Stacks

Chapter 6

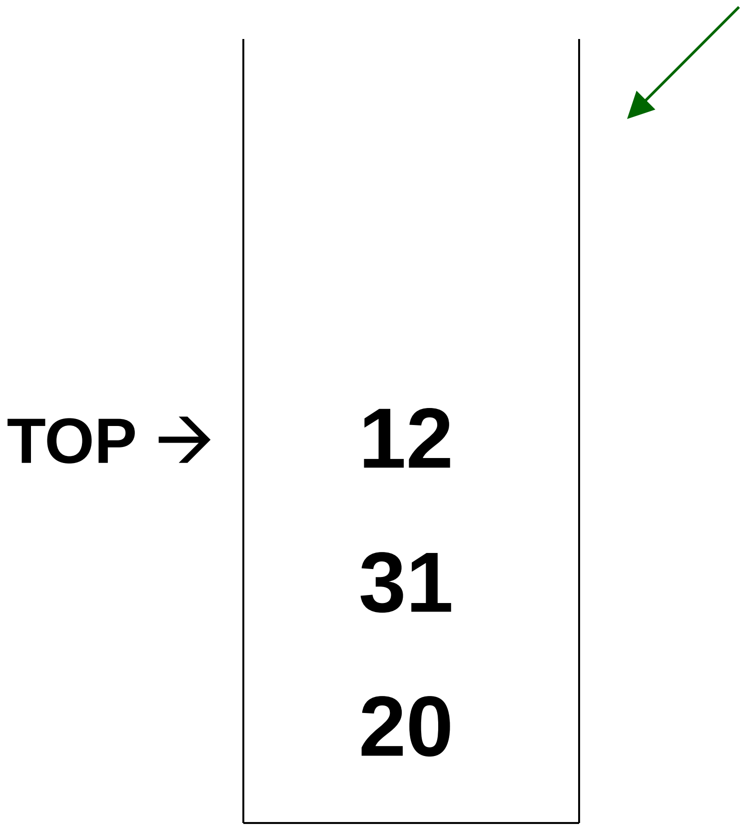
Fundamentals

- A <u>stack</u> is a sequence of data elements (of the same type) arranged one after another conceptually.
- An element can be added to the top of the stack only. (“PUSH”)
- An element can be removed from the top of the stack only. (“POP”)

Applications of Stacks

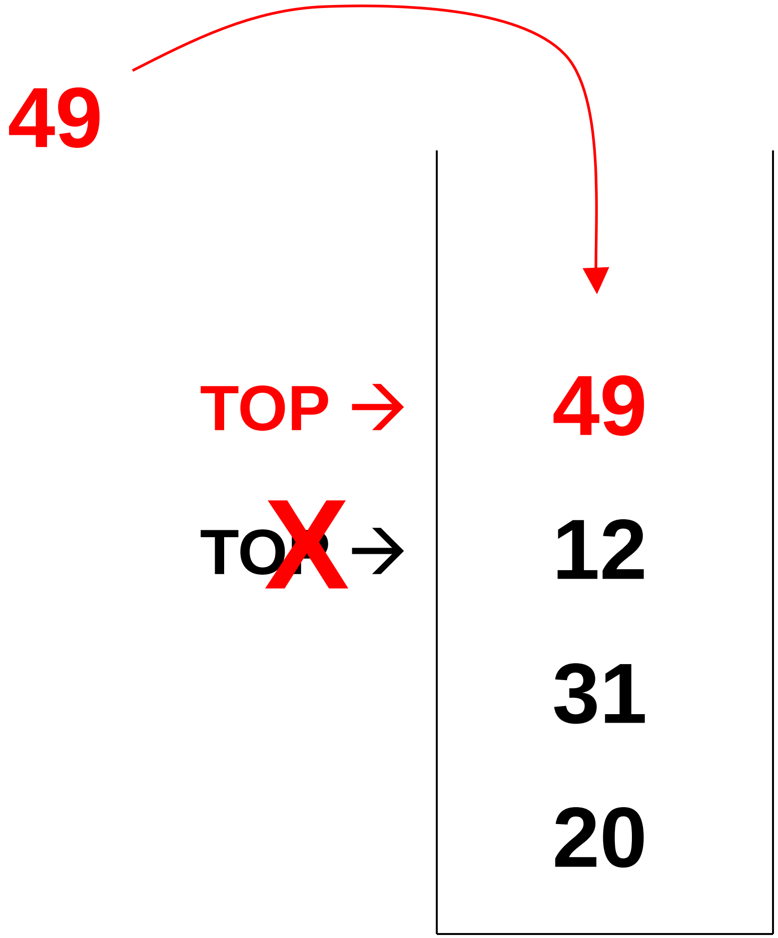
- Operating systems
  - keeping track of method calls in a running program
- Compilers
  - conversion of arithmetic expressions to machine code

Conceptual Picture

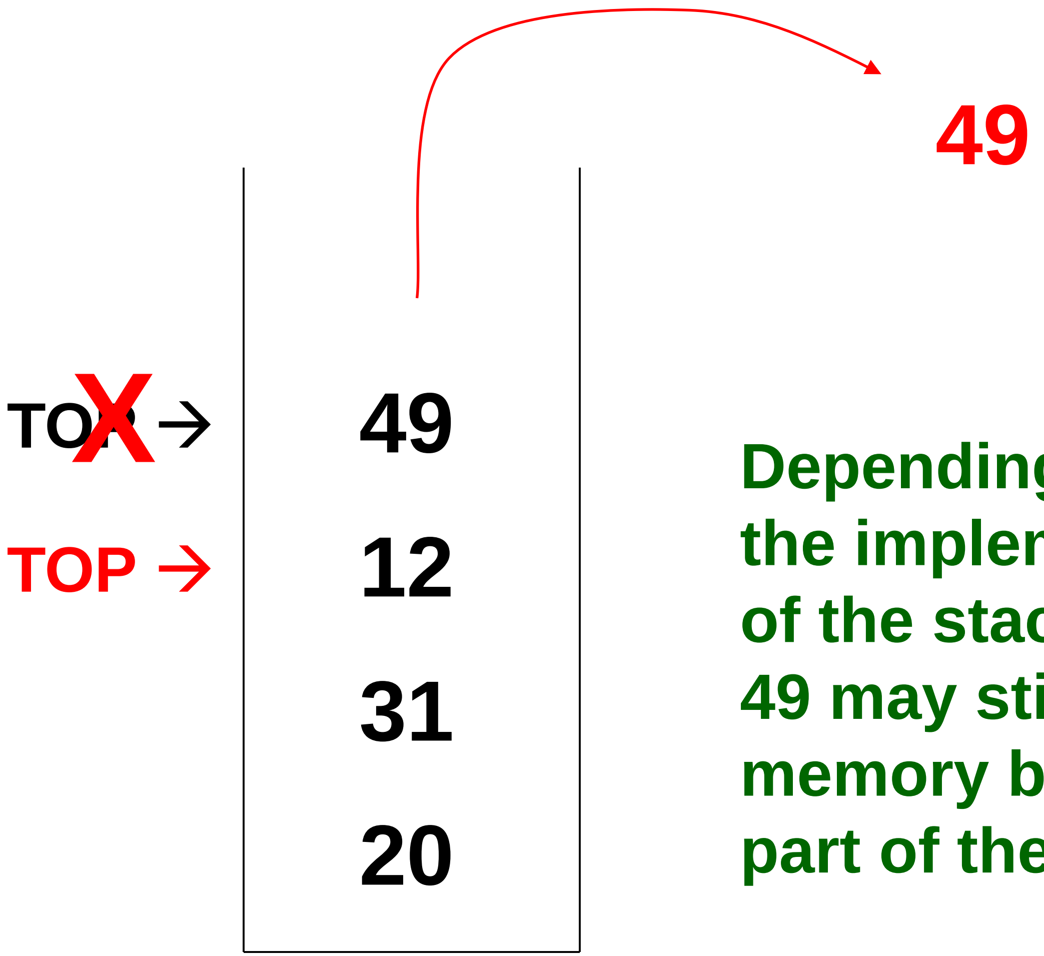


Depending on the implementation of a stack, it may or may not have a maximum capacity.

PUSH



POP



Depending on the implementation of the stack, the 49 may still be in memory but is not part of the stack.

Implementation of a Stack

- **ARRAYS**

12	**31**	**20**				

top

20	**31**	**12**				

top

Which is more efficient?

Basic Stack Operations

- **Constructor – create an empty stack**
- **isEmpty – is the stack empty?**
- **push – push an element on to the top of the stack (if the stack is not full)**
- **pop – remove the top element from the stack (if the stack is not empty)**

Extended Stack Operations

- **peek – examine the top element of the stack without removing it (if the stack is not empty)**

```
if not isEmpty()
    temp ← pop()
    push(temp)
    return(temp)
```

- **size – return the number of elements on the stack**

  How would you implement this using the basic operations?

An IntStack using Arrays

```
public class IntStack implements
  Cloneable {

  public final int CAPACITY = 100;
  private int[] data;
  private int top;

  // IntStack methods (clone not shown)
}
```

IntStack (arrays) (cont'd)

```
public IntStack()
{
    top = -1;
    data = new int[CAPACITY];
}
public boolean isEmpty()
{
    return (top == -1);
}
```

IntStack (arrays) (cont’d)

```
public void push(int item)
{
    if (top == CAPACITY-1)
        throw new FullStackException();
    top++;
    data[top] = item;
}
```

IntStack (arrays) (cont'd)

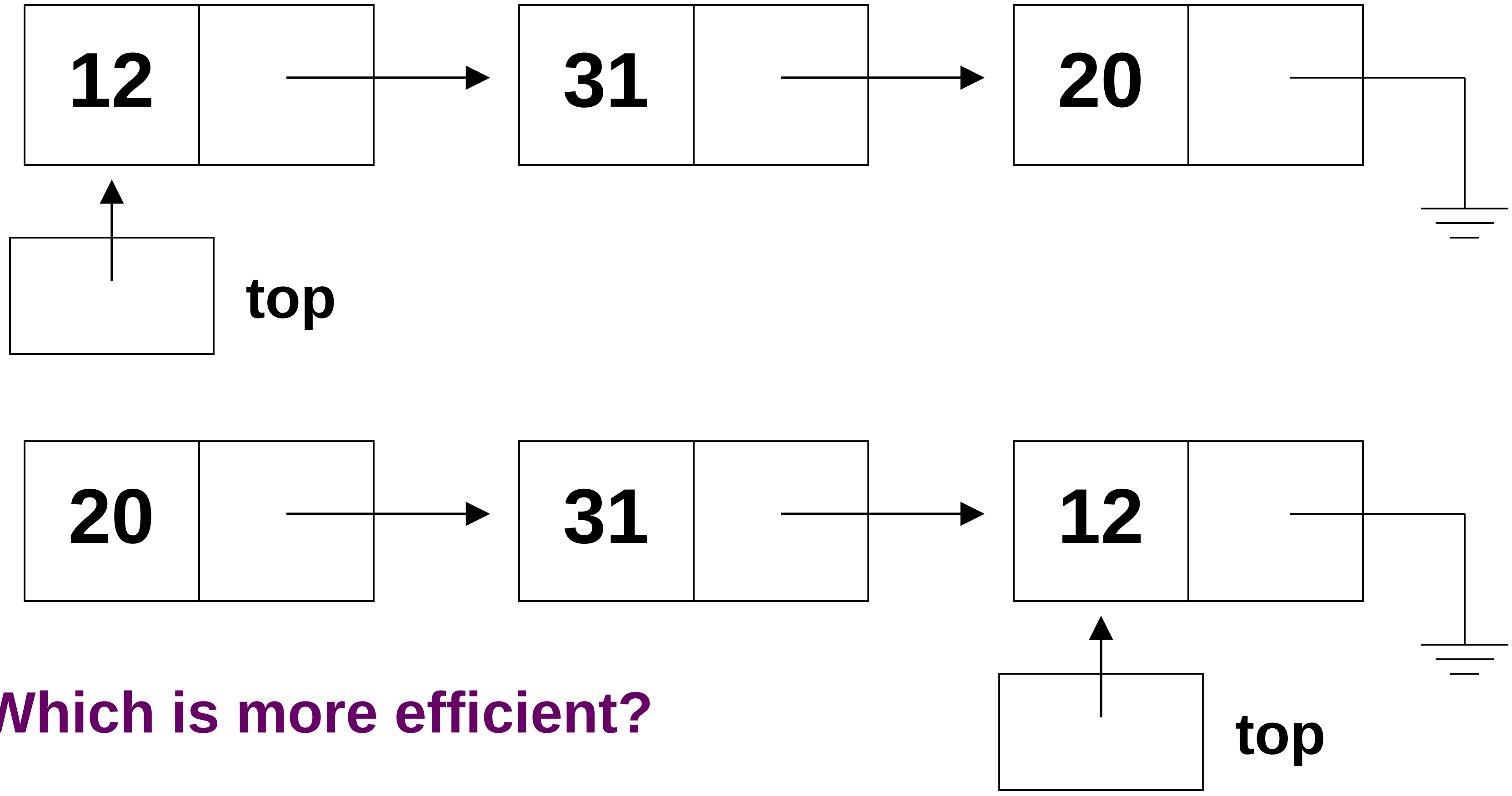
```
public int pop()
{
    int answer;
    if (top == -1) // isEmpty()
        throw new EmptyStackException();
    answer = data[top];
    top--;
    return answer;
}
```

IntStack (arrays) (cont'd)

```
public int peek()
{
  int answer;
  if (top == -1) // isEmpty()
    throw new EmptyStackException();
  answer = data[top];
  return answer;
}
```

Implementation of a Stack

- **LINKED LISTS**



Which is more efficient?

An IntStack using Lists

```
public class IntStack implements
  Cloneable {

  private IntNode top;

  // IntStack methods (clone not shown)
}
```

IntStack (lists) (cont'd)

```
public IntStack()
{
     top = null;
}
public boolean isEmpty()
{
     return (top == null);
}
```

IntStack (lists) (cont'd)

```
public void push(int item)
{
    IntNode newNode
        = new IntNode(item);
    newNode.setLink(top);
    top = newNode;
}
```

Stack Overflow?

IntStack (lists) (cont'd)

```
public int pop()
{
    int answer;
    if (top == null) // isEmpty()
        throw new EmptyStackException();
    answer = top.getData();
    top = top.getLink();
    return answer;
}
```

IntStack (lists) (cont'd)

```
public int peek()
{
  int answer;
  if (top == null) // isEmpty()
    throw new EmptyStackException();
  answer = top.getData();
  return answer;
}
```

Balanced Parentheses

- **An arithmetic expression has balanced parenthesis if and only if:**
  - **the number of left parentheses of each type is equal to the number of right parentheses of each type**
  - **each right parenthesis of a given type matches to a left parenthesis of the same type to its left and all parentheses in between are balanced correctly.**

Examples

- **({A + B} – C)**
  Balanced
- **({A + B) – C}**
  Not balanced
- **({A + B} – [C / D])**
  Balanced
- **(({A + B} – C) / D))**
  Not balanced

Algorithm

CHECK FOR BALANCED PARENTHESES

- **Scan the expression from left to right.**
  - **For each left parenthesis that is found, push on the stack.**
  - **For each right parenthesis that is found,**
    - **If the stack is empty, return false (too many right parentheses)**
    - **Otherwise, pop the top parenthesis from the stack:**
      - **If the left and right parentheses are of the same type, discard. Otherwise, return false.**

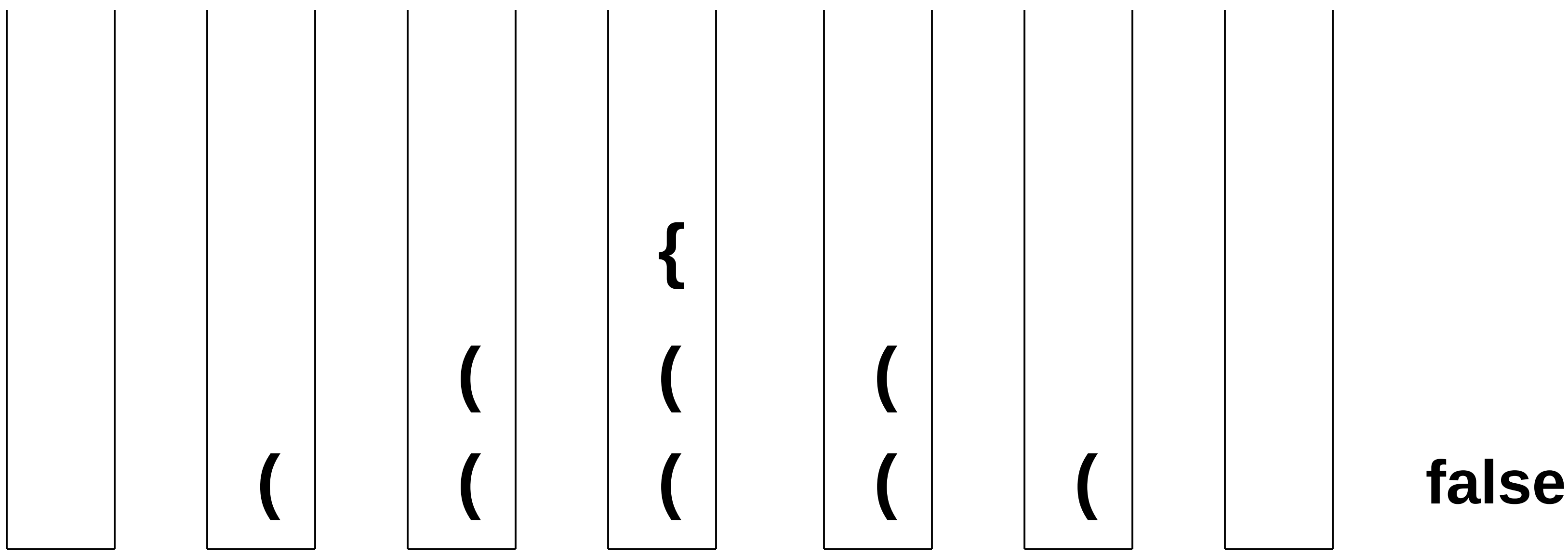
Algorithm (cont’d)

CHECK FOR BALANCED PARENTHESES

- **If the stack is empty when the scan is complete, return true.**
  Otherwise, return false. (too many left parentheses)

Trace

- **(({A + B} – C) / D))**
- **Stack trace:**



Evaluating Expressions

- **An expression is fully parenthesized if every operator has a pair of balanced parentheses marking its left and right operands.**
- **Not fully-parenthesized:**
  3 * (5 + 7) – 9   (2 - 4) * (5 - 7) + 8
- **Fully-parenthesized:**
  ((3 * (5 + 7)) – 9)   (((2 - 4)*(5 - 7)) + 8)

General Idea

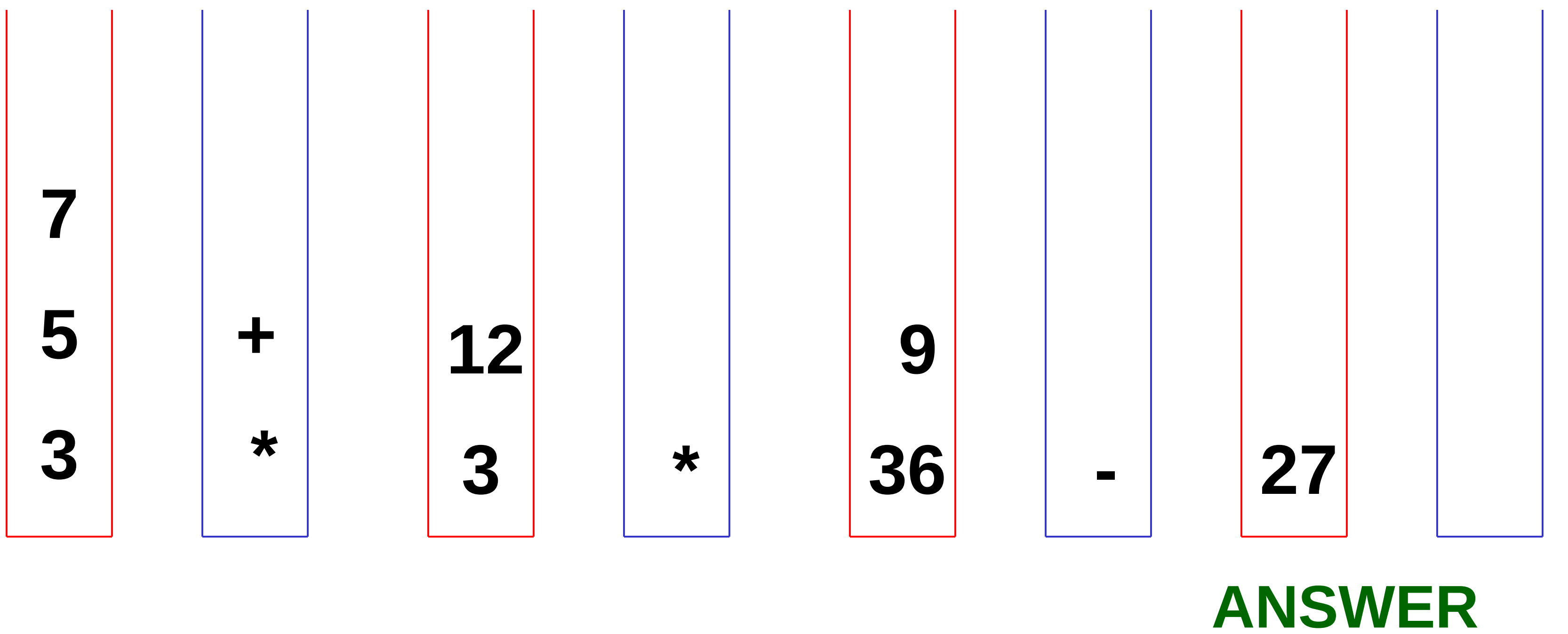
- **The first operation to perform is surrounded by the innermost set of balanced parentheses.**
- **Example: ((3 * (5 + 7)) – 9) First op: +**
- **By reading expression from left to right, first operator comes immediately before first right parenthesis.**
- **Replace that subexpression with its result and search for next right parenthesis, etc.**
- **Example: ((3 * 12) – 9) = (36 – 9) = 27**

General Idea (cont'd)

- **How do we keep track of operands and operators as we read past them in the expression from left to right?**
- **Use two stacks:
one for operands and one for operators.**
- **When we encounter a right parenthesis, pop off one operator and two operands, perform the operation, and push the result back on the operand stack.**

Trace

- **((3 * (5 + 7)) – 9)**
- **Stack traces:** **operands** **operators**



Algorithm

- **Let each operand or operator or parenthesis symbol be a <u>token</u>.**
- **Let NumStack store the operands.**
- **Let OpStack store the operations.**
- **For each token in the input expression do**

    If token = operand, NumStack.push(token)

    If token = operator, OpStack.push(token)

Algorithm (cont'd)

If token = ")",

operand2 ← NumStack.pop()

operand1 ← NumStack.pop()

operator ← OpStack.pop()

result ← operand1 operator operand2

NumStack.push(result)

If token = "(", ignore token

- **After expression is parsed, answer ← NumStack.pop()**

Arithmetic Expressions

- **Infix notation:**
  operator is between its two operands

  3 + 5 (5 + 7) * 9 5 + (7 * 9)

- **Prefix notation:**
  operator precedes its two operands

  + 3 5 * + 5 7 9 + 5 * 7 9

- **Postfix notation:**
  operator follows its two operands

  3 5 + 5 7 + 9 * 5 7 9 * +

NO PAREN-THESES

Precedence of Operators

- **Multiplication and division (higher precedence) are performed before addition and subtraction (lower precedence)**
- **Operators in balanced parentheses are performed before operators outside of the balanced parentheses.**
- **If two operators are of the same precedence, they are evaluated left to right.**

Example

- **Infix expression:**

A + B * (C * D – E / F) / G – H

6 4 1 3 2 5 7

- **What is its prefix equivalent?**

– + A / * B – * C D / E F G H

- **What is its postfix equivalent?**

A B C D * E F / – * G / + H –

1 2 3 4 5 6 7

Evaluating a Postfix Expression

- **Let each operand or operator be a <u>token</u>.**
- **Let NumStack store the operands.**
- **For each token in the input expression do**

  If token = operand, push(token)

  If token = operator,

  operand2 ← pop()

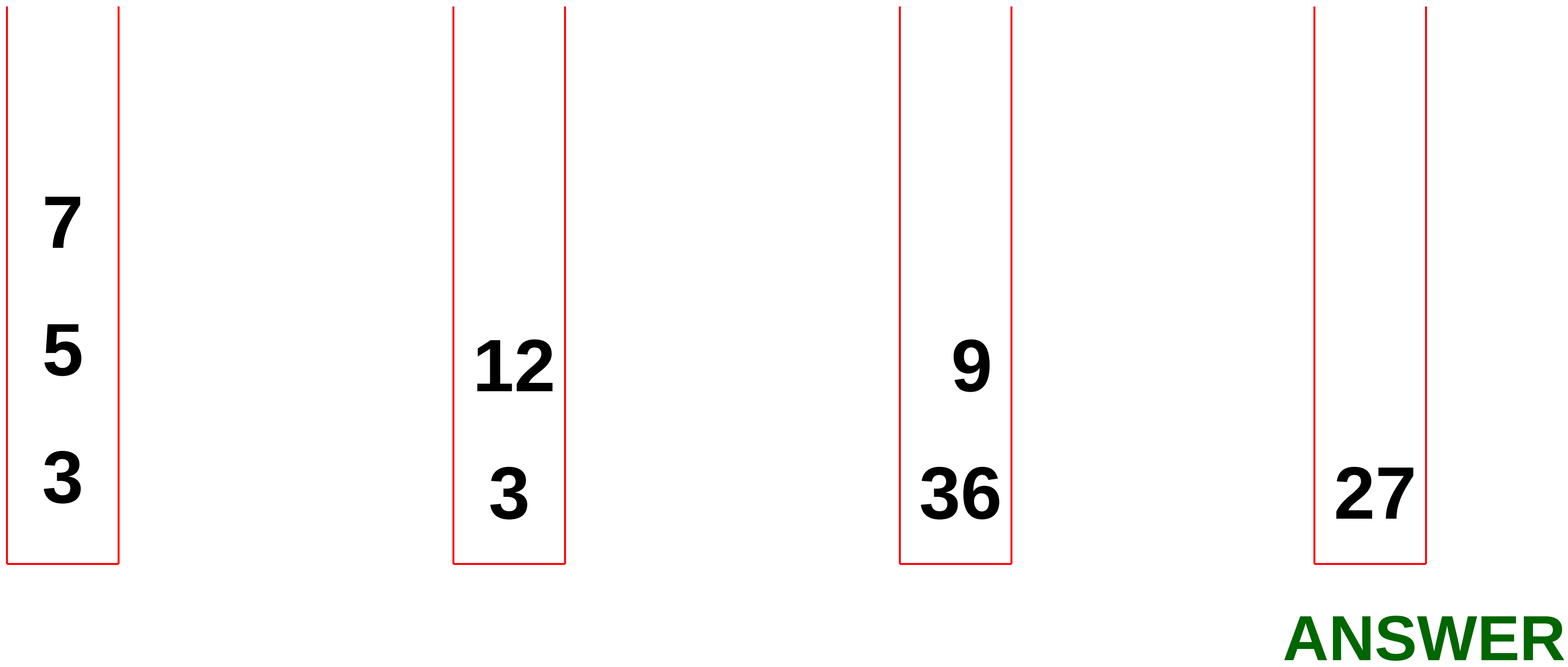
  operand1 ← pop()

  result ← operand1 operator operand2

  push(result)
- **answer ← pop()**

Trace

- **INFIX:** **3 * (5 + 7) – 9**
- **POSTFIX:** **3 5 7 + * 9 -**
- **Stack traces:** **operands**



Translating Infix to Postfix

Fully-Parenthesized Expressions

- **Let each operand, operator, or parenthesis be a token.**
- **Let OpStack store the operators.**
- **Let postfix string P = “” (empty string)**
- **For each token in the input expression do**

    If token = operand, append operand to P

    If token = operator, push(token)

    If token = “)”, append pop() to P

    If token = “(“, ignore

Trace

- **Infix: ((3 * (5 + 7)) – 9)**

Stack (sideways)	**Postfix String**
empty	**3**
*****	**3**
*****	**3 5**
*** +**	**3 5**
*** +**	**3 5 7**
*****	**3 5 7 +**
empty	**3 5 7 + ***
–	**3 5 7 + ***
–	**3 5 7 + * 9**
empty	**3 5 7 + * 9 –**

Another Example

Infix: ((3 * (5 + 7)) – 9)

Postfix: 3 5 7 + * 9 –

Infix: (((2 – 4)*(5 – 7)) + 8)

Postfix: 2 4 – 5 7 – * 8 +

Another Example

Infix: ((3 * (5 + 7)) – 9)

2 1 3

Postfix: 3 5 7 + * 9 –

1 2 3

Infix: (((2 – 4)*(5 – 7)) + 8)

1 3 2 4

Postfix: 2 4 – 5 7 – * 8 +

1 2 3 4

Translating Infix to Postfix

General Expressions

- **Define a precedence function**
- ***prec*: token → {0,1,2,3}**
- **let "$" represent empty stack**
- **token precedence**

token	precedence
"$"	0
"("	1
"+", "-"	2
"*", "/"	3

Translating Infix to Postfix

General Expressions

- **Let each operand, operator, or parenthesis be a token.**
- **Let OpStack be a character stack that stores the operators or other special symbols (“(” and “$”).**
- **Let postfix string P = “” (empty string)**

Translating Infix to Postfix (cont'd)

General Expressions

1. **push("$")**
2. **For each token in the input expression do**

   a. If token = operand,

   append token to P

   b. if token = "(",

   push(token)

Translating Infix to Postfix (cont'd)

General Expressions

c. if token = “)”,

topOp ← pop()

while topOp ≠ “(”

append topOp to P

topOp ← pop()

Translating Infix to Postfix (cont'd)

General Expressions

d. if token = operator,

topOp ← peek()

while *prec*(topOp) ≥ *prec*(token)

append pop() to P

topOp ← peek()

push(token)

Translating Infix to Postfix (cont'd)

General Expressions

3. At end of infix expression,

topOp ← pop()

while topOp ≠ "$" do

append topOp to P

topOp ← pop()

Trace

A + B * (C * D - E / F) / G – H

- **Stack *(sideways)*** **Postfix String**

Stack *(sideways)*	Postfix String
$	
$	**A**
$ +	**A**
$ +	**A B**
$ + *	**A B**
$ + * (	**A B**
$ + * (	**A B C**

Trace (cont'd)

A + B * (C * D - E / F) / G – H

• **Stack *(sideways)***	**Postfix String**
$ + * (	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

Trace (cont'd)

A + B * (C * D - E / F) / G – H

• **Stack** ***(sideways)***	**Postfix String**
$ + * (- /	**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 / + H**
empty	**A B C D * E F / - * G / + H -**