1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Fundamental data types.

- **Value**: collection of objects.
- **Operations**: `insert`, `remove`, `iterate`, `test if empty`.
- **Intent is clear when we insert**.
- **Which item do we remove?**

**Stack**. Examine the item most recently added.  `LIFO = "last in first out"`

**Queue**. Examine the item least recently added.  `FIFO = "first in first out"`
Separate interface and implementation.
Ex: stack, queue, bag, priority queue, symbol table, union-find, ....

Benefits.
• **Client** can't know details of implementation ⇒
  client has many implementation from which to choose.
• Implementation can't know details of client needs ⇒
  many clients can re-use the same implementation.
• **Design**: creates modular, reusable libraries.
• **Performance**: use optimized implementation where it matters.

**Client**: program using operations defined in interface.
**Implementation**: actual code implementing operations.
**Interface**: description of data type, basic operations.
- stacks
  - resizing arrays
  - queues
  - generics
  - iterators
  - applications
**Stack API**

**Warmup API.** Stack of strings data type.

```java
public class StackOfStrings {
    StackOfStrings() {
        create an empty stack
    }
    void push(String s) {
        insert a new item onto stack
    }
    String pop() {
        remove and return the item most recently added
    }
    boolean isEmpty() {
        is the stack empty?
    }
    int size() {
        number of items on the stack
    }
}
```

**Warmup client.** Reverse sequence of strings from standard input.
Stack test client

Read strings from standard input.
- If string equals "-", pop string from stack and print.
- Otherwise, push string onto stack.

```java
class StackOfStrings {
    public void push(String item) {
        // Implementation...
    }
    public String pop() {
        // Implementation...
    }
}

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

% more tobe.txt
% java StackOfStrings < tobe.txt
Stack: linked-list representation

Maintain pointer to first node in a linked list; insert/remove from front.

```
StdIn  StdOut

- be

or

not

to

- to

- be

- not

Insert at front of linked list

Remove from front of linked list
```
Stack pop: linked-list implementation

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

save item to return
String item = first.item;

delete first node
first = first.next;

return saved item
return item;
Stack push: linked-list implementation

inner class

private class Node
{
    String item;
    Node next;
}

save a link to the list

Node oldfirst = first;

create a new node for the beginning

first = new Node();

set the instance variables in the new node

first.item = "not";
first.next = oldfirst;
Stack: linked-list implementation in Java

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

    private class Node {
        String item;
        Node next;
    }

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

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

    public String pop() {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```
**Proposition.** Every operation takes constant time in the worst case.

**Proposition.** A stack with \( N \) items uses \( \sim 40 \) \( N \) bytes.

**Remark.** Analysis includes memory for the stack (but not the strings themselves, which the client owns).
Array implementation of a stack.

- Use array $s[]$ to store $N$ items on stack.
- $\text{push}()$: add new item at $s[N]$.
- $\text{pop}()$: remove item from $s[N-1]$.

Defect. Stack overflows when $N$ exceeds capacity. [stay tuned]
public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity)
    {  s = new String[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void push(String item)
    {  s[N++] = item;  }

    public String pop()
    {  return s[--N];  }
}
Stack considerations

Overflow and underflow.
• Underflow: throw exception if pop from an empty stack.
• Overflow: use resizing array for array implementation. [stay tuned]

Loitering. Holding a reference to an object when it is no longer needed.

```java
public String pop()
{
    String item = s[--N];
    s[N] = null;
    return item;
}
```

Null items. We allow null items to be inserted.
stacks
resizing arrays
queues
generics
iterators
applications
Stack: resizing-array implementation

Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.
• `push()`: increase size of array `s[]` by 1.
• `pop()`: decrease size of array `s[]` by 1.

Too expensive.
• Need to copy all item to a new array.
• Inserting first \( N \) items takes time proportional to \( 1 + 2 + \ldots + N \sim N^2 / 2 \).

Challenge. Ensure that array resizing happens infrequently.
Stack: resizing-array implementation

Q. How to grow array?
A. If array is full, create a new array of twice the size, and copy items.

```java
public ResizingArrayStackOfStrings()
{  s = new String[1];  }

public void push(String item)
{
    if (N == s.length) resize(2 * s.length);
    s[N++] = item;
}

private void resize(int capacity)
{
    String[] copy = new String[capacity];
    for (int i = 0; i < N; i++)
        copy[i] = s[i];
    s = copy;
}
```

Consequence. Inserting first $N$ items takes time proportional to $N$ (not $N^2$).
Cost of inserting first $N$ items. \[ N + (2 + 4 + 8 + \ldots + N) \sim 3N. \]
Stack: resizing-array implementation

Q. How to shrink array?

First try.
• `push()`: double size of array `s[]` when array is full.
• `pop()`: halve size of array `s[]` when array is one-half full.

Too expensive in worst case.
• Consider push-pop-push-pop-... sequence when array is full.
• Each operation takes time proportional to $N$.

<table>
<thead>
<tr>
<th>$N$</th>
<th>to</th>
<th>be</th>
<th>or</th>
<th>not</th>
<th>to</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
<td>to</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>4</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
<td>to</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>4</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Q. How to shrink array?

Efficient solution.

• `push()`: double size of array `s[]` when array is full.
• `pop()`: halve size of array `s[]` when array is one-quarter full.

```java
public String pop()
{
    String item = s[--N];
    s[N] = null;
    if (N > 0 && N == s.length/4) resize(s.length/2);
    return item;
}
```

Invariant. Array is between 25% and 100% full.
Stack: resizing-array implementation trace

<table>
<thead>
<tr>
<th>push()</th>
<th>pop()</th>
<th>N</th>
<th>a.length</th>
<th>a[]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>to</td>
<td>0</td>
<td>1</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>be</td>
<td>1</td>
<td>1</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>or</td>
<td>2</td>
<td>2</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>not</td>
<td>3</td>
<td>4</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>to</td>
<td>4</td>
<td>4</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>- to</td>
<td>5</td>
<td>8</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>be</td>
<td>6</td>
<td>8</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>- be</td>
<td>7</td>
<td>8</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>- not</td>
<td>8</td>
<td>8</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>that</td>
<td>9</td>
<td>8</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>- that</td>
<td>10</td>
<td>8</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>- or</td>
<td>11</td>
<td>4</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>- be</td>
<td>12</td>
<td>2</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>is</td>
<td>13</td>
<td>2</td>
<td>null</td>
<td></td>
</tr>
</tbody>
</table>

Trace of array resizing during a sequence of push() and pop() operations.
Stack resizing-array implementation: performance

**Amortized analysis.** Average running time per operation over a worst-case sequence of operations.

**Proposition.** Starting from an empty stack, any sequence of $M$ push and pop operations takes time proportional to $M$.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Best</th>
<th>Worst</th>
<th>Amortized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Construct</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Push</td>
<td>1</td>
<td>$N$</td>
<td>1</td>
</tr>
<tr>
<td>Pop</td>
<td>1</td>
<td>$N$</td>
<td>1</td>
</tr>
<tr>
<td>Size</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Order of growth of running time for resizing stack with $N$ items
Proposition. Uses between \( \sim 8N \) and \( \sim 32N \) bytes to represent a stack with \( N \) items.

- \( \sim 8N \) when full.
- \( \sim 32N \) when one-quarter full.

Remark. Analysis includes memory for the stack (but not the strings themselves, which the client owns).
Tradeoffs. Can implement a stack with either a resizing array or a linked list; client can use interchangeably. Which one is better?

Linked-list implementation.
• Every operation takes constant time in the worst case.
• Uses extra time and space to deal with the links.

Resizing-array implementation.
• Every operation takes constant amortized time.
• Less wasted space.
- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
## Queue API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QueueOfStrings()</td>
<td>create an empty queue</td>
</tr>
<tr>
<td>void enqueue(String s)</td>
<td>insert a new item onto queue</td>
</tr>
<tr>
<td>String dequeue()</td>
<td>remove and return the item least recently added</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the queue empty?</td>
</tr>
<tr>
<td>int size()</td>
<td>number of items on the queue</td>
</tr>
</tbody>
</table>

**Diagram:**
- **enqueue** to add items to the queue.
- **dequeue** to remove and return the last item added.
Queue: linked-list representation

Maintain pointer to first and last nodes in a linked list; insert/remove from opposite ends.
Queue dequeue: linked-list implementation

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

Remark. Identical code to linked-list stack pop().
Queue enqueue: linked-list implementation

inner class

private class Node
{
    String item;
    Node next;
}

save a link to the last node

Node oldlast = last;

create a new node for the end

Node last = new Node();
last.item = "not";
last.next = null;

link the new node to the end of the list

oldlast.next = last;
public class LinkedQueueOfStrings
{
    private Node first, last;

    private class Node
    {  /* same as in StackOfStrings */  }

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

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

    public String dequeue()
    {
        String item = first.item;
        first       = first.next;
        if (isEmpty()) last = null;
        return item;
    }
}
Queue: resizing array implementation

Array implementation of a queue.

- Use array q[] to store items in queue.
- enqueue(): add new item at q[tail].
- dequeue(): remove item from q[head].
- Update head and tail modulo the capacity.
- Add resizing array.

<table>
<thead>
<tr>
<th></th>
<th>null</th>
<th>null</th>
<th>the</th>
<th>best</th>
<th>of</th>
<th>times</th>
<th>null</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<td>head</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>tail</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>capacity</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

0 1 2 3 4 5 6 7 8 9
- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Parameterized stack

We implemented: `StackOfStrings`.
We also want: `StackOfURLs`, `StackOfInts`, `StackOfVans`, ....

**Attempt 1.** 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.

@#$*! most reasonable approach until Java 1.5.
We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 2. Implement a stack with items of type Object.
• Casting is required in client.
• Casting is error-prone: run-time error if types mismatch.

```java
StackOfObjects s = new StackOfObjects();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```
Parameterized stack

We implemented: StackOfStrings.
We also want: StackOfURLs, StackOfInts, StackOfVans, ....

Attempt 3. Java generics.
• Avoid casting in client.
• Discover type mismatch errors at compile-time instead of run-time.

Guiding principles. Welcome compile-time errors; avoid run-time errors.
public class LinkedStackOfStrings
{
    private Node first = null;

    private class Node
    {
        String item;
        Node next;
    }

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

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

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

public class Stack<Item>
{
    private Node first = null;
    private class Node
    {
        Item item;
        Node ...   }
    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
Generic stack: array implementation

public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity)
    {  s = new String[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void push(String item)
    {  s[N++] = item;  }

    public String pop()
    {  return s[--N];  }
}

public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {  s = new Item[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void push(Item item)
    {  s[N++] = item;  }

    public Item pop()
    {  return s[--N];  }
}

the way it should be

public class FixedCapacityStackOfStrings
{
    private String[] s;
    private int N = 0;

    public FixedCapacityStackOfStrings(int capacity)
    {  s = new String[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void push(String item)
    {  s[N++] = item;  }

    public String pop()
    {  return s[--N];  }
}

public class FixedCapacityStack<Item>
{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity)
    {  s = new Item[capacity];  }

    public boolean isEmpty()
    {  return N == 0;  }

    public void push(Item item)
    {  s[N++] = item;  }

    public Item pop()
    {  return s[--N];  }
}

@#$%^! generic array creation not allowed in Java
Generic stack: array implementation

```java
public class FixedCapacityStack<Item>
{
   private Item[] s;
   private int N = 0;

   public FixedCapacityStack(int capacity)
   {  s = (Item[]) new Object[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(Item item)
   {  s[N++] = item;  }

   public Item pop()
   {  return s[--N];  }
}
```

the way it is

```java
public class FixedCapacityStackOfStrings
{
   private String[] s;
   private int N = 0;

   public FixedCapacityStackOfStrings(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   {  return N == 0;  }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {  return s[--N];  }
}
```

the ugly cast
Generic data types: autoboxing

Q. What to do about primitive types?

Wrapper type.
• Each primitive type has a wrapper object type.
• Ex: Integer is wrapper type for int.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Behind-the-scenes casting.

```
Stack<Integer> s = new Stack<Integer>();
s.push(17); // s.push(new Integer(17));
int a = s.pop(); // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for any type of data.
- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.

Java solution. Make stack implement the `Iterable` interface.
Q. What is an **Iterable**?
A. Has a method that returns an **Iterator**.

Q. What is an **Iterator**?
A. Has methods `hasNext()` and `next()`.

Q. Why make data structures **Iterable**?
A. Java supports elegant client code.

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();
}
```

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

```
for (String s : stack)
   StdOut.println(s);
```

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() { return current != null; }
        public void remove() { /* not supported */ }
        public Item next()
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
Stack iterator: array implementation

```java
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator()
    { return new ReverseArrayIterator(); }

    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() { return i > 0; }
        public void remove()     { /* not supported */ }
        public Item next()       { return s[--i]; }
    }
}
```

<table>
<thead>
<tr>
<th>s[]</th>
<th>i</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>it</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>was</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>the</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>best</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>of</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>times</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>null</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Iteration: concurrent modification

Q. What if client modifies the data structure while iterating?
A. A fail-fast iterator throws a `ConcurrentModificationException`.

```
for (String s : stack)
    stack.push(s);
```

To detect:
- Count total number of `push()` and `pop()` operations in `Stack`.
- Save current count in `*Iterator` subclass upon creation.
- Check that two values are still equal when calling `next()` and `hasNext()`.
Main application. Adding items to a collection and iterating (when order doesn't matter).

Implementation. Stack (without pop) or queue (without dequeue).
• stacks
• resizing arrays
• queues
• generics
• iterators

• applications
Java collections library

**List interface.** `java.util.List` is API for ordered collection of items.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>List()</code></td>
<td>create an empty list</td>
</tr>
<tr>
<td><code>boolean isEmpty()</code></td>
<td>is the list empty?</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>number of items</td>
</tr>
<tr>
<td><code>void add(Item item)</code></td>
<td>append item to the end</td>
</tr>
<tr>
<td><code>Item get(int index)</code></td>
<td>return item at given index</td>
</tr>
<tr>
<td><code>Item remove(int index)</code></td>
<td>return and delete item at given index</td>
</tr>
<tr>
<td><code>boolean contains(Item item)</code></td>
<td>does the list contain the given item?</td>
</tr>
<tr>
<td><code>Iterator&lt;Item&gt; iterator()</code></td>
<td>iterator over all items in the list</td>
</tr>
</tbody>
</table>

**Implementations.** `java.util.ArrayList` uses resizing array; `java.util.LinkedList` uses linked list.
Java collections library

**java.util.Stack.**
- Supports `push()`, `pop()`, `size()`, `isEmpty()`, and iteration.
- Also implements `java.util.List` interface from previous slide, including, `get()`, `remove()`, and `contains()`.
- Bloated and poorly-designed API (why?) ⇒ don't use.

**java.util.Queue.** An interface, not an implementation of a queue.

**Best practices.** Use our implementations of `Stack`, `Queue`, and `Bag`. 
War story (from COS 226)

Generate random open sites in an $N$-by-$N$ percolation system.

- Jenny: pick $(i, j)$ at random; if already open, repeat.
  Takes $\sim c_1 N^2$ seconds.
- Kenny: create a `java.util.LinkedList` of $N^2$ closed sites.
  Pick an index at random and delete.
  Takes $\sim c_2 N^4$ seconds.

Lesson. Don't use a library until you understand its API!
This course. Can't use a library until we've implemented it in class.
Stack 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.
- ...

Java
Adobe PostScript

Compilers: Principles, Techniques, and Tools
Function calls

How a compiler implements a function.

- 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.

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

p = 216, q = 192

p = 192, q = 24

p = 24, q = 0
Goal. Evaluate infix expressions.

Two-stack algorithm. [E. W. Dijkstra]
• Value: push onto the value stack.
• Operator: push onto the operator stack.
• Left parenthesis: ignore.
• Right parenthesis: 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 demo
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());
    }
}
Correctness

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

\[
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
\]

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.
Stack-based programming languages

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

\[
(1 \ (\ (2\ 3\ +)\ (4\ 5\ \times)\ \times)\ +)
\]

Observation 2. All of the parentheses are redundant!

\[
1\ 2\ 3\ +\ 4\ 5\ \times\ \times\ +
\]

Bottom line. Postfix or "reverse Polish" notation.
Applications. Postscript, Forth, calculators, Java virtual machine, …
**PostScript**

**PostScript.** [Warnock-Geschke 1980s]

- Postfix program code.
- Turtle graphics commands.
- Variables, types, text, loops, conditionals, functions, ...

Simple virtual machine, but not a toy.

- Easy to specify published page.
- Easy to implement in printers.
- Revolutionized the publishing world.
PostScript applications

**Algorithms, 3rd edition.** Figures created directly in PostScript.

```postscript
%! 72 72 translate
/kochR
{ 2 copy ge { dup 0 rlineto }
{ 3 div
 2 copy kochR 60 rotate
 2 copy kochR -120 rotate
 2 copy kochR 60 rotate
 2 copy kochR
} ifelse
pop pop
} def
0 0 moveto 81 243 kochR
0 81 moveto 27 243 kochR
0 162 moveto 9 243 kochR
0 243 moveto 1 243 kochR
stroke
```

**Algorithms, 4th edition.** Figures created using enhanced version of `StdDraw` that saves to PostScript for vector graphics.
Queue applications

Familiar 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.
- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.
**M/M/1 queuing model**

**M/M/1 queue.**

- Customers arrive according to **Poisson process** at rate of $\lambda$ per minute.
- Customers are serviced with rate of $\mu$ per minute.

interarrival time has exponential distribution $\Pr[X \leq x] = 1 - e^{-\lambda x}$

service time has exponential distribution $\Pr[X \leq x] = 1 - e^{-\mu x}$

---

**Q.** What is average wait time $W$ of a customer in system?

**Q.** What is average number of customers $L$ in system?
$M/M/1$ queuing model: example simulation

<table>
<thead>
<tr>
<th></th>
<th>arrival</th>
<th>departure</th>
<th>wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>15</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>17</td>
<td>23</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>19</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td>30</td>
<td>9</td>
</tr>
</tbody>
</table>
**M/M/1 queuing model: event-based simulation**

```java
public class MM1Queue {
    public static void main(String[] args) {
        double lambda = Double.parseDouble(args[0]);  // arrival rate
        double mu = Double.parseDouble(args[1]);  // service rate
        double nextArrival = StdRandom.exp(lambda);
        double nextService = nextArrival + StdRandom.exp(mu);

        Queue<Double> queue = new Queue<Double>();  // queue of arrival times
        Histogram hist = new Histogram("M/M/1 Queue", 60);

        while (true) {
            while (nextArrival < nextService) {
                queue.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }

            double arrival = queue.dequeue();
            double wait = nextService - arrival;
            hist.addDataPoint(Math.min(60, (int) (Math.round(wait))));
            if (queue.isEmpty()) nextService = nextArrival + StdRandom.exp(mu);
            else nextService = nextService + StdRandom.exp(mu);
        }
    }
}
```
**M/M/1 queuing model: experiments**

**Observation.** If service rate $\mu$ is much larger than arrival rate $\lambda$, customers get good service.

```plaintext
% java MM1Queue .2 .333
```
Observation. As service rate $\mu$ approaches arrival rate $\lambda$, services goes to h***.
Observation. As service rate $\mu$ approaches arrival rate $\lambda$, services goes to $h^{**}$.

% java MM1Queue .2 .21
**M/M/1 queuing model: analysis**

**M/M/1 queue.** Exact formulas known.

Wait time $W$ and queue length $L$ approach infinity as service rate approaches arrival rate

Little’s Law

$$W = \frac{1}{\mu - \lambda}, \quad L = \lambda W$$

More complicated queueing models. Event-based simulation essential!

Queueing theory. See ORF 309.