CSE 143
Lecture 8

More Stacks and Queues;
Complexity (Big-Oh)

reading: 13.1 - 13.3

slides created by Marty Stepp
http://www.cs.washington.edu/143/

Stack/queue exercise

- A *postfix expression* is a mathematical expression but with the operators written after the operands rather than before.

  `1 + 1` becomes `1 1 +`

  `1 + 2 * 3 + 4` becomes `1 2 3 * + 4 +`

  - supported by many kinds of fancy calculators
  - never need to use parentheses
  - never need to use an = character to evaluate on a calculator

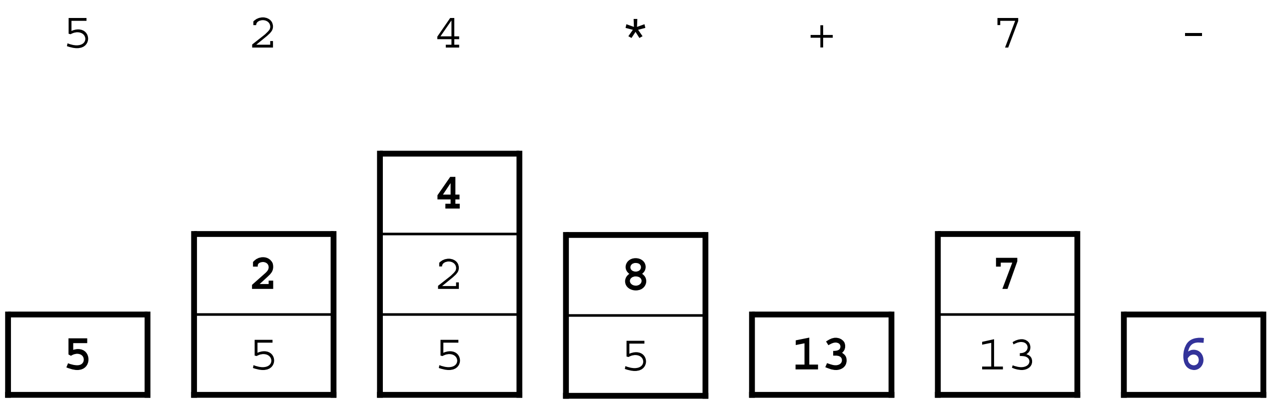
- Write a method `postfixEvaluate` that accepts a postfix expression string, evaluates it, and returns the result.
  - All operands are integers; legal operators are `+` , `-`, `*`, and `/`

  `postFixEvaluate("5 2 4 * + 7 -")` returns `6`

Postfix algorithm

- The algorithm: Use a **stack**
  - When you see an operand, push it onto the stack.
  - When you see an operator:
    - pop the last two operands off of the stack.
    - apply the operator to them.
    - push the result onto the stack.
  - When you're done, the one remaining stack element is the result.

`"5 2 4 * + 7 -"`



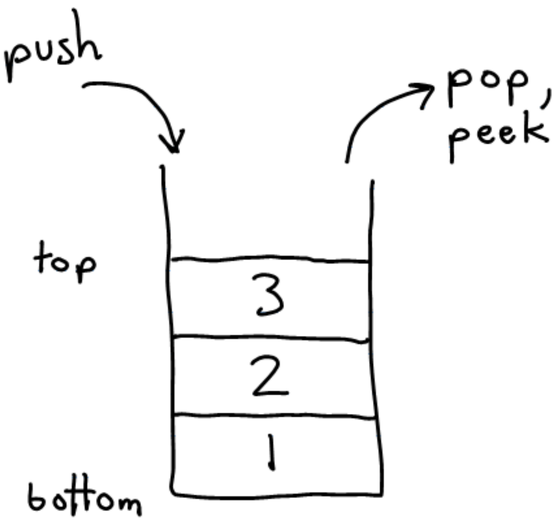
Exercise solution

```
// Evaluates the given prefix expression and returns its result.
// Precondition: string represents a legal postfix expression
public static int postfixEvaluate(String expression) {
    Stack<Integer> s = new Stack<Integer>();
    Scanner input = new Scanner(expression);
    while (input.hasNext()) {
        if (input.hasNextInt()) {     // an operand (integer)
            s.push(input.nextInt());
        } else {                      // an operator
            String operator = input.next();
            int operand2 = s.pop();
            int operand1 = s.pop();
            if (operator.equals("+")) {
                s.push(operand1 + operand2);
            } else if (operator.equals("-")) {
                s.push(operand1 - operand2);
            } else if (operator.equals("*")) {
                s.push(operand1 * operand2);
            } else {
                s.push(operand1 / operand2);
            }
        }
    }
    return s.pop();
}
```

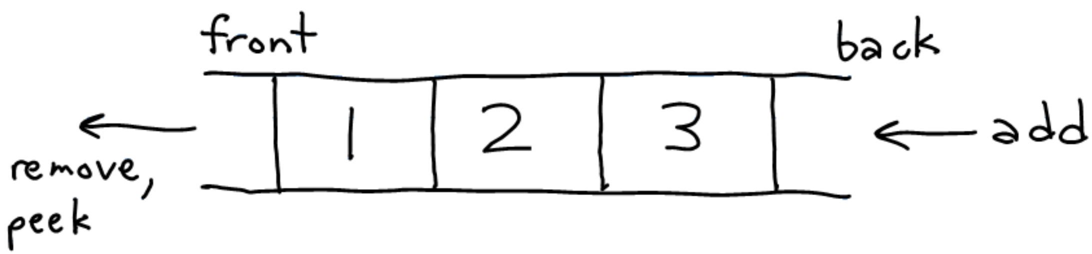
Stack/queue motivation

- Sometimes it is good to have a collection that is less powerful, but is optimized to perform certain operations very quickly.

- Stacks and queues do few things, but they do them efficiently.



stack



queue

Runtime Efficiency (13.2)

- **efficiency**: A measure of the use of computing resources by code.
  - can be relative to speed (time), memory (space), etc.
  - most commonly refers to run time

- Assume the following:
  - Any single Java statement takes the same amount of time to run.
  - A method call's runtime is measured by the total of the statements inside the method's body.
  - A loop's runtime, if the loop repeats N times, is N times the runtime of the statements in its body.

Efficiency examples

```
statement1;
statement2;
statement3;

for (int i = 1; i <= N; i++) {
    statement4;
}

for (int i = 1; i <= N; i++) {
    statement5;
    statement6;
    statement7;
}
```

3

N

3N

4N + 3

Efficiency examples 2

```
for (int i = 1; i <= N; i++) {
    for (int j = 1; j <= N; j++) {
        statement1;
    }
}
```

N^2

```
for (int i = 1; i <= N; i++) {
    statement2;
    statement3;
    statement4;
    statement5;
}
```

$4N$

$N^2 + 4N$

- How many statements will execute if N = 10? If N = 1000?

Algorithm growth rates (13.2)

- We measure runtime in proportion to the input data size, N.
  - **growth rate**: Change in runtime as N changes.

- Say an algorithm runs $\mathbf{0.4N^3 + 25N^2 + 8N + 17}$ statements.
  - Consider the runtime when N is *extremely large* .
  - We ignore constants like 25 because they are tiny next to N.
  - The highest-order term (N^3) dominates the overall runtime.
  - We say that this algorithm runs "on the order of" N^3.
  - or $\mathbf{O(N^3)}$ for short ("**Big-Oh** of N cubed")

Complexity classes

- **complexity class**: A category of algorithm efficiency based on the algorithm's relationship to the input size N.

Class	Big-Oh	If you double N, ...	Example
constant	$O(1)$	unchanged	10ms
logarithmic	$O(\log_2 N)$	increases slightly	175ms
linear	$O(N)$	doubles	3.2 sec
log-linear	$O(N \log_2 N)$	slightly more than doubles	6 sec
quadratic	$O(N^2)$	quadruples	1 min 42 sec
cubic	$O(N^3)$	multiplies by 8	55 min
...	...	...	...
exponential	$O(2^N)$	multiplies drastically	$5 * 10^{61}$ years

Collection efficiency

- Efficiency of various operations on different collections:

Method	ArrayList	SortedIntList	Stack	Queue
`add` (or `push`)	O(1)	O(N)	O(1)	O(1)
`add(`**index**`,` **value**`)`	O(N)		-	-
`indexOf`	O(N)	O(?)	-	-
`get`	O(1)	O(1)	-	-
`remove`	O(N)	O(N)	O(1)	O(1)
`set`	O(1)	O(1)	-	-
`size`	O(1)	O(1)	O(1)	O(1)

Binary search (13.1, 13.3)

- **binary search** successively eliminates half of the elements.
  - *Algorithm:* Examine the middle element of the array.
    - If it is too big, eliminate the right half of the array and repeat.
    - If it is too small, eliminate the left half of the array and repeat.
    - Else it is the value we're searching for, so stop.

  - Which indexes does the algorithm examine to find value **22**?
  - What is the runtime complexity class of binary search?

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*
value	-4	-1	0	2	3	5	6	8	11	14	22	29	31	37	56

Binary search runtime

- For an array of size N, it eliminates ½ until 1 element remains.

  N, N/2, N/4, N/8, ..., 4, 2, 1

  - How many divisions does it take?

- Think of it from the other direction:
  - How many times do I have to multiply by 2 to reach N?

    1, 2, 4, 8, ..., N/4, N/2, N
  - Call this number of multiplications "x".

    $2^x = N$

    $\mathbf{x = \log_2 N}$

- Binary search is in the **logarithmic** complexity class.

Range algorithm

What complexity class is this algorithm? Can it be improved?

```
// returns the range of values in the given array;
// the difference between elements furthest apart
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int maxDiff = 0;      // look at each pair of values
    for (int i = 0; i < numbers.length; i++) {
        for (int j = 0; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]);
            if (diff > maxDiff) {
                maxDiff = diff;
            }
        }
    }
    return diff;
}
```

Range algorithm 2

The algorithm is **O(N²)**. A slightly better version:

```
// returns the range of values in the given array;
// the difference between elements furthest apart
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int maxDiff = 0;      // look at each pair of values
    for (int i = 0; i < numbers.length; i++) {
        for (int j = i + 1; j < numbers.length; j++) {
            int diff = Math.abs(numbers[j] - numbers[i]);
            if (diff > maxDiff) {
                maxDiff = diff;
            }
        }
    }
    return diff;
}
```

Range algorithm 3

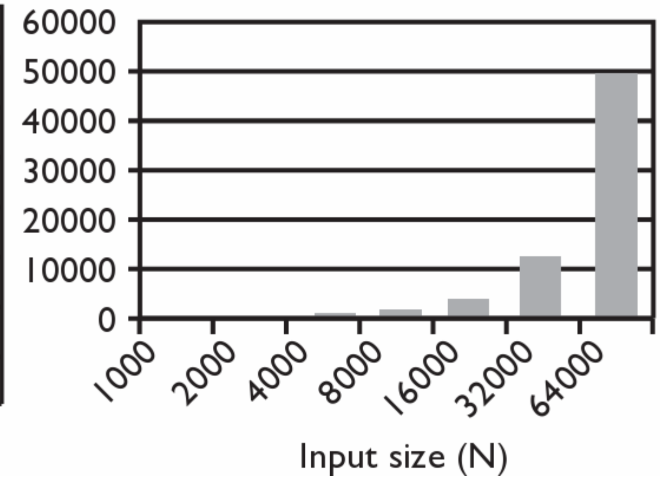
This final version is O(N). It runs MUCH faster:

```
// returns the range of values in the given array;
// example: range({17, 29, 11, 4, 20, 8}) is 25
public static int range(int[] numbers) {
    int max = numbers[0];      // find max/min values
    int min = max;
    for (int i = 1; i < numbers.length; i++) {
        if (numbers[i] < min) {
            min = numbers[i];
        }
        if (numbers[i] > max) {
            max = numbers[i];
        }
    }
    return max - min;
}
```

Runtime of first 2 versions

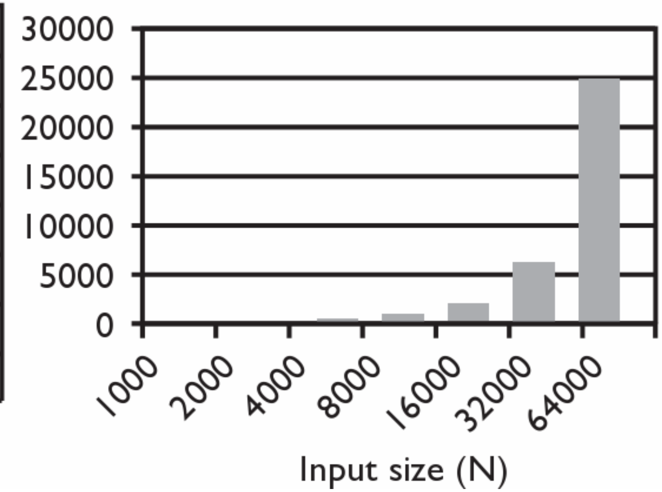
- Version 1:

N	Runtime (ms)
1000	15
2000	47
4000	203
8000	781
16000	3110
32000	12563
64000	49937



- Version 2:

N	Runtime (ms)
1000	16
2000	16
4000	110
8000	406
16000	1578
32000	6265
64000	25031



Runtime of 3rd version

- Version 3:

N	Runtime (ms)
1000	0
2000	0
4000	0
8000	0
16000	0
32000	0
64000	0
128000	0
256000	0
512000	0
1e6	0
2e6	16
4e6	31
8e6	47
1.67e7	94
3.3e7	188
6.5e7	453
1.3e8	797
2.6e8	1578