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/
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 -"

```
  5  2  4  *  +  7  -
  5  2  4  8  13  7  6
```

5  2  4  *  +  7  -
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.
• **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

\[
\begin{align*}
\text{statement1; } & \quad 3 \\
\text{statement2; } & \quad 3 \\
\text{statement3; } & \quad 3 \\
\text{for (int } i = 1; i \leq N; i++) \{ & \\
\text{statement4; } & \quad N \\
\} \\
\text{for (int } i = 1; i \leq N; i++) \{ & \\
\text{statement5; } & \quad 3N \\
\text{statement6; } & \\
\text{statement7; } & \\
\} \\
\end{align*}
\]
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;
}

• 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")
**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>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>exponential</td>
<td>$O(2^N)$</td>
<td>multiplies drastically</td>
<td>$5 \times 10^{61}$ years</td>
</tr>
</tbody>
</table>
### Collection efficiency

- Efficiency of various operations on different collections:

<table>
<thead>
<tr>
<th>Method</th>
<th>ArrayList</th>
<th>SortedIntList</th>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (or push)</td>
<td>O(1)</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(index, value)</td>
<td>O(N)</td>
<td></td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>indexOf</td>
<td>O(N)</td>
<td>O(?)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>get</td>
<td>O(1)</td>
<td>O(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>remove</td>
<td>O(N)</td>
<td>O(N)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>set</td>
<td>O(1)</td>
<td>O(1)</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>size</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
• **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>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>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>-4</td>
<td>-1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>22</td>
<td>29</td>
<td>31</td>
<td>37</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 **logarithmic** complexity class.
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;
}
The algorithm is $O(N^2)$. A slightly better version:

```java
// 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;
}
```
This final version is $O(N)$. It runs MUCH faster:

```java
// 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:**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>15</td>
</tr>
<tr>
<td>2000</td>
<td>47</td>
</tr>
<tr>
<td>4000</td>
<td>203</td>
</tr>
<tr>
<td>8000</td>
<td>781</td>
</tr>
<tr>
<td>16000</td>
<td>3110</td>
</tr>
<tr>
<td>32000</td>
<td>12563</td>
</tr>
<tr>
<td>64000</td>
<td>49937</td>
</tr>
</tbody>
</table>

- **Version 2:**
  
<table>
<thead>
<tr>
<th>N</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>16</td>
</tr>
<tr>
<td>2000</td>
<td>16</td>
</tr>
<tr>
<td>4000</td>
<td>110</td>
</tr>
<tr>
<td>8000</td>
<td>406</td>
</tr>
<tr>
<td>16000</td>
<td>1578</td>
</tr>
<tr>
<td>32000</td>
<td>6265</td>
</tr>
<tr>
<td>64000</td>
<td>25031</td>
</tr>
</tbody>
</table>
Runtime of 3rd version

• Version 3:

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