B E F O R E  W E  S T A R T
Announcements

• EX1 (Algo Analysis I) due **TONIGHT 11:59pm PDT**
  - You can use late days on exercises, just like projects!

• P2 (Maps) and EX2 (Algo Analysis II) released today

• Summations Reference published (on course calendar under Wednesday’s lecture)
  - Section handout has a cheat-sheet version
P2: Maps

• Implement everyone’s good pal: the Hash Map!
• Like P1, look at multiple data structures under single ADT
  - But this time, we have the algorithmic analysis tools to reason about more complicated situations (especially Case Analysis!)
• 3 Parts:
  - ArrayMap
  - ChainedHashMap
  - Experiments
• Start early! In particular, ChainedHashMap iterator can take a long time!
Welcome to the Data Structures Part™

• We’re now armed with a toolbox stuffed full of analysis tools
  - Wednesday was the last algorithmic analysis lecture
  - It’s time to apply this theory to more practical topics!

• Today, we’ll take our first deep dive using those tools on a data structure: Hash Maps!
Learning Objectives

After this lecture, you should be able to...

1. Compare the relative pros/cons of various Map implementations, especially given a design like the ones we cover today

2. Trace operations in a Separate Chaining Hash Map on paper (such as insertion, getting an element, resizing)

3. Implement a Separate Chaining Hash Map in code (P2)

4. Differentiate between the “worst” and “in practice” runtimes of a Separate Chaining Hash Map, and describe what assumptions allow us to consider the “in practice” case
Lecture Outline

MAP ADT

1. ArrayMap
   - As seen on Project 2
   - FASTER: Jump directly to element, only int keys

2. DirectAccessMap
   - FASTER: Jump directly to element, only int keys

3. SimpleHashMap
   - MORE FLEXIBLE: Hash function supports any type of key

4. SeparateChaining HashMap
   - YOUR BEST FRIEND: Addresses limitations with hash collisions, but still fast!

Review
Lecture Outline

3. SeparateChaining HashMap
   YOUR BEST FRIEND: Addresses limitations with hash collisions, but still fast!

2. SimpleHashMap
   MORE FLEXIBLE: Hash function supports any type of key

1. DirectAccessMap
   FASTER: Jump directly to element, only int keys

Review

MAP ADT
The Map ADT

- **Map**: an ADT representing a set of distinct keys and a collection of values, where each key is associated with one value.
  - Also known as a **dictionary**
  - If a key is already associated with something, calling `put(key, value)` replaces the old value

- **Used all over the place**
  - It’s hard to work on a big project without needing one sooner or later
  - CSE 143 introduced:
    - `Map<String, Integer> map1 = new HashMap<>();`
    - `Map<String, String> map2 = new TreeMap<>();`

---

**MAP ADT**

**State**
- Set of keys, Collection of values
- Count of keys

**Behavior**
- `put(key, value)` add value to collection, associated with key
- `get(key)` return value associated with key
- `containsKey(key)` return if key is associated
- `remove(key)` remove key and associated value
- `size()` return count
- `clear()` remove all
- `iterator()` get an iterator
## Review Implementing a Map with an Array

<table>
<thead>
<tr>
<th>MAP ADT</th>
<th>ArrayMap&lt;K, V&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>State</strong></td>
<td>State</td>
</tr>
<tr>
<td>Set of keys, Collection of values</td>
<td>Pair&lt;K, V&gt;[] data</td>
</tr>
<tr>
<td>Count of keys</td>
<td></td>
</tr>
<tr>
<td><strong>Behavior</strong></td>
<td>Behavior</td>
</tr>
<tr>
<td>put(key, value) add value to collection, associated with key</td>
<td>put find key, overwrite value if there. Otherwise create new pair, add to next available spot, grow array if necessary</td>
</tr>
<tr>
<td>get(key) return value associated with key</td>
<td>get scan all pairs looking for given key, return associated item if found</td>
</tr>
<tr>
<td>containsKey(key) return if key is associated</td>
<td>containsKey scan all pairs, return if key is found</td>
</tr>
<tr>
<td>remove(key) remove key and associated value</td>
<td>remove scan all pairs, replace pair to be removed with last pair in collection</td>
</tr>
<tr>
<td>size() return count</td>
<td>size return count of items in dictionary</td>
</tr>
</tbody>
</table>

### Big-Oh Analysis – (if key is the last one looked at / not in the dictionary)

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>get()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>remove()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>

### Big-Oh Analysis – (if the key is the first one looked at)

<table>
<thead>
<tr>
<th>Method</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>get()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>remove()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>

### Example

```java
put('b', 97)
put('e', 20)
```

<table>
<thead>
<tr>
<th>Index</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>('a', 1)</td>
</tr>
<tr>
<td>1</td>
<td>('b', 97)</td>
</tr>
<tr>
<td>2</td>
<td>('c', 3)</td>
</tr>
<tr>
<td>3</td>
<td>('d', 4)</td>
</tr>
<tr>
<td>4</td>
<td>('e', 20)</td>
</tr>
</tbody>
</table>

```java
```
# Review Implementing a Map with Linked Nodes

## MAP ADT

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set of keys, Collection of values</td>
<td>put(key, value) add value to collection, associated with key</td>
</tr>
<tr>
<td>Count of keys</td>
<td>get(key) return value associated with key</td>
</tr>
<tr>
<td></td>
<td>containsKey(key) return if key is associated</td>
</tr>
<tr>
<td></td>
<td>remove(key) remove key and associated value</td>
</tr>
<tr>
<td></td>
<td>size() return count</td>
</tr>
</tbody>
</table>

```java
containsKey('c')
get('d')
put('b', 20)
```

## LinkedMap<K, V>

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>front</td>
<td>put if key is unused, create new with pair, add to front of list, else replace with new value</td>
</tr>
<tr>
<td>size</td>
<td>get scan all pairs looking for given key, return associated item if found</td>
</tr>
<tr>
<td></td>
<td>containsKey scan all pairs, return if key is found</td>
</tr>
<tr>
<td></td>
<td>remove scan all pairs, skip pair to be removed</td>
</tr>
<tr>
<td></td>
<td>size return count of items in dictionary</td>
</tr>
</tbody>
</table>

```
front
```

## Big O Analysis – (if key is the last one looked at / not in the dictionary)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>get()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>remove()</td>
<td>O(n) linear</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>

## Big O Analysis – (if the key is the first one looked at)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>put()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>get()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>containsKey()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>remove()</td>
<td>O(1) constant</td>
</tr>
<tr>
<td>size()</td>
<td>O(1) constant</td>
</tr>
</tbody>
</table>
Could we do better?

- put, get, and remove have $\Theta(n)$ runtimes. Could we use a $\Theta(1)$ operation to improve?
- What about array indexing?
  - data[i] (array access) and data[i] = 2 (array update) are constant runtime!
  - What if we could jump directly to the requested key?
  - We could simplify the problem: only allow integer keys
Lecture Outline

**Review**

1. **ArrayMap**
   - **DirectAccessMap**
   - **SimpleHashMap**

2. **FASTER**: Jump directly to element, only int keys

3. **MORE FLEXIBLE**: Hash function supports any type of key

4. **SeparateChaining HashMap**
   - **YOUR BEST FRIEND**: Addresses limitations with hash collisions, but still fast!
DirectAccessMap

• put, get, and remove have $\Theta(n)$ runtimes. Could we use a $\Theta(1)$ operation to improve?
• What about array indexing?
  - data[i] (array access) and data[i] = 2 (array update) are constant runtime!
  - What if we could jump directly to the requested key?
  - We could simplify the problem: \textbf{only allow integer keys}

\begin{verbatim}
put(3, “Alex”)
get(3)
\end{verbatim}

<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>data</td>
<td></td>
<td></td>
<td></td>
<td>Alex</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

DirectAccessMap<K, V>

<table>
<thead>
<tr>
<th>State</th>
<th>data[]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Behavior</td>
<td>size</td>
</tr>
<tr>
<td>put item at given index</td>
<td>get item at given index</td>
</tr>
<tr>
<td>containsKey if data[] null at index, return false, return true otherwise</td>
<td></td>
</tr>
<tr>
<td>remove nullify element at index</td>
<td></td>
</tr>
<tr>
<td>size return count of items in dictionary</td>
<td></td>
</tr>
</tbody>
</table>
DirectAccessMap Implementation

```java
public void put(int key, V value) {
    this.array[key] = value;
}

public boolean containsKey(int key) {
    return this.array[key] != null;
}

public V get(int key) {
    return this.array[key];
}

public void remove(int key) {
    this.array[key] = null;
}
```

DirectAccessMap\(<K, V>\>

<table>
<thead>
<tr>
<th>State</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>data[]</td>
<td>put put item at given index</td>
</tr>
<tr>
<td>size</td>
<td>get get item at given index</td>
</tr>
<tr>
<td></td>
<td>containsKey if data[] null at index, return false, return true otherwise</td>
</tr>
<tr>
<td></td>
<td>remove nullify element at index</td>
</tr>
<tr>
<td></td>
<td>size return count of items in dictionary</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operation</th>
<th>Case</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>get(key)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>containsKey(key)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>
Pros and Cons of DirectAccessMap

What’s a benefit of using it? What’s a drawback?
Pros and Cons of DirectAccessMap

• Super Fast!
  - Everything is $\Theta(1)$

• Wasted Space
  - Say we want to store 0 and 999999999. This implementation would waste all the space inbetween 😞

• Only Integer Keys
  - Would be nice to store any type of data 😞
  - But note what’s so useful here: being able to go quickly from key to array index
Can We Store Any Integer?

IDEA 1
• Create a GIANT array with every possible integer as an index
• Problems:
  - Can we allocate an array big enough?
  - Super wasteful

IDEA 2
• Create a smaller array, with a translation from integer keys into available indices
• Problems:
  - How can we construct a translation?
Hash Functions

- **Hash Function**: any function that can be used to map data of an arbitrary size to fixed-size values.
  
  - We want to translate from the set of all integers to the set of valid indexes in our array.

\[
9002 \mod 10 = 2 \quad \text{(so store it in index 2 of the array)}
\]

- One simple approach: take the key and \% (mod) it by size of the array.
# Mod: Remainder

- The `%` operator computes the remainder from integer division.

\[
\begin{array}{c|c|c}
14 & \% & 4 \\
\hline
12 & \% & 2 \\
\hline
2 & \% & 2 \\
\end{array}
\quad
\begin{array}{c|c|c}
218 & \% & 5 \\
\hline
20 & \% & 3 \\
\hline
18 & \% & 3 \\
\hline
15 & \% & 3 \\
\end{array}
\]

Equivalent, to find \(a \% b\) (for \(a, b > 0\)):

\[
\begin{array}{l}
\text{while}(a > b-1) \\
a -= b; \\
\text{return } a;
\end{array}
\]

- Applications of `%` operator:
  - Obtain last digit of a number: \(230857 \% 10\) is 7
  - See whether a number is odd: \(7 \% 2\) is 1, \(42 \% 2\) is 0
  - Limit integers to specific range: \(8 \% 12\) is 8, \(18 \% 12\) is 6

For more review/practice, check out [https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic](https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/a/what-is-modular-arithmetic)
Lecture Outline

MAP ADT

1. ArrayMap
   - Review

2. DirectAccessMap
   - FASTER: Jump directly to element, only int keys

3. SimpleHashMap
   - MORE FLEXIBLE: Hash function supports any type of key

4. SeparateChaining HashMap
   - YOUR BEST FRIEND: Addresses limitations with hash collisions, but still fast!

- FASTER
- MORE FLEXIBLE
- YOUR BEST FRIEND
SimpleHashMap: “% by size” as Hash Function

**IMPLEMENTATION**

```java
public void put(int key, int value) {
    data[hashToValidIndex(key)] = value;
}
```

```java
public V get(int key) {
    return data[hashToValidIndex(key)];
}
```

```java
public int hashToValidIndex(int k) {
    return k % this.data.length;
}
```

<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>data</td>
<td>I</td>
<td>&lt;3</td>
<td>Hash</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Maps</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

put(0, “I”)       0 % 10 = 0
put(8, “Maps”)    8 % 10 = 8
put(11, “<3”)    11 % 10 = 1
put(23, “Hash”)  23 % 10 = 3
What input will cause a problem?

```
put(0, "I")          0 % 10 = 0
put(8, "Maps")      8 % 10 = 8
put(11, "<3")       11 % 10 = 1
put(23, "Hash")     23 % 10 = 3
```

---

**IMPLEMENTATION**

```java
public void put(int key, int value) {
    data[hashToValidIndex(key)] = value;
}

public V get(int key) {
    return data[hashToValidIndex(key)];
}

public int hashToValidIndex(int k) {
    return k % this.data.length;
}
```
SimpleHashMap: Collisions?!

```
public void put(int key, int value) {
    data[hashToValidIndex(key)] = value;
}

public V get(int key) {
    return data[hashToValidIndex(key)];
}

public int hashToValidIndex(int k) {
    return k % this.data.length;
}
```

<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>data</td>
<td>I</td>
<td>&lt;3</td>
<td>Hash</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Maps</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

put(0, “I”) 0 % 10 = 0
put(8, “Maps”) 8 % 10 = 8
put(11, “<3”) 11 % 10 = 1
put(23, “Hash”) 23 % 10 = 3
put(20, “We”) 20 % 10 = 0

IMPLEMENTATION
Lecture Outline

Review

1. ArrayMap
   - FASTER: Jump directly to element, only int keys

2. DirectAccessMap
   - MORE FLEXIBLE: Hash function supports any type of key

3. SimpleHashMap
   - YOUR BEST FRIEND: Addresses limitations with hash collisions, but still fast!

4. SeparateChaining HashMap

MAP ADT
Handling Collisions

• Two common strategies to handle collisions:

1. Separate Chaining
   "Chain" together multiple values stored in a single bucket

   We’ll focus on separate chaining this quarter, much more common in practice

2. Open Addressing
   If a bucket is taken, find a new bucket using some strategy:
   - Linear Probing
   - Quadratic Probing
   - Double Hashing

Bonus topic beyond the scope of the class
Separate Chaining

• If two values want to live in the same index, let’s just let them be roommates!

• Each index is a “bucket”
  - Linked Nodes are a common implementation for these bucket “chains”

• When item x hashes to index h:
  - If bucket at h is empty, create new list with x
  - Else, add x to the list
Separate Chaining

• If two values want to live in the same index, let’s just let them be roommates!

• Each index is a “bucket”
  - Linked Nodes are a common implementation for these bucket “chains”

• When item x hashes to index h:
  - If bucket at h is empty, create new list with x
  - Else, add x to the list

• But if multiple keys can hash to the same index, need to store the key too!
Separate Chaining

• Implementation of get/put/containsKey very similar

PSEUDOCODE

public boolean get(int key) {
    int bucketIndex = key % data.length;
    loop through each pair in data[bucketIndex]
        if pair.key == key
            return pair.value
    return null if we get here
}

Let’s analyze the runtime. First, are there different possible states for this HashMap to make the code faster or slower, assuming n key/value pairs are already stored?
Separate Chaining Worst Case

- It’s possible that everything hashes to the same bucket by chance!
  - get would take $\Theta(n)$ time 😞

- Consider get(51)
  - Use hash function (% 10) to get index (5)
  - Check every element in bucket for key 51

- We’ve lost that $\Theta(1)$ runtime

PSEUDOCODE

```java
public boolean get(int key) {
    int bucketIndex = key % data.length;
    loop through each pair in data[bucketIndex]
    if pair.key == key
        return pair.value
    return null if we get here
}
```
Separate Chaining Best Case

- However, if everything is spread evenly across the buckets, get takes $\Theta(1)$

- Consider get(22)
  - Use hash function (% 10) to get index (2)
  - Check the single element in bucket for key 22 – a constant time operation!

- Key to a successful Hash Map implementation: how can we keep the buckets as close to this distribution as possible?
Separate Chaining... In Practice

• A well-implemented separate chaining hash map will stay very close to the best case
  - Most of the time, operations are fast. Rarely, do an expensive operation that restores the map close to best case.

• How to stay close to best case?
  - Good distribution & Resizing!

• We can describe the “in-practice” case as what *almost always* happens:
  - (1) items are fairly evenly distributed
  - (2) assume resizing doesn’t occur
    - This is similar to the concept of “amortized”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Case</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>put(key, value)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>In-practice</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>get(key)</td>
<td>In-practice</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>average</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>remove(key)</td>
<td>best</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>In-practice</td>
<td>Θ(1)</td>
</tr>
<tr>
<td></td>
<td>worst</td>
<td>Θ(n)</td>
</tr>
</tbody>
</table>
Resizing

- The runtime to scan each bucket is creeping up
  - If we don’t intervene, our in-practice runtime is going to hit $\Theta(n)$
    - number of buckets is a constant, so $n / (\# \text{ buckets})$ is $\Theta(n)$
Resizing

Don’t forget to re-distribute your keys!

If we just expand the buckets array, several values are hashed in the wrong place.

How to Resize:
1. Expand the buckets array
2. For every element in the old hash table, re-distribute!
   Recompute its position by taking the mod with the new length.
When to Resize?

• In ArrayList, we were forced to resize when we ran out of room
  - In SeparateChainingHashMap, never \textit{forced} to resize, but we want to make sure the buckets don’t get too long for good runtime

• How do we quantify “too full”?  
  - Look at the average bucket size: number of elements / number of buckets

\[
\lambda = \frac{n}{c}
\]

\begin{align*}
0 &: \\
1 &: (1, \text{red}) \rightarrow (6, \text{pink}) \\
2 &: (22, \text{tan}) \rightarrow (7, \text{blue}) \rightarrow (77, \text{aqua}) \\
3 &: (8, \text{lilac}) \rightarrow (53, \text{puce}) \\
4 &: (4, \text{orange}) \\
\end{align*}

\[\lambda = \frac{8}{5} = 1.6\]
When to Resize?

• In ArrayList, we were forced to resize when we ran out of room
  - In SeparateChainingHashMap, never forced to resize, but we want to make sure the buckets don’t get too long for good runtime

• How do we quantify “too full”?
  - Look at the average bucket size: number of elements / number of buckets

• If we resize when $\lambda$ hits some constant value like 1:
  - We expect to see 1 element per bucket: constant runtime!
  - If we double the capacity each time, the expensive resize operation becomes less and less frequent

LOAD FACTOR $\lambda$

- $n$: total number of key/value pairs
- $c$: capacity of the array (# of buckets)

$$\lambda = \frac{n}{c}$$
Hashing

• What about non-integer data?
  - Remember the definition -- **Hash Function**: any function that can be used to map data of an arbitrary size to fixed-size values.

  ![Diagram of hash function with examples]

• Considerations for Hash Functions:
  1. **Deterministic** – same input should generate the same output
  2. **Efficient** – reasonable runtime
  3. **Uniform** – inputs spread “evenly” across output range
Hashing

Implementation 1: Simple aspect of values
public int hashCode(String input) {
    return input.length();
}

Pro: super fast
Con: lots of collisions!

Implementation 2: More aspects of value
public int hashCode(String input) {
    int output = 0;
    for(char c : input) {
        out += (int)c;
    }
    return output;
}

Pro: still really fast
Con: some collisions

Implementation 3: Multiple aspects of value + math!
public int hashCode(String input) {
    int output = 1;
    for (char c : input) {
        int nextPrime = getNextPrime();
        out *= Math.pow(nextPrime, (int)c);
    }
    return Math.pow(nextPrime, input.length());
}

Pro: few collisions
Con: slower, gigantic integers
Hashing

• Fortunately, experts have made most of these design decisions for us!
  - All objects in Java have a .hashCode() method that does some magic to make a “good” hash for any object type (e.g. String, ArrayList, Scanner)
  - The built-in hashCode() has a good distribution/not a lot of collisions

• More precisely, hashCode() just gets us an int representation: then we % by size

1. call key.hashCode() to get int representation of object
2. Mod (%) by the number of buckets to get our index
**Review**  Iterators

- **Iterator**: a Java interface that dictates how a collection of data should be traversed. Can only move forward and in a single pass.

  ```java
  Iterator Interface
  Behavior
  hasNext() – true if elements remain
  next() – returns next element
  ```

  hasNext() – returns true if the iteration has more elements yet to be examined

  next() – returns the next element in the iteration and moves the iterator forward to next item

  Two ways to *use* an iterator in Java:

  ```java
  ArrayList<Integer> list;
  Iterator itr = list.iterator();
  while (itr.hasNext()) {
      int item = itr.next();
  }
  ```

  ```java
  ArrayList<Integer> list;
  for (int i : list) {
      int item = i;
  }
  ```
P2 Reminders

- Implementing an iterator for a Hash Map is **complex!**
  - You need to iterate through the elements of a bucket, but when you reach the end of the chain, **have to move to the next bucket**
  - “you’re not iterating over some linear data structure, you’re playing 2D chess”
  - Howard Xiao

- Start early! P2 is out for over 1.5 weeks, but for good reason!
  - Especially the ChainedHashMap iterator

- Remember to read the entire Tips section of the instructions!