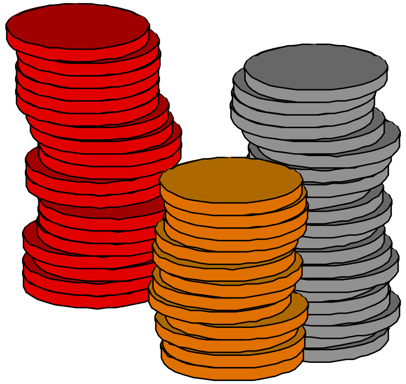
Stacks

Stacks

- ADT
- Implementation(s)
- Applications
  - Checking for Balanced Parentheses, Brackets, and Braces in an Infix Algebraic Expression
  - Transforming an Infix Expression to a Postfix Expression
  - Evaluating Postfix Expressions
  - Evaluating Infix Expressions

What is a stack?

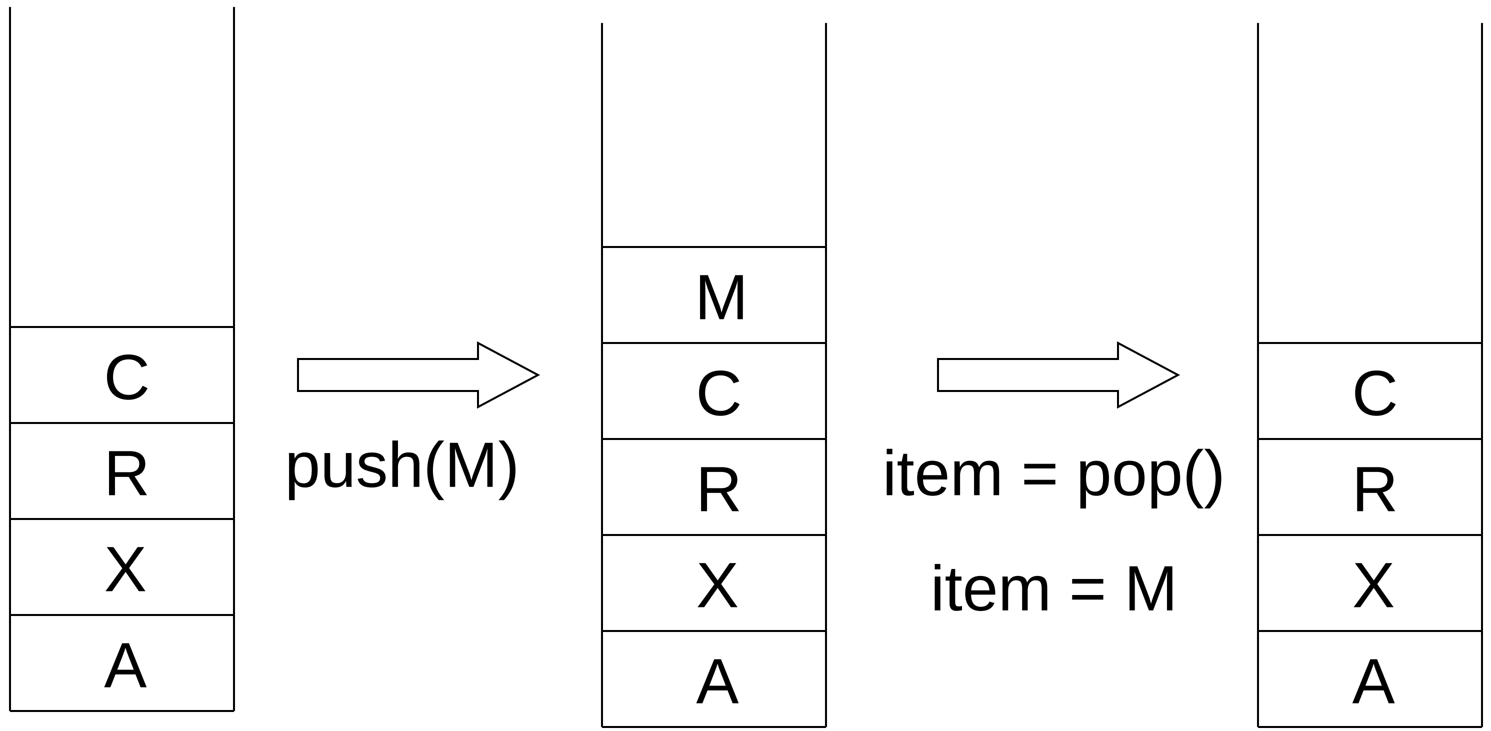
- It is an ordered group of homogeneous items
- Stack principle: LAST IN FIRST OUT = LIFO
  - It means: the last element inserted is the first one to be removed
- Only access to the stack is the top element
  - consider trays in a cafeteria
    - to get the bottom tray out, you must first remove all the elements above



Stack

- *Push*
  - the operation to place a new item at the top of the stack
- *Pop*
  - the operation to remove the next item from the top of the stack

Stack



Stack Applications

- Real life
  - Pile of books
  - Plate trays
- More applications related to computer science
  - Program execution stack
  - Evaluating expressions

Array-based Stack Implementation

- Allocate an array of some size (pre-defined)
  - Maximum N elements in stack
- Bottom stack element stored at element 0
- last index in the array is the *top*
- Increment *top* when one element is pushed, decrement after pop

Stack Implementation: CreateS, isEmpty, isFull

```
#define MAX_STACK_SIZE 100 /* maximum stack size */
typedef struct {
        int key;
        /* other fields */
        } element;

element stack[MAX_STACK_SIZE];
int top = -1;
Boolean isEmpty(Stack) { if(top< 0) return FALSE; }
Boolean isFull(Stack) { if (top >= MAX_STACK_SIZE-1) return FALSE;
```

Push

```
void push(int *top, element item)
{
 /* add an item to the global stack */
    if (*top >= MAX_STACK_SIZE-1)  {
        printf ("Stack is Full\n");
        return;            /* returns  error message*/
     }
    *top=*top+1;
    stack[*top] = item;
   }
```

Pop

```
element pop(int *top)
{
 /* return the top element from the stack */
    if (*top == -1)
        printf ("Stack is Empty\n");
         return; /* returns  error message*/
    item= stack[*top];
    *top--;
    return item;
 }
```

Notes on *push()* and *pop()*

- Other ways to do this even if using arrays
  - Keep a *size* variable that tracks how many items in the list
  - Keep a *maxSize* variable that stores the maximum number of elements the stack can hold (size of the array)
    - you would have to do this in a language like C/C++

Implementing a Stack: Linked List

- Advantages:
  - always constant time to push or pop an element
  - can grow to an infinite size
- Disadvantages
  - the common case is the slowest of all the implementations
- Basic implementation
  - list is initially empty
  - *push( )* method adds a new item to the head of the list
  - *pop( )* method removes the head of the list

Additional Notes

- It should appear obvious that linked lists are very well suited for stacks
  - *addHead()* and *deleteHead()* are basically the *push()* and *pop()* methods
- Again, no need for the *isFull()* method
  - list can grow to an infinite size

List-based Stack Implementation: Push

```
void push(element  *top, element item)
  {
    /* add an element to the top of the stack */
        element  *temp =   (element *) malloc (sizeof (node));
        if (IS_FULL(*temp)) {
                fprintf(stderr, " The memory is full\n");
                    /*printf( "The memory is full\n" );*/
                exit(1);
        }
        temp->item = item;
        temp->next= top;
        top= temp;
  }
```

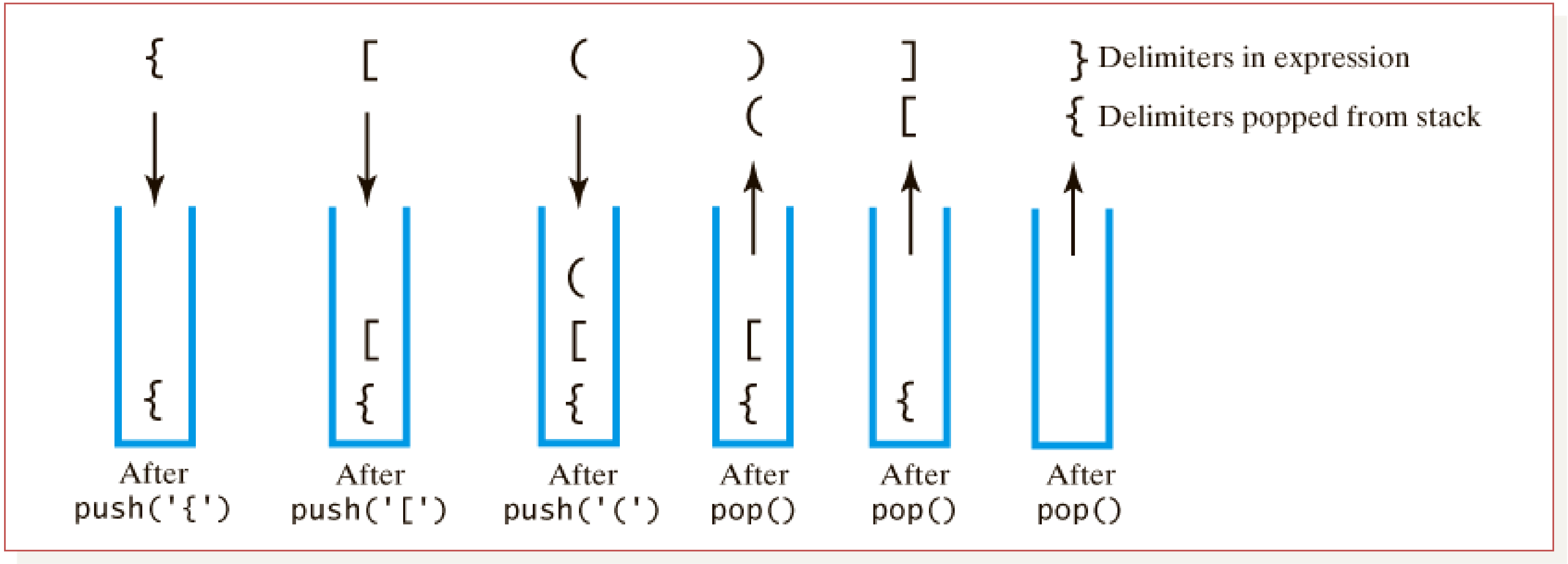
Pop

```
element pop(element *top) {
/* delete an element from the stack */
        element *temp = top;
        element item;
        if (IS_EMPTY(temp))  {
                fprintf(stderr,  “The stack is empty\n”);
                exit(1);
         }
        item = temp->item;
        top = temp->next;
        free(temp);
        return item;
}
```

Algorithm Analysis

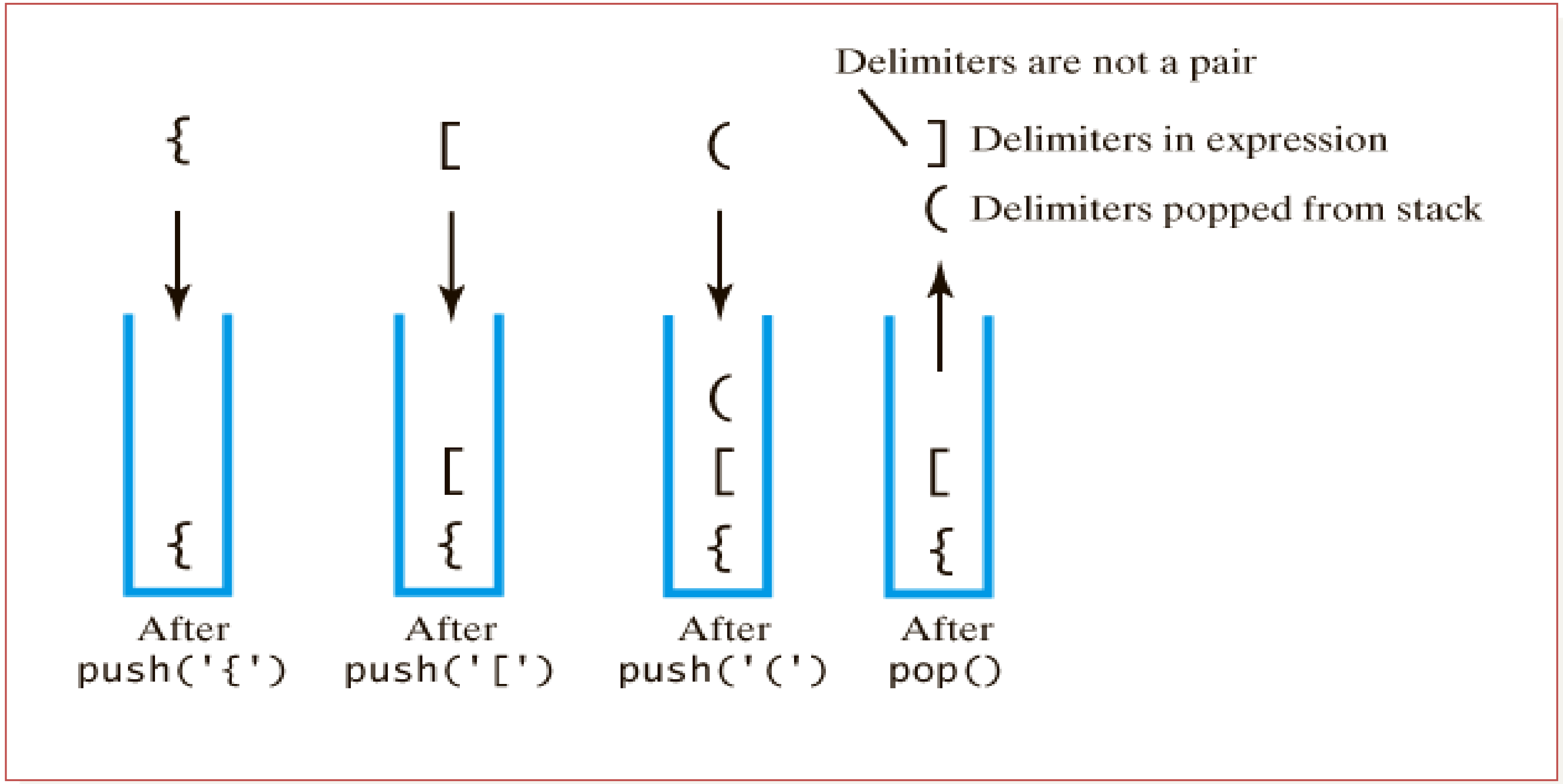
- push O(1)
- pop O(1)
- isEmpty O(1)
- isFull O(1)

Checking for Balanced (), [], {}



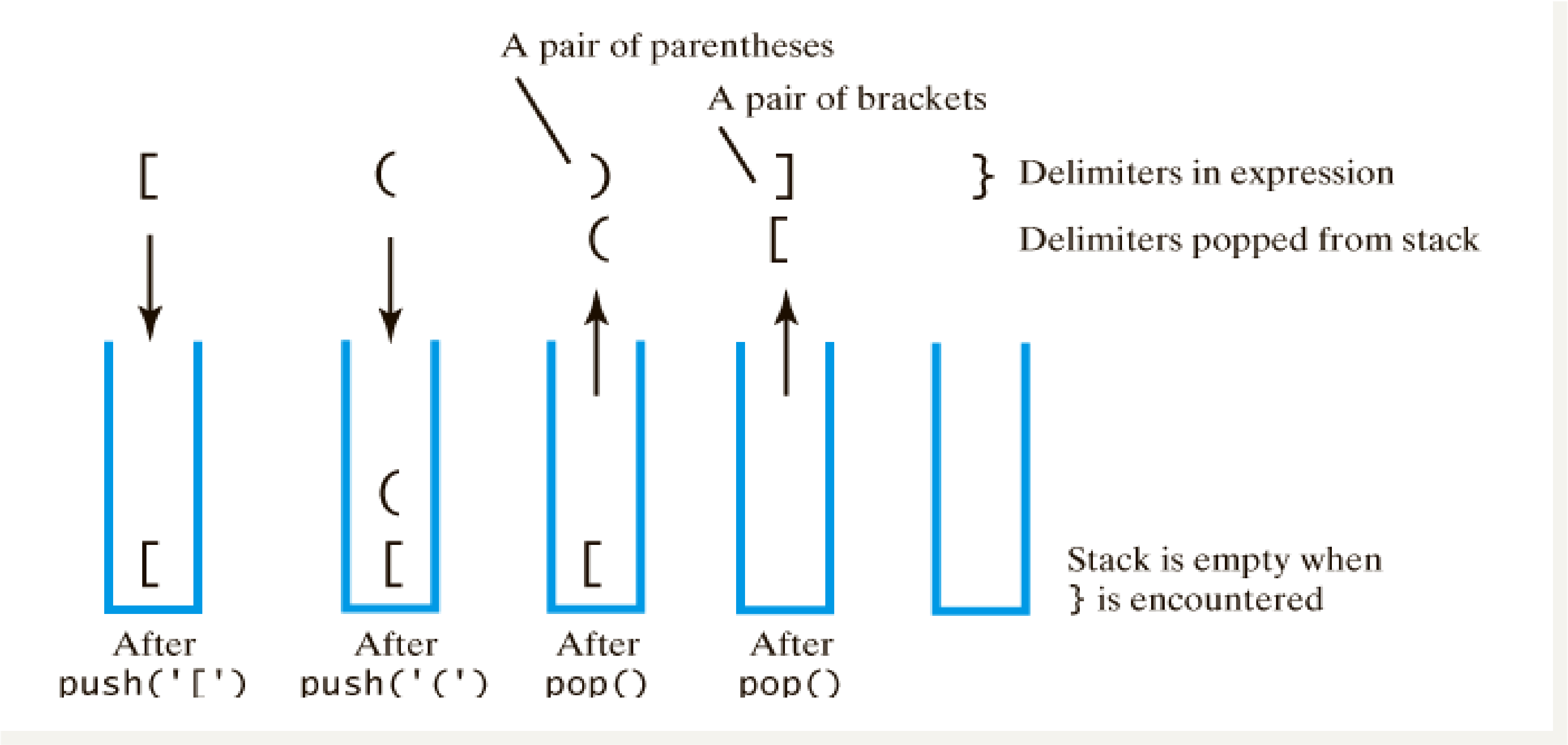
The contents of a stack during the scan of an expression that contains the balanced delimiters { [( )] }

Checking for Balanced `()`, `[]`, `{}`



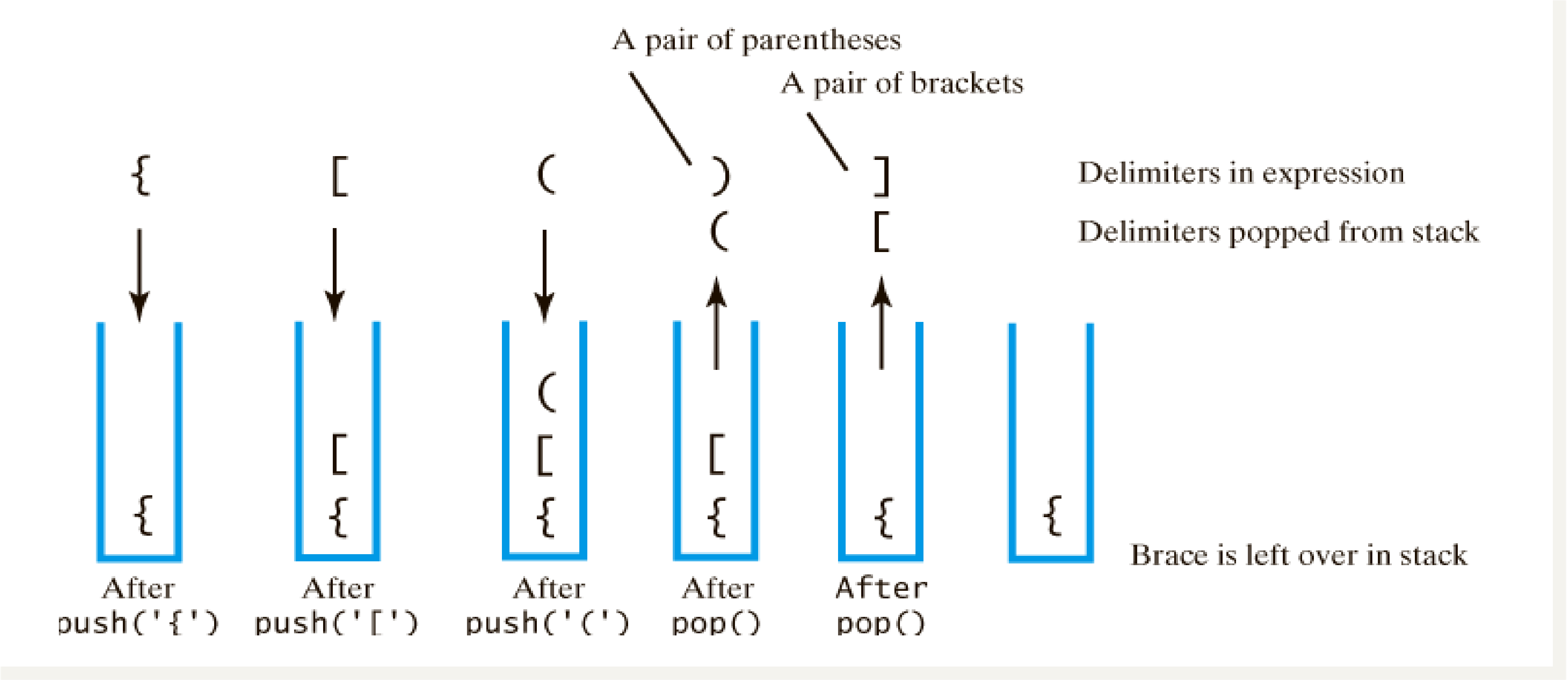
The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{ [ ( ] ) }`

Checking for Balanced (), [], {}



The contents of a stack during the scan of an expression that contains the unbalanced delimiters [()]}

Checking for Balanced `()`, `[]`, `{}`



The contents of a stack during the scan of an expression that contains the unbalanced delimiters `{[()]`

Checking for Balanced (), [], {}

```
Algorithm checkBalance(expression)
// Returns true if the parentheses, brackets, and braces in an expression are paired correctly.
isBalanced = true
while ( (isBalanced == true) and not at end of expression)
{       nextCharacter = next character in expression
        switch (nextCharacter)
        {       case '(': case '[': case '{':
                        Push nextCharacter onto stack
                        break
                 case ')': case ']': case '}':
                        if (stack is empty)  isBalanced = false
                        else
                        {       openDelimiter = Pop  top of stack
                                isBalanced = true or false according to whether openDelimiter and
                                        nextCharacter are a pair of delimiters
                        }
                        break
        } //End of switch case
} // End of While loop
if (stack is not empty)  isBalanced = false
return isBalanced
```

Using a Stack to Process Algebraic Expressions

- Infix expressions
  - Binary operators appear between operands
    - `a+b`   `a*b+c`
- Prefix expressions
  - Binary operators appear before operands
    - `+ab`   `+*abc`
- Postfix expressions
  - Binary operators appear after operands
    - `ab+`   `abc*+`
  - Easier to process – no need for parentheses nor precedence

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	a	
$+$	a	$+$
b	$a\ b$	$+$
$*$	$a\ b$	$+\ *$
c	$a\ b\ c$	$+\ *$
	$a\ b\ c\ *$	$+$
	$a\ b\ c\ *\ +$	

Converting the infix expression `a + b * c` to postfix form

Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	a	
$-$	a	$-$
b	$a\ b$	$-$
$+$	$a\ b\ -$	
	$a\ b\ -$	$+$
c	$a\ b - c$	$+$
	$a\ b - c\ +$	

Converting infix expression to postfix form:

a – b + c

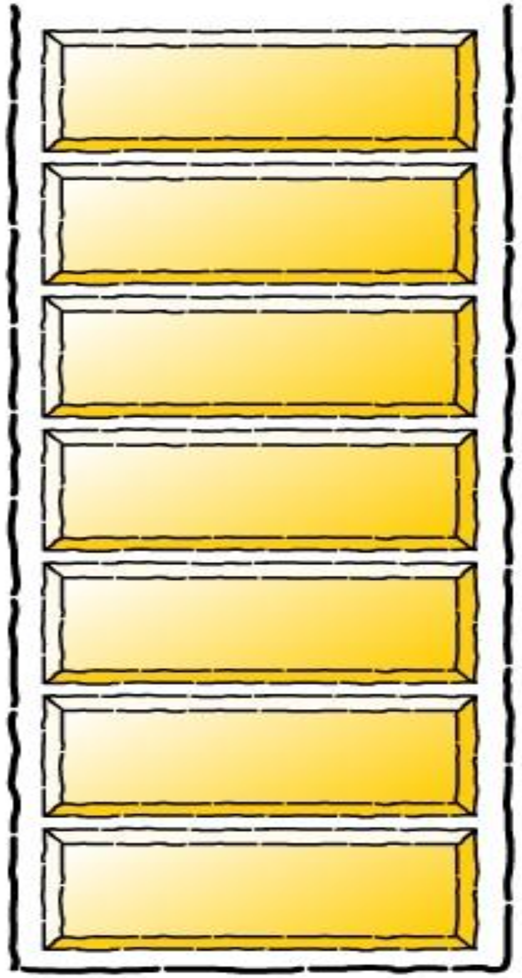
Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	*a*	
^	*a*	^
b	*a b*	^
^	*a b*	^ ^
c	*a b c*	^ ^
	a b c ^	^
	a b c ^ ^	

Converting infix expression to postfix form:

`a ^ b ^ c`

Infix to postfix conversion



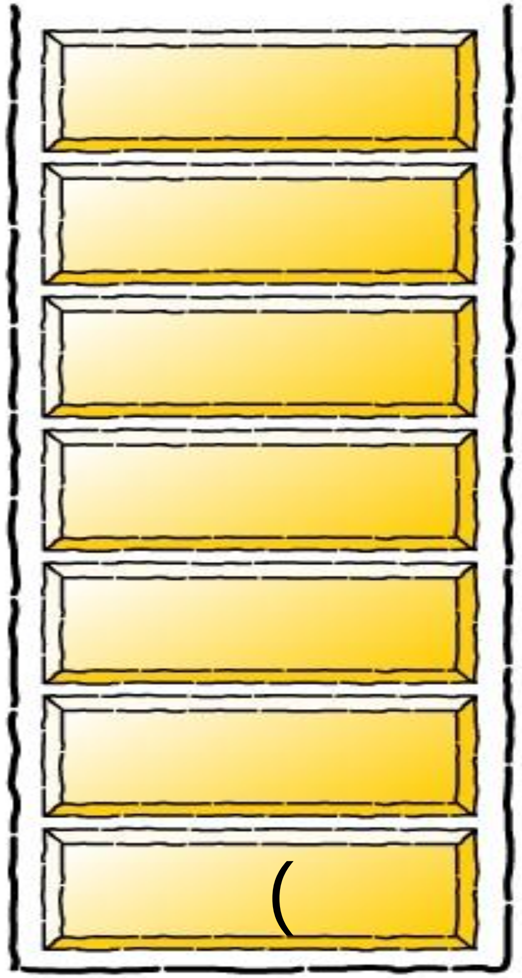
infixVect

(a + b - c) * d – (e + f)

postfixVect

Infix to postfix conversion

stackVect



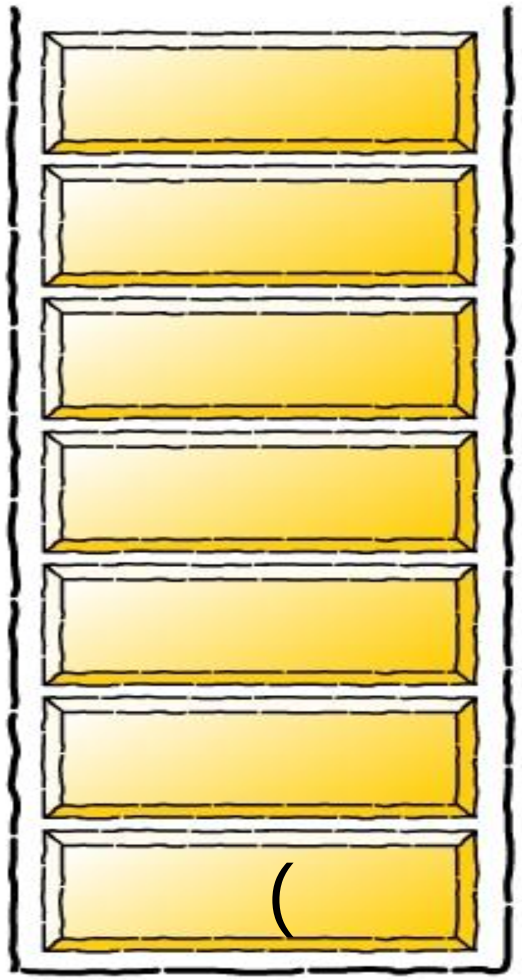
infixVect

a + b - c) * d – (e + f)

postfixVect

Infix to postfix conversion

stackVect



infixVect

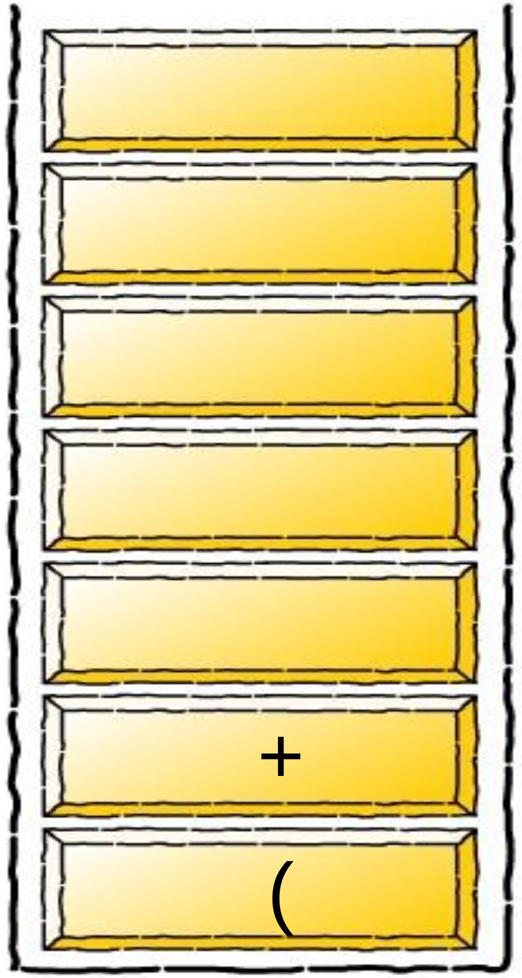
+ b - c) * d – (e + f)

postfixVect

a

Infix to postfix conversion

stackVect



infixVect

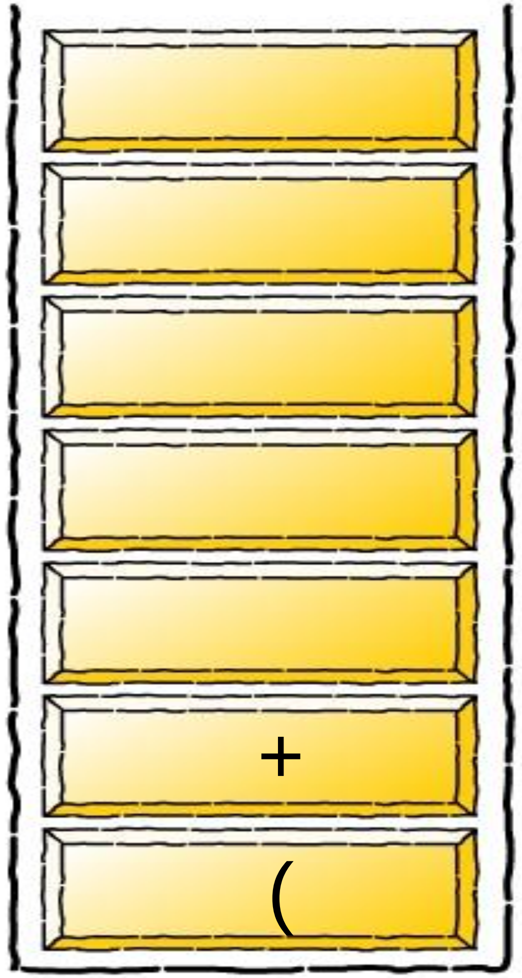
b - c) * d – (e + f)

postfixVect

a

Infix to postfix conversion

stackVect



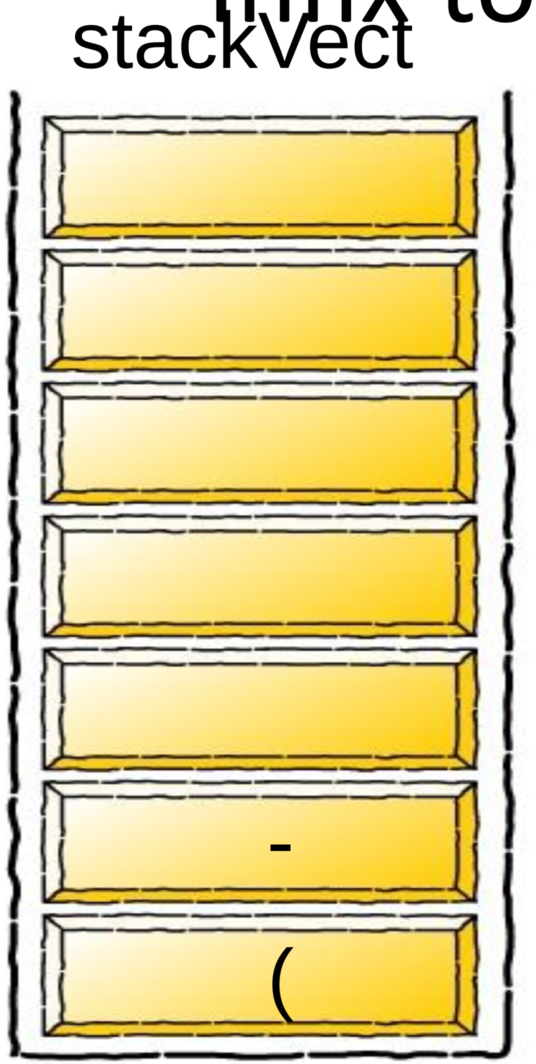
infixVect

- c) * d – (e + f)

postfixVect

a b

Infix to postfix conversion



infixVect

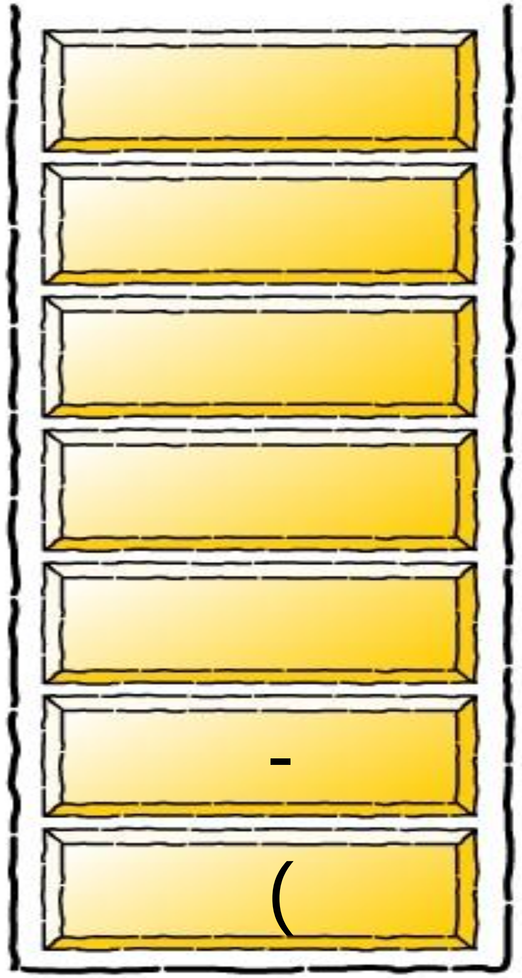
c) * d – (e + f)

postfixVect

a b +

Infix to postfix conversion

stackVect



infixVect

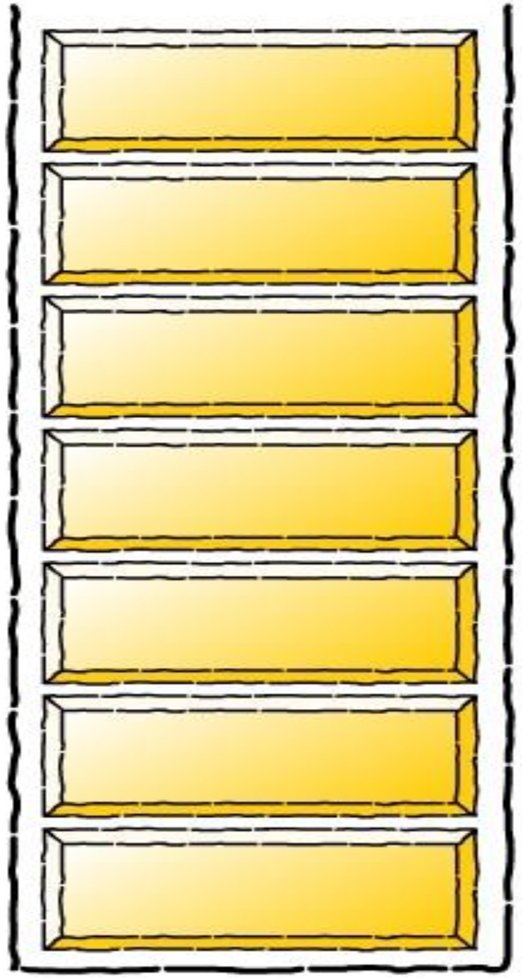
) * d – (e + f)

postfixVect

a b + c

Infix to postfix conversion

stackVect



infixVect

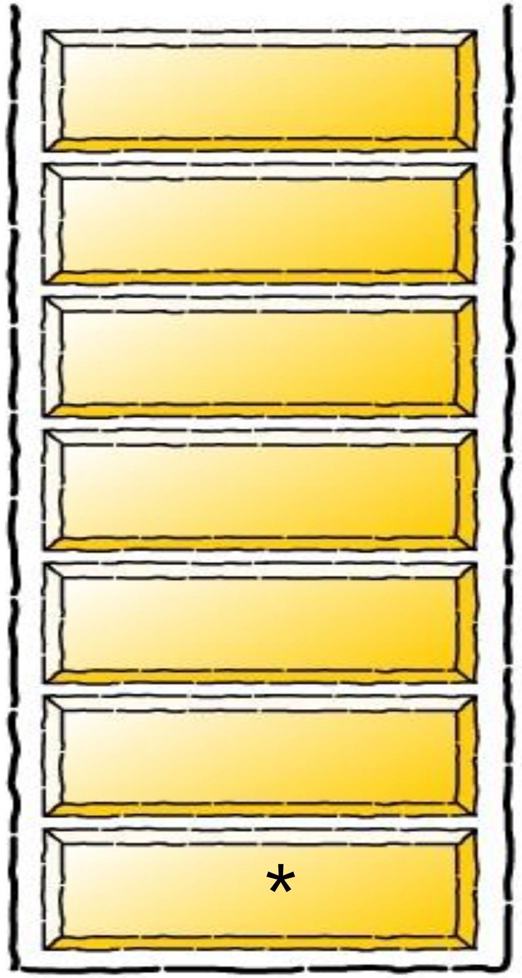
* d – (e + f)

postfixVect

a b + c -

Infix to postfix conversion

stackVect



infixVect

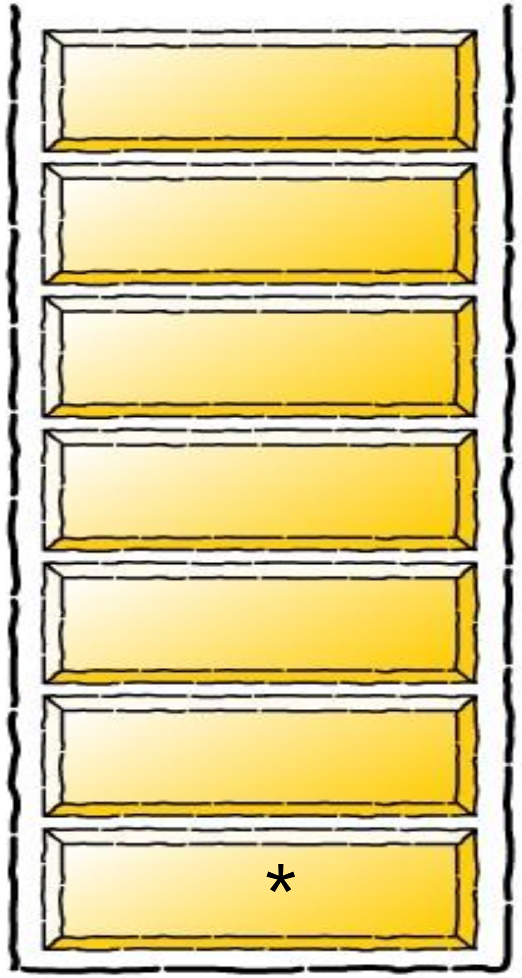
d – (e + f)

postfixVect

a b + c -

Infix to postfix conversion

stackVect



infixVect

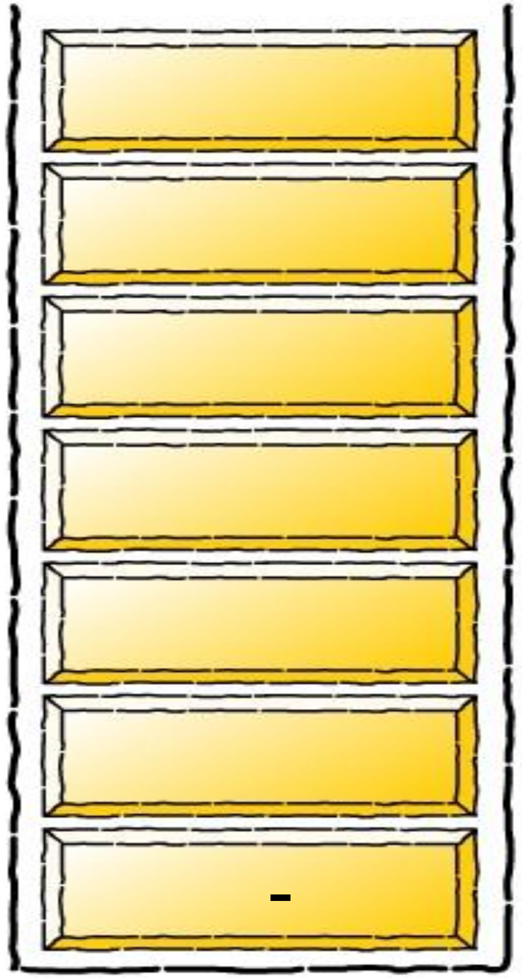
– (e + f)

postfixVect

a b + c - d

Infix to postfix conversion

stackVect



infixVect

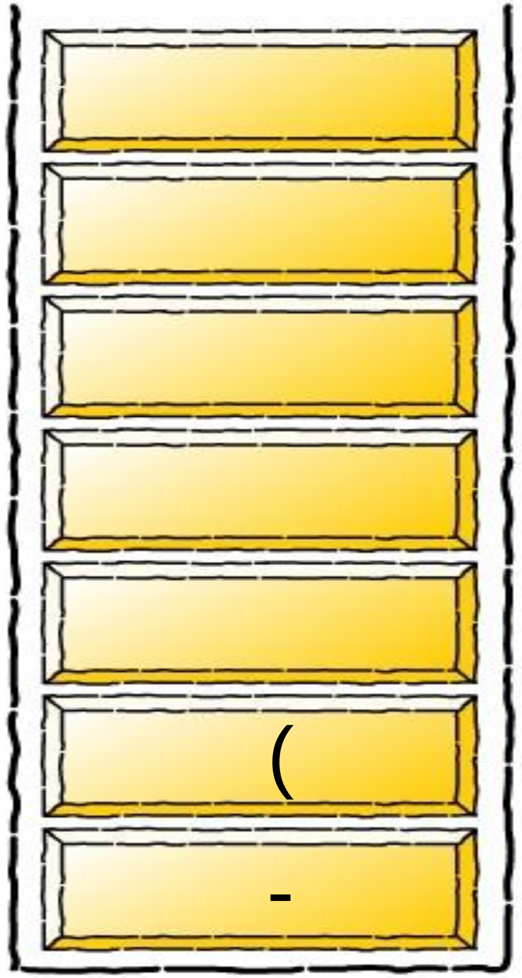
(e + f)

postfixVect

a b + c – d *

Infix to postfix conversion

stackVect



infixVect

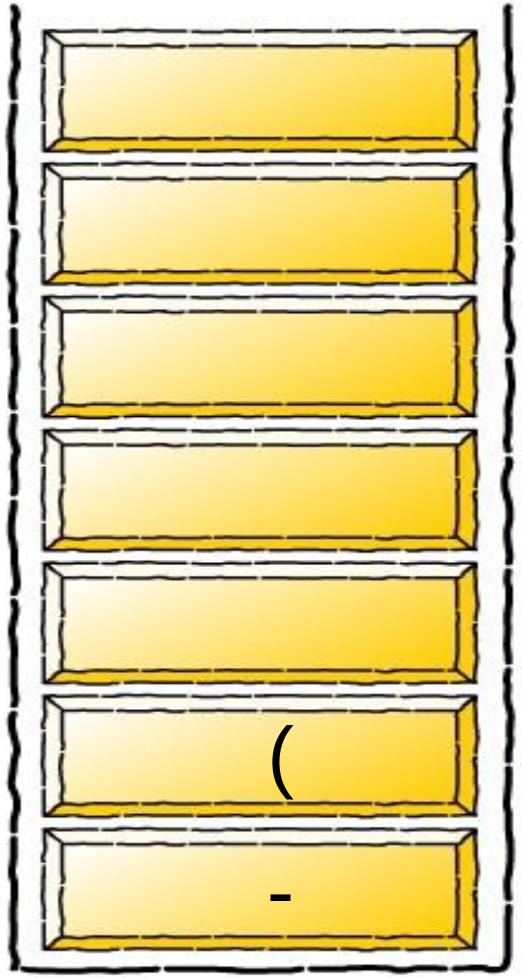
e + f)

postfixVect

a b + c – d *

Infix to postfix conversion

stackVect



infixVect

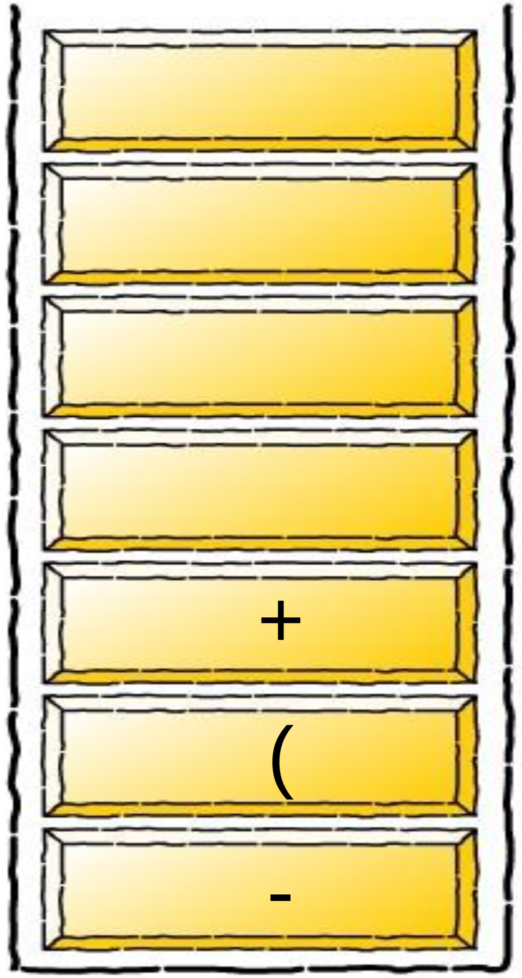
+ f)

postfixVect

a b + c – d * e

Infix to postfix conversion

stackVect



infixVect

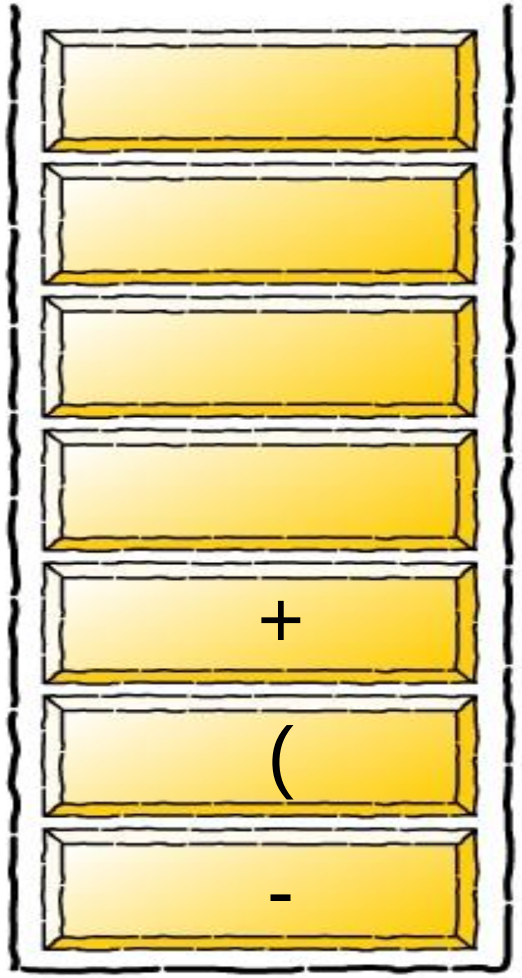
f)

postfixVect

a b + c – d * e

Infix to postfix conversion

stackVect



infixVect

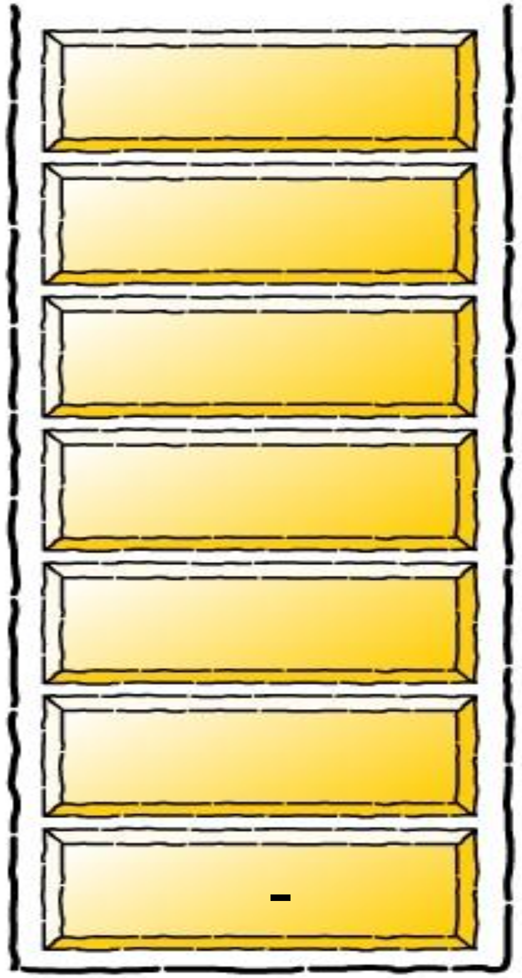
)

postfixVect

a b + c – d * e f

Infix to postfix conversion

stackVect



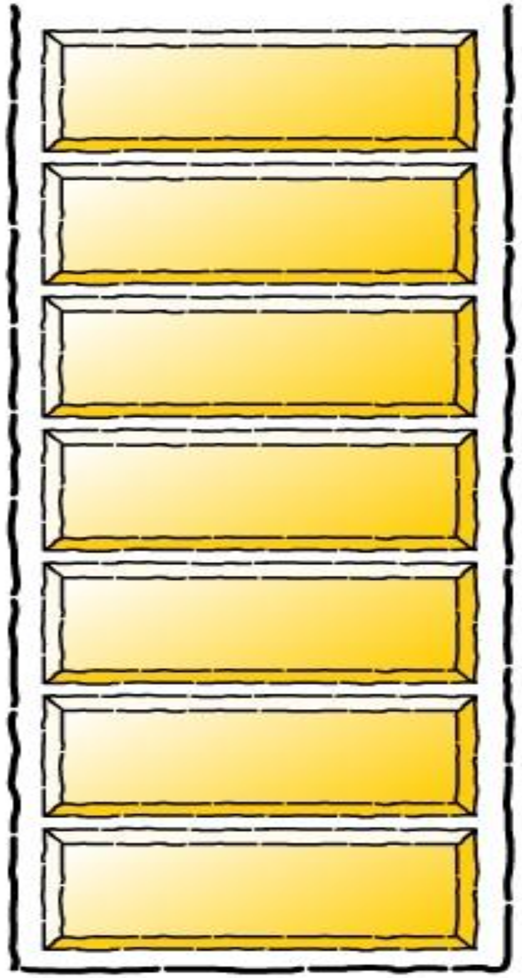
infixVect

postfixVect

a b + c – d * e f +

Infix to postfix conversion

stackVect



infixVect

postfixVect

a b + c – d * e f + -

Another Example: Transforming Infix to Postfix

Next Character	Postfix	Operator Stack (bottom to top)
a	a	
/	a	/
b	$a\ b$	/
*	$a\ b$ /	
	$a\ b$ /	*
(	$a\ b$ /	* (
c	$a\ b$ / c	* (
+	$a\ b$ / c	* (+
(	$a\ b$ / c	* (+ (
d	$a\ b$ / $c\ d$	* (+ (
−	$a\ b$ / $c\ d$	* (+ (−
e	$a\ b$ / $c\ d\ e$	* (+ (−
)	$a\ b$ / $c\ d\ e$ −	* (+ (
	$a\ b$ / $c\ d\ e$ −	* (+
)	$a\ b$ / $c\ d\ e$ − +	* (
	$a\ b$ / $c\ d\ e$ − +	*
	$a\ b$ / $c\ d\ e$ − + *	

Steps to convert the infix expression `a / b * ( c + ( d – e ) )` to postfix form.

Infix-to-Postfix Algorithm

Symbol in Infix	Action
Operand	Append to end of output expression
Operator ^	Push ^ onto stack
Operator +,-, *, or /	Pop operators from stack, append to output expression until stack empty or top has lower precedence than new operator. Then push new operator onto stack
Open parenthesis	Push (onto stack
Close parenthesis	Pop operators from stack, append to output expression until we pop an open parenthesis). Discard both parentheses.

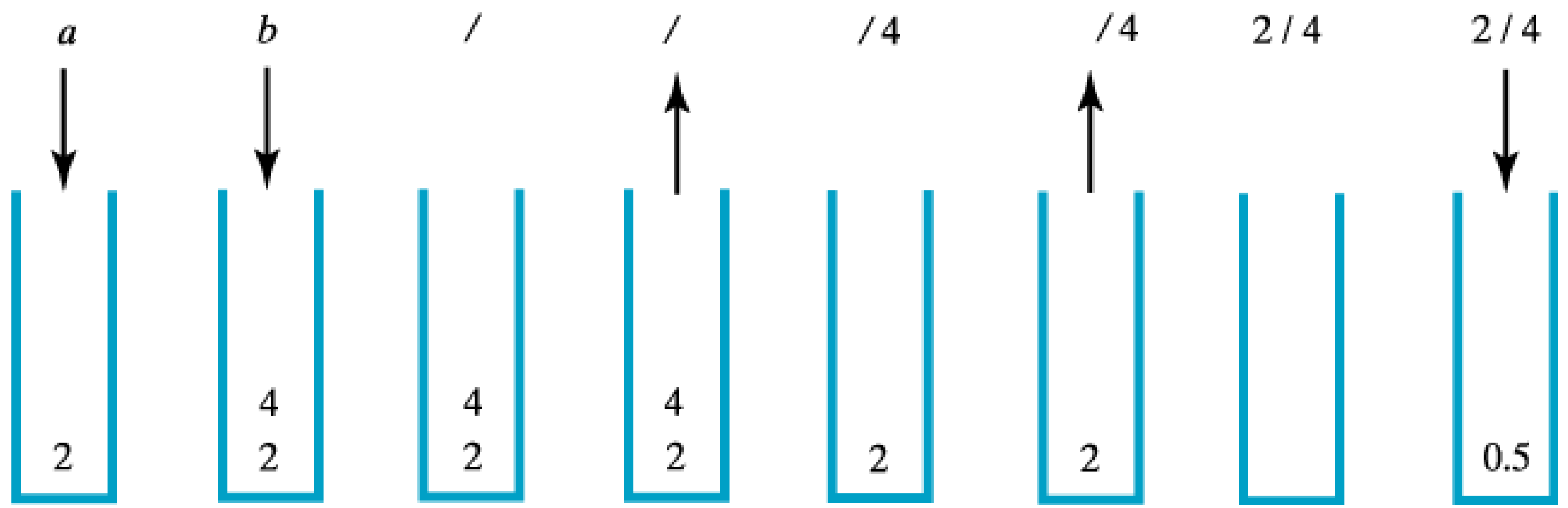
Infix to postfix conversion

Read the tokens from a vector infixVect of tokens (strings) of an infix expression

- When the *token* is an operand
  - Add it to the end of the vector postfixVect of token (strings) that is used to store the corresponding postfix expression
- When the *token* is a left or right parenthesis or an operator
  - If the *token x* is "("
    - Push the token x to the end of the vector stack of token (strings) that simulates a stack
  - if the *token* x is ")"
    - Repeatedly pop a token y from stack and push that token y to postfixVect until "(" is encountered in the end of stack. Then pop "(" from stack.
    - If stack is already empty before finding a "(", that expression is not a valid expression.
  - if the *token x* is a regular operator
    - **Step 1**: Check the token y **currently** at the top of stack.
    - **Step 2**: **If** (case 1) stack is **not** empty **and** (case 2) y is **not** "(" **and** (case 3) y is an operator of **higher or equal** precedence than that of x, then pop the token y from stack and push the token y to postfixVect, and **go to Step 1 again**.
    - **Step 3**: **If** (case 1) stack is already empty **or** (case 2) y is "(" **or** (case 3) y is an operator of **lower precedence** than that of x, then push the token x into stack.

When all tokens in infixVect are processed as described above, repeatedly pop a token y from stack and push that token y to postfixVect until stack is empty

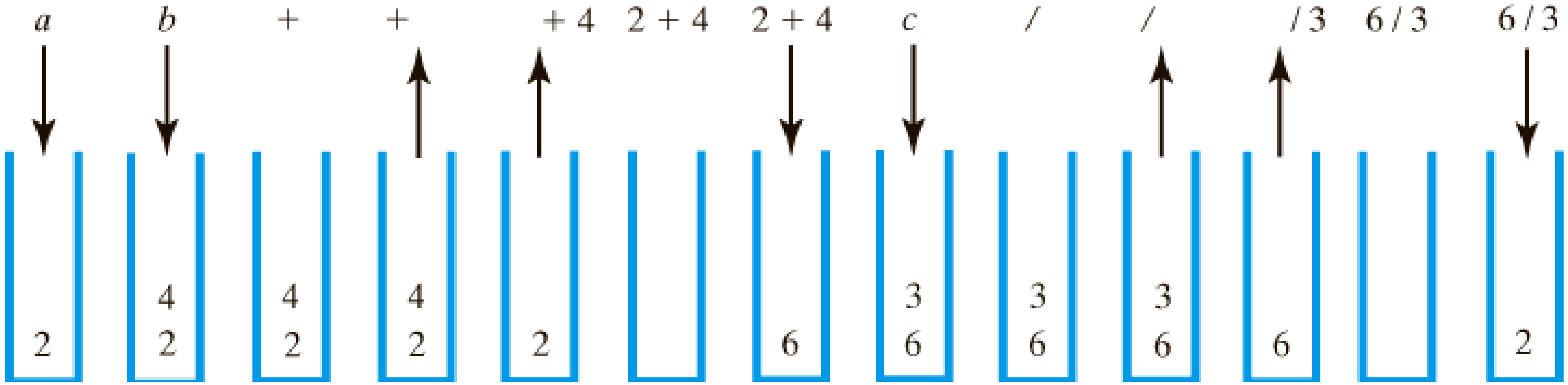
Evaluating Postfix Expression



The stack during the evaluation of the postfix expression

`a b /` when `a` is 2 and `b` is 4

Evaluating Postfix Expression



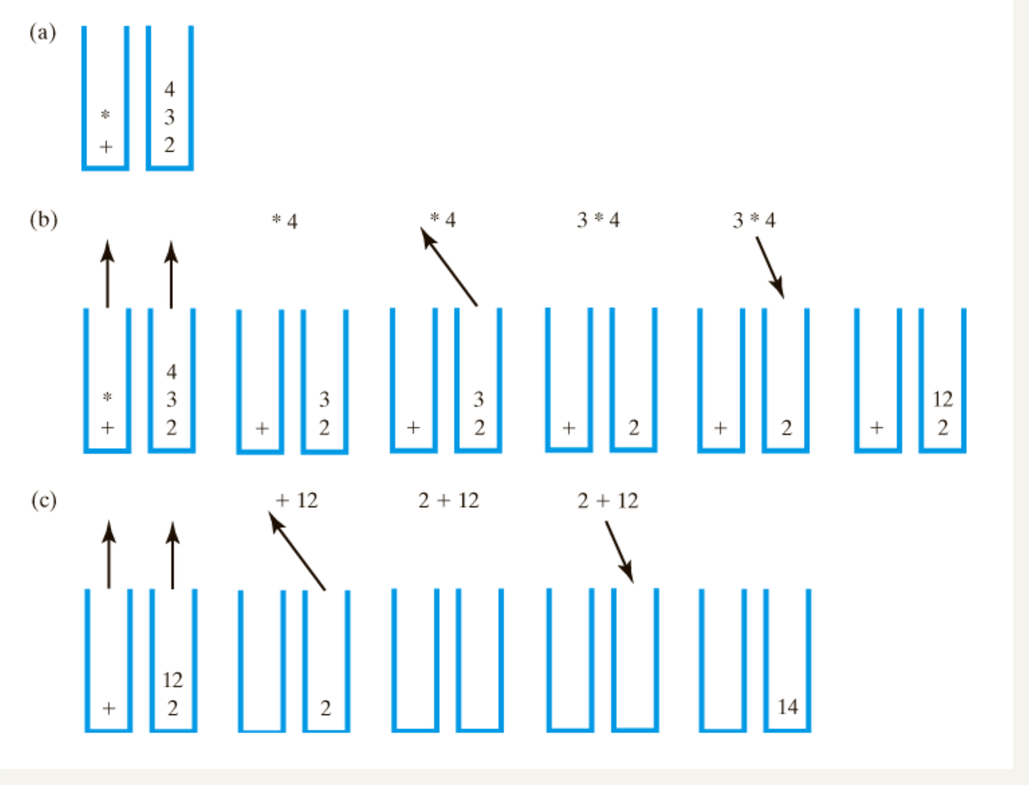
The stack during the evaluation of the postfix expression

`a b + c /` when `a` is 2, `b` is 4 and `c` is 3

Evaluates a postfix expression

```
Algorithm evaluatePostfix(postfix)      // Evaluates a postfix expression.
Stack = a new empty stack
while (postfix has characters left to parse)
{     nextCharacter = next nonblank character of postfix
      switch (nextCharacter)
      {     case variable:
                  Stack.push(value of the variable nextCharacter)
                  break
            case '+': case '-': case '*': case '/': case '^':
                  operandTwo = Stack.pop()
                  operandOne = Stack.pop()
                  result = the result of the operation in nextCharacter and its operands
                              operandOne and operandTwo
                  Stack.push(result)
                  break
            default: break
      }
}
return Stack.peek()
```

Evaluating Infix Expressions



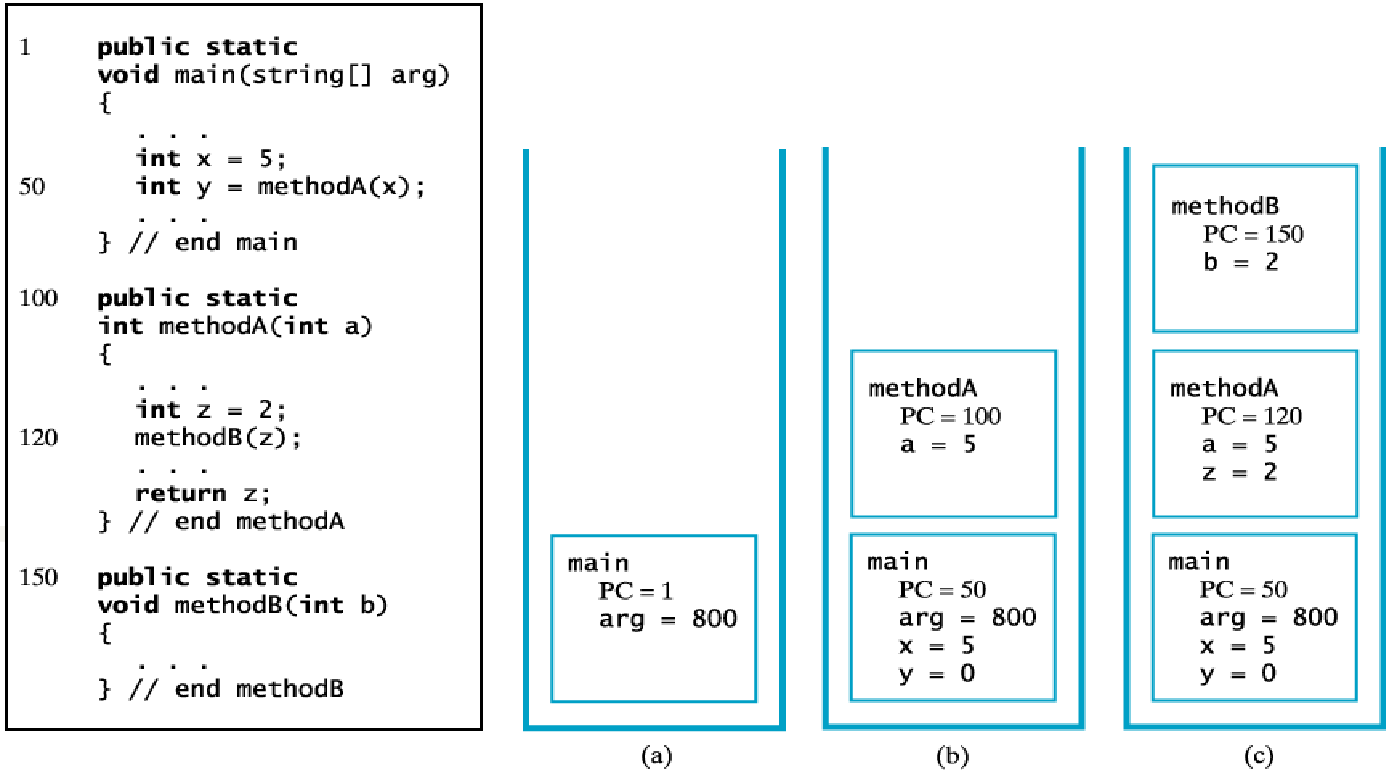
Two stacks during evaluation of a + b * c when a = 2, b = 3, c = 4;

(a) after reaching end of expression;
(b) while performing multiplication;
(c) while performing the addition

The Program Stack

- When a method is called
  - Runtime environment creates activation record
  - Shows method's state during execution
- Activation record pushed onto the program stack (Java stack)
  - Top of stack belongs to currently executing method
  - Next method down is the one that called current method

The Program Stack



Program

Program stack at three points in time (PC is the program counter)

The program stack at 3 points in time; (a) when **`main`** begins execution; (b) when **`methodA`** begins execution, (c) when **`methodB`** begins execution.

Recursive Methods

- A recursive method making many recursive calls
  - Places many activation records in the program stack
  - Thus, the reason recursive methods can use much memory
- Possible to replace recursion with iteration by using a stack

Slides and figures have been collected from various publicly available Internet sources for preparing the lecture slides of IT2001 course. I acknowledge and thank all the original authors for their contribution to prepare the content.