1.3 Stacks and Queues

- stacks
- resizing arrays
- queues
- generics
- 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.  
**Queue.** Examine the item least recently added.

LIFO = "last in first out"

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
- applications
**Stack API**

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<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>

**Potential client use case.** Reverse sequence of strings from standard input.
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
```

```
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;

oldfirst
first

create a new node for the beginning

first = new Node();

oldfirst
first

set the instance variables in the new node

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

```java
public class LinkedListStackOfStrings {
    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;
    }
}
```
**Stack: linked-list implementation performance**

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

**Proposition.** A stack with $N$ items uses $\sim 40 \cdot N$ 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 String)

8 bytes (reference to 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).
Fixed-capacity stack: array implementation

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

```
least recently added

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</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>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
```

N

Defect. Stack overflows when $N$ exceeds capacity. [stay tuned]
Fixed-capacity 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];
    }
}
```

- use to index into array; then increment N
- decrement N; then use to index into array

(a cheat (stay tuned))
Stack considerations

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

Null items. We allow null items to be added.
Duplicate items. We allow an item to be added more than once.
Loitering. Holding a reference to an object when it is no longer needed.

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

```
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
- 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 = \( N + (2 + 4 + \ldots + 2(N-1)) \approx N^2. \)

**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 \textit{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. $N + (2 + 4 + 8 + \ldots + N) \sim 3N$. 

"repeated doubling"
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>

**doubling and halving operations**

**order of growth of running time for resizing array stack with $N$ items**
Stack: amortized cost of adding to a stack

Cost of adding first $N$ 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: 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).
Stack implementations: resizing array vs. linked list

**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.
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- applications
public class QueueOfStrings

<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 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>

**enqueue**

**dequeue**
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`.

![Diagram showing a queue implemented using a linked list with pointers `first` and `last`. The nodes are connected in a linear fashion with arrows indicating the direction of enqueue and dequeue operations.](image-url)
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().
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

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;
    }
}
Queue: resizing-array implementation

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

<table>
<thead>
<tr>
<th>( q[] )</th>
<th>least recently added</th>
<th>most recently added</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>null</td>
<td>the</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>best</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>of</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>times</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>6</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>7</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>8</td>
</tr>
<tr>
<td>null</td>
<td></td>
<td>null</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>null</td>
<td>head</td>
<td>tail</td>
</tr>
</tbody>
</table>

head | tail | capacity = 10
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);  // 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 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]; }
}

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 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 ugly cast
Unchecked cast

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.

```
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.
```
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>s[]</td>
<td>it</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

linked-list representation

```

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:
  - `hasNext()` methods returns `false` when there are no more items
  - `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.util.Iterator interface

public interface Iterator<
Item>
{
    boolean hasNext();
    Item next();
    void remove(); ← optional; use at your own risk
}
```

```
java.lang.Iterable interface

public interface Iterable<
Item>
{
    Iterator<Item> 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;
        }
    }
}
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]; }
    }
}
Bag API

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>public class Bag&lt;Item&gt; 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 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
- applications
Java collections library

List interface. java.util.List is API for a sequence of items.

```java
public interface List<Item> extends Iterable<Item>
{
    List() // create an empty list
    boolean isEmpty() // is the list empty?
    int size() // number of items
    void add(Item item) // add item to the end
    Item get(int index) // return item at given index
    Item remove(int index) // return and delete item at given index
    boolean contains(Item item) // does the list contain the given item?
    Iterator<Item> iterator() // iterator over all items in the list
    ;
}
```

Implementations. java.util.ArrayList uses a resizing array;
java.util.LinkedList uses doubly-linked list. Caveat: only some operations are efficient.
Java collections library

`java.util.Stack`

- Supports `push()`, `pop()`, and iteration.
- Inherits from `java.util.Vector`, which implements `java.util.List` interface.

Java 1.3 bug report (June 27, 2001)

The iterator method on `java.util.Stack` iterates through a Stack from the bottom up. One would think that it should iterate as if it were popping off the top of the Stack.

status (closed, will not fix)

It was an incorrect design decision to have Stack extend Vector ("is-a" rather than "has-a"). We sympathize with the submitter but cannot fix this because of compatibility.
Java collections library

\texttt{java.util.Stack}.

- Supports \texttt{push()}, \texttt{pop()}, and iteration.
- Inherits from \texttt{java.util.Vector}, which implements \texttt{java.util.List} interface.

\texttt{java.util.Queue}. An interface, not an implementation of a queue.

\textbf{We suggest.} Implement your own Stack and Queue following the textbook to be sure you understand what’s going on.
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.
Dijkstra's two-stack algorithm demo

infix expression (fully parenthesized)

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

Operator stack: [ ]

Value stack: [ ]
Dijkstra's two-stack algorithm

**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.

infix expression (fully parenthesized)

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>1</td>
<td>+</td>
<td>(</td>
<td>(</td>
<td>2</td>
<td>+</td>
<td>3</td>
</tr>
</tbody>
</table>
Dijkstra's two-stack algorithm

**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.
Dijkstra's two-stack algorithm

**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.
Dijkstra's two-stack algorithm

**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.
Dijkstra's two-stack algorithm

**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.

![Stack Diagram](image-url)
Dijkstra's two-stack algorithm

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.

\[
\begin{array}{c}
\text{value stack} \\
1 \\
\end{array} \quad \begin{array}{c}
\text{operator stack} \\
+ \\
\end{array}
\]

\[
(1 + (2 + 3) * (4 * 5))
\]
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

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.

\[
\begin{array}{c}
\text{1} \\
\text{+}
\end{array}
\]

value stack operator stack

\[
\begin{array}{c}
( \ 1 \ + \ ( \ 2 \ + \ 3 \ ) \ ) \ast \ ( \ 4 \ast \ 5 \ ) \ ) \ )
\end{array}
\]
Dijkstra's two-stack algorithm

**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.

```
(   1   +   (   (   2   +   3   )   *   (   4   *   5   )   )   )   )
```
Dijkstra's two-stack algorithm

**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.
Dijkstra's two-stack algorithm

**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.

---

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

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.
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.
Dijkstra's two-stack algorithm

**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.

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

value stack operator stack
Dijkstra's two-stack algorithm

**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.

---

(1 + (2 + 3) * (4 * 5))
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

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.

```
(value stack)  (operator stack)

( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

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.
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.

<table>
<thead>
<tr>
<th>Value stack</th>
<th>Operator stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
</tr>
<tr>
<td>5</td>
<td>*</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
</tr>
</tbody>
</table>

( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.

---

(value stack)  

- 4
- 5
- 1

(operator stack)  

- *
- *
- +
Dijkstra's two-stack algorithm

**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.

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

![Value stack and operator stack](image)
Dijkstra's two-stack algorithm

**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.
Dijkstra's two-stack algorithm

**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.

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

value stack | operator stack
---|---
5 | *
4 | *
5 | +
1 |
Dijkstra's two-stack algorithm

**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.

![Value stack and operator stack diagram]

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

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.

\[
\begin{array}{c}
5 \\
+ \\
1
\end{array}
\quad
\begin{array}{c}
* \\
+
\end{array}
\quad
\begin{array}{c}
5 \\
* \\
4 \\
=
\end{array}
\quad
20
\]

\[
(1 + (2 + 3) \times (4 \times 5))
\]
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.

```
  ( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.

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

value stack operator stack
Dijkstra's two-stack algorithm

**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.

\[
20 * 5 = 100
\]

(value stack) \hspace{1cm} (operator stack)

\[
\begin{array}{cccccc}
1 & + & \frac{1}{2} & + & 3 & \frac{1}{*} & \frac{4}{*} & 5 & \frac{1}{)} & \frac{)}
\end{array}
\]
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Dijkstra's two-stack algorithm

**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.

```
(1 + ( (2 + 3) * (4 * 5) ) )
```
Dijkstra's two-stack algorithm

**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.

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

\[ 100 + 1 = 101 \]
Dijkstra's two-stack algorithm

**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.
**Dijkstra's two-stack algorithm**

**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.
Dijkstra's two-stack algorithm

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.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```
Arithmetic expression evaluation

**Goal.** Evaluate infix expressions.

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

**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
Correctness

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

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

as if the original input were:

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

Repeating the argument:

\[
( 1 + ( 5 \times 20 ) )
( 1 + 100 )
101
\]

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