Example binary search tree
Search: method **contains**

- How will it be different for a BST than for a general tree?
  - At each node, if we don’t find what we’re looking for, we know whether to look left or right. We do not have to check both.
BST insertion

• We observed that insertions always add a new leaf.

• Challenge: Try to find a case where we can’t insert by adding a leaf. (There is no such case!)
BST deletion

• Then we considered deleting a root farther up
  – If we could find a leaf whose value could go in the root, we would avoid big structural changes.
  – We observed that the largest value on the left, or the smallest on the right, would work.
  – We wrote the values in the tree in order, and saw that this makes sense.
The Collection ADT

- BSTs can implement the Collection ADT.
- Data: 0 or more items
- Operations:
  - Insert
  - Search
  - Delete
- Search and delete are *by value*, not by position. Unlike a list, there is no notion of position.
- Unlike a stack or queue, order of insertion has no bearing on deletion.
Comparing implementations

• We now know several data structures that can implement Collection, including
  – Sorted Python list
  – Linked list
  – BST
• Which is most efficient?
Collection via sorted Python list

• Pros and cons of keeping it sorted?

• Efficiency of search:

• Efficiency of insertion:

• Efficiency of deletion:
Aside: Worst-case analysis

• For some algorithms, the running time on a given size of input *varies*.
• It depends on other properties of the inputs.
• Examples?
Collection via (unsorted) linked list

- Pros and cons of keeping it sorted?
- Efficiency of search:
- Efficiency of insertion:
- Efficiency of deletion:
Collection via BST

• Efficiency of search:

• Efficiency of insertion:

• Efficiency of deletion:
BST height

• O(h) sounds no better than unsorted lists – linear time.
• But h is the height of our tree, and can be much smaller than the size of our data!
• What is the relationship between
  – number of nodes (size of our data) and
  – height of our tree
Collection via BST, revisited

- Efficiency of search:

- Efficiency of insertion:

- Efficiency of deletion:
### Comparison (worst case)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Balanced BST</th>
<th>Sorted Python list</th>
<th>Linked list</th>
</tr>
</thead>
<tbody>
<tr>
<td>search</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>insert</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>delete</td>
<td>O(log n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>

• So a BST provides the best of both worlds: fast search, plus fast updates.

• But *only if* the tree is balanced!
Keeping trees balanced

• You’ll learn about balanced trees in csc263.
  – (The rotations you did in lab are part of the algorithm.)

• Balanced trees are very important in databases.
• To be fast with large amounts of data, we use branching factors like 1,024!
  – This is a situation where constants matter
  – Height $\log_{1024}(n)$ is much less than $\log_2(n)$

• You’ll learn about databases in csc343, and their implementation in csc443.
Cost-benefit analysis

• Often, we invest extra time on one operation to save time on another.

• Example: Keeping a Python list sorted.
  – Insert and delete are slower
  – Search is faster

• Example: Keeping a tree balanced.
  – Insert and delete are slower
  – Search is faster

• Is it worth it? You will learn this sort of “amortized” analysis in csc263.
  – Hint: Pay attention to probability in your stats class!
Why always analyze worst case?

• The worst case doesn’t always happen!
• What about best case?
  – Not terribly useful. We must be prepared for anything.
• What about average case?
  – This is very useful.
  – Example: If we don’t try to keep a tree balanced, it is unlikely to have height n.
  – But how likely is it to end up how balanced?
  – Tricky!
• You will learn average-case analysis in csc263.
  – Hint: Again, pay attention to probability in your stats class!
An instance of Tree
An instance of **Tree**
An instance of **BinarySearchTree**
An instance of **BinarySearchTree**