CSE 211

More Stacks and Queues;
Complexity (Big-Oh)

reading: 13.1 - 13.3

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

Notes on the homework

• You must implement the LetterInventory API exactly as described.
• The structure of your testing code is up to you, as long as it works and is easy to read and maintain.
• You must verify that methods that should throw exceptions do throw exceptions.
• If you wish, you may verify that methods that should not throw exceptions do not throw exceptions.

Type Parameters (Generics)

ArrayList<Type> name = new ArrayList<Type>();

• When declaring or constructing an ArrayList (or LinkedList or List or Stack or Queue) you must specify the type of elements it will contain between < and >.
  – This is called a type parameter or a generic class.
  – Allows the same ArrayList class to store lists of different types.

ArrayList<String> names = new ArrayList<String>();
names.add("Mr. Rogers");
names.add("Big Bird");

ArrayList of primitives?

• The type you specify when creating an ArrayList must be an object type; it cannot be a primitive type.

// illegal -- int cannot be a type parameter
ArrayList<int> list = new ArrayList<int>();

• But we can still use ArrayList with primitive types by using special classes called wrapper classes in their place.

// creates a list of ints
ArrayList<Integer> list = new ArrayList<Integer>();
Wrapper classes

- A wrapper is an object whose sole purpose is to hold a primitive value.

- Once you construct the list, use it with primitives as normal:

  ```java
  ArrayList<Double> grades = new ArrayList<Double>();
  grades.add(3.2);
  grades.add(2.7);
  ...
  double myGrade = grades.get(0);
  ```

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Stack/queue exercise

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

  - 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.

  
  \[
  \text{"5 2 4 * + 7 -"}
  \]

  \[
  \begin{array}{cccccc}
  5 & 2 & 4 & + & 7 & - \\
  5 & 2 & 4 & 8 & 7 & 6
  \end{array}
  \]

Exercise solution

```java
// 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>(){
        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 if (operator.equals("/")) {
                s.push(operand1 / operand2);
            }
            return s.pop();
        }
    }
    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.

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;
}
```

```
N
> 4N + 3
```

```
N2
```

Efficiency examples 2

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

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

```
N2 + 4N
```

```
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 \( 0.4N^3 + 25N^2 + 8N + 17 \) 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 \( O(N^3) \) for short ("Big-Oh of \( N \) cubed")

(For formal definitions and details, take CIS 313 and 314.)

Complexity classes

- complexity class: A category of algorithm efficiency based on the algorithm’s relationship to the input size \( N \).

<table>
<thead>
<tr>
<th>Class</th>
<th>Big-Oh</th>
<th>If you double ( N ), ...</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>( O(1) )</td>
<td>unchanged</td>
<td>10ms</td>
</tr>
<tr>
<td>logarithmic</td>
<td>( O(\log_2 N) )</td>
<td>increases slightly</td>
<td>175ms</td>
</tr>
<tr>
<td>linear</td>
<td>( O(N) )</td>
<td>doubles</td>
<td>3.2 sec</td>
</tr>
<tr>
<td>log-linear</td>
<td>( O(N \log_2 N) )</td>
<td>slightly more than doubles</td>
<td>6 sec</td>
</tr>
<tr>
<td>quadratic</td>
<td>( O(N^2) )</td>
<td>quadruples</td>
<td>1 min 42 sec</td>
</tr>
<tr>
<td>cubic</td>
<td>( O(N^3) )</td>
<td>multiplies by 8</td>
<td>55 min</td>
</tr>
<tr>
<td>exponential</td>
<td>( O(2^N) )</td>
<td>multiplies drastically</td>
<td>5 * 10^{11}</td>
</tr>
</tbody>
</table>

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?

<table>
<thead>
<tr>
<th>index</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-4</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>11</td>
</tr>
<tr>
<td>9</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>29</td>
</tr>
<tr>
<td>12</td>
<td>31</td>
</tr>
<tr>
<td>13</td>
<td>37</td>
</tr>
<tr>
<td>14</td>
<td>56</td>
</tr>
</tbody>
</table>
Binary search runtime

- For an array of size N, it eliminates $\frac{1}{2}$ until 1 element remains.
  \[ N, \frac{N}{2}, \frac{N}{4}, \frac{N}{8}, \ldots, 4, 2, 1 \]
  - 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?
    \[ 1, 2, 4, 8, \ldots, \frac{N}{4}, \frac{N}{2}, N \]
  - Call this number of multiplications "x".
    \[ 2^x = N \]
    \[ x = \log_2 N \]

- Binary search is in the \textit{logarithmic} complexity class.

Range algorithm

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

```java
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 last algorithm is $O(N^2)$. A slightly better version:

```java
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 maxDiff;
}
```

Range algorithm 3

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

```java
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 | 118
  - 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
  - 1024000 | 248
  - 2048000 | 16
  - 4096000 | 31
  - 8192000 | 47
  - 16384000 | 94
  - 32768000 | 188
  - 65536000 | 433
  - 131072000 | 797
  - 262144000 | 1378