Stacks & Queues

COMP 2210 – Dr. Hendrix

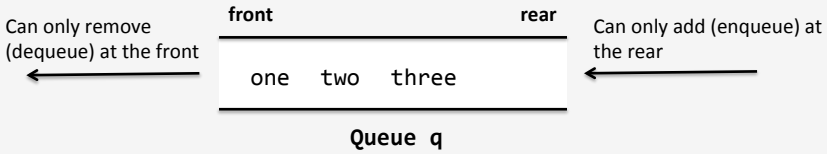
AUBURN UNIVERSITY

SAMUEL GINN COLLEGE OF ENGINEERING

Google define:queue

"A queue (pronounced /ˈkjuː/ kew) is a particular kind of collection in which the entities in the collection are kept in order and the principal (or only) operations on the collection are the addition of entities to the rear terminal position and removal of entities from the front terminal position. This makes the queue **a First-In-First-Out (FIFO) data structure.**"

Queue (data structure)
From Wikipedia, the free encyclopedia



Queue q

```
q.enqueue("one");
q.enqueue("two");
q.enqueue("three");
```

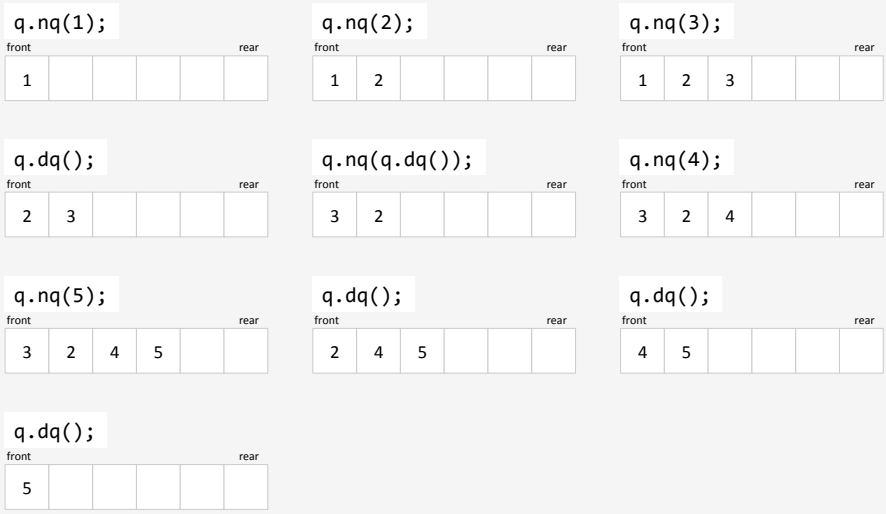
```
q.dequeue();
```

Physical analogy: A waiting line.

Queue API

```
public interface QueueInterface<T>
{
   //  Adds one element to the rear of the queue
   public void enqueue (T element);

   //  Removes and returns the element at the front of the queue
   public T dequeue();

   //  Returns without removing the element at the front of the queue
   public T first();

   //  Returns true if the queue contains no elements
   public boolean isEmpty();

   //  Returns the number of elements in the queue
   public int size();

   //  Returns an Iterator for the queue
   public Iterator<T> iterator();
}
```


Queue behavior – enqueue/add and dequeue/remove



Google define:stack

"In computer science, a stack is a **last in, first out (LIFO) abstract data type and data structure**. A stack can have any abstract data type as an element, but is characterized by only two fundamental operations**: push and pop**. ..."

Stack (data structure)
From Wikipedia, the free encyclopedia

Can only add (**push**) at the top → Can only delete (**pop**) from the top

```
s.push("one");
s.push("two");
s.push("three");
```

top

three
two
one

Stack s

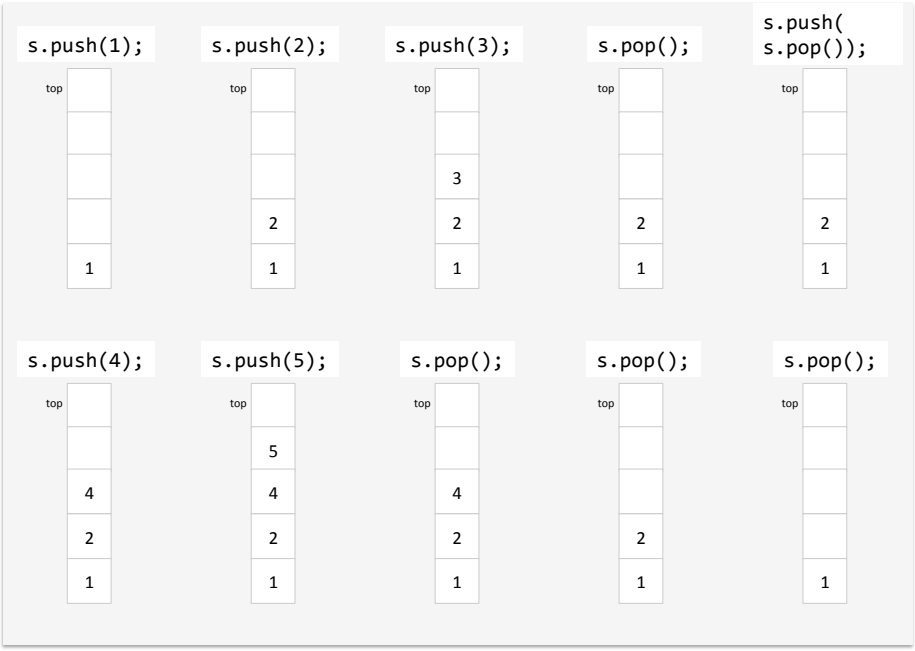
```
s.pop();
```

Physical analogy: Cafeteria trays.

Stack API

```
public interface StackInterface<T>
{
   //  Adds one element to the top of this stack
   public void push (T element);

   //  Removes and returns the top element from this stack
   public T pop();

   //  Returns without removing the top element of this stack
   public T peek();

   //  Returns true if this stack contains no elements
   public boolean isEmpty();

   //  Returns the number of elements in this stack
   public int size();

  //  Returns an Iterator for the stack
   public Iterator<T> iterator();
}
```

Stack behavior – push/add and pop/remove

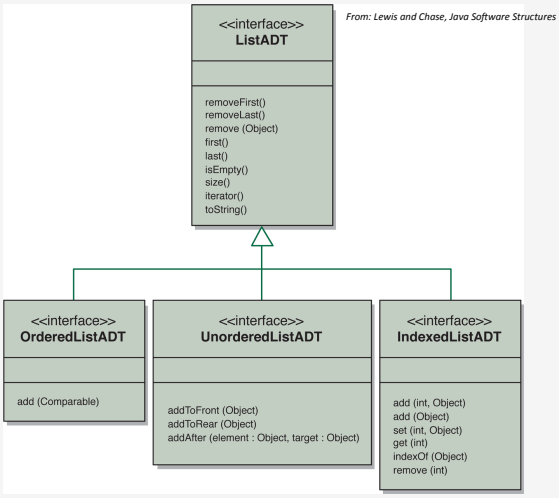


COMP 2210 • Dr. Hendrix • 7

implementation choices

COMP 2210 • Dr. Hendrix • 8

Reuse the List collection



Inheritance

```
public class ArrayStack<T> extends ArrayUnorderedList<T> implements StackInterface<T>
{


   public void push(T element) { this.addToRear(element); }

   public T pop() { this.removeLast(); }

}
```

O(1)

```
public class LinkedStack<T> extends LinkedUnorderedList<T> implements StackInterface<T>
{


   public void push(T element) { this.addToFront(element); }

   public T pop() { this.removeFirst(); }
}
```

O(1)

We inherit too much! Override unsupported methods.

Careless Coding Causes Killer Kangas

Mutant Marsupials Take Up Arms Against Australian Air Force

The reuse of some object-oriented code has caused tactical headaches for Australia's armed forces. As virtual reality simulators assume larger roles in helicopter combat training, programmers have gone to great lengths to increase the realism of their scenarios, including detailed landscapes and - in the case of the Northern Territory's Operation Phoenix- herds of kangaroos (since disturbed animals might well give away a helicopter's position).

The head of the Defense Science & Technology Organization's Land Operations/Simulation division reportedly instructed developers to model the local marsupials' movements and reactions to helicopters. Being efficient programmers, they just re-appropriated some code originally used to model infantry detachment reactions under the same stimuli, changed the mapped icon from a soldier to a kangaroo, and increased the figures' speed of movement.

Eager to demonstrate their flying skills for some visiting American pilots, the hotshot Aussies "buzzed" the virtual kangaroos in low flight during a simulation. The kangaroos scattered, as predicted, and the visiting Americans nodded appreciatively... then did a double-take as the kangaroos reappeared from behind a hill and launched a barrage of Stinger missiles at the hapless helicopter. (Apparently the programmers had forgotten to remove that part of the infantry coding.)

The lesson?

Objects are defined with certain attributes, and any new object defined in terms of an old one inherits all the attributes. The embarrassed programmers had learned to be careful when reusing object-oriented code, and the Yanks left with a newfound respect for Australian wildlife.

Simulator supervisors report that pilots from that point onward have strictly avoided kangaroos, just as they were meant to.

From June 15, 1999 Defense Science and Technology Organization Lecture Series, Melbourne, Australia, and staff reports

(An urban legend, based on a nugget of truth.)

Composition

```
public class ArrayStack<T> implements StackInterface<T>
{
   private ArrayUnorderedList<T> stack;

   public void push(T element) { stack.addToRear(element); }

   public T pop() { stack.removeLast(); }
}
```

O(1)

```
public class LinkedStack<T> implements StackInterface<T>
{
   private LinkedUnorderedList<T> stack;

   public void push(T element) { stack.addToFront(element); }

   public T pop() { stack.removeFirst(); }
}
```

O(1)

These classes allow only stack behavior but still enjoy the benefits of reuse.

Josh Bloch *Effective Java, 2nd Edition*

Item 16: Prefer composition over inheritance

Item 17: Design and document for inheritance or else prohibit it

Roll our own stack – arrays

```
import java.util.Iterator;

public class ArrayStack<T> implements StackInterface<T>
{
    private T[] stack;
    private int top;

    public ArrayStack(int capacity) { . . . }

    public void push(T element) { . . . }

    public T pop() { . . . }

    public T peek() { . . . }

    public int size() { . . . }

    public boolean isEmpty() { . . . }

    public Iterator<T> iterator() { . . . }

}
```

Nothing new here.

Static memory, thus capacity.

These *should* all be O(1).

ArrayStack – push and pop

```
import java.util.Iterator;

public class ArrayStack<T> implements StackInterface<T>
{
    private T[] stack;    private int top;

    public ArrayStack(int capacity) {
       stack = (T[]) new Object[capacity];
       top = 0;
    }

    public void push(T element) {
       if (size() == stack.length)
          expandCapacity();

       stack[top] = element;
       top++;
    }

    public T pop() {
       if (isEmpty()) return null;

       top--; T result = stack[top];
       return result;
    }
}
```

O(N) **O(1)** (amortized)

O(1)

O(1)

Time complexity of push()

O(1)

amortized

Should reduce the array capacity when it becomes too sparse.

More later …

Roll our own stack – nodes

```
import java.util.Iterator;


public class LinkedStack<T> implements StackInterface<T>
{
    private Node<T> top;
    private int count;

    public LinkedStack() { . . . }

    public void push(T element) { . . . }

    public T pop() { . . . }

    public T peek() { . . . }

    public int size() { . . . }

    public boolean isEmpty() { . . . }

    public Iterator<T> iterator() { . . . }

  }
```

Nothing new here.

Dynamic memory, no capacity.

These *should* all be O(1).

LinkedStack – push and pop

```
import java.util.Iterator;


public class LinkedStack<T> implements StackInterface<T>
{
    private Node<T> top;   private int count;

    public LinkedStack() {
       top = null;
       count = 0;
    }

    public void push(T element) {
       top = new Node<T>(element, top);
       count++;
    }

    public T pop() {
       if (isEmpty()) return null;

       T result = top.getElement();
       top = top.getNext();
       count--;
       return result;
    }
}
```

O(1)

O(1)

No amortization needed.

These methods are always O(1).

Roll our own queue – arrays

```
import java.util.Iterator;

public class ArrayQueue<T> implements QueueInterface<T>
{
    private T[] queue;
    private int rear;

    public ArrayQueue(int capacity) { . . . }

    public void enqueue(T element) { . . . }

    public T dequeue() { . . . }

    public T first() { . . . }

    public int size() { . . . }

    public boolean isEmpty() { . . . }

    public Iterator<T> iterator() { . . . }

  }
```

Nothing new here.

Static memory, thus capacity.

These *should* all be O(1).

ArrayQueue – enqueue and dequeue

```
import java.util.Iterator;

public class ArrayQueue<T> implements QueueInterface<T>
{
    private T[] queue;    private int rear;

    public ArrayQueue(int capacity) {
       queue = (T[]) new Object(capacity);
       rear = 0;
    }

    public void enqueue(T element) {
       if (size() == queue.length)
          expandCapacity();

       queue[rear] = element;
       rear++;
    }

    public T dequeue() {
       if (isEmpty()) return null;

       T result = queue[0]; rear--; shiftLeft();
       queue[rear] = null; return result;
    }
}
```

O(N) — **O(1)** (amortized)

O(1)

O(N)

Worst case for enqueue is O(N).

Amortized worst-case cost for enqueue is O(1).

Worst case for dequeue is O(N).

We can't amortize the cost of dequeue, because shiftLeft() is always called.

Should reduce the array capacity when it becomes too sparse.

More later …

An array-based circular queue

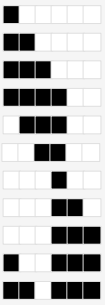
```
import java.util.Iterator;


public class ArrayQueue<T> implements QueueInterface<T>
{
    private T[] queue;
    private int rear;     private int front;

    public ArrayQueue(int capacity) { . . . }

    public void enqueue(T element) { . . . }

    public T dequeue() { . . . }

    public T first() { . . . }

    public int size() { . . . }

    public boolean isEmpty() { . . . }

    public Iterator<T> iterator() { . . . }

  }
```

Revised strategy:

Don't anchor the front at index 0.

Instead, keep track of the front and rear position, incrementing rear on every enqueue and incrementing front on every dequeue.

The elements are stored contiguously between front and rear, with wrap-around possible.



Circular ArrayQueue – enqueue and dequeue

```
import java.util.Iterator;


public class ArrayQueue<T> implements QueueInterface<T>
{
    private T[] queue;     private int front, rear;

    public ArrayQueue(int capacity) {
       queue = (T[]) new Object(capacity);
       front = 0; rear = 0;
    }

    public void enqueue(T element) {                    O(1)
       queue[rear] = element;                           (amortized)
       rear = (rear + 1) % queue.length;

       if (front == rear) expandArray();
    }

    public T dequeue() {                                O(1)
       if (front == rear) return null;

       T result = queue[front]; queue[front] = null;
       front = (front + 1) % queue.length;
       return result;
    }
}
```

Worst case for enqueue is O(N).

Amortized cost for enqueue is O(1).

Worst case for dequeue is O(1).

Same purpose, but different.

No shifting!

Roll our own queue – nodes

```
import java.util.Iterator;


public class LinkedQueue<T> implements QueueInterface<T>
{
    private Node<T> front, rear;
    private int count;

    public LinkedQueue() { . . . }

    public void enqueue(T element) { . . . }

    public T dequeue() { . . . }

    public T first() { . . . }

    public int size() { . . . }

    public boolean isEmpty() { . . . }

    public Iterator<T> iterator() { . . . }

  }
```

Nothing new here.

Dynamic memory, no capacity.

These *should* all be O(1).

LinkedQueue – enqueue and dequeue

```
import java.util.Iterator;


public class LinkedQueue<T> implements QueueInterface<T>
{
    private Node<T> front, rear;   private int count;

    public LinkedQueue() {
       front, rear = null;
       count = 0;
    }

    public void enqueue(T element) {
       Node<T> n = new Node<T>(element, top);
       if (isEmpty()) front = n;
       else rear.setNext(n);
       rear = n; count++;
    }

    public T dequeue() {
       if (isEmpty()) return null;
       T result = front.getElement();
       front = front.getNext(); count--;
       if (isEmpty()) rear = null;
       return result;
    }
}
```

O(1)

O(1)

No amortization needed.

These methods are always O(1).

Dynamic array resizing

```
public class ArrayBasedCollection<T> {

  private T[] elements;

  public ArrayBasedCollection() {
  . . .
     elements = (T[]) new Object[1];
  . . .
  }

  public void add(T e) {
  . . .
   if (size() == elements.length) {
          resize(elements.length * 2);
   }
  . . .
  }

  public T remove(T e) {
  . . .
   if (size() < (elements.length / 4)) {
          resize(elements.length / 2);
   }
  . . .
  }
}
```

Starting capacity 1.

Double capacity when full.

Halve capacity when less than 25% full.

Performance

	Stack		Queue		
method	**Array**	**Nodes**	**Array**	**Nodes**	**Circular Array**
add(element)	O(1)*	O(1)	O(1)*	O(1)	O(1)*
remove()	O(1)*	O(1)	O(N)	O(1)	O(1)*
look()	O(1)	O(1)	O(1)	O(1)	O(1)

**amortized*

applications

stack machines

A simple infix calculator

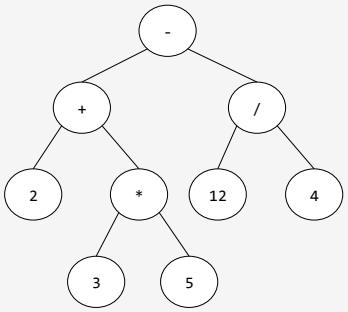
```
Enter an arithmetic expression: 2 + 3 * 5 – 12 / 4

      2 + 3 * 5 – 12 / 4 = 14
```

Get the infix as a tokenized stream of Strings.

2	+	3	*	5	-	12	/	4

This is harder than it looks!

We would need to ***parse*** the entire expression because of precedence and associativity before we could evaluate it.



A simpler infix calculator

```
Enter an arithmetic expression: ((2 + (3 * 5)) – (12 / 4))

      ((2 + (3 * 5)) – (12 / 4)) = 14
```

Get the infix as a tokenized stream of Strings.

(	(	2	+	(	3	*	5	)	)	-	(	12	/	4	)	)

This can be evaluated with a single left-to-right scan – no parsing.

We'll use a variation on Dijkstra's "shunting yard" algorithm.

Evaluating infix

(	(	2	+	(	3	*	5	)	)	-	(	12	/	4	)	)

```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating infix

(	(	2	+	(	3	*	5	)	)	-	(	12	/	4	)	)

```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating infix


```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating infix


```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating infix


```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating infix


```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

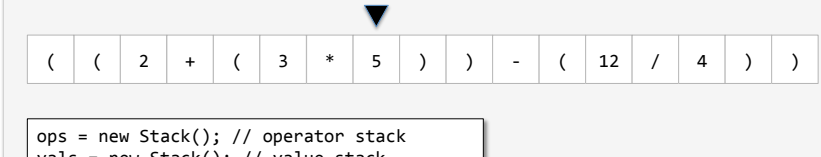
Evaluating infix


```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating infix

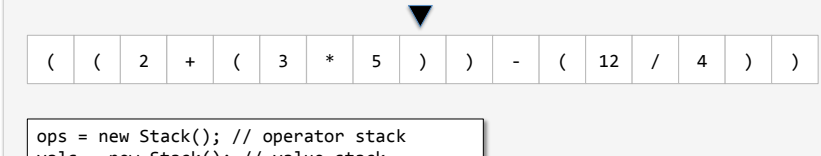

```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating infix



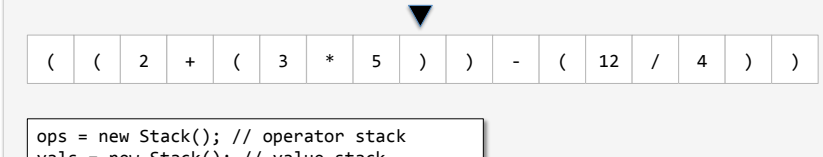

```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating infix




```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

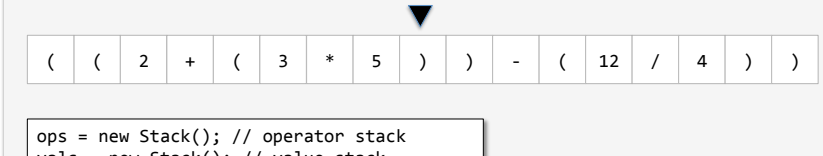
Evaluating infix




```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

op: *

Evaluating infix

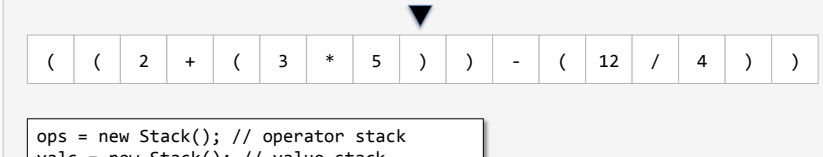



```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

op: *

rightVal: 5

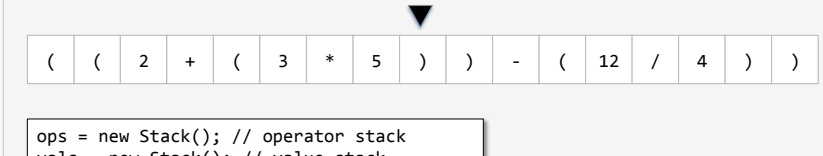
Evaluating infix




```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

op: *
rightVal: 5
leftVal: 3

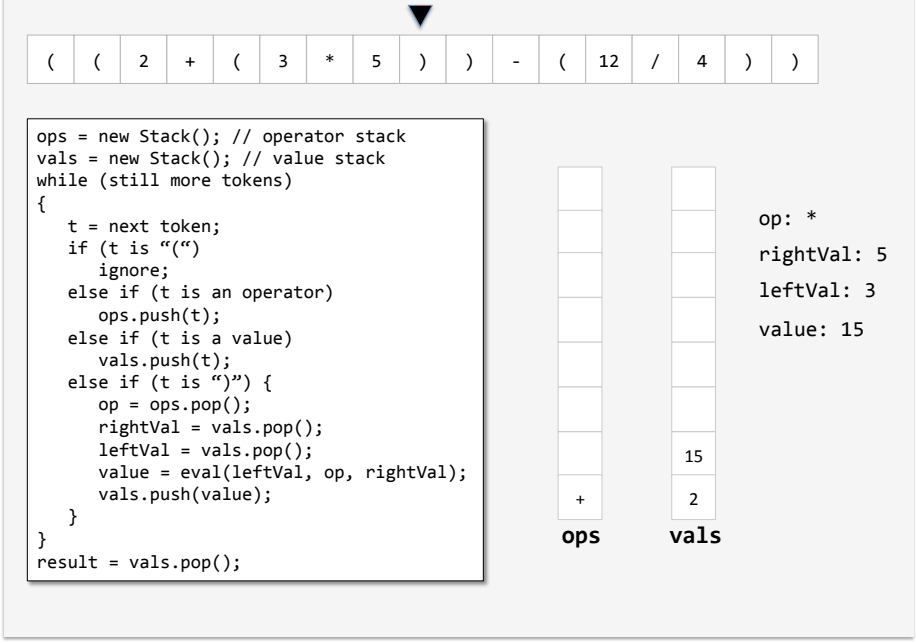
Evaluating infix



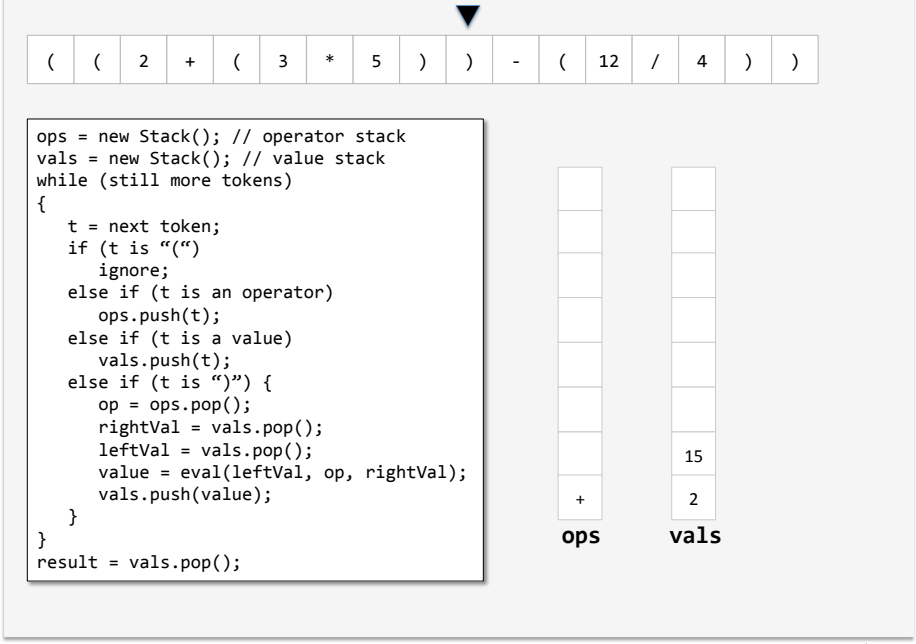

```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

op: *
rightVal: 5
leftVal: 3
value: 15

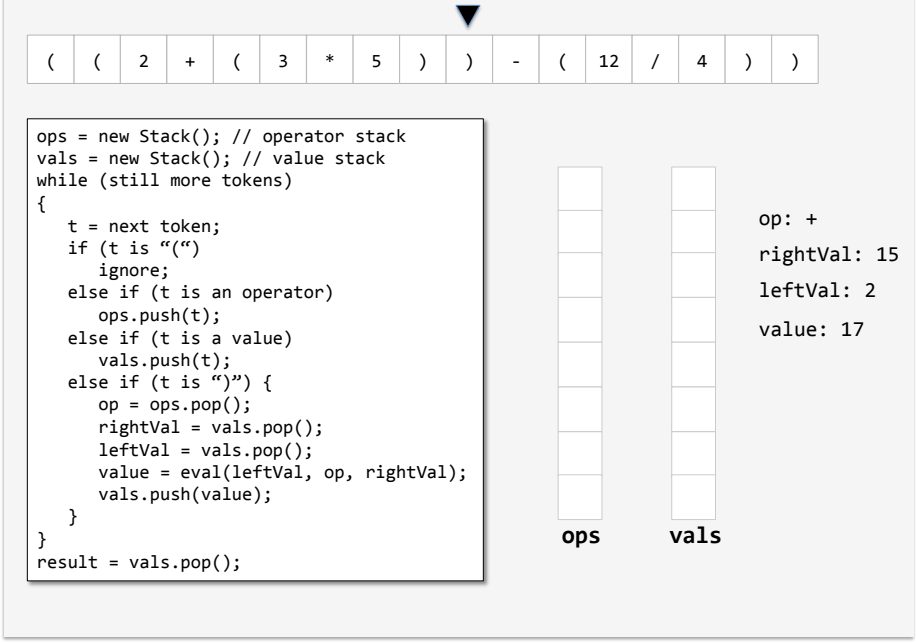
Evaluating infix



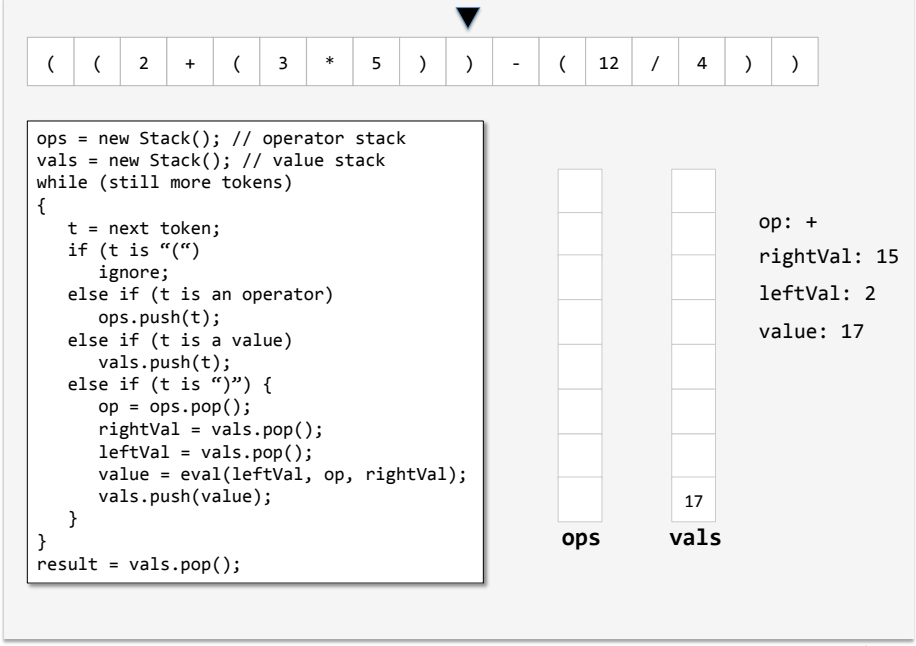
Evaluating infix



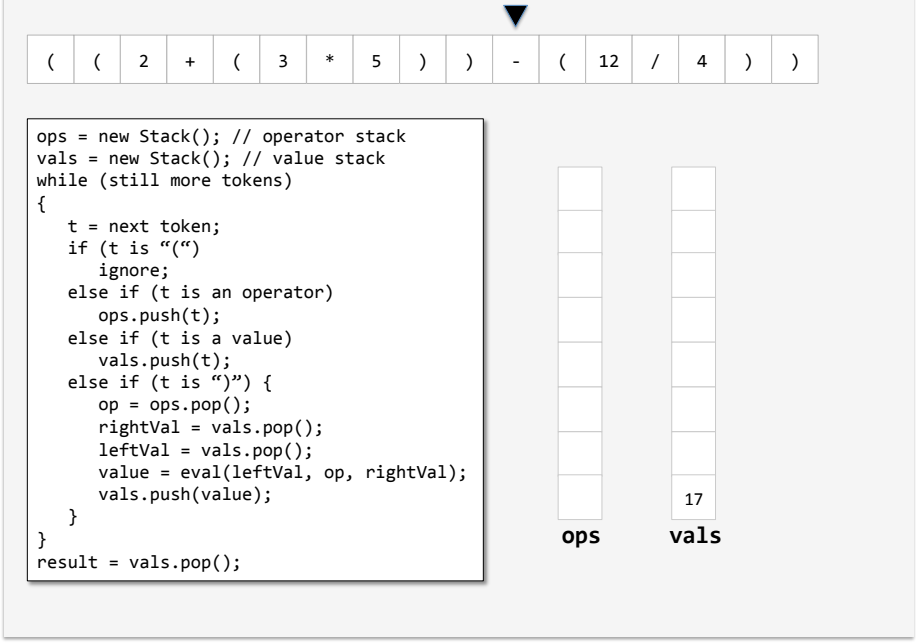
Evaluating infix



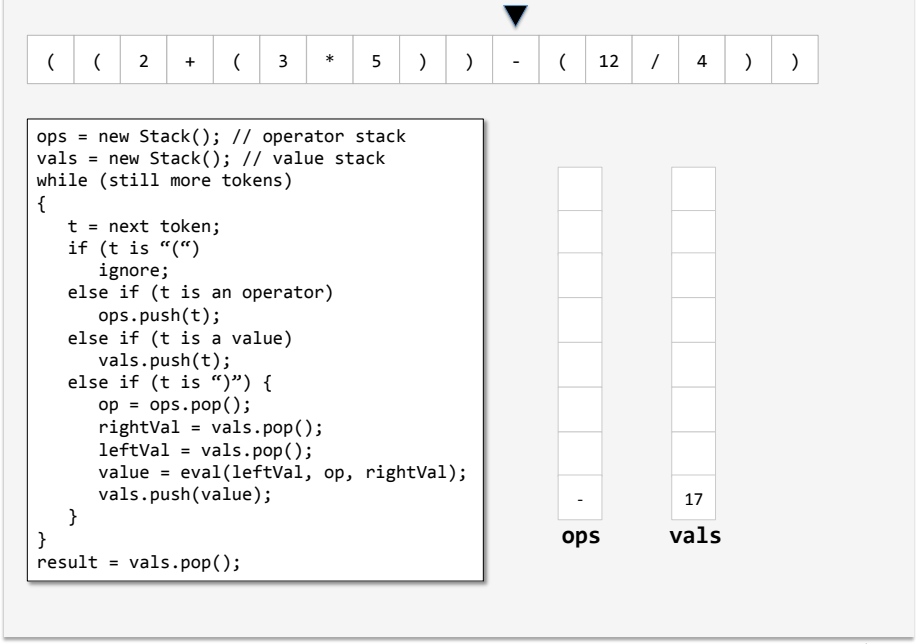
Evaluating infix



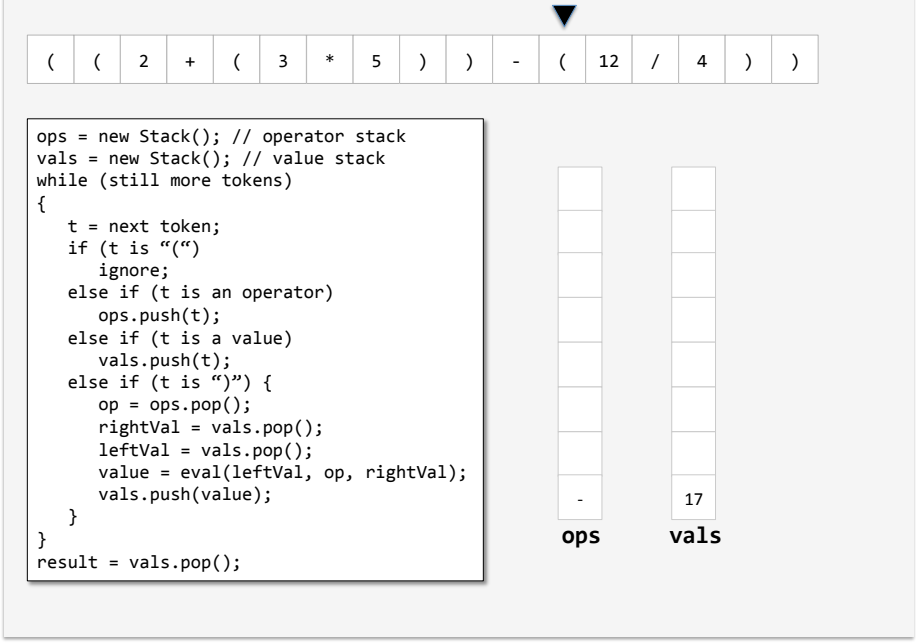
Evaluating infix



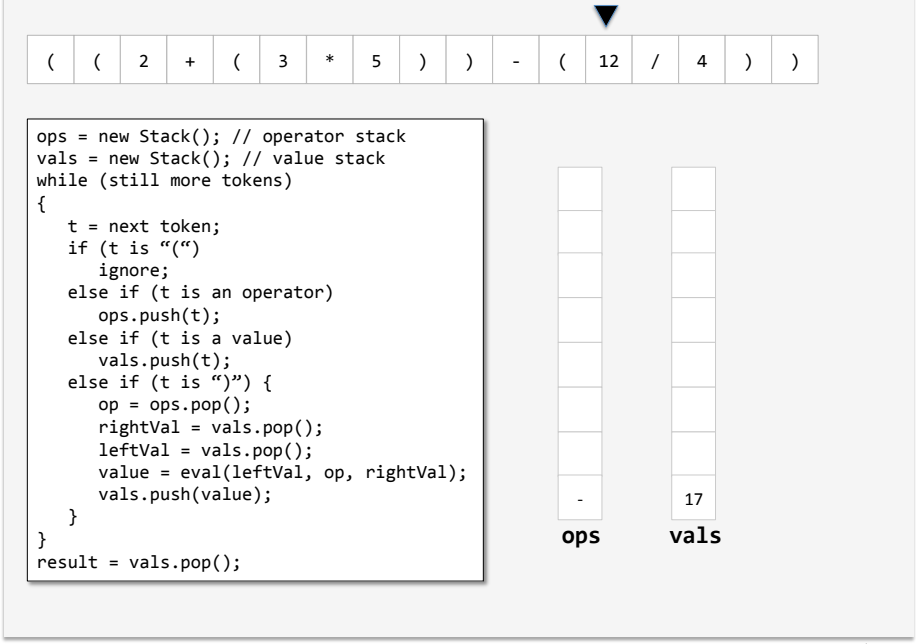
Evaluating infix



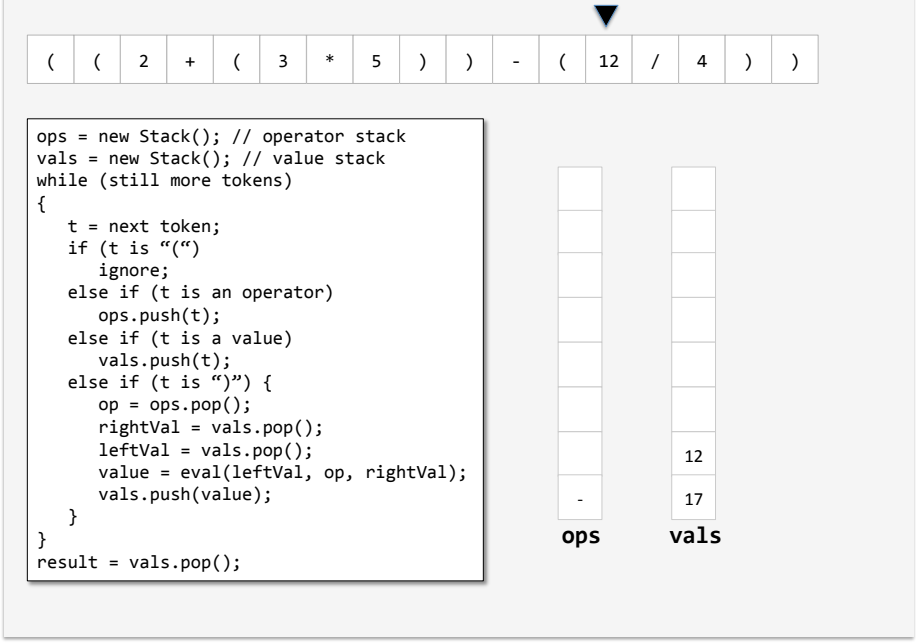
Evaluating infix



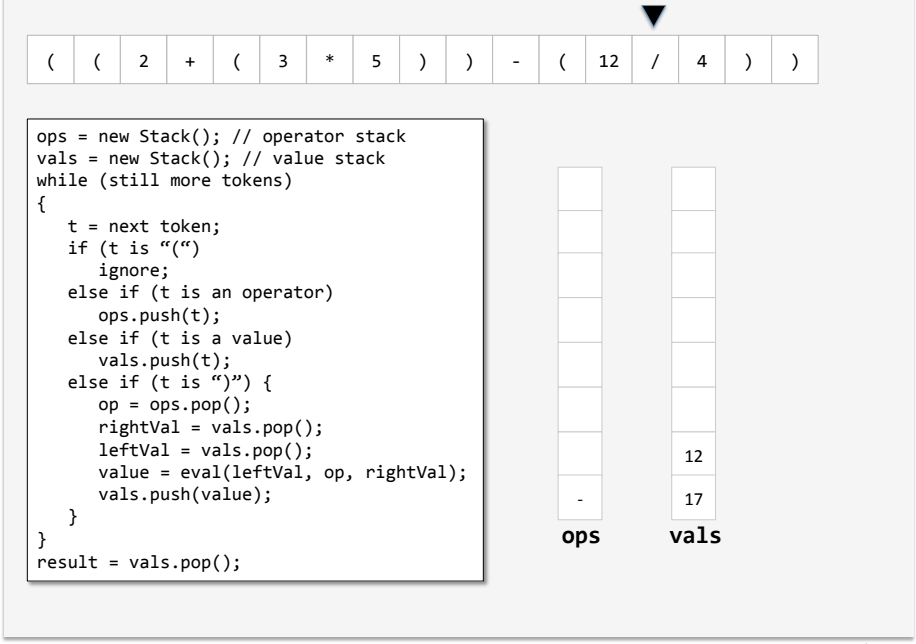
Evaluating infix



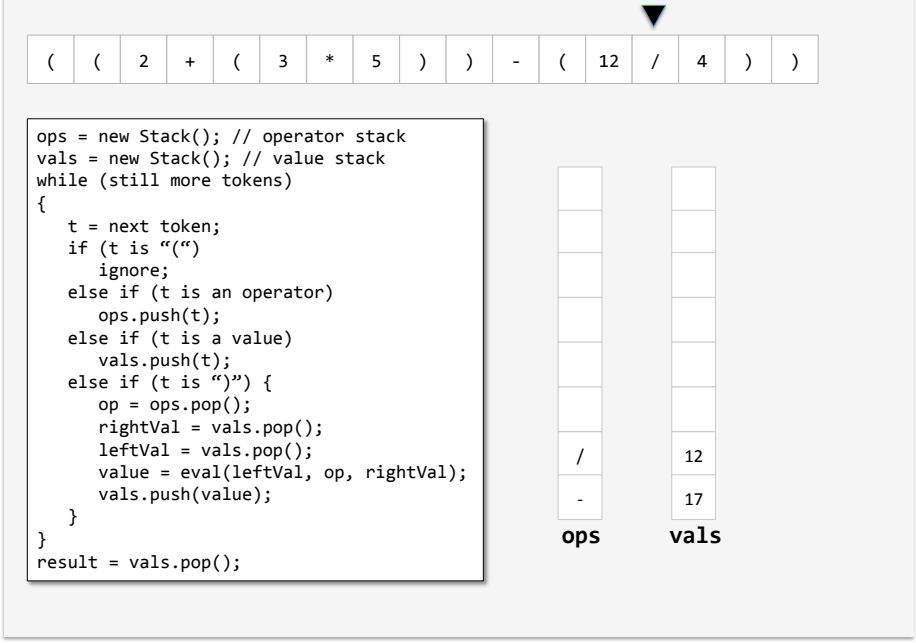
Evaluating infix



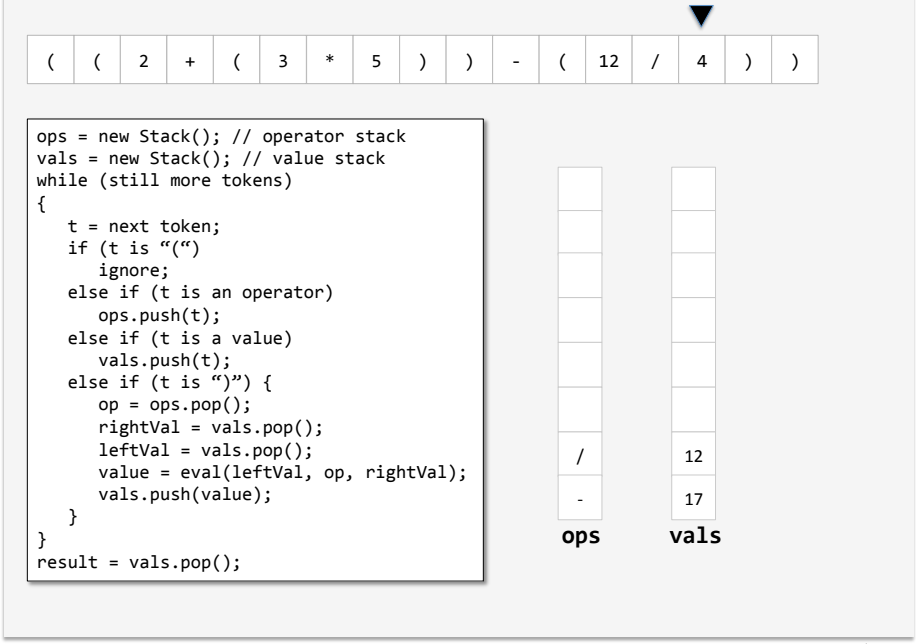
Evaluating infix



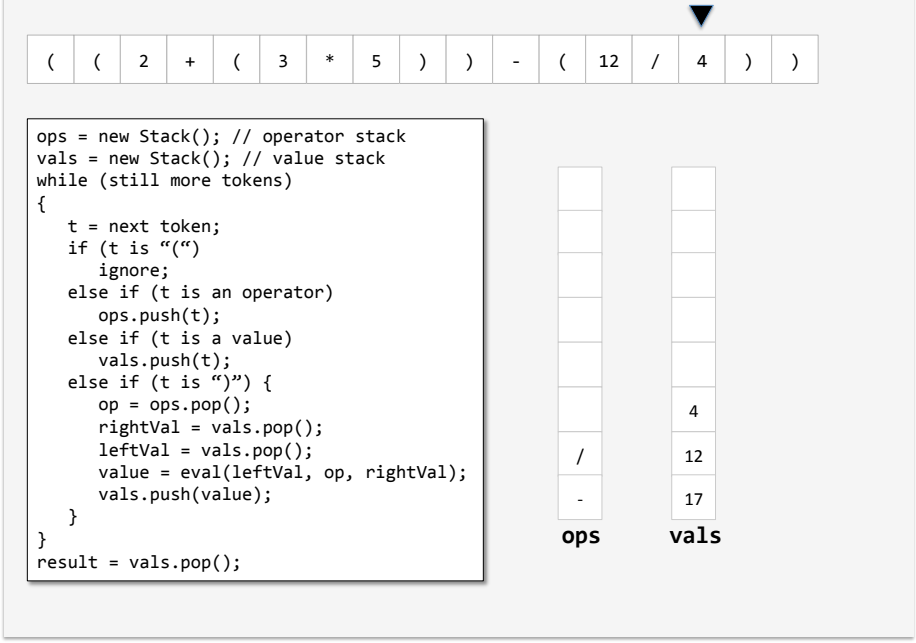
Evaluating infix



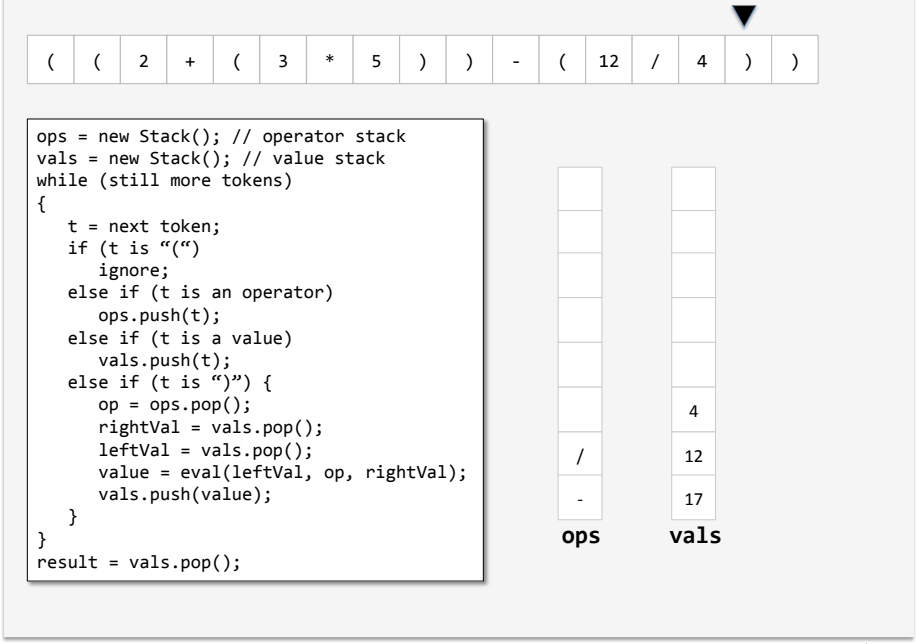
Evaluating infix



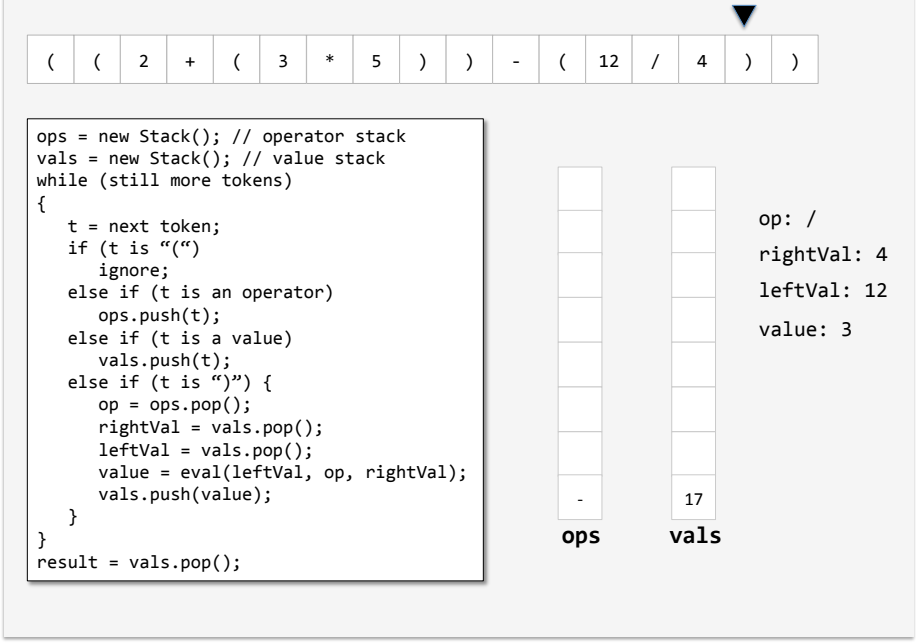
Evaluating infix



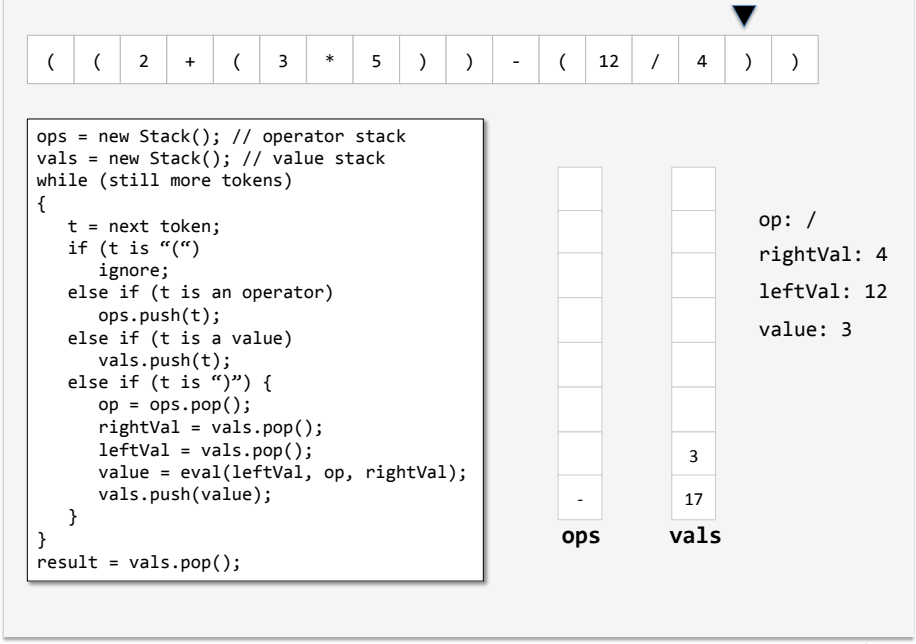
Evaluating infix



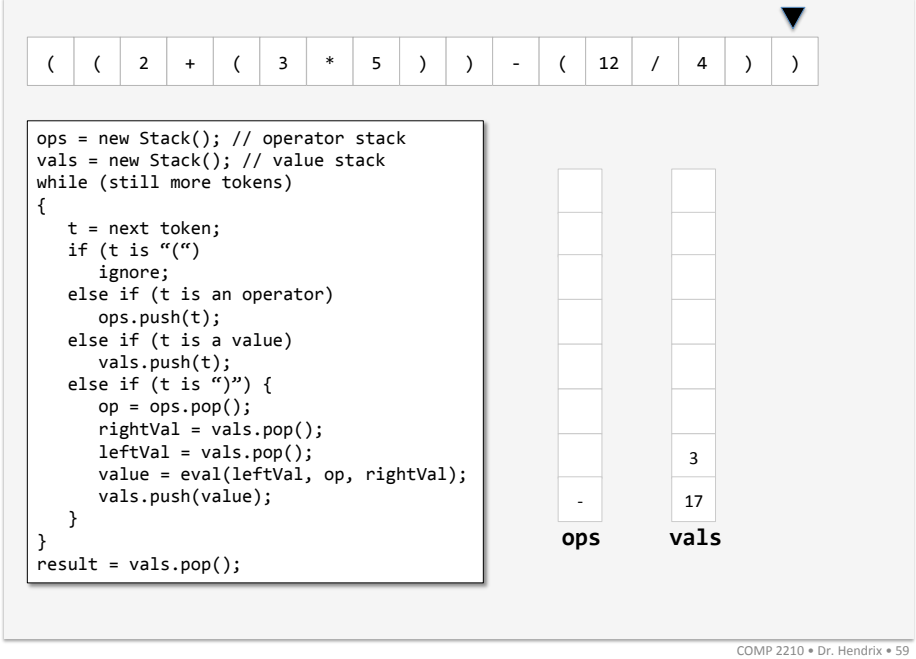
Evaluating infix



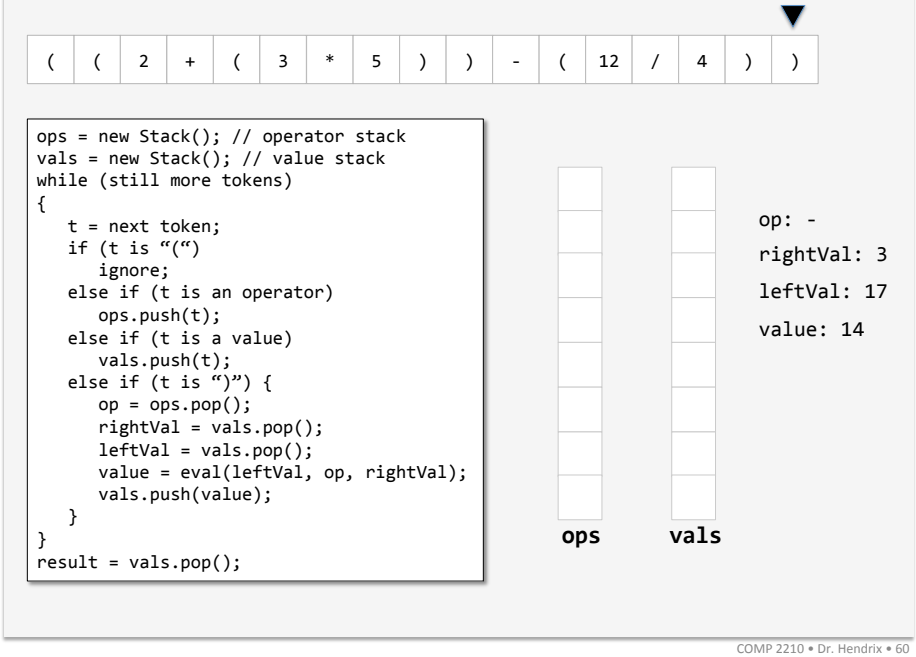
Evaluating infix



Evaluating infix



Evaluating infix



Evaluating infix


```
ops = new Stack(); // operator stack
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is "(")
      ignore;
   else if (t is an operator)
      ops.push(t);
   else if (t is a value)
      vals.push(t);
   else if (t is ")") {
      op = ops.pop();
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, op, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

op: -
rightVal: 3
leftVal: 17
value: 14

Postfix

Jan Łukasiewicz

"I came upon the idea of a parenthesis-free notation in 1924 . . ."

We can come upon a similar idea by making two observations:

(1) The infix evaluation algorithm would compute the same result if we moved each operator adjacent to its corresponding right parenthesis.

```
((2 + (3 * 5)) – (12 / 4))  ⟶  ((2 (3 5 *) +) (12 4 /) -)
```

(2) This makes all the parentheses redundant.

```
((2 (3 5 *) +) (12 4 /) -)  ⟶  2 3 5 * + 12 4 / -
```

Postfix

(Or, "reverse Polish notation" - RPN)

Infix	Binary operators appear between operands	3 + 4	
Prefix	Binary operators appear before operands	+ 3 4	*(Jan Łukasiewicz)*
Postfix	Binary operators appear after operands	3 4 +	*(Charles L. Hamblin)*

Evaluating postfix `2 3 5 * + 12 4 / -`

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

Evaluating postfix `2 3 5 * + 12 4 / -`

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

rightVal: 5

leftVal: 3

value: 15

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

vals

rightVal: 5

leftVal: 3

value: 15

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

vals

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

rightVal: 15

leftVal: 2

value: 17

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

rightVal: 15

leftVal: 2

value: 17

Evaluating postfix 2 3 5 * + 12 4 / -

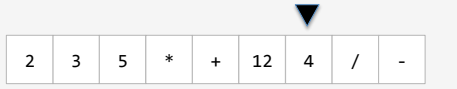

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

Evaluating postfix 2 3 5 * + 12 4 / -


```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating postfix 2 3 5 * + 12 4 / -




```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating postfix 2 3 5 * + 12 4 / -

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating postfix 2 3 5 * + 12 4 / -

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

Evaluating postfix 2 3 5 * + 12 4 / -

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

rightVal: 4

leftVal: 12

value: 3

Evaluating postfix 2 3 5 * + 12 4 / -

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

vals

rightVal: 4

leftVal: 12

value: 3

Evaluating postfix 2 3 5 * + 12 4 / -

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```

vals

Evaluating postfix 2 3 5 * + 12 4 / -

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

```
rightVal: 3
leftVal: 17
value: 14
```

Evaluating postfix

`2 3 5 * + 12 4 / -`

2	3	5	*	+	12	4	/	-

```
vals = new Stack(); // value stack
while (still more tokens)
{
   t = next token;
   if (t is a value)
      vals.push(t);
   else if (t is an operator) {
      rightVal = vals.pop();
      leftVal = vals.pop();
      value = eval(leftVal, t, rightVal);
      vals.push(value);
   }
}
result = vals.pop();
```


vals

```
rightVal: 3
leftVal: 17
value: 14
```


Who cares??

http://h20331.www2.hp.com/hpsub/us/en/rpn-calculator.html

http://www.hpmuseum.org/

http://en.wikipedia.org/wiki/PostScript

http://www.adobe.com/products/postscript/

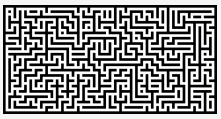
http://java.sun.com/docs/books/vmspec/2nd-edition/html/Compiling.doc.html

http://en.wikipedia.org/wiki/Java_bytecode

http://en.wikipedia.org/wiki/Stack_machine

maze search

Maze assumptions



A ***maze*** is a tour puzzle that has fixed walls and passages with complex branching possible (multicursal).

A *labyrinth* is a tour puzzle that has fixed walls and passages with no branching (unicursal).

(1) We will solve 2D mazes. A (rectangular) labyrinth can be considered a special case of this.

(2) We will represent a maze as a 2D grid of positions.

Each position is either open or blocked.

Open: *Blocked:*

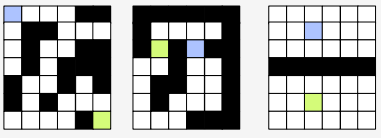
Exactly one open position is designated the start, and exactly one is designated the finish.

Start: *Finish:*

Movement is in only four directions: up, down, left, right. No diagonal moves.

(3) No minotaurs.

Examples:



Maze search

Let's assume this is our goal: Find the finish position.

We might want to do more later, like remembering how we got there, make sure it's the shortest path, etc. But for now, just finding the finish position is all we care about.

Strategy:

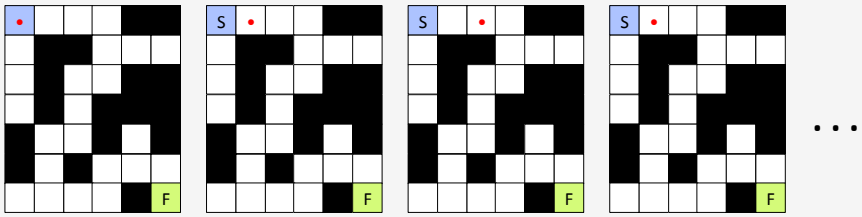
```
while (the current position is not the finish) {
   choose an adjacent open position and move there
}
```

"random mouse"

Simple, yes. Usable, no.

This is basically what we want to do, but we have to put some constraints on the choice of moves to avoid going in circles.

The red dot (•) indicates where we are.



Maze search

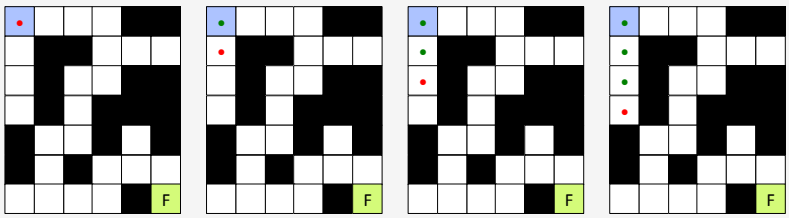
Goal: Find the finish position.

Strategy:

```
while (the current position is not the finish) {
   choose an adjacent position such that:
      - the position is open
      - we have not been there before
   move to this new position and mark it visited
}
```

We have to drop breadcrumbs as we go so we'll recognize positions that we've already visited.

The red dot (•) indicates where we are. The green dot (•) indicates where we've been.



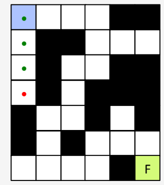
What now? There is no position that meets our selection criteria.

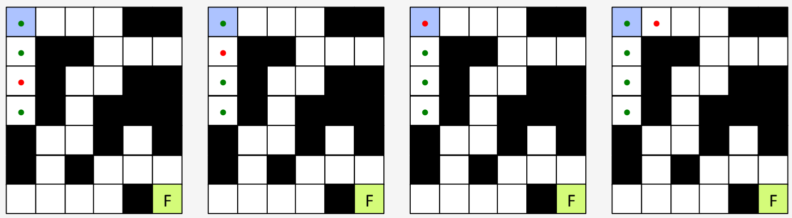
When we reach a dead end, we have to **backtrack**.

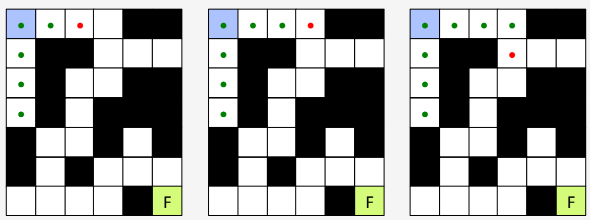
Maze search

Strategy:

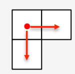
```
while (the current position is not the finish) {
   if (∃ adj position open and not yet visited)
       move to such a position
       mark it visited
   else
       backtrack
}
```







When we have a choice of moves, how do we decide?



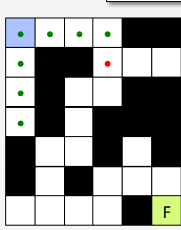
(1) Choose randomly

(2) Apply a heuristic

Maze search

Strategy:

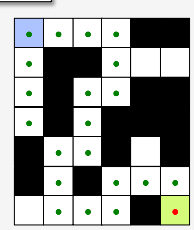
```
while (the current position is not the finish) {
   if (∃ adj position open and not yet visited)
       move to such a position
       mark it visited
   else
       backtrack
}
```



Since we can backtrack, the choice we make will only affect how quickly we find the finish.

If there is a path from start to finish, our strategy will find it.

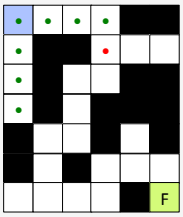
Assuming we make the right choices at each branch, then the maze will be:



Maze search

Strategy:

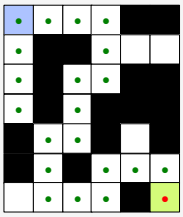
```
while (the current position is not the finish) {
   if (∃ adj position open and not yet visited)
       move to such a position
       mark it visited
   else
       backtrack
}
```



Since we can backtrack, the choice we make will only affect how quickly we find the finish.

If there is a path from start to finish, our strategy will find it.

Assuming we make the right choices at each branch, then the maze will be:



How to implement forward and backward moves?

Use a stack:

top = where we are (•)

push = move forward

pop = move backward

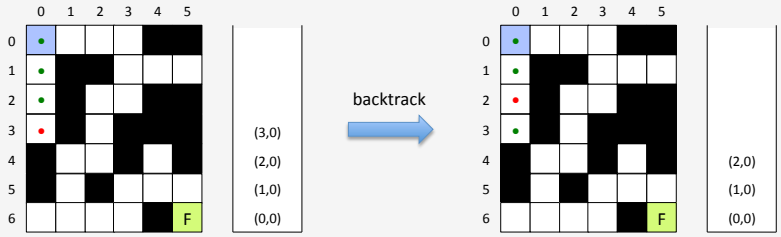
What does the stack contain while the algorithm is running?

What does the stack contain when the algorithm stops?

Maze search

Strategy:

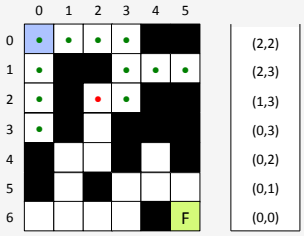
```
push the start position onto the stack
while (current position is not the finish) {
   if (∃ adj position open and not yet visited)
       push such a position onto the stack
       mark it visited
   else
       pop the stack
}
```



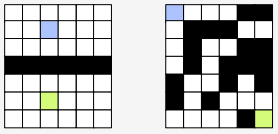
Maze search

Strategy:

```
push the start position onto the stack
while (current position is not the finish) {
   if (∃ adj position open and not yet visited)
       push such a position onto the stack
       mark it visited
   else
       pop the stack
}
```



What if the maze has no solution?



We need to modify the loop bound to take this into account.

The while loop has to stop if either of these things happen:

- We find the finish position.
- We have no more *viable* moves.

Maze search

DFS:

```
push the start position onto the stack
while ((current position != finish) && (stack not empty)) {
   if (∃ adj position open and not yet visited)
       push such a position onto the stack
       mark it visited
   else
       pop the stack
}
```

This is a **depth-first** search strategy.

We explore paths in the maze as deeply as possible. When we reach a dead end, only then do we backtrack to the closest branch.

Alternative? A **breadth-first** search strategy.

We explore the immediate neighborhood first, then branch out.

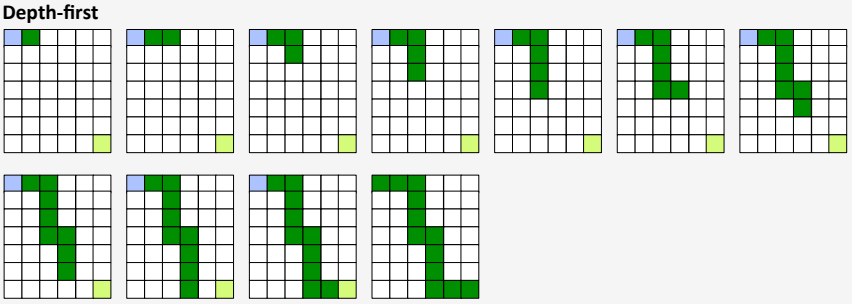
That is, explore the current position's neighbors, then their neighbors, then their neighbors ...

BFS:

```
enqueue the start position onto the queue and mark it visited
while ((current position != finish) && (queue not empty)) {
    dequeue the current position from the queue
    enqueue all its viable neighbors and mark them visited
}
```

Depth-first and Breadth-first

Depth-first



Breadth-first