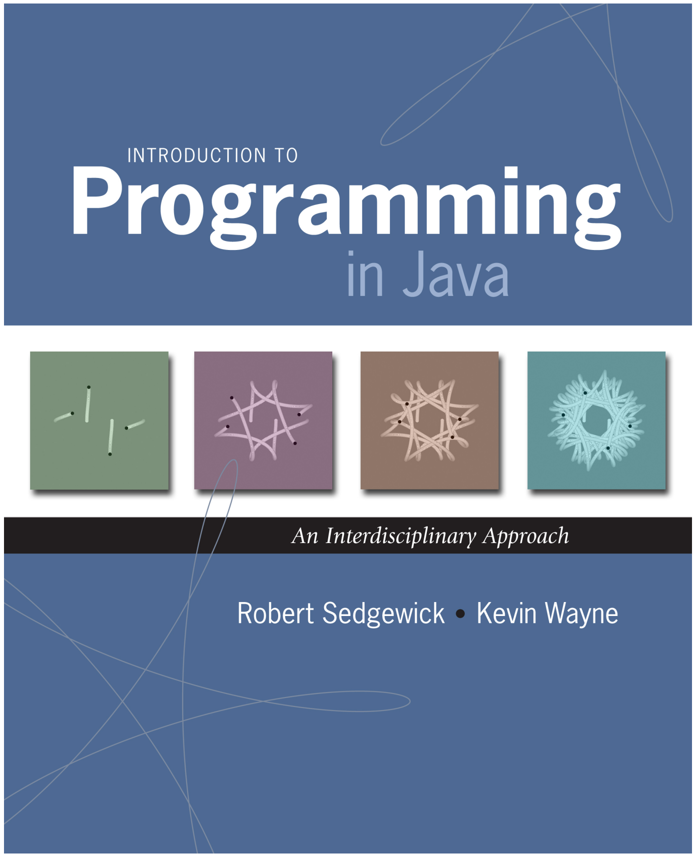
4.3 Stacks and Queues



Data Types and Data Structures

Data types.

- Set of values.
- Set of operations on those values.
- Some are built in to Java: `int`, `double`, `char`, . . .
- Most are not: `Complex`, `Picture`, `Stack`, `Queue`, `Graph`, …

this lecture

Data structures.

- Represent data or relationships among data.
- Some are built into Java: arrays, `String`,…
- Most are not: linked list, circular list, tree, sparse array, graph, . . .

this lecture TSP assignment next lecture

Collections

Fundamental data types.

- Set of operations (add, remove, test if empty) on generic data.
- Intent is clear when we insert.
- Which item do we remove?

Stack. [LIFO = last in first out] ← this lecture

- Remove the item most recently added.
- Ex: cafeteria trays, Web surfing.

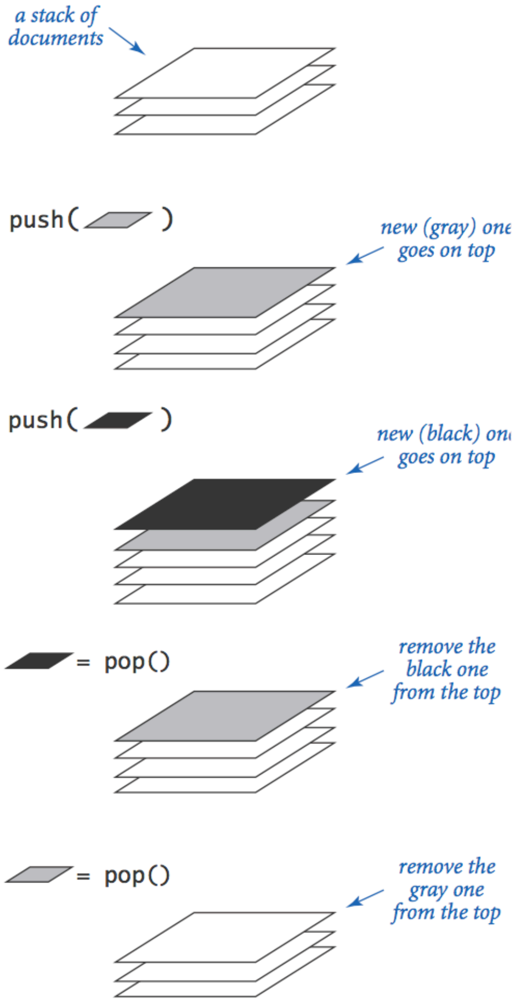
Queue. [FIFO = first in, first out]

- Remove the item least recently added.
- Ex: Registrar's line.

Symbol table. ← next lecture

- Remove the item with a given key.
- Ex: Phone book.

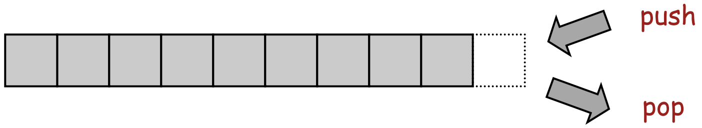
Stacks


a stack of documents
push( )
new (gray) one goes on top
push( )
new (black) on goes on top
= pop()
remove the black one from the top
= pop()
remove the gray one from the top

Stack API

public class *StackOfStrings

	*StackOfStrings()	*create an empty stack*
boolean	isEmpty()	*is the stack empty?*
void	push(String item)	*push a string onto the stack*
String	pop()	*pop the stack*



```
public class Reverse {
   public static void main(String[] args) {
      StackOfStrings stack = new StackOfStrings();
      while (!StdIn.isEmpty())
         stack.push(StdIn.readString());
      while (!stack.isEmpty())
         StdOut.println(stack.pop());
   }
}
```

Stack Client Example 1: Reverse

```
public class Reverse {
   public static void main(String[] args) {
      StackOfStrings stack = new StackOfStrings();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         stack.push(s);
      }
      while (!stack.isEmpty()) {
         String s = stack.pop ();
         StdOut.println(s);
      }
   }
}
```

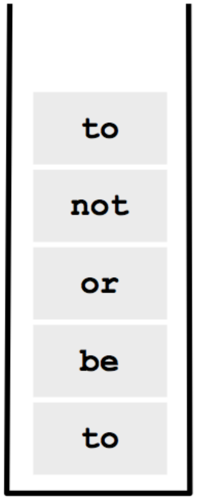
```
% more tiny.txt
it was the best of times

% java Reverse < tiny.txt
times of best the was it
```

stack contents when standard input is empty

Stack Client Example 2: Test Client

```
public static void main(String[] args) {
   StackOfStrings stack = new StackOfStrings();
   while (!StdIn.isEmpty()) {
      String s = StdIn.readString();
      if (s.equals("-"))
         StdOut.println(stack.pop());
      else
         stack.push(s);
   }
}
```

```
% more test.txt
to be or not to - be - - that - - - is

% java StackOfStrings < test.txt
to be not that or be
```



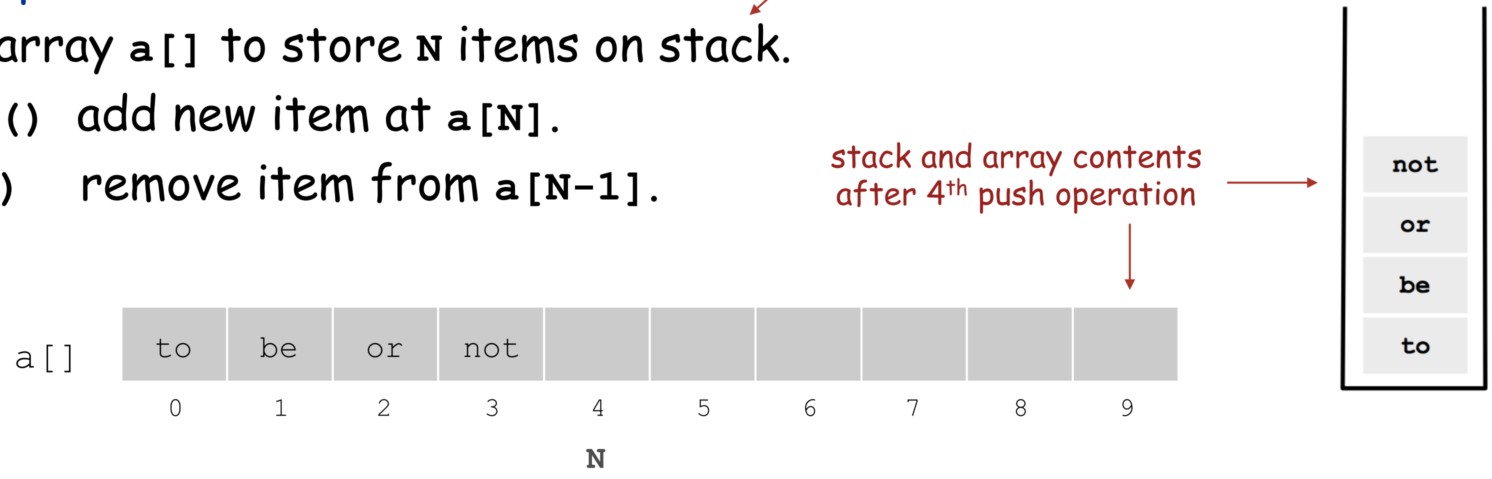
stack contents just before first pop operation

Stack: Array Implementation

how big to make array? [stay tuned]

Array implementation of a stack.

- Use array `a[]` to store `N` items on stack.
- `push()` add new item at `a[N]`.
- `pop()` remove item from `a[N-1]`.



stack and array contents after 4th push operation

temporary solution: make client provide capacity

```
public class ArrayStackOfStrings {
   private String[] a;
   private int N = 0;

   public ArrayStackOfStrings(int max) { a = new String[max]; }
   public boolean isEmpty()      { return (N == 0);  }
   public void push(String item) { a[N++] = item;    }
   public String pop()           { return a[--N];    }
}
```

Array Stack: Test Client Trace

	StdIn	StdOut	N	a[]				
				0	1	2	3	4
			0					
push	to		1	to				
	be		2	to	be			
	or		3	to	be	or		
	not		4	to	be	or	not	
	to		5	to	be	or	not	to
pop	-	to	4	to	be	or	not	to
	be		5	to	be	or	not	be
	-	be	4	to	be	or	not	be
	-	not	3	to	be	or	not	be
	that		4	to	be	or	that	be
	-	that	3	to	be	or	that	be
	-	or	2	to	be	or	that	be
	-	be	1	to	be	or	that	be
	is		2	to	is	or	not	to

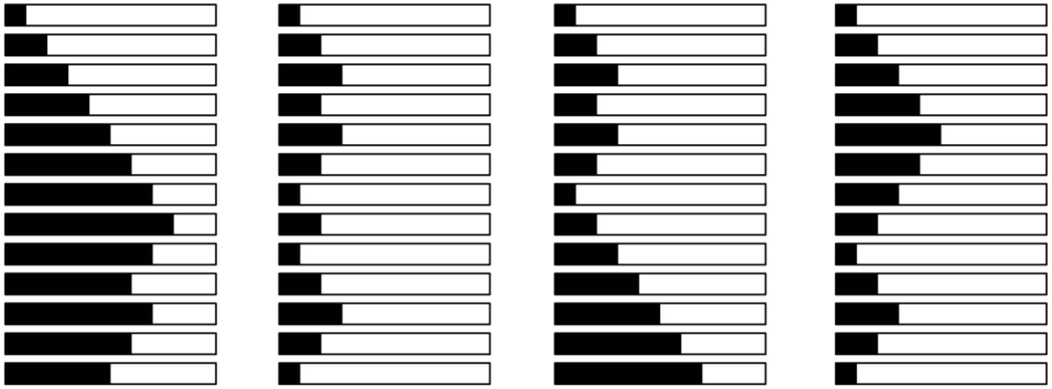
Array Stack: Performance

Running time. Push and pop take constant time.

Memory. Proportional to client-supplied capacity, not number of items.

Problem.

- API does not call for capacity (bad to change API).
- Client might use multiple stacks.
- Client might not know what capacity to use.



Challenge. Stack implementation where size is not fixed ahead of time.

Linked Lists

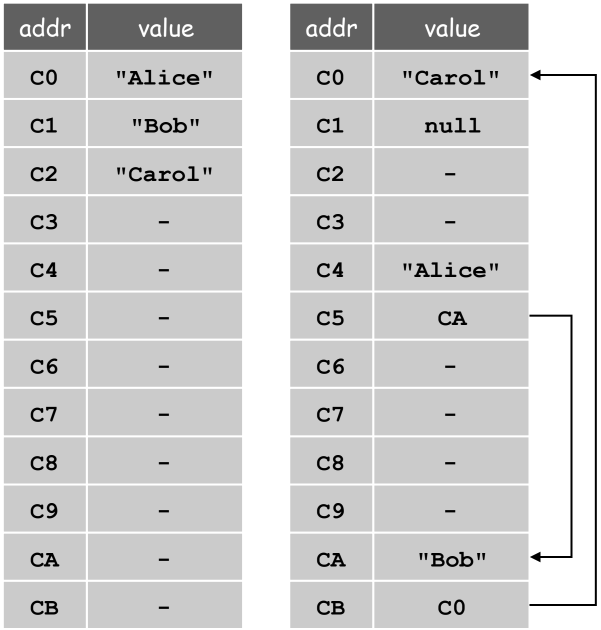
Sequential vs. Linked Allocation

Sequential allocation. Put object one after another.

- TOY: consecutive memory cells.
- Java: array of objects.

Linked allocation. Include in each object a link to the next one.

- TOY: link is memory address of next object.
- Java: link is reference to next object.



addr	value
C0	"Alice"
C1	"Bob"
C2	"Carol"
C3	-
C4	-
C5	-
C6	-
C7	-
C8	-
C9	-
CA	-
CB	-

array

addr	value
C0	"Carol"
C1	null
C2	-
C3	-
C4	"Alice"
C5	CA
C6	-
C7	-
C8	-
C9	-
CA	"Bob"
CB	C0

linked list

Key distinctions.

get i^{th} element

- Array: random access, fixed size.
- Linked list: sequential access, variable size.

get next element

Linked Lists

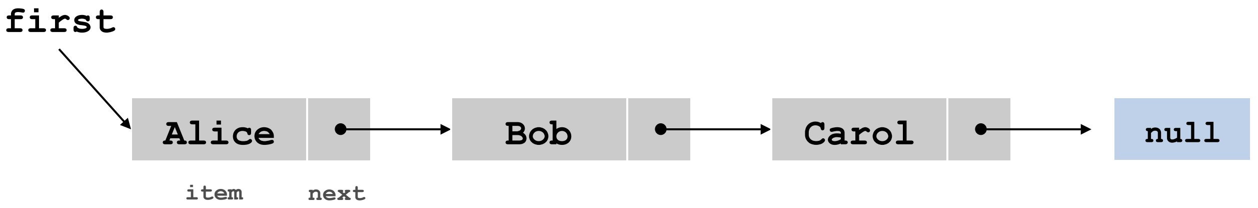
Linked list.

- A recursive data structure.
- An item plus a pointer to another linked list (or empty list).
- Unwind recursion: linked list is a sequence of items.

Node data type.

- A reference to a `String`.
- A reference to another `Node`.

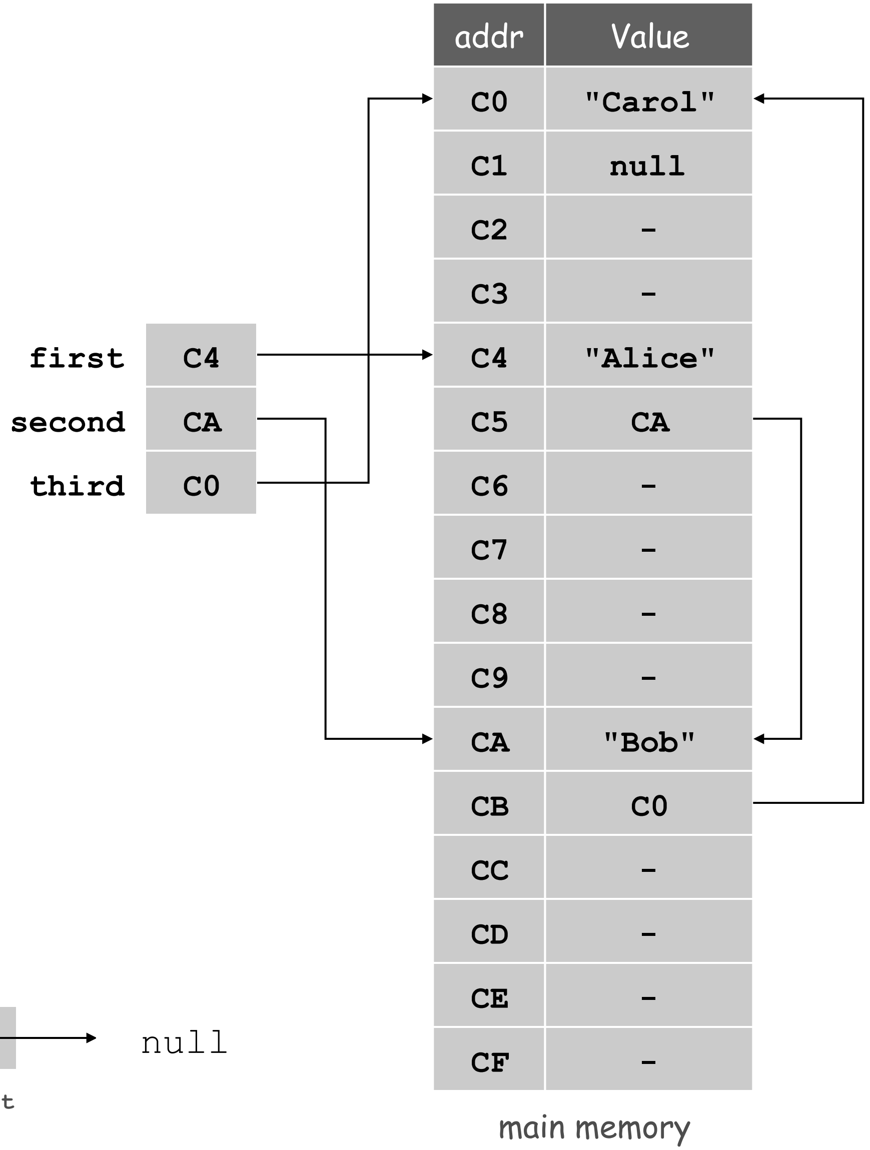
```
public class Node {
   private String item;
   private Node next;
}
```



special pointer value `null` terminates list

Building a Linked List

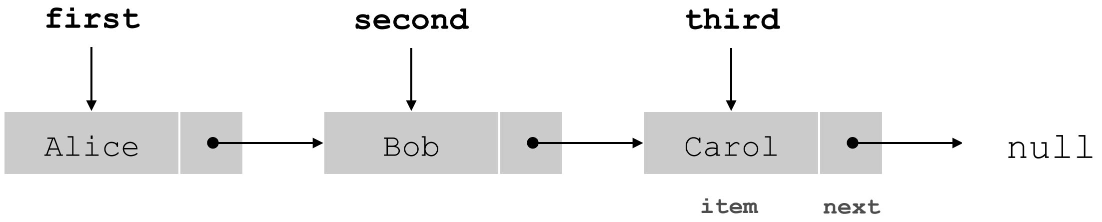
```
Node third  = new Node();
third.item  = "Carol";
third.next  = null;

Node second = new Node();
second.item = "Bob";
second.next = third;

Node first  = new Node();
first.item  = "Alice";
first.next  = second;
```



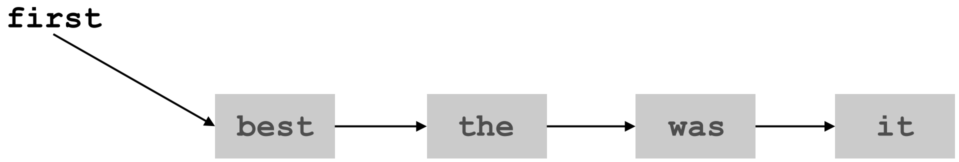
first	C4
second	CA
third	C0

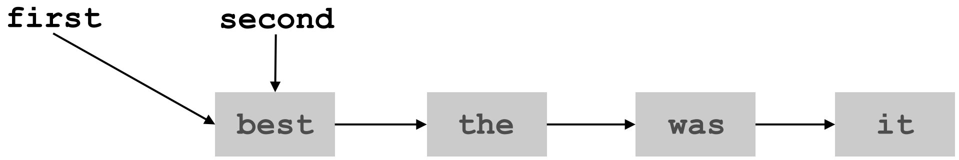
addr	Value
C0	"Carol"
C1	null
C2	-
C3	-
C4	"Alice"
C5	CA
C6	-
C7	-
C8	-
C9	-
CA	"Bob"
CB	C0
CC	-
CD	-
CE	-
CF	-

main memory

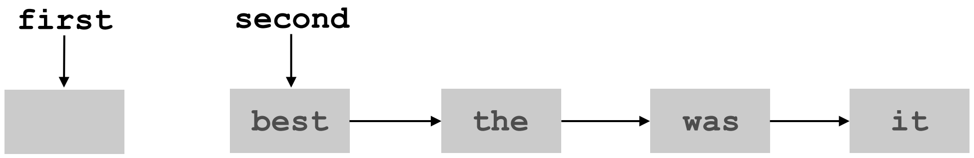


Stack Push: Linked List Implementation

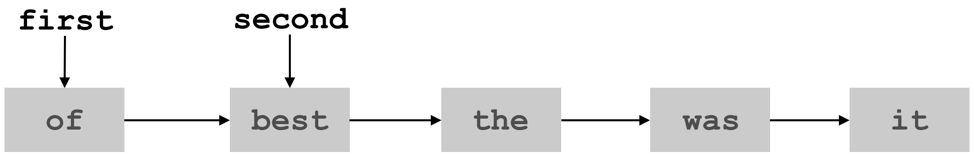




```
Node second = first;
```

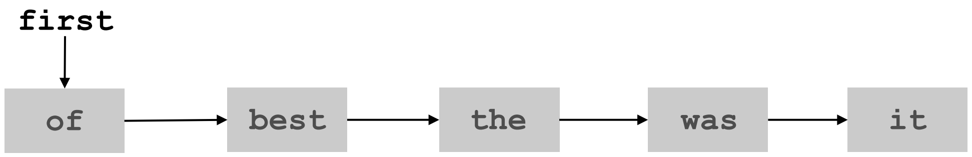


```
first = new Node();
```



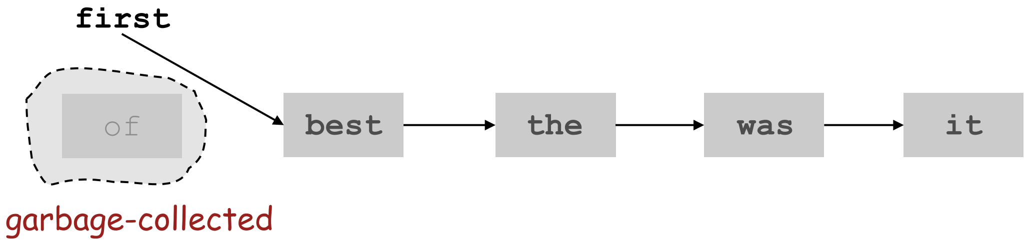
```
first.item = "of";
first.next = second;
```

Stack Pop: Linked List Implementation

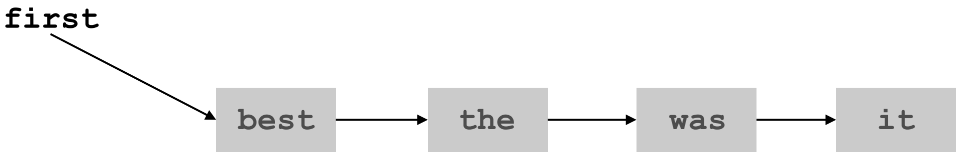


"of"

```
String item = first.item;
```



```
first = first.next;
```



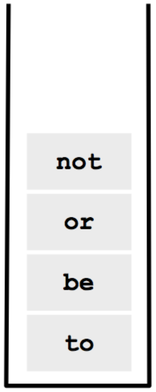
```
return item;
```

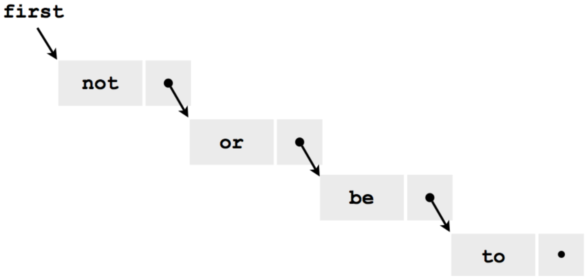
Stack: Linked List Implementation

```
public class LinkedStackOfStrings {
   private Node first = null;

   private class Node {
      private String item;
      private Node next;
   }

   public boolean isEmpty() { return first == null; }

   public void push(String item) {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public String pop() {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

"inner class"

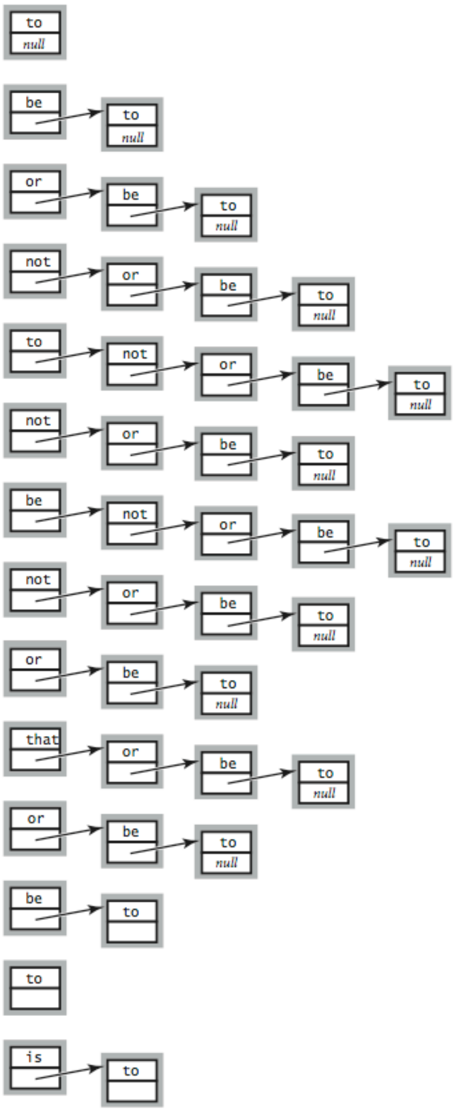
stack and linked list contents after 4th push operation





Linked List Stack: Test Client Trace

	StdIn	StdOut
push	to	
	be	
	or	
	not	
	to	
pop	-	to
	be	
	-	be
	-	not
	that	
	-	that
	-	or
	-	be
	is	



Linked List Stack: Performance

Running time. Push and pop take constant time.

Memory. Proportional to number of items in stack.

Stack Data Structures: Tradeoffs

Two data structures to implement **`Stack`** data type.

Array.

- Every push/pop operation take constant time.
- But… must fix maximum capacity of stack ahead of time.

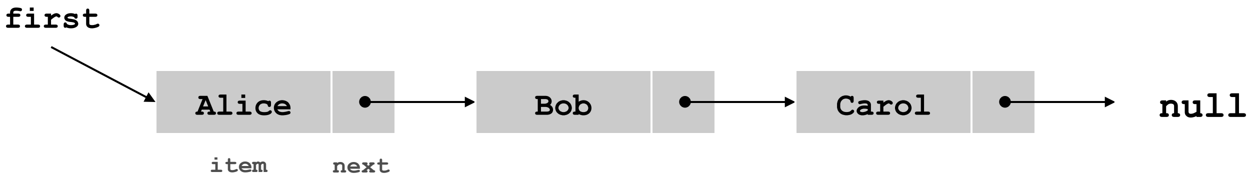
Linked list.

- Every push/pop operation takes constant time.
- But… uses extra space and time to deal with references.

List Processing Challenge 1

Q. What does the following code fragment do?

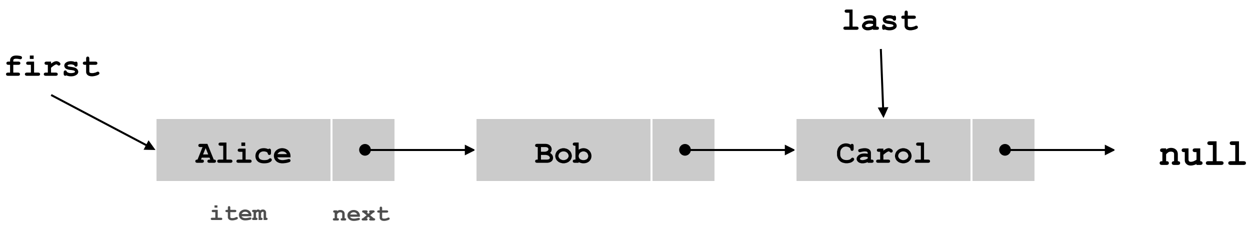
```
for (Node x = first; x != null; x = x.next) {
   StdOut.println(x.item);
}
```



List Processing Challenge 2

Q. What does the following code fragment do?

```
Node last = new Node();
last.item = StdIn.readString();
last.next = null;
Node first = last;
while (!StdIn.isEmpty()) {
   last.next = new Node();
   last = last.next;
   last.item = StdIn.readString();
   last.next = null;
}
```



Parameterized Data Types

Parameterized Data Types

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfInts`, …

Strawman. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

Generics

Generics. Parameterize stack by a single type.

"stack of apples"

parameterized type

```
Stack<Apple> stack = new Stack<Apple>();
Apple  a = new Apple();
Orange b = new Orange();
stack.push(a);
stack.push(b);    // compile-time error
a = stack.pop();
```

sample client

can't push an orange onto
a stack of apples

Generic Stack: Linked List Implementation

```
public class Stack<Item> {
   private Node first = null;

   private class Node {
      private Item item;
      private Node next;
   }

   public boolean isEmpty() { return first == null; }

   public void push(Item item) {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public Item pop() {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```

parameterized type name
(chosen by programmer)

Autoboxing

Generic stack implementation. Only permits reference types.

Wrapper type.

- Each primitive type has a wrapper reference type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast from primitive type to wrapper type.

Autounboxing. Automatic cast from wrapper type to primitive type.

```
Stack<Integer> stack = new Stack<Integer>();
stack.push(17);        // autobox   (int -> Integer)
int a = stack.pop();   // autounbox (Integer -> int)
```

Stack Applications

Real world applications.

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

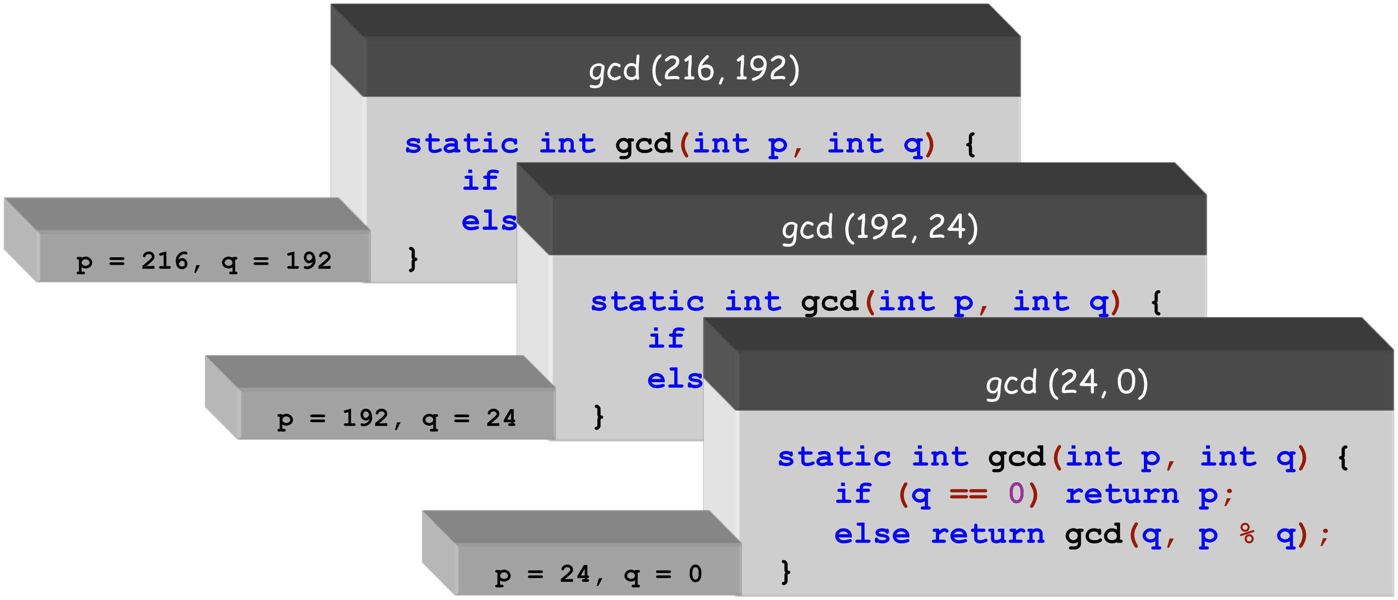
Function Calls

How a compiler implements functions.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



gcd (216, 192)

```
static int gcd(int p, int q) {
   if
   els
}
```

p = 216, q = 192

gcd (192, 24)

```
static int gcd(int p, int q) {
   if
   els
}
```

p = 192, q = 24

gcd (24, 0)

```
static int gcd(int p, int q) {
   if (q == 0) return p;
   else return gcd(q, p % q);
}
```

p = 24, q = 0

Arithmetic Expression Evaluation

Goal. Evaluate infix expressions.

(1 + ((2 + 3) * (4 * 5)))

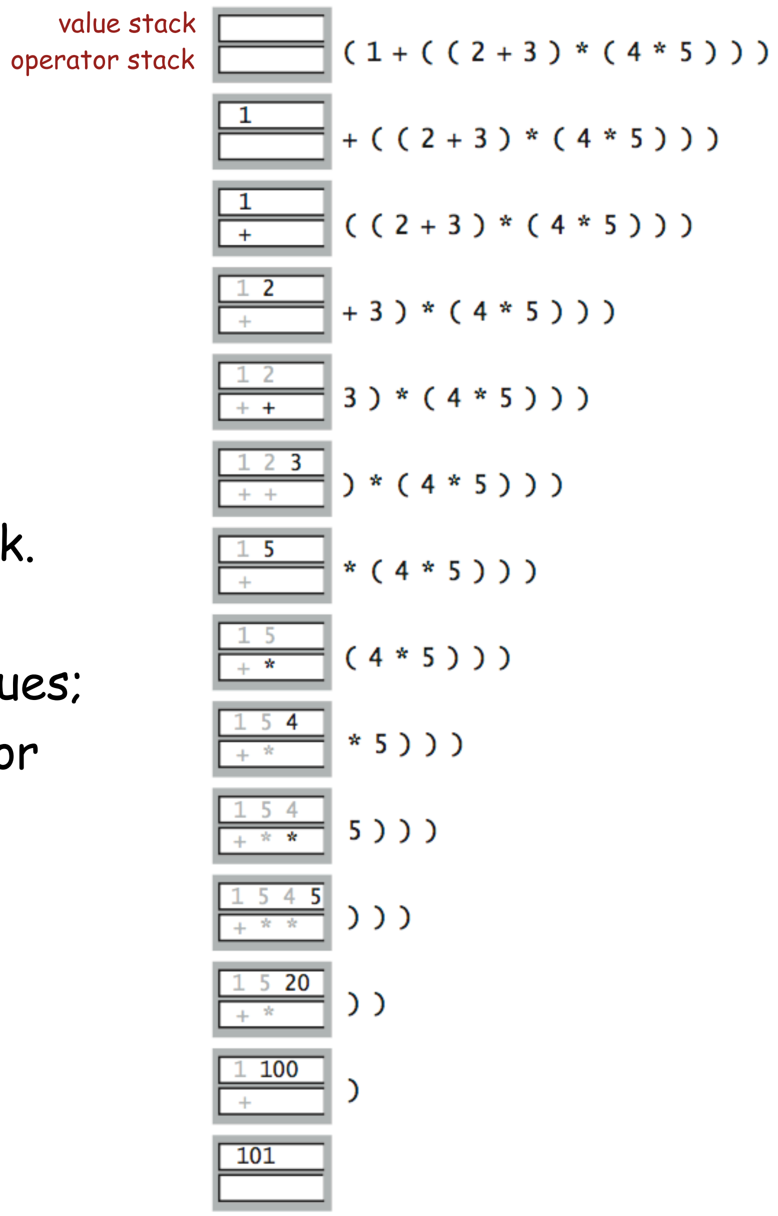
operand

operator

Two stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!



Arithmetic Expression Evaluation

```
public class Evaluate {
   public static void main(String[] args) {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")")) {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Correctness

Why correct? When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

So it's as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions. More ops, precedence order, associativity, whitespace.

```
1 + (2 - 3 - 4) * 5 * sqrt(6*6 + 7*7)
```

Stack-Based Programming Languages

Observation 1. Remarkably, the 2-stack algorithm computes the same value if the operator occurs *after* the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

Observation 2. All of the parentheses are redundant!

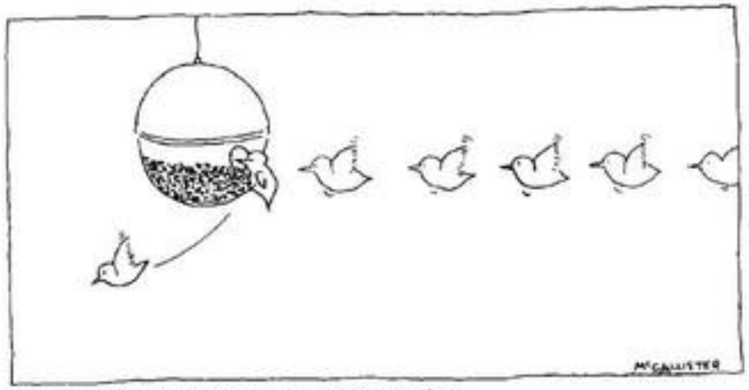
```
1 2 3 + 4 5 * * +
```

Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, …

Queues



Drawing by McCallister; © 1977 *The New Yorker* Magazine, Inc.

(CNN.COM GRPAHIC)

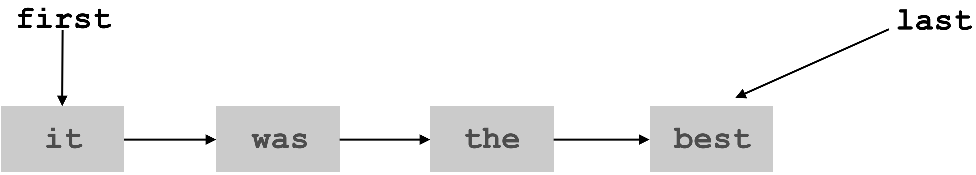
Queue API

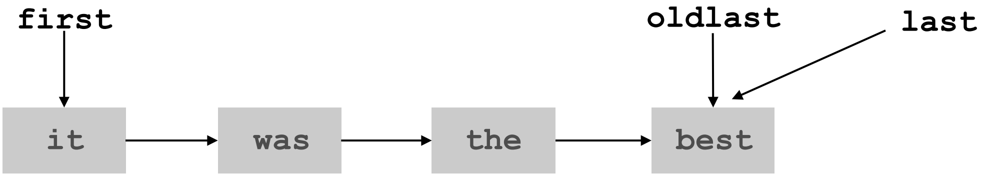
`public class Queue<Item>`

	`Queue<Item>()`	*create an empty queue*
`boolean`	`isEmpty()`	*is the queue empty?*
`void`	`enqueue(Item item)`	*enqueue an item*
`Item`	`dequeue()`	*dequeue an item*
`int`	`length()`	*queue length*

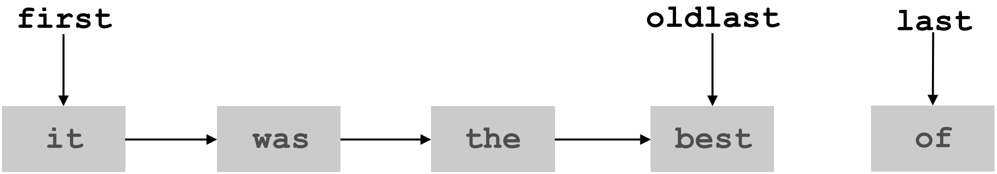
```
public static void main(String[] args) {
   Queue<String> q = new Queue<String>();
   q.enqueue("Vertigo");
   q.enqueue("Just Lose It");
   q.enqueue("Pieces of Me");
   q.enqueue("Pieces of Me");
   while(!q.isEmpty())
      StdOut.println(q.dequeue());
}
```

Enqueue: Linked List Implementation

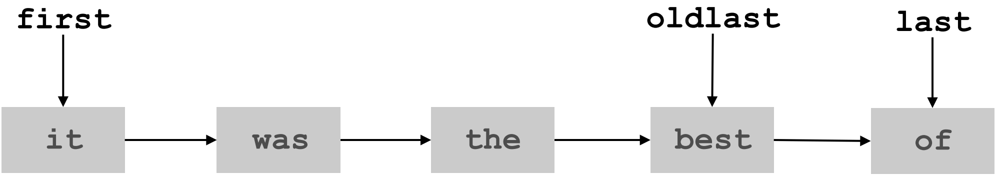




```
Node oldlast = last;
```

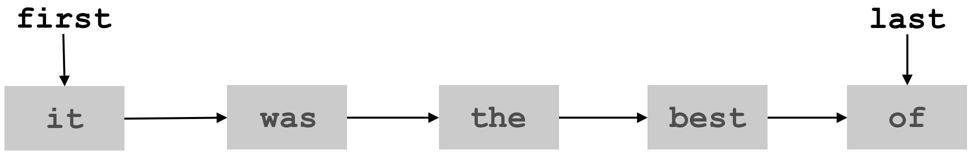


```
last = new Node();
last.item = "of";
last.next = null;
```

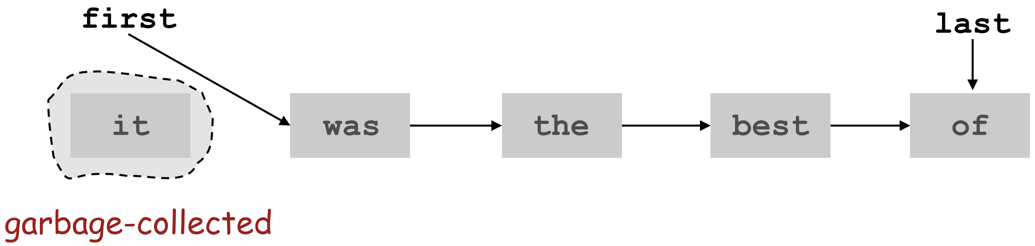


```
oldlast.next = last;
```

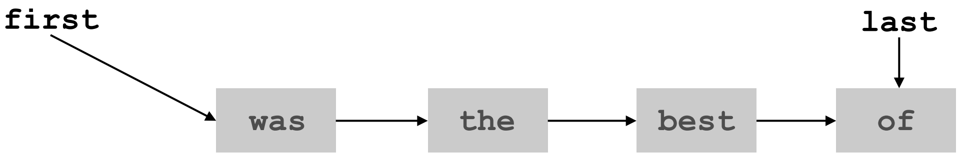
Dequeue: Linked List Implementation



```
String item = first.item;
```



```
first = first.next;
```



```
return item;
```

Queue: Linked List Implementation

```java
public class Queue<Item> {
   private Node first, last;

   private class Node { Item item; Node next; }

   public boolean isEmpty() { return first == null; }

   public void enqueue(Item item) {
      Node oldlast = last;
      last = new Node();
      last.item = item;
      last.next = null;
      if (isEmpty()) first = last;
      else           oldlast.next = last;
   }

   public Item dequeue() {
      Item item = first.item;
      first     = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```

Queue Applications

Some applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.

- Guitar string.
- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

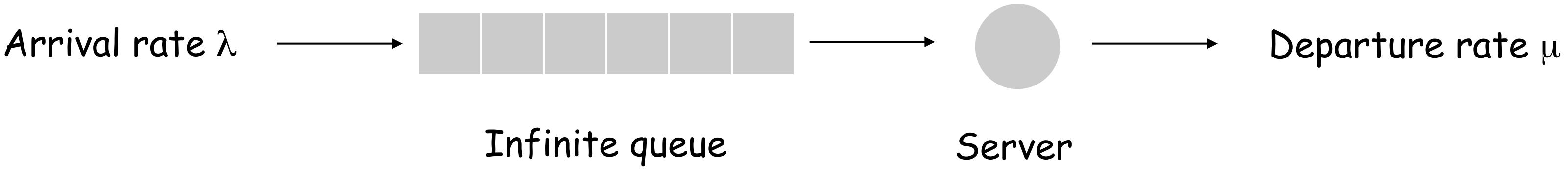
M/D/1 Queuing Model

M/D/1 queue.

- Customers are serviced at fixed rate of μ per minute.
- Customers arrive according to Poisson process at rate of λ per minute.

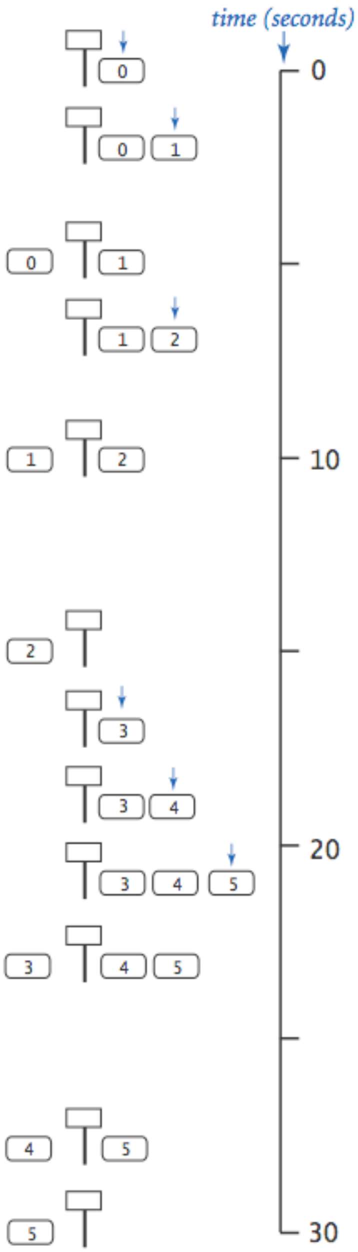
inter-arrival time has exponential distribution

$$\Pr[X \le x] \;=\; 1 - e^{-\lambda x}$$



Q. What is average wait time W of a customer?

Q. What is average number of customers L in system?



	arrival	*departure*	*wait*
0	0	5	5
1	2	10	8
2	7	15	8
3	17	23	6
4	19	28	9
5	21	30	9

Event-Based Simulation

```
public class MD1Queue {
   public static void main(String[] args) {
      double lambda = Double.parseDouble(args[0]);
      double mu     = Double.parseDouble(args[1]);
      Queue<Double> q = new Queue<Double>();
      double nextArrival = StdRandom.exp(lambda);
      double nextService = nextArrival + 1/mu;
      while(true) {
```

arrival

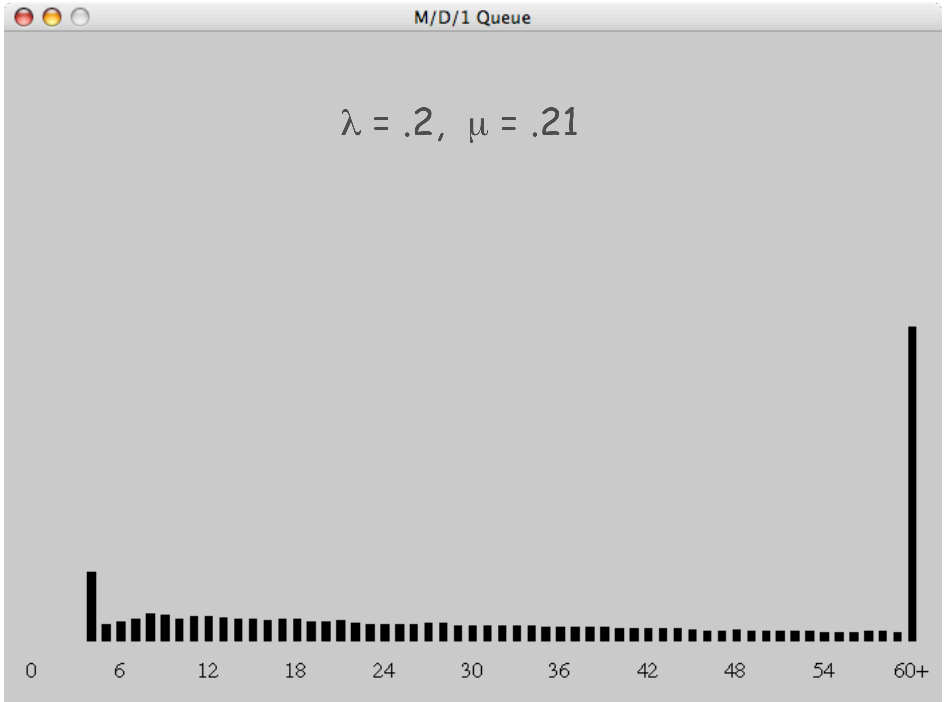
```
         if (nextArrival < nextService) {
            q.enqueue(nextArrival);
            nextArrival += StdRandom.exp(lambda);
         }
```

service

```
         else {
            double wait = nextService - q.dequeue();
            // add waiting time to histogram
            if (q.isEmpty()) nextService = nextArrival + 1/mu;
            else             nextService = nextService + 1/mu;
         }
      }
   }
}
```

M/D/1 Queue Analysis

Observation. As service rate approaches arrival rate, service goes to h***.



see ORFE 309 ↙

Queueing theory. $W = \frac{\lambda}{2\mu(\mu-\lambda)} + \frac{1}{\mu}, \quad L = \lambda W$

Little's law

Summary

Stacks and queues are fundamental ADTs.

- Array implementation.
- Linked list implementation.
- Different performance characteristics.

Many applications.