Let's analyze BST's `contains()` and `add()` performance

- Just focus on worst case.

<table>
<thead>
<tr>
<th>Operations</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>add an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>??</td>
</tr>
<tr>
<td>contains an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>??</td>
</tr>
<tr>
<td>remove an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>??</td>
</tr>
</tbody>
</table>
Let's analyze BST's `contains()` and `add()` performance

- Just focus on worst case:
  - Takes the longest path from root to a leaf (there's a term for that)
  - So it's $O(H)$, where $H$ is the height of the tree

<table>
<thead>
<tr>
<th>Operations</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>add an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$</td>
</tr>
<tr>
<td>contains an item?</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$</td>
</tr>
<tr>
<td>remove an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$</td>
</tr>
</tbody>
</table>
But BSTs have different shapes and therefore different heights

- Sometimes, BSTs are short and squatty (left; what we want)
- Other times they approximate lists (right; what we want to avoid)

Same items built differently
Shapes of Trees

- **Full Binary Tree**: All nodes have either 0 or 2 children

"Full, but not balanced"
Shapes of Trees

- **Full Binary Tree**: All nodes have either 0 or 2 children
- **Perfect Binary Tree**: Full binary tree with \( \log n \) height, where \( n \) is the number of nodes (proof to come)
Shapes of Trees

- **Full Binary Tree**: All nodes have either 0 or 2 children
- **Perfect Binary Tree**: Full binary tree with $\log n$ height, where $n$ is the number of nodes (proof to come)
- **Balanced Binary Tree**: Perfect except for lowest level, missing nodes on the right

![Diagram of trees]

"Full, but not balanced"  "Perfect"  "Balanced"
Shapes of Trees

- *Unbalanced BSTs* are to be avoided
  - Worst case (height = number of nodes)
    - Just a LinkedList that happens to satisfy BST properties (*i.e.*, that it is sorted)
BST Performance

- BSTs generally perform better than lists for add/find/remove
  - As long as you keep the tree balanced, that is!
    - Harder to do (CS 361/455 topic: Red-Black trees, AVL trees, B+trees)

<table>
<thead>
<tr>
<th>Operations</th>
<th>ArrayList</th>
<th>LinkedList</th>
<th>Binary Search Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>add an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$ avg/worst *</td>
</tr>
<tr>
<td>contains an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$ avg/worst *</td>
</tr>
<tr>
<td>remove an item</td>
<td>$O(n)$ avg/worst</td>
<td>$O(n)$ avg/worst</td>
<td>$O(H)$ avg/worst *</td>
</tr>
</tbody>
</table>

* Note:
  $H = \log n$, when tree is balanced
  $H = n$, when tree is unbalanced
BST Removal Complexity

- Case 1: Node is a leaf
  - Find the node: \( O(H) \)
  - Remove it (set reference to null): \( O(1) \)
  - Total: \( O(H) \)

- Case 2: Node has one child
  - Find the node: \( O(H) \)
  - Save reference of its left or right child: \( O(1) \)
  - Remove the node: \( O(1) \)
  - Link up child: \( O(1) \)
  - Total: \( O(H) \)
Case 3: Node has both children

- Find the node: $O(H)$
- Find the in-order predecessor (how?): $O(H)$
- Remove the node: $O(1)$
- Replace with in-order predecessor: $O(1)$
- Remove in-order predecessor (how?): $O(H)$
- Total: $O(H)$
Outline

- Tree Terminologies
  - Relationships
  - Tree Node Classifications
  - Subtrees
  - Properties
- Binary Trees
  - Implementation
  - Traversal
- Binary Search Tree
  - Implementation
- Conclusion
Implementing BSTs

Things worth considering

- `BinarySearchTree<E>` is just a special form of `BinaryTree<E>`
- The data it stores must implement `Comparable`
- Implements the `SearchTree<E>` interface

```java
public interface SearchTree<E> {
    void add(E item);
    boolean contains(E target);
    void remove(E target);
}
```

Let's update the class diagram
BinarySearchTree<E> Class

Important about E: It must be comparable

```java
public class BinarySearchTree<E extends Comparable<E>> extends BinaryTree<E> implementsSearchTree<E> {
    /**
     * Creates an empty BST
     */
    public BinarySearchTree() {
        super();
    }

    /**
     * Creates a BST with the given root
     * @param newRoot reference to the root node
     */
    public BinarySearchTree(Node<E> newRoot) {
        super(newRoot);
    }

    // methods that implement SearchTree (omitted for now)
}
```

BST inherits all BinaryTree methods and the Node nested class!
Outline

- Tree Terminologies
  - Relationships
  - Tree Node Classifications
  - Subtrees
  - Properties

- Binary Trees
  - Traversal
  - Implementation

- Binary Search Trees
  - Java's TreeSet<E> Class

- Conclusion
Java's TreeSet<E> Class

- Java does have native support for BSTs
  - But they're called TreeSets
    - They're actually Red-Black Trees
    - BSTs that are always balanced

- Import the class to use it:

```java
import java.util.TreeSet;
```
Java's TreeSet<E> Class (Cont.)

- Sample Usage...
  - Lots more methods available in a TreeSet! Check out the API.
  - https://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

```java
TreeSet<Integer> my_bst = new TreeSet<>();
my_bst.add(50);
my_bst.add(100);
my_bst.add(20);
my_bst.add(10);
my_bst.add(15);
my_bst.add(250);
System.out.println(my_bst.last());
> 250

System.out.println(my_bst.first());
> 10

my_bst.remove(10);
System.out.println(my_bst.first());
> 15
```
Outline

- Tree Terminologies
  - Relationships
  - Tree Node Classifications
  - Subtrees
  - Properties

- Binary Trees
  - Traversal
  - Implementation

- Binary Search Trees
  - Java's TreeSet<E> Class

- Conclusion
Conclusion

- Trees allow us to organize hierarchies that are present in data
  - Generally can have any number of children
  - Children only have links to more children (no links back "up" the tree)

- Binary Trees
  - A tree in which all nodes have 0 to 2 children
  - Lots of uses in practice: expression trees, Huffman trees, BSTs, ...

- Binary Search Trees
  - A binary tree in which its nodes are ordered
  - On average performs better than lists
    - As long as the tree is balanced
Exam 2 tomorrow
- @10am in Lab
- Budget for 60 min
- These APIs are given with the Exam: List, Stack, Queue

Hwk 6 (Huffman Encoding) Posted
- Due 11/25
- Find team members now!
Exam 2 feedback

- Things you didn't expect, or expected but didn't appear?

Reminders

- Hwk 6 due 11/25
- Hwk 5 currently being graded

Last time...

- Finished recursion
- Today: Trees
Last time:
- Tree concepts; binary trees
  - Recursive nature of (binary trees)
- Wrote
  - Node<E> nested class
  - 3 BinaryTree constructors
    - Only way to add Nodes so far is with 3-argument constructor (bottom up)
    - E getData(), boolean ifLeaf()

Today:
- Traversals
- Start BSTs
- Tree concepts; binary trees
  - Recursive nature of (binary trees)
- Wrote
  - Node<E> nested class
  - 3 BinaryTree constructors
    – Only way to add Nodes so far is with 3-argument constructor (bottom up)
  - E getData(), boolean ifLeaf()

- Traversals
- Start BSTs
Reminders:
- Hwk 6 due Monday 11/25
- Exam 2 graded
  - Grades don't incl. bonus
    - Recursive code reading
    - Queue programming was a little tricky.

Last time...
- Binary Search Trees (BSTs)
- Today: More BSTs
Binary Search Tree (BST) Review

- Draw a BST that stores the following items
  - 5, 20, 25, 30, 45, 50

- How do we search this tree?
  - 30, 42

- Now add:
  - 0, 20 again...

- Traverse this tree...
  - In-order Traversal