1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Stacks and queues

Fundamental data types.
- Value: collection of objects.
- Operations: add, remove, iterate, test if empty.
- Intent is clear when we add.
- 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"
Client, implementation, API

Separate client and implementation via API.

**API:** description of data type, basic operations.

**Client:** program using operations defined in API.

**Implementation:** actual code implementing operations.

**Benefits.**

- **Design:** creates modular, reusable libraries.
- **Performance:** substitute optimized implementation when it matters.

**Ex.** Stack, queue, bag, priority queue, symbol table, union-find, ....
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Stack API

Warmup API. Stack of strings data type.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class StackOfStrings</td>
<td></td>
</tr>
<tr>
<td>StackOfStrings()</td>
<td>create an empty stack</td>
</tr>
<tr>
<td>void push(String item)</td>
<td>add a new string to stack</td>
</tr>
<tr>
<td>String pop()</td>
<td>remove and return the string most recently added</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>is the stack empty?</td>
</tr>
<tr>
<td>int size()</td>
<td>number of strings on the stack</td>
</tr>
</tbody>
</table>

Warmup client. Reverse sequence of strings from standard input.
Sample client

**Warmup client.** Reverse sequence of strings from standard input.
- Read string and push onto stack.
- Pop string and print.

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

% more tinyTale.txt
it was the best of times ...

% java ReverseStrings < tinyTale.txt
... times of best the was it
[ignoring newlines]
How to implement a stack with a singly-linked list?

A. least recently added

```
        ↓
  it → was → the → best → of → null
```

B. most recently added

```
        ↓
of → best → the → was → it → null
```

C. None of the above.

D. I don't know.
Stack: linked-list implementation

- Maintain pointer `first` to first node in a singly-linked list.
- Push new item before `first`.
- Pop item from `first`.

```
most recently added

of  ->  best  ->  the  ->  was  ->  it  ->  null

first
```
Stack pop: linked-list implementation

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

inner class Decode {
    String item;
    Decode next;
}
```

**Save item to return**

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

**Delete first node**

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

**Return saved item**

```java
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;
oldfirst
first

create a new node for the beginning
first = new Node();
first
oldfirst

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 {
        private String item;
        private 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;
    }
}
```

private inner class (access modifiers for instance variables don't matter)
Stack: linked-list implementation performance

**Proposition.** Every operation takes constant time in the worst case.

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

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

16 bytes (object overhead)

8 bytes (inner class extra overhead)

8 bytes (reference to item String)

8 bytes (reference to next Node)

40 bytes allocated per stack node

**Remark.** This accounts for the memory for the stack (but not memory for the strings themselves, which the client owns).
How to implement a fixed-capacity stack with an array?

A. least recently added

<table>
<thead>
<tr>
<th>it</th>
<th>was</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>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>
</tbody>
</table>

B. most recently added

<table>
<thead>
<tr>
<th>times</th>
<th>of</th>
<th>best</th>
<th>the</th>
<th>was</th>
<th>it</th>
<th>null</th>
<th>null</th>
<th>null</th>
<th>null</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tbody>
</table>

C. *None of the above.*

D. *I don't know.*
**Fixed-capacity stack: array implementation**

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

**least recently added**

```
  s[]
  +---+---+---+---+---+---+---+---+---+---+
  | it | was | the | best | of | times | null | null | null | null |
  +---+---+---+---+---+---+---+---+---+---+
  | 0  | 1   | 2   | 3    | 4  | 5    | 6    | 7    | 8    | 9    |
```

$N$ \hspace{2cm} capacity = 10

**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]; }
}

Fixed-capacity stack: array implementation
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()
{ return s[--N]; }
```

Loitering

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

this version avoids "loitering":
garbage collector can reclaim memory for an object only if no remaining references
1.3 Bags, Queues, and Stacks

- 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 items to a new array, for each operation.
- Array accesses to add first `N` items $= N + (2 + 4 + \ldots + 2(N - 1)) \sim N^2$.

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;
}
```

Array accesses to add first $N = 2^i$ items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N$. 

- 1 array access per push
- k array accesses to double to size k (ignoring cost to create new array)
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>full</th>
<th>to</th>
<th>be</th>
<th>or</th>
<th>not</th>
</tr>
</thead>
<tbody>
<tr>
<td>push(&quot;to&quot;)</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
<tr>
<td>pop()</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
<tr>
<td>push(&quot;be&quot;)</td>
<td>to</td>
<td>be</td>
<td>or</td>
<td>not</td>
</tr>
</tbody>
</table>
Stack: resizing-array implementation

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: performance

Amortized analysis. Starting from an empty data structure, 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></th>
<th>typical</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 array stack with $N$ items
Stack resizing-array implementation: memory usage

**Proposition.** A ResizingArrayStackOfStrings uses $\sim 8N$ to $\sim 32N$ bytes of memory for a stack with $N$ items.

- $\sim 8N$ when full.
- $\sim 32N$ when one-quarter full.

```java
public class ResizingArrayStackOfStrings {
    private String[] s; // 8 bytes x array size
    private int N = 0;
    ...
}
```

**Remark.** This accounts for the memory for the stack (but not the memory for strings themselves, which the client owns).
Tradeoffs. Can implement a stack with either resizing 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.

Resizing-array implementation.
- Every operation takes constant amortized time.
- Less wasted space.

\[
N = 4 \quad \begin{array}{cccccccc}
  & to & be & or & not & null & null & null & null \\
  first \to & not & or & be & to & null
\end{array}
\]
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
# Queue API

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>public class QueueOfStrings()</td>
<td></td>
</tr>
<tr>
<td>QueueOfStrings()</td>
<td>create an empty queue</td>
</tr>
<tr>
<td>void enqueue(String item)</td>
<td>add a new string to queue</td>
</tr>
<tr>
<td>String dequeue()</td>
<td>remove and return the string 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 strings on the queue</td>
</tr>
</tbody>
</table>

![Queue illustration](image)
How to implement a queue with a singly-linked linked list?

A. most recently added

```
| times | of | best | the | was | it | null |
```

B. least recently added

```
| it | was | the | best | of | times | null |
```

C. None of the above.

D. I don't know.
Queue: linked-list implementation

- Maintain one pointer `first` to first node in a singly-linked list.
- Maintain another pointer `last` to last node.
- Dequeue from `first`.
- Enqueue after `last`.

![Queue Diagram]
Queue dequeue: 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;

Remark. Identical code to linked-list stack pop().
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

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

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 LinkedStackOfStrings */ }

    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;
    }
}
How to implement a fixed-capacity queue with an array?

A. least recently added

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>it</td>
<td>was</td>
<td>the</td>
<td>best</td>
<td>of</td>
<td>times</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

B. most recently added

```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>times</td>
<td>of</td>
<td>best</td>
<td>the</td>
<td>was</td>
<td>it</td>
<td>null</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
```

C. None of the above.

D. I don't know.
Queue: resizing-array implementation

- 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**: “circular buffer”
- Add resizing array.

```
least recently added   most recently added

<table>
<thead>
<tr>
<th>q[]</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></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>
</tbody>
</table>

head                              tail                              capacity = 10
```

**Q.** How to resize?
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Parameterized stack

We implemented: StackOfStrings.

We also want: StackOfURLs, StackOfInts, StackOfApples, StackOfOranges, ... 

Solution in Java: generics.

Stack<Apple> stack = new Stack<Apple>();
Apple apple = new Apple();
Orange orange = new Orange();
stack.push(apple);
stack.push(orange);  \[\text{compile-time error}\]...
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

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 way it should be

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 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]; }
}
```

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

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

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

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

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

the way it is

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

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

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

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

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

the ugly cast
Unchecked cast

% javac FixedCapacityStack.java
Note: FixedCapacityStack.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

% javac -Xlint:unchecked FixedCapacityStack.java
FixedCapacityStack.java:26: warning: [unchecked] unchecked cast
found    : java.lang.Object[]
required: Item[]
    a = (Item[]) new Object[capacity];
    ^
1 warning

Q. Why does Java make me cast (or use reflection)?
Short answer. Backward compatibility.
Long answer. Need to learn about type erasure and covariant arrays.
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.

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

Bottom line. Client code can use generic stack for any type of data.
Which of the following is the correct way to declare and initialize an empty stack of characters?

A. `Stack<Character> stack = new Stack<>();`

B. `Stack stack = new Stack<Character>();`

C. `Stack<Character> stack = new Stack<Character>();`

D. `Stack<char> stack = new Stack<char>();`

E. None of the above.
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.

resizing-array representation

<table>
<thead>
<tr>
<th>i</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>it</td>
<td>was</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>

linked-list representation

- first
  - times
  - of
  - best
  - the
  - was
  - it
  - null
- current

Java solution. Use a “foreach” loop.
Foreach loop

Java provides elegant syntax for iteration over collections.

To make user-defined collection support foreach loop:
- Data type must have a method named `iterator()`.
- The `iterator()` method returns an object that has two core methods:
  - the `hasNext()` methods returns `false` when there are no more items.
  - the `next()` method returns the next item in the collection.
Iterators

To support foreach loops, Java provides two interfaces.

- **Iterator interface**: `next()` and `hasNext()` methods.
- **Iterable interface**: `iterator()` method that returns an `Iterator`.
- Both should be used with generics.

```java
class java.util.Iterator {
    boolean hasNext();
    Item next();
    void remove(); ← optional; use at your own risk
}
```

```java
class java.lang.Iterable {
    public Iterator iterator();
}
```

Type safety.

- Data type must use these interfaces to support foreach loop.
- Client program won't compile if implementation doesn't.
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]; }
    }
}
```

```
s[]
<table>
<thead>
<tr>
<th>i</th>
<th>it</th>
<th>was</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>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
```
Main application. Adding items to a collection and iterating (when order doesn't matter).

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

Implementation. Stack (without pop) or queue (without dequeue).
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
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 logo](image1.png)
![Adobe PostScript 3 logo](image2.png)
![Undo button in a Web browser](image3.png)
![Compilers book cover](image4.png)
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);
}
```

- p = 216, q = 192
- p = 192, q = 24
- p = 24, q = 0
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 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!
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("")) /* noop */ ;
            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
Stack-based programming languages

Observation 1. Dijkstra's two-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, ...
TECHNICAL INTERVIEW QUESTIONS
**Queue With Two Stacks**

**Problem.** Implement a queue with two stacks so that:

- Each queue op uses a constant *amortized* number of stack ops.
- At most constant extra memory (besides two stacks).

**Applications.**

- Job interview.
- Implement an *immutable* or *persistent* queue.
- Implement a queue in a (purely) *functional programming language*.

![Haskell](image1.png) ![Lisp](image2.png) ![OCaml](image3.png) ![Standard ML](image4.png)