Lecture 21: Hashing

08/12/22
Upcoming

• Checkpoint 8 due Sunday 8/14 @ 11:59pm
• A6 Resubmission due **Tuesday** 8/16 @ 11:59pm
• A8 due **Tuesday** 8/16 @ 11:59pm
  • **Cannot be resubmitted!**
  • Late days allowed, but the last day of IPL is Wednesday, 8/17
Runtime Efficiency of `contains`

- Array, ArrayList, LinkedList:
- TreeSet:
- HashSet:
Runtime Efficiency of `contains`:

- Array, ArrayList, LinkedList: $O(N)$
- TreeSet: $O(\log N)$
- HashSet: $O(1)$
Arrays

- **Random access:** we can jump straight to any index in an array

<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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>0</td>
<td>11</td>
<td>5</td>
<td>-1</td>
<td>24</td>
<td>2</td>
<td>3</td>
<td>7</td>
<td>0</td>
<td>49</td>
</tr>
</tbody>
</table>
Really Big Array – my idea 😊

- Store Set of student ids: 0 – 9,999,999
  - add(id)
  - contains(id)

<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>…</th>
<th>9,999,999</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Hashing

• **hash**: To map a value to an integer index.
• **hash table**: An array that stores elements via hashing.
• **hash function**: A function that maps values to indexes.
# Hashing Example

- Hash function: $h(x) = x \mod 10$
- Add: 3, 16, 24, 300
- Contains: 16, 27

<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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Bad Hash Functions (why?)

- \( h(x) = 1 \)

- \( h(x) = \text{rand.nextInt}() \)
Bad Hash Functions (why?)

• $h(x) = 1$
  Everything hashes to the same index – lots of collisions!

• $h(x) = \text{rand.nextInt}()$
  Not consistent – we can’t find our elements after we put them in our set!
Good Hash Functions

• Maps a value to a number
  • passing in the same value should always give the same result

• Results from a hash function should be distributed over a range
  • very bad if everything hashes to 1!
  • should "look random"

• Should be “fast”
Hashing Objects

• Object class – superclass of everything
  • public String toString()
  • public int hashCode()
    • This is a built-in hash function!
Hashing Strings

• How would we write a hash function for String objects?
String's hashCode

• The `hashCode` function inside `String` objects looks like this:

```java
public int hashCode() {
    int hash = 0;
    for (int i = 0; i < this.length(); i++) {
        hash = 31 * hash + this.charAt(i);
    }
    return hash;
}
```

• As with any general hashing function, collisions are possible.
  • Example: "Ea" and "FB" have the same hash value.
Let’s implement our own HashSet!
Collisions

- **collision**: When hash function maps 2 values to same index.

Example: $h(x) = x \% 10$

```java
set.add(24);
set.add(7);
set.add(54);
```

<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>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Separate Chaining

- **chaining**: Resolving collisions by storing a list at each index.
  - add/contains/remove must traverse lists, but the lists are short
Practical points

• Use known hash functions – don’t reinvent the wheel!
• When you override `hashCode()` you must always override `equals()` as well! (and vice versa)
• Use prime numbers for table sizes
• Rehash when the hash table gets too crowded
Rehashing

- **rehash**: Growing to a larger array when the table is too full.
  - Cannot simply copy the old array to a new one. (Why not?)

- **load factor**: ratio of \( \frac{\# \text{ of elements}}{\text{hash table length}} \)
  - many collections rehash when load factor \( \approx .75 \)
  - can use big prime numbers as hash table sizes to reduce collisions