (Algorithm)

1) Reverse the input string.
2) Examine the next element in the input.
3) If it is operand, add it to output string.
4) If it is Closing parenthesis, push it on stack.
5) If it is an operator, then
   i) If stack is empty, push operator on stack.
   ii) If the top of stack is closing parenthesis, push operator on stack.
   iii) If it has same or higher priority than the top of stack, push operator on stack.
   iv) Else pop the operator from the stack and add it to output string, repeat step 5.
6) If it is a opening parenthesis, pop operators from stack and add them to output string until a closing parenthesis is encountered. Pop and discard the closing parenthesis.
7) If there is more input go to step 2
8) If there is no more input, unstack the remaining operators and add them to output string.
9) Reverse the output string.

Example

- Suppose we want to convert 2*3/(2-1)+5*(4-1) into Prefix form:

  Reversed Expression:)1-4(*5+)1-2(/3*2

Char Scanned	Stack Contents (Top on right)	Prefix Expression (right to left)
)	)	
1	)	1
-	)-	1
4	)-	14
(	Empty	14-
*	*	14-
5	*	14-5
+	+	14-5*
)	+)	14-5*

1	+)	14-5*1
-	+)-	14-5*1
2	+)-	14-5*12
(	+	14-5*12-
/	+/	14-5*12-
3	+/	14-5*12-3
*	+/*	14-5*12-3
2	+/*	14-5*12-32
	Empty	14-5*12-32*/+

- Reverse the output string : +/*23-21*5-41
- So, the final Prefix Expression is +/*23-21*5-41

Inf_pre.c

End