1.3 Bags, Queues, and Stacks

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

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

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

**Queue.** Examine the item least recently added.  
FIFO = "first in first out"
Client, implementation, interface

Separate interface and implementation.
Ex: stack, queue, bag, priority queue, symbol table, union-find, ....

Benefits.

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

---

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

**Implementation:** actual code implementing operations.

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

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

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

**Warmup client.** Reverse sequence of strings from standard input.
How to implement a stack with a linked list?

A. Can't be done efficiently with a singly-linked list.

B. \( \text{it} \rightarrow \text{was} \rightarrow \text{the} \rightarrow \text{best} \rightarrow \text{of} \rightarrow \text{null} \)

C. \( \text{of} \rightarrow \text{best} \rightarrow \text{the} \rightarrow \text{was} \rightarrow \text{it} \rightarrow \text{null} \)
Stack: linked-list implementation

- Maintain pointer \texttt{first} to first node in a singly-linked list.
- Push new item before \texttt{first}.
- Pop item from \texttt{first}.

\texttt{null}

\begin{tikzpicture}[->,>=stealth',shorten >=1pt,auto,node distance=2.8cm,thick,main node/.style={fill=black!20,rectangle,rounded corners,font={\small}}]

\node[main node] (of) {of};
\node[main node] (best) [right of=of] {best};
\node[main node] (the) [right of=best] {the};
\node[main node] (was) [right of=the] {was};
\node[main node] (it) [right of=was] {it};
\node[main node] (null) [right of=it] {null};

\path (of) edge (best)
(best) edge (the)
(the) edge (was)
(was) edge (it)
(it) edge (null);
\end{tikzpicture}
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**

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

**create a new node for the beginning**

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

**set the instance variables in the new node**

```java
first.item = "not";
first.next = oldfirst;
```

**save a link to the list**

```java
Node oldfirst = first;
```
Stack: linked-list implementation in Java

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

    private class Node {
        String item;
        Node next;
    }

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

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

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

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.

inner class

```java
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 per stack node

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

A. Can't be done efficiently with an array.

B. 

```
<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>
```

top of stack

C. 

```
<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>
```

top of stack
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 $s[N-1]$.

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

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

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

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

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

Overflow and underflow.

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

Null items. We allow null items to be inserted.

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 outstanding 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 insert first \( N \) items = \( N + (2 + 4 + \ldots + 2(N - 1)) \sim N^2 \).

**Challenge.** Ensure that array resizing happens infrequently.
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 insert first $N = 2^i$ items. $N + (2 + 4 + 8 + \ldots + N) \sim 3N$. 

1 array access per push \hspace{1cm} 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 \( N \).

\[
\begin{array}{ccccccc}
N = 5 & \text{to} & \text{be} & \text{or} & \text{not} & \text{to} & \text{null} & \text{null} & \text{null} \\
N = 4 & \text{to} & \text{be} & \text{or} & \text{not} \\
N = 5 & \text{to} & \text{be} & \text{or} & \text{not} & \text{to} & \text{null} & \text{null} & \text{null} \\
N = 4 & \text{to} & \text{be} & \text{or} & \text{not} \\
\end{array}
\]
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.
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>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>
Stack resizing-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.

```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.

![Diagram of linked list with first node labeled not, and subsequent nodes labeled be, or, and to, with null pointers after each node.}

```
N = 4   to   be   or   not   null   null   null   null
```

```
first ----> not ----> or ----> be ----> to
          |    |    |    |    |
          |    |    |    |    |
          null null null null
```
1.3 Bags, Queues, and Stacks

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

```java
public class QueueOfStrings {
    QueueOfStrings() {
        // create an empty queue
    }
    void enqueue(String item) {
        // insert a new string onto queue
    }
    String dequeue() {
        // remove and return the string least recently added
    }
    boolean isEmpty() {
        // is the queue empty?
    }
    int size() {
        // number of strings on the queue
    }
}
```
How to implement a queue with a linked list?

A. Can't be done efficiently with a singly-linked list.

B. 

- back of queue
  - times
  - of
  - best
  - the
  - was
  - it
  - null

- front of queue

C. 

- front of queue
  - it
  - was
  - the
  - best
  - of
  - times
  - null

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

A. Can't be done efficiently with an array.

B.  

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<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>
<td>null</td>
<td>null</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>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

C.  

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<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>
<td>null</td>
<td>null</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>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
Queue: resizing-array implementation

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

<table>
<thead>
<tr>
<th>q[]</th>
<th>front of queue</th>
<th>back of queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>null</td>
<td>the</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>2</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, 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, ....

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

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

run-time error
Parameterized stack

We implemented: StackOfStrings.

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

Attempt 3. Java generics.

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.

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

    private class Node
    {
        String item;
        Node next;
    }

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

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

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

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

    private class Node
    {
        Item item;
        Node 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 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:

```java
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 .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 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

the way it is
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> s = new Stack<Integer>();
s.push(17); // s.push(Integer.valueOf(17));
int a = s.pop(); // int a = s.pop().intValue();
```

Bottom line. Client code can use generic stack for any type of data.
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
**Iteration**

**Design challenge.** Support iteration over stack items by client, without revealing the internal representation of the stack.

![Diagram of iteration through a stack]

---

**Java solution.** Make stack implement the `java.lang.Iterable` interface.
Iterators

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.

“foreach” statement (shorthand)

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

equivalent code (longhand)

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

**java.lang.Iterable interface**

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

**java.util.Iterator interface**

```java
public interface Iterator<Item>
{
  boolean hasNext();
  Item next();
  void remove(); // optional; use at your own risk
}
```
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]; }
    }
}
Questions:

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

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

Q. How to detect?
A. 
  - Count total number of `push()` and `pop()` operations in Stack.
  - Save counts in *Iterator* subclass upon creation.
  - If, when calling `next()` and `hasNext()`, the current counts do not equal the saved counts, throw exception.
1.3 Bags, Queues, and Stacks

- stacks
- resizing arrays
- queues
- generics
- iterators
- applications
Java collections library

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
</table>
| `List<
Item>`                  | implements `Iterable<
Item>`                    |
| `List()`                    | create an empty list                             |
| boolean `isEmpty()`         | is the list empty?                               |
| int `size()`                | number of items                                  |
| void `add(Item item)`       | append 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 resizing array;
`java.util.LinkedList` uses linked list. caveat: only some operations are efficient
Java collections library

java.util.Stack.

- Supports push(), pop(), and iteration.
- Extends java.util.Vector, which implements java.util.List interface from previous slide, including get() and remove().
- Bloated and poorly-designed API (why?)

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

`java.util.Stack`.
- Supports `push()`, `pop()`, and iteration.
- Extends `java.util.Vector`, which implements `java.util.List` interface from previous slide, including `get()` and `remove()`.
- Bloated and poorly-designed API (why?)

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

Best practices. Use our implementations of Stack, Queue, and Bag.
War story (from Assignment 1)

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.ArrayList of $N^2$ closed sites.
  Pick an index at random and delete.
  Takes $\sim c_2 N^4$ seconds.

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

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

Java
Adobe PostScript
Compilers
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.
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!
Dijkstra's two-stack algorithm demo

infix expression
(fully parenthesized)

value stack
operator stack

operand
operator

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

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

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, ...