1.3 Stacks and Queues

- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications
**Stacks and queues**

**Fundamental data types.**
- Values: sets of objects.
- Operations: *insert*, *remove*, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

**Stack.** Remove the item most recently added.
**Analogy.** Cafeteria trays, Web surfing.

**Queue.** Remove the item least recently added.
**Analogy.** Registrar's line.
Client, implementation, interface

Separate interface and implementation.
Ex: stack, queue, 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
- dynamic resizing
- queues
- generics
- iterators
- applications
Stack API

Warmup. Stack of strings objects.

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

Challenge. Reverse sequence of strings from standard input.
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);
    }
}
Stack pop: linked-list implementation

```java
String item = first.item;
first = first.next;
return item;
```
Stack push: linked-list implementation

1. **Save a link to the list**
   ```
   Node oldfirst = first;
   ```

2. **Create a new node for the beginning**
   ```
   first = new Node();
   ```

3. **Set the instance variables in the new node**
   ```
   first.item = "not";
   first.next = oldfirst;
   ```
public class StackOfStrings
{
    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()
    {
        if (isEmpty()) throw new RuntimeException();
        String item = first.item;
        first = first.next;
        return item;
    }
}
Stack: linked-list implementation performance

**Proposition.** Using a linked-list implementation of a stack, every operation takes constant time in the worst case.

**Proposition.** Uses $\sim 40 \, N$ bytes to represent a stack with $N$ items.

- assume **64-bit machine** (8 bytes per reference)
- extra overhead for inner class

```java
class Node {
    String item;
    Node next;
}
```

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

**Lesson.** “Swollen” pointers can use up memory on 64-bit machines!
Stack: array implementation

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]
Stack: array implementation

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

    public StackOfStrings(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];
    }
}
```

This version avoids "loitering":
garbage collector reclaims memory only if no outstanding references

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

decrement N;
then use to index into array
- stacks
- **dynamic resizing**
- queues
- generics
- iterators
- applications
Problem. Requiring client to provide capacity does not implement API!

Q. How to grow and shrink array?

First try.
- \texttt{push}(): increase size of \texttt{s[]} by 1.
- \texttt{pop}(): decrease size of \texttt{s[]} by 1.

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

Challenge. Ensure that array resizing happens infrequently.
Stack: dynamic-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 StackOfStrings() {  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 $N^2$).
Stack: amortized cost of adding to a stack

Cost of inserting first $N$ items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N$.

1 array accesses per push
128

k array accesses to double to size $k$

one gray dot for each operation

red dots give cumulative average

0
0
128
128

number of push() operations
Q. How to shrink array?

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

Too expensive.
• Consider push-pop-push-pop-... sequence when array is full.
• Takes time proportional to $N$ per operation in worst case.
Stack: dynamic-array implementation

Q. How to shrink array?

Efficient solution.

- **push()**: double size of s[] when array is full.
- **pop()**: halve size of 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: dynamic-array implementation trace

<table>
<thead>
<tr>
<th>StdIn</th>
<th>StdOut</th>
<th>N</th>
<th>a.length</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>0</td>
<td>1</td>
<td>null</td>
</tr>
<tr>
<td>to</td>
<td>1</td>
<td>1</td>
<td></td>
<td>to</td>
</tr>
<tr>
<td>be</td>
<td>2</td>
<td>2</td>
<td></td>
<td>be</td>
</tr>
<tr>
<td>or</td>
<td>3</td>
<td>4</td>
<td></td>
<td>to</td>
</tr>
<tr>
<td>not</td>
<td>4</td>
<td>4</td>
<td></td>
<td>be</td>
</tr>
<tr>
<td>to</td>
<td>5</td>
<td>8</td>
<td></td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>to</td>
<td>4</td>
<td>8</td>
<td>be</td>
</tr>
<tr>
<td>be</td>
<td>5</td>
<td>8</td>
<td></td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>be</td>
<td>4</td>
<td>8</td>
<td>be</td>
</tr>
<tr>
<td>-</td>
<td>not</td>
<td>3</td>
<td>8</td>
<td>be</td>
</tr>
<tr>
<td>that</td>
<td>4</td>
<td>8</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>-</td>
<td>that</td>
<td>3</td>
<td>8</td>
<td>null</td>
</tr>
<tr>
<td>-</td>
<td>or</td>
<td>2</td>
<td>4</td>
<td>null</td>
</tr>
<tr>
<td>-</td>
<td>be</td>
<td>1</td>
<td>2</td>
<td>null</td>
</tr>
<tr>
<td>is</td>
<td>2</td>
<td>2</td>
<td></td>
<td>is</td>
</tr>
</tbody>
</table>

Amortized analysis. This doubling and halving strategy is a judicious tradeoff between wasting space (by setting the size of the array to be too big and leaving empty slots) and wasting time (by reorganizing the array after each insertion). The specific strategy in `DoublingStackOfStrings` guarantees that the stack never overflows and never becomes less than one-quarter full (unless the stack is empty, in which case the array size is 1). If you are mathematically inclined, you might enjoy proving this fact with mathematical induction (see Exercise 4.3.20). More important, we can prove that the cost of doubling and halving is always absorbed (to within a constant factor) in the cost of other stack operations. Again, we leave the details to an exercise for the mathematically inclined, but the idea is simple: when the stack size is less than one-fourth the array size. Then, after the array is halved, it will be about half full and can accommodate a substantial number of `push()` and `pop()` operations before having to change the size of the array again. This characteristic is important: for example, if we were to use to policy of halving the array when the stack size is one-half the array size, then the resulting array would be full, which would mean it would be doubled for a `push()`, leading to the possibility of an expensive cycle of doubling and halving.
Amortized analysis. Average running time per operation over a worst-case sequence of operations.

Proposition. Starting from empty stack (with dynamic resizing), any sequence of $M$ push and pop operations takes time proportional to $M$. 

<table>
<thead>
<tr>
<th></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>

running time for doubling stack with $N$ items
Stack dynamic array implementation: memory usage

**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).
Stack implementations: dynamic array vs. linked list

Tradeoffs. Can implement a stack with either dynamic array or 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.

Dynamic-array implementation.
- Every operation takes constant amortized time.
- Less wasted space.
- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications
Queue API

public class QueueOfStrings

QueueOfStrings() create an empty queue

void enqueue(String s) insert a new item onto queue

String dequeue() remove and return the item least recently added

boolean isEmpty() is the queue empty?

int size() number of items on the queue
Queue test client

```java
public static void main(String[] args)
{
    QueueOfStrings q = new QueueOfStrings();
    while (!StdIn.isEmpty())
    {
        String item = StdIn.readString();
        if (item.equals("-")) StdOut.print(q.dequeue());
        else                  q.enqueue(item);
    }
}
```

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

% java QueueOfStrings < tobe.txt
to be or not to be
```
Queue dequeue: linked-list implementation

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

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 QueueOfStrings {
    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;
    }
}

special cases for empty queue
Queue: linked-list trace

```
StdIn  StdOut
  to   to
  be   be
  or   be
  not  not
  to   to
  -    to
  be   be
  -    be
  -    or
  that that
  -    not
  to    be
  -    be
  is    is
```
Queue: dynamic 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 dynamic resizing.
Parameterized stack

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

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.
Parameterized stack

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

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);
s.push(b);
    a = (Apple) (s.pop());
```

run-time error
Parameterized stack

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

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 next;
    }

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

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

    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
Generic stack: array implementation

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

    public StackOfStrings(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];
    }
}
```

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

    public Stack(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

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

    public StackOfStrings(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];
    }
}
```

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

    public Stack(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

definition

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

   public ArrayStack()
   {  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];  }
}

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

   public StackOfStrings(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];  }
}
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
- dynamic resizing
- 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.

**Iterable interface**

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

**Iterator interface**

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

**"foreach" statement**

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

**equivalent code**

```java
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next();
    StdOut.println(s);
}
```
Stack iterator: linked-list implementation

```java
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;
        }
    }
}
```
import java.util.Iterator;

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

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

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

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

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

```java
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()`.
- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications
Summary

Three ADTs for processing collections of objects:
• Stack
• Queue
• Bag

Generic implementations
• reusable code (don’t need new implementation for new type)
• compile-time type checks

Iteration
• stack: LIFO order
• queue: FIFO order
• bag: arbitrary order

easy implementation:
stack without pop() or queue without get()
**Bag**

**API**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>public class Bag&lt;Item&gt;</strong> implements Iterable&lt;Item&gt;</td>
<td></td>
</tr>
<tr>
<td>Bag()</td>
<td>create an empty bag</td>
</tr>
<tr>
<td>void add(Item x)</td>
<td>add an item</td>
</tr>
<tr>
<td>int size()</td>
<td>number of items in bag</td>
</tr>
<tr>
<td>Iterable&lt;Item&gt; iterator()</td>
<td>iterator for all items in bag</td>
</tr>
</tbody>
</table>

**Typical Client**

(average the numbers on StdIn)

```java
public static void main(String[] args) {
    Bag<Double> numbers = new Bag<Double>();
    while (!StdIn.isEmpty())
        numbers.add(StdIn.readDouble());
    int N = numbers.size();
    double sum = 0.0;
    for (Double s : numbers) sum += s;
    double avg = sum/N;
    StdOut.println("Average: " + avg);
}
```

**Sweet spot:** Save for iteration where order doesn't matter.
### Queue API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Queue()</code></td>
<td>create an empty queue</td>
</tr>
<tr>
<td><code>void enqueue(Item x)</code></td>
<td>add an item</td>
</tr>
<tr>
<td><code>Item dequeue()</code></td>
<td>remove the least recently added item</td>
</tr>
<tr>
<td><code>int size()</code></td>
<td>number of items in queue</td>
</tr>
<tr>
<td><code>Iterable&lt;Item&gt; iterator()</code></td>
<td>iterator for all items in queue</td>
</tr>
</tbody>
</table>

#### Typical Client

```java
public static int[] readInts(String name)
{
    In in = new In(name);
    Queue<Integer> q = new Queue<Integer>();
    while (!in.isEmpty())
        q.enqueue(in.readInt());
    int N = q.size();
    int[] a = new int[N];
    for (int i = 0; i < N; i++)
        a[i] = q.dequeue();
    return a;
}
```

**Key Point:** don't need to know file size

**Sweet spot:** Save for later use where order **does** matter.
Stack

API

```java
public class Stack<Item> implements Iterable<Item>

Stack() create an empty stack
void push(Item x) add an item
Item pop() remove the most recently added item
int size() number of items in queue
Iterable<Item> iterator() iterator for all items in queue

sample client
(print the strings on StdIn in reverse order)

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

Sweet spot: Support recursive computation (stay tuned).
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>List()</td>
<td>create an empty list</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the list empty?</td>
</tr>
<tr>
<td>int size()</td>
<td>number of items</td>
</tr>
<tr>
<td>void add(Item item)</td>
<td>append item to the end</td>
</tr>
<tr>
<td>Item get(int index)</td>
<td>return item at given index</td>
</tr>
<tr>
<td>Item remove(int index)</td>
<td>return and delete item at given index</td>
</tr>
<tr>
<td>boolean contains(Item item)</td>
<td>does the list contain the given item?</td>
</tr>
<tr>
<td>Iterator&lt;Item&gt; iterator()</td>
<td>iterator over all items in the list</td>
</tr>
</tbody>
</table>

Implementations. `java.util.ArrayList` uses dynamic 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 ⇒ don't use.

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

Best practices. Use our implementations of Stack, Queue, and Bag.
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.
- ...
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.

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

Goal. Evaluate infix expressions.

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

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!

55
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

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.
Observation 1. 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!

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

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.

```
%! 100 100 moveto
100 300 lineto
300 300 lineto
300 100 lineto
stroke
```

units are points (72 per inch)

define a path

draw the path

its output
PostScript

Page description language.
• Explicit stack.
• Full computational model
• Graphics engine.

Basics.
• %!: “I am a PostScript program."
• Literal: “push me on the stack."
• Function calls take arguments from stack.
• Turtle graphics built in.

a PostScript program

```postscript
%! 72 72 moveto 0 72 rlineto 72 0 rlineto 0 -72 rlineto -72 0 rlineto 2 setlinewidth stroke
```

its output
PostScript

Data types.
- Basic: integer, floating point, boolean, ...
- Graphics: font, path, curve, ....
- Full set of built-in operators.

Text and strings.
- Full font support.
- \texttt{show} (display a string, using current font).
- \texttt{cvs} (convert anything to a string).

\begin{verbatim}
%! /Helvetica-Bold findfont 16 scalefont setfont
72 168 moveto
(Square root of 2:) show
72 144 moveto
2 sqrt 10 string cvs show
\end{verbatim}

Square root of 2:
1.41421
Variables (and functions).

- Identifiers start with `/`.
- `def` operator associates id with value.
- Braces.
- `args on stack`.

PostScript

```postscript
%! /box
{
/sz exch def
0 sz rlineto
sz 0 rlineto
0 sz neg rlineto
sz neg 0 rlineto
} def
72 144 moveto
72 box
288 288 moveto
144 box
2 setlinewidth
stroke
```
PostScript

For loop.
• “from, increment, to” on stack.
• Loop body in braces.
• for operator.

If-else conditional.
• Boolean on stack.
• Alternatives in braces.
• if operator.

... (hundreds of operators)
PostScript applications


%!
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 queue.**

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

**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

$M/M/1$ queue.

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

The service time has exponential distribution $\Pr[X \leq x] = 1 - e^{-\mu x}$
M/M/1 queuing model: example simulation
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>();
        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);
        }
    }
}
```
Observation. If service rate $\mu$ is much larger than arrival rate $\lambda$, customers get good service.
M/M/1 queuing model: experiments

Observation. As service rate $\mu$ approaches arrival rate $\lambda$, services goes to h***.

```
% java MM1Queue .2 .25
```
Observation. As service rate $\mu$ approaches arrival rate $\lambda$, services goes to h***.

% java-MM1Queue .2 .21
M/M/1 queueing 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.