Warm Up: Give pseudocode to calculate the height of a Tree

treeHeight(root){
    height = 0;
    ???
    return height;
}
Tree Height

treeHeight(root){
    height = 0;
    if (root.left != Null){
        height = max(height, treeHeight(root.left));
    }
    if (root.right != Null){
        height = max(height, treeHeight(root.right));
    }
    return height;
}
Dictionary (Map) ADT

• Contents:
  • Sets of key+value pairs
  • Keys must be comparable

• Operations:
  • insert(key, value)
    • Adds the (key,value) pair into the dictionary
    • If the key already has a value, overwrite the old value
      • Consequence: Keys cannot be repeated
  • find(key)
    • Returns the value associated with the given key
  • delete(key)
    • Remove the key (and its associated value)
<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Time to insert</th>
<th>Time to find</th>
<th>Time to delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
Less Naïve attempts

• Binary Search Trees (today)
• Tries (Project)
• AVL Trees (next week)
• B-Trees (next week)
• HashTables (week after)
• Red-Black Trees (not included in this course)
• Splay Trees (not included in this course)
Naïve attempts

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Time to insert</th>
<th>Time to find</th>
<th>Time to delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary Search Tree (W.C.)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary Search Tree (average)</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>
Tree Height

treeHeight(root){
    height = 0;
    if (root.left != Null){
        height = max(height, treeHeight(root.left));
    }
    if (root.right != Null){
        height = max(height, treeHeight(root.right));
    }
    return height;
}
More Tree “Vocab”

• Traversal:
  • An algorithm for “visiting” every node in a tree

• Pre-Order Traversal:
  • Root, Left Subtree, Right Subtree

• In-Order Traversal:
  • Left Subtree, Root, Right Subtree

• Post-Order Traversal
  • Left Subtree, Right Subtree, Root
Name that Traversal!

A-orderTraversal(root){
    if (root.left != Null){
        process(root.left);
    }
    if (root.right != Null){
        process(root.right);
    }
    process(root);
}

BorderTraversal(root){
    process(root);
    if (root.left != Null){
        process(root.left);
    }
    if (root.right != Null){
        process(root.right);
    }
}

C-orderTraversal(root){
    if (root.left != Null){
        process(root.left);
    }
    process(root);
    if (root.right != Null){
        process(root.right);
    }
}

Binary Search Tree

- Binary Tree
  - Definition:
- Order Property
  - All keys in the left subtree are smaller than the root
  - All keys in the right subtree are larger than the root
- Why?
Are these BSTs?
Find Operation (recursive)

```java
find(key, root) {
    if (root == Null) {
        return Null;
    }
    if (key == root.key) {
        return root.value;
    }
    if (key < root.key) {
        return find(key, root.left);
    }
    if (key > root.key) {
        return find(key, root.right);
    }
    return Null;
}
```
Find Operation (iterative)

```java
find(key, root)
    while (root != Null && key != root.key)
        if (key < root.key)
            root = root.left;
        else if (key > root.key)
            root = root.right;
    
    if (root == Null)
        return Null;
    return root.value;
```
Insert Operation (iterative)

insert(key, value, root) {
    if (root == Null) { this.root = new Node(key, value); }
    parent = Null;
    while (root != Null && key != root.key) {
        parent = root;
        if (key < root.key) { root = root.left; }
        else if (key > root.key) { root = root.right; }
    }
    if (root != Null) { root.value = value; }
    else if (key < parent.key) { parent.left = new Node(key, value); }
    else { parent.right = new Node(key, value); }
}

Note: Insert happens only at the leaves!
Delete Operation (iterative)

delete(key, root){
    while (root != Null && key != root.key){
        if (key < root.key){ root = root.left; }
        else if (key > root.key){ root = root.right; }
    }
    if (root == Null){ return; }
    // Now root is the node to delete, what happens next?
}
Delete – 3 Cases

• 0 Children (i.e. it’s a leaf)

• 1 Child

• 2 Children
Finding the Max and Min

- **Max of a BST:**
  - Right-most Thing

- **Min of a BST:**
  - Left-most Thing

```java
maxNode(root){
    if (root == Null){ return Null; }
    while (root.right != Null){
        root = root.right;
    }    
    return root;
}

minNode(root){
    if (root == Null){ return Null; }
    while (root.left != Null){
        root = root.left;
    }    
    return root;
}
```
Delete Operation (iterative)

dele(key, root){
    while (root != Null && key != root.key){
        if (key < root.key){ root = root.left; }
        else if (key > root.key){ root = root.right; }
    }
    if (root == Null){ return; }
    if (root has no children){
        make parent point to Null Instead;
    }
    if (root has one child){
        make parent point to that child instead;
    }
    if (root has two children){
        make parent point to either the max from the left or min from the right
    }
}
Worst Case Analysis

• For each of Find, insert, Delete:
  • Worst case running time matches height of the tree
• What is the maximum height of a BST with \( n \) nodes?
Improving the worst case

• How can we get a better worst case running time?
“Balanced” Binary Search Trees

• We get better running times by having “shorter” trees
• Trees get tall due to them being “sparse” (many one-child nodes)
• Idea: modify how we insert/delete to keep the tree more “full”
Idea 1: Both Subtrees of Root have same # Nodes
Idea 2: Both Subtrees of Root have same height
Idea 3: Both Subtrees of every Node have same # Nodes
Idea 4: Both Subtrees of every Node have same height
Teaser: AVL Tree

• A Binary Search tree that maintains that the left and right subtrees of every node have heights that differ by at most one.
  • Not too weak (ensures trees are short)
  • Not too strong (works for any number of nodes)