L7: Runtime Efficiency

Alex Steiger
CompSci 201: Spring 2024
2/5/24
Logistics, Coming up

• Today 2/6
  • Project 1 NBody due today
  • Project 2 Markov releasing tomorrow (due in 2 weeks)

• Wednesday 2/8
  • APT 3 due

• Next Monday 2/13
  • Midterm Exam 1
    • Covers everything through THIS week, up to and including asymptotic analysis / Big O
    • Example/Practice exams available this evening
Midterm Exams

See details on [course website assignments and grading page](#)

- 60 minutes, in-class exam
- Multiple choice + short answer
- Reason about algorithms, data structures, code.
- Can bring 1 double sided reference sheet (8.5x11 in), write/type whatever notes you like.
- No electronic devices out during exam
Exam Grades and Missing Exams

• Three midterm exams scheduled: \((E_1, E_2, E_3)\)
• Final exam has 3 corresponding parts: \((F_1, F_2, F_3)\)
  • Worth 11% of grade

• Overall exam grade has four exam parts
  • Part 1, Part 2, Part 3, Final: 11% each
  • Part \(i\) grade: \(\max(E_i, F_i)\)

• Meaning the final exam serves in part:
  • As a makeup, if you need to miss a midterm, and/or
  • As an opportunity to demonstrate more learning, if you’re unhappy with your midterm score.
Midterm 1 Material/Concepts

• Lectures 1-8
  • WOTO answers being added to early slides tonight

• Discussion 1-4
  • Solutions to documents on schedule since D1

• Project 0, Project 1

• APT 1-3, required problems
  • Optional/challenge not expected, but great practice
Midterm 1 Material/Concepts

• Java
  • Methods (return types, parameters)
  • Classes/Objects (instance variables, constructors)
  • Primitive types (e.g., int, double,...) vs. primitive types (e.g., Integer, Double, String, [any class])
  • Immutability (e.g. of primitive types and Strings)
  • Static vs. non-static
  • Overriding methods (e.g., .equals, .hashCode, .toString)

• List/Set/Map ADTs
  • Methods / APIs
  • ArrayList impl. of List
  • HashMap impl. of Map (hash tables) using ArrayList for buckets
    • .equals/.hashCode contract
    • Efficiency of implementation’s methods, e.g., of .contains

• Asymptotic Analysis / Runtime Efficiency
  • Big O notation
  • Analyze runtime of code snippets with respect to parameter size
Wrapping up Maps

HashMap methods at a high level

Always start by getting the hash = Math.abs(key.hashCode()) % list.size()

• put(key, value)
  • Add <key, value> pair to “bucket” (list) at index hash if existing pair with same key
  • Otherwise replace existing pair with given value

• get(key)
  • Return value paired with key in bucket at index hash (or return null if no such pair)

• containsKey(key)
  • Check if key exists in bucket at index hash

Absolute value and % (remainder when dividing by) list size ensures valid index

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>&lt;“hi”, 5&gt; &lt;“fain”, 104&gt;</td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>&lt;“ok”, 3&gt;</td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
Revisiting Hashing Efficiency

• Runtime of `get()`, `put()`, and `containsKey()`
  = Time to get the hash
  + Time to search “bucket” by calling `.equals()` on everything in the bucket

⇒ HashMaps are faster with more buckets
Simple uniform hashing assumption (SUHA)

- Suppose we hash N pairs to M buckets.

**Simple uniform hashing assumption:**
Any element (i.e., key for HashMap, value for HashSet) is equally likely to hash into any bucket, independently of where any other element hashes to. [CLRS]

- Probability any two unequal elements hash into the same bucket: 1/M
  - Spread of pairs to buckets looks random (but is not).
  - Ways to design such hash functions, not today
  - We make this assumption to analyze efficiency in theory, can verify runtime performance in practice
Implications of SUHA

• Expected number of pairs per bucket under SUHA? N/M [N pairs, M buckets].

• Stronger statements are true: Very high probability that any bucket has approximately N/M pairs.

• Runtime implication?
  • Time to get the hash
  • Time to search over the “bucket” at hash index
  • Calling .equals() on everything in the bucket

Expect ~ N/M pairs to search

Constant, does not depend on N or M.
Memory/Runtime Tradeoff

• N pairs, M buckets, assuming SUHA / good `hashCode()`

• **Case 1: N >> M** – too many pairs in too few buckets
  • Runtime inefficient

• **Case 2: M >> N** – too many buckets, not many pairs
  • Runtime efficient, NOT memory efficient

• **Case 3: M slightly larger than N** – sweet spot
  • Runtime efficient, memory usage slightly more than an array/ArrayList
Load Factor and HashMap Growth

• N pairs, M buckets

• **Load factor** = maximum N/M ratio allowed
  • Java default is 0.75

• Whenever N/M exceeds the load factor?
  • Create a new larger table, rehash/copy everything
  • Double the size, just like ArrayList!
    • Geometric growth pattern for amortized efficiency
  • Called *resizing*
Hash table resizing

| jshell>  Math.abs("cs".hashCode()) % 4 | 0 |
|        $15 => 0                     |   |
| jshell>  Math.abs("hi".hashCode()) % 4 | 1 |
|        $16 => 1                     |   |
| jshell>  Math.abs("ok".hashCode()) % 4 | 0 |
|        $17 => 0                     |   |

| jshell>  Math.abs("cs".hashCode()) % 8 | 0 |
|        $19 => 0                      |   |
| jshell>  Math.abs("hi".hashCode()) % 8 | 1 |
|        $20 => 1                      |   |
| jshell>  Math.abs("ok".hashCode()) % 8 | 4 |
|        $21 => 4                      |   |

| 0       | "cs", 201 |
| 1       | "hi", 5   |
| 2       |           |
| 3       | "ok", 3   |

Resizing
WOTO

Go to duke.is/v/syyy

Not graded for correctness, just participation.

Try to answer without looking back at slides and notes.

But do talk to your neighbors!
2. Which methods must be correctly implemented in order for a HashSet/HashMap to function correctly? Select all that apply. * □

- ✔ equals() for the key objects
-  □ equals() for the value objects
- ✔ hashCode() for the key objects
-  □ hashCode() for the value objects
3. Suppose you store one million (1,000,000) Keys in a HashSet where the hashCode() of all the keys returns 0 but none of the keys are equal to each other (according to equals()). What would you expect when calling contains() on the HashSet? *

○ Incorrect behavior, returning the wrong value

○ Correct and efficient behavior, constant time

✓ Correct and inefficient behavior, comparable to contains in ArrayList

○ None of the above
4. Suppose a HashSet/Map performs a resizing operation to double the number of buckets every time it reaches a load factor of 1. Assume a good implementation of hashCode() for the keys / the simple uniform hashing assumption. When performing N add/put operations with unique keys, the best characterization of the runtime complexity of add/put is... *

- Constant time
- Amortized constant time
- Expected constant time
- Amortized expected constant time
Revisiting guarantees

```java
public class HashSet<E>
extends AbstractSet<E>
implements Set<E>, Cloneable, Serializable
```

This class implements the Set interface, backed by a hash table (actually a HashMap instance). It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time. This class permits the null element.

This class offers constant time performance for the basic operations (add, remove, contains and size), assuming the hash function disperses the elements properly among the buckets. Iterating over this set requires time proportional to the sum of the HashSet instance's size (the number of elements) plus the "capacity" of the backing HashMap instance (the number of buckets). Thus, it's very important not to set the initial capacity too high (or the load factor too low) if iteration performance is important.

Java API documentation

Constant **amortized** time operations **in expectation** under SUHA (practically: assuming the hash function distributes unequal keys).
Runtime Efficiency, an Empirical Look at String Concatenation
Two methods for repeated concatenation

Method A: Using String object and basic + operator

```java
public static String repeatConcatA(int reps, String toConcat) {
    String result = new String();
    for (int i=0; i<reps; i++) {
        result += toConcat;
    }
    return result;
}
```

Method B: Using StringBuilder object and append method

```java
public static String repeatConcatB(int reps, String toConcat) {
    StringBuilder result = new StringBuilder();
    for (int i=0; i<reps; i++) {
        result.append(toConcat);
    }
    return result.toString();
}
```
Empirical timing experiment

Can see the code on GitLab here.

```java
public class StringConcatTiming {
    static final int NUM_TRIALS = 100;
    static final int REPS_PER_TRIAL = 1024;
    static final String TO_CONCAT = "201";

    public static void main(String[] args) {
        long totalTime = 0;
        for (int trial=0; trial<NUM_TRIALS; trial++) {
            long startTime = System.nanoTime();
            //repeatConcatA(REPS_PER_TRIAL, TO_CONCAT);
            repeatConcatB(REPS_PER_TRIAL, TO_CONCAT);
            long endTime = System.nanoTime();
            totalTime += (endTime - startTime);
        }
        double avgTime = (double)totalTime / NUM_TRIALS;
        System.out.printf("Avg time per trial is %f ms", avgTime*1E-6);
    }
}
```

static final used for constants here

Going to time both methods separately.
Empirical results

![Graph showing Empirical results](image)

- **Empirical results**
- **Method A [String] (ms)**
- **Method B [StringBuilder] (ms)**
Empirical results in more detail

<table>
<thead>
<tr>
<th>Reps</th>
<th>MethodA (ms)</th>
<th>MethodB (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.384</td>
<td>0.050</td>
</tr>
<tr>
<td>2048</td>
<td>1.136</td>
<td>0.061</td>
</tr>
<tr>
<td>4096</td>
<td>3.443</td>
<td>0.077</td>
</tr>
<tr>
<td>8192</td>
<td>12.244</td>
<td>0.099</td>
</tr>
<tr>
<td>16384</td>
<td>41.754</td>
<td>0.143</td>
</tr>
<tr>
<td>32768</td>
<td>147.719</td>
<td>0.207</td>
</tr>
</tbody>
</table>

Multiply reps by 2 multiplies runtime by 4. Quadratic complexity.

Multiply reps by 2 multiplies runtime by ~2. Linear complexity.
Empirical results in more detail

<table>
<thead>
<tr>
<th>Reps</th>
<th>MethodA ns/rep</th>
<th>MethodB ns/rep</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.375</td>
<td>0.048</td>
</tr>
<tr>
<td>2048</td>
<td>0.555</td>
<td>0.030</td>
</tr>
<tr>
<td>4096</td>
<td>0.841</td>
<td>0.019</td>
</tr>
<tr>
<td>8192</td>
<td>1.495</td>
<td>0.012</td>
</tr>
<tr>
<td>16384</td>
<td>2.548</td>
<td>0.009</td>
</tr>
<tr>
<td>32768</td>
<td>4.508</td>
<td>0.006</td>
</tr>
</tbody>
</table>

Runtime / rep increasing, *greater than linear* complexity.

Runtime / rep not increasing, at most *linear* complexity.
What's going on? Documentation?

[docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String](docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/String)

**Class String**

java.lang.Object

java.lang.String

**All Implemented Interfaces:**

Serializable, CharSequence, Comparable<String>, Constable, ConstantDesc

```java
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence, Constable, ConstantDesc
```

The `String` class represents character strings. All string literals in Java programs, such as "abc", are implemented as `String` objects. **Strings are constant; their values cannot be changed after they are created**. String buffers support mutable strings.
methodA revisited

public static String repeatConcatA(int reps, String toConcat) {
    String result = new String();
    for (int i=0; i<reps; i++) {
        result += toConcat;
    }
    return result;
}

How many characters will be copied per iteration if toConcat is “201”?

• i=0: 3
• i=1: 6
• i=2: 9
• ...
• On iteration i, need to copy \(3 \times (i+1)\) characters!

String is immutable; line 22 creates a new string and copies result then toConcat.
How many total characters are copied? Algebra!

**method A:** for \( i \) from 0 to \( \text{reps} - 1 \),
copy \( 3(i+1) \) characters per iteration.

\[
\sum_{i=0}^{\text{reps}-1} 3(i + 1) = 3(\text{reps}) + 3 \left( \sum_{i=0}^{\text{reps}-1} i \right)
\]

\[
= 3(\text{reps}) + 3 \left( \frac{\text{reps}}{2} \right) (0 + \text{reps} - 1)
\]

\[
= \frac{3}{2} (\text{reps}^2 + \text{reps})
\]

Arithmetic series formula:
\[
\sum_{i=1}^{n} a_i = \left( \frac{n}{2} \right) (a_1 + a_n)
\]

2/5/24

CompSci 201, Spring 2024, Runtime Efficiency
Abstracting, Intro to Big O Notation (Preview for next time)

• The $\frac{3}{2}$ in $\frac{3}{2} \text{reps}^2$ doesn’t tell us much about how the performance scales with the size of reps.

• Often, we use asymptotic notation, especially **Big O notation** to abstract away constants.

• For example: Let $N = \text{reps}$, then we say that the asymptotic runtime complexity is $O(N^2)$.
  - If you ~double $N$, you ~quadruple the runtime
Two general Big O rules

1. Can drop constants
   - $2N + 3 \Rightarrow O(N)$
   - $0.001N + 1,000,000 \Rightarrow O(N)$

2. Can drop lower order terms
   - $2N^2 + 3N \Rightarrow O(2N^2) \Rightarrow O(N^2)$
   - $N + \log(N) \Rightarrow O(N)$
   - $2^N + N^2 \Rightarrow O(2^N)$
## Hierarchy of some common complexity classes

<table>
<thead>
<tr>
<th>Big O</th>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(2^N)$</td>
<td>Exponential</td>
<td>Calculate all subsets of a set</td>
</tr>
<tr>
<td>$O(N^3)$</td>
<td>Cubic</td>
<td>Multiply N x N matrices</td>
</tr>
<tr>
<td>$O(N^2)$</td>
<td>Quadratic</td>
<td>Loop over all <em>pairs</em> from N things</td>
</tr>
<tr>
<td>$O(N \log(N))$</td>
<td>Nearly-linear</td>
<td>Sorting algorithms</td>
</tr>
<tr>
<td>$O(N)$</td>
<td>Linear</td>
<td>Loop over N things</td>
</tr>
<tr>
<td>$O(\log(N))$</td>
<td>Logarithmic</td>
<td>Binary search a sorted list</td>
</tr>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
<td>Addition, array access, etc.</td>
</tr>
</tbody>
</table>
How does StringBuilder work?

“Every string builder has a capacity. As long as the length of the character sequence contained in the string builder does not exceed the capacity, it is not necessary to allocate a new internal buffer. If the internal buffer overflows, it is automatically made larger.” - StringBuilder JDK 17 documentation.

• But how does it grow?
• Geometrically! Like ArrayList, HashMap, ...
  • Still linear amortized complexity, for same reasons
StringBuilder is like an ArrayList of characters

• Suppose we run the code:

```java
StringBuilder() sb = new StringBuilder(3);
sb.append("hi");
sb.append("ya");
```

Array representing StringBuilder:

```
<table>
<thead>
<tr>
<th>h</th>
<th>i</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>h</td>
<td>i</td>
<td>y</td>
</tr>
</tbody>
</table>
```

Initial buffer/array capacity
How many total characters are copied with a StringBuilder?

Suppose we

• start with capacity 3,
• append a length 3 string \texttt{reps} times, and
• double when out of capacity.

\[
\approx \log_2(3 \cdot \text{reps})
\]

\[
3 \cdot \text{reps} + \sum_{i=0}^{\approx \log_2(3 \cdot \text{reps})} 2^i = 3 \cdot \text{reps} + 6 \cdot \text{reps}
\]

\[
= 9 \cdot \text{reps}
\]

The “good case” copies

From doubling and copying the array

Geometric series formula:

\[
\sum_{i=0}^{n} a r^i = a \left( \frac{1 - r^{n+1}}{1 - r} \right)
\]
public static String repeatConcatB(int reps, String toConcat) {
    StringBuilder result = new StringBuilder();
    for (int i=0; i<reps; i++) {
        result.append(toConcat);
    }
    System.out.printf("String builder capacity is %d characters%n", result.capacity());
    System.out.printf("Result length is %d characters%n", result.length());
    return result.toString();
}

String builder capacity is 147454 characters
Result length is 98304 characters

Final StringBuilder is using about 146k / 98k ~ = 1.5 times as much memory as necessary. Very common tradeoff in data structures!
What’s the real difference between methodA and methodB?

• methodA: Copies roughly $\frac{3}{2}(\text{reps}^2 - \text{reps})$
• methodB: copies roughly $9 \cdot \text{reps}$ characters.

<table>
<thead>
<tr>
<th>Reps</th>
<th>~MethodA char copies (millions)</th>
<th>MethodB char copies (millions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>1.5</td>
<td>0.009</td>
</tr>
<tr>
<td>2000</td>
<td>6</td>
<td>0.018</td>
</tr>
<tr>
<td>4000</td>
<td>24</td>
<td>0.036</td>
</tr>
<tr>
<td>8000</td>
<td>95</td>
<td>0.072</td>
</tr>
<tr>
<td>16000</td>
<td>383</td>
<td>0.144</td>
</tr>
<tr>
<td>32000</td>
<td>1535</td>
<td>0.288</td>
</tr>
</tbody>
</table>
WOTO

Go to duke.is/m/pm9u

Not graded for correctness, just participation.

Try to answer without looking back at slides and notes.

But do talk to your neighbors!
How many total characters must be copied by the code on lines 8 and 9? Remember that Strings are immutable in Java. *

- 5
- 7
- 9
- 10
- 15
- 30

```java
7    String s = "hi";
8    s += "hey";
9    s += s;
```
Suppose method A has linear complexity and takes 10 ms to run on an input of size N. About what would you expect the runtime to be for an input of size 2*N?

* 20 ms

- 10 ms
- 40 ms
- 100 ms
Suppose method B has quadratic complexity and takes 10 ms to run on an input of size N. About what would you expect the runtime to be for an input of size 2*N? *

- 10 ms
- 20 ms
- 40 ms ✔
- 100 ms
Here is another String concatenation method. Suppose the input string s has a small number of characters, say 3. As a function of the parameter reps, how would you characterize the runtime complexity of the method? Hint: As a function of reps, how many total characters will be copied across all iterations of the loop? *  

7  
8  public static String concatAlot(int reps, String s)  
9    for (int i=0; i<reps; i++) {    
10      s += s;  
11    }  
12  return s;

- Constant
- Linear
- Quadratic
- Exponential