Last Time

• Trees
  – Definitions and terms
  – UML layout
  – Adding nodes, retrieving information, and deleting nodes
Computational Efficiency of Trees

• Add: need to traverse the tree to find the parent we want to add a child to, then primitive operations
• Get: need to traverse the tree to find the node with the information we require, then return
• Delete: need to traverse the tree to find the node we want to delete, then many primitive operations

• Tree traversal: could find the node at the root, or on the second node checked, or on the third, fourth, $n^{th}$, $n-1^{th}$, etc.
  – Average case: correct node found on the $n/2$ node checked.
  – Tree traversal is $O(n)$, therefore add, get, and delete are all $O(n)$.
  – Well, that’s not great...
Binary Search Trees

- Occasionally called **sorted trees**
- Rather than specifying where a node should be added in the tree, we’ll place it naturally given the other nodes already in the tree
  - Typical usage:
    - smaller = left
    - larger = right
How does that help us computationally?

• Add: start at the root, and go left if we’re trying to insert something smaller, right if we’re trying to insert something larger.
  – Repeat until a null location is reached, then insert.
  – As we drop down each level of the tree, the number of available positions where we can insert the node is cut in half.
  – Cutting something in half is a log operation, so add is now $O(\log(n))$!
How does that help us computationally?

• Get: start at the root, and go left if we’re trying to find something smaller, right if we’re trying to find something larger.
  – Repeat until either the data is found (which can be returned), or until a null location is reached (that information is not stored in the tree, so return null/false)
  – As we drop down each level of the tree, the number of available positions where we can find the node is cut in half.
  – Cutting something in half is a log operation, so get is now $O(\log(n))$!
How does that help us computationally?

• Delete: start at the root, and go left if we’re trying to delete something smaller, right if we’re trying to delete something larger.
  – Repeat until either the node is found (which can be deleted with primitive operations), or until a null location is reached (that node is not stored in the tree, so return null/false)
  – As we drop down each level of the tree, the number of available positions where we can find the node is cut in half.
  – Cutting something in half is a log operation, so delete is now $O(\log(n))$!
Average Case vs. Worst Case

• An average binary tree is fairly balanced. A worst case binary tree is a linked list.
  – In the worst case:
    • Add is O(n)
    • Get is O(n)
    • Delete is O(n)
  – The worst case is pretty unlikely...
## Comparing Data Structures

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Add</th>
<th>Get</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>O(1) (unless resize)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Linked List</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Stack</td>
<td>O(1)</td>
<td>O(1) (as long as we want what’s on top)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Queue</td>
<td>O(1)</td>
<td>O(1) (as long as we want the front)</td>
<td>O(1)</td>
</tr>
<tr>
<td>ArrayList</td>
<td>O(1) (unless resize)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Tree</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
<td>O(log(n))</td>
</tr>
<tr>
<td>Heap</td>
<td>?</td>
<td>Stay tuned...</td>
<td>?</td>
</tr>
</tbody>
</table>
Any Questions?