1. What’s the load factor?

2. Which of the following is a good choice of threshold for the load factor before we resize?
   - n
   - log n
   - 1
Announcements

• Current Assignments: P2 (Due in 1.5 weeks), EX2 (Due Friday)
  - Tonight is the late cutoff for P1, Wednesday is the late cutoff for EX1
  - P2: Iterator Overview video now available!

• Lectures are now also available via Panopto
  - Panoptos will be linked under each lecture on website within a few hours
  - Still available via Zoom tab on Canvas – use whichever one you prefer

• Anonymous feedback: sound quality (lecture, review videos)! Thanks for letting us know 😊
  - For section videos: find higher-quality audio in video description

• Course Feedback survey sent tonight
  - Now that you’ve seen lectures, sections, projects, and exercises – please let us know what’s working for you and what could be improved!
Exam I Logistics

• You’ll have 48 hours (no time limit once you start or anything. Just submit by the deadline!)

• Released 7/24 12:01 AM PDT

• Due 7/25 11:59 PM PDT
  - No late submissions!

• Open notes and open friends: you’ll be able to work on and submit in groups

• Covers content up through this Friday’s lecture (Memory & Caching, B-Trees)

• Will focus on conceptual questions. Be prepared to explain why things are true and defend your reasoning
Learning Objectives

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

1. Describe the properties of a good hash function and the role of a hash function in making a Hash Map efficient

2. Evaluate invariants based on their strength and maintainability, and come up with the invariants for data structure implementations

3. Compare and contrast the properties and runtimes of BTs, BSTs, and AVL Trees

4. Describe the AVL invariant, explain how it affects AVL Tree runtimes, and compare it with the BST invariant
Lecture Outline

• Hash Map (Wrap-Up)
  - Hashing & Applications

• Binary Trees & Binary Search Trees (BSTs)

• Case Analysis on BSTs

• Choosing a Good Invariant (AVL Trees)
**Review** 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!
Review  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 on this particular operation
    - 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>
Review 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.

• 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();
}

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

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: super fast
Con: lots of collisions!

Pro: still really fast
Con: some collisions

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
Content Hashing: Applications

• Caching:
  - You’ve downloaded a large video file. You want to know if a new version is available. Rather than re-downloading the entire file, compare your file’s hash value with the server’s hash value.

• File Verification / Error Checking:
  - Same implementation
  - Can be used to verify files on your machine, files spread across multiple servers, etc.

• Fingerprinting
  - Git hashes ("identification")
  - Ad tracking ("identification"): see https://panopticlick.eff.org/
  - YouTube ContentID ("duplicate detection")
Content Hashing: Defining a Salient Feature

• Hash function implementors can choose what’s salient:
  - hash(“cat”) == hash(“CAT”) ???

• What’s salient in detecting that an image or video is unique?

• What’s salient in determining that a user is unique?
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.

  ![Iterator Interface]

  **Iterator Interface**

<table>
<thead>
<tr>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>hasNext() – true if elements remain</td>
</tr>
<tr>
<td>next() – returns next element</td>
</tr>
</tbody>
</table>

  **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 available for over 1.5 weeks, but for good reason!
  - Especially the ChainedHashMap iterator

• Remember to read the entire Tips section of the instructions!
Lecture Outline

• Hash Map (Wrap-Up)
  - Hashing & Applications

• Binary Trees & Binary Search Trees (BSTs)

• Case Analysis on BSTs

• Choosing a Good Invariant (AVL Trees)
143 Review Binary Trees

- A **binary tree** is a collection of nodes where each node has at most 1 parent and anywhere from 0 to 2 children
  - Similar to linked lists (just add an extra child field!)

- **Root node**: the single node with no parent, “top” of the tree. Often called the ‘overallRoot’

- **Leaf node**: a node with no children

- **Subtree**: a node and all its descendants

- **Height**: the number of edges contained in the longest path from root node to any leaf node

```java
public class Node<K> {
    K data;
    Node<K> left;
    Node<K> right;
}
```
143 Review  Tree Height

• What is the height of the following binary trees?
  - **Height**: the number of edges contained in the longest path from root node to any leaf node

  ![Tree Diagrams]

  overallRoot
  
  height: 2

  overallRoot
  
  height: 0

  overallRoot
  
  height: -1 (or N/A)
Other Useful Binary Tree Numbers

For a binary tree of height $h$:

Max number of leaves: $2^h$
Max number of nodes: $2^{h+1} - 1$

Min number of leaves: 1
Min number of nodes: $h + 1$
# 143 Review  Binary Search Tree (BST)

**• Invariants**

- The *rules* for your data structure or algorithm
- Whenever you implement any operation on your data structure:
  - You know the invariants are true at the beginning. Great! Simpler code, fewer cases
  - But you must leave the invariants true at the end.
- Accidentally violating invariants a common source of bugs. Defensive programming: check invariants at beginning/end of methods!

---

**Binary Search Tree Invariant:**

For every node with key $k$:

- The left subtree has only keys smaller than $k$.
- The right subtree has only keys greater than $k$.

---

```java
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true; // Assuming the method returns true when key is found
    } else if (node.key == key) {
        return false; // Assuming the method returns false for non-matching keys
    } else if (node.key > key) {
        return this.containsKeyBST(node.left, key); // Assumes left subtree
    } else if (node.key < key) {
        return this.containsKeyBST(node.right, key); // Assumes right subtree
    }
    return false;
}
```
BST Ordering Applies *Recursively*
**Aside   Anything Can Be a Map**

• Want to make a tree implement the Map ADT?
  - No problem – just add a value field to the nodes, so each node represents a key/value pair.

```java
public class Node<K, V> {
    K key;
    V value;
    Node<K, V> left;
    Node<K, V> right;
}
```

• For simplicity, we’ll just talk about the keys
  - Interactions between nodes are based off of keys (e.g. BST sorts by keys)
  - In other words, keys determine where the nodes go
Lecture Outline

• Hash Map (Wrap-Up)
  - Hashing & Applications

• Binary Trees & Binary Search Trees (BSTs)

• Case Analysis on BSTs

• Choosing a Good Invariant (AVL Trees)
Binary Tree vs. BST: containsKey(5)

**Without BST Invariant**

- 10
- 9
- 1
- 3
- 2
- 30
- 14
- 5

**With BST Invariant**

- 9
- 3
- 10
- 2
- 14
- 30
- 5
- 1
Binary Tree vs. BST: containsKey(5)

public boolean containsKeyBinaryTree(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        return containsKeyBT(node.left) || containsKeyBT(node.right);
    }
}

public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left);
        } else {
            return containsKeyBST(node.right);
        }
    }
}

RECURSIVE PATTERN

2 Constant size Input

Θ (n)

1 Halving the Input

Θ (log n)

Where n is the number of nodes in the tree
**BST containsKey runtime**

```java
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left);
        } else {
            return containsKeyBST(node.right);
        }
    }
}
```

For `containsKey` on a BST, what are some interesting cases for us to consider?

What’s the best? The worst? Are there other interesting cases?

For example, consider what values of key could change the behavior on the above tree.
For containsKey on a BST, what are some interesting cases for us to consider?

What’s the best? The worst? Are there other interesting cases?
For example, consider what values of key could change the behavior on this tree.

**Best Case**

key=9 seems pretty nice! Find it immediately. Θ(1)

**Worst Case**

key=100 is terrible. Have to search the entire tree (and you don’t even get the satisfaction of returning true). Θ(log n)

We can analyze with recurrences:

\[
T(n) = \begin{cases} 
T \left( \frac{n}{2} \right) + 1 & \text{if } n > 1 \\
3 & \text{otherwise}
\end{cases}
\]

\[T(n) = \Theta(\log n)\]
Is it possible to do worse than $\Theta(\log n)$? 😈

• Sources of variation that affect the runtime:
  1. Number of elements in the tree (n)
  2. Position of searched-for key in tree
  3. Arrangement of elements in tree

Our new recurrence:

$$T(n) = \begin{cases} T(n - 1) + 1 & \text{if } n > 1 \\ 3 & \text{otherwise} \end{cases}$$

$$T(n) = \Theta(n)$$
**BST Extremes**

- Here are two different extremes our BST could end up in:

  **Perfectly balanced** – for every node, its descendants are split evenly between left and right subtrees.

  **Degenerate** – for every node, all of its descendants are in the right subtree.
Can we do better?

• Key observation: what ended up being important was the height of the tree!
  - **Height**: the number of edges contained in the longest path from root node to any leaf node
  - In the worst case, this is the number of recursive calls we’ll have to make

• If we can limit the height of our tree, the BST invariant can take care of quickly finding the target
  - How do we limit?
  - Let’s try to find an invariant that forces the height to be short
Lecture Outline

• Hash Map (Wrap-Up)
  - Hashing & Applications

• Binary Trees & Binary Search Trees (BSTs)

• Case Analysis on BSTs

• Choosing a Good Invariant (AVL Trees)
In Search of a “Short BST” Invariant: Take 1

• What about this?

PUBLIC

BST Height Invariant
The height of the tree must not exceed $\Theta(\log n)$

• This is technically what we want (would be amazing if true on entry)
• But how do we implement it so it’s true on exit?
  - Should the insertBST method rebuild the entire tree balanced every time?
    This invariant is too broad to have a clear implementation
• Invariants are tools – more of an art than a science, but we want to pick one that is specific enough to be maintainable