CSE 373: Data Structures and Algorithms

Lecture 10: Trees II

Implementing Set with BST

- Each Set entry adds a node to tree
  - Node contains String element, references to left/right subtree
- Tree organized for binary search
  - Quickly search or place to insert/remove element

Implementing Set with BST (cont.)

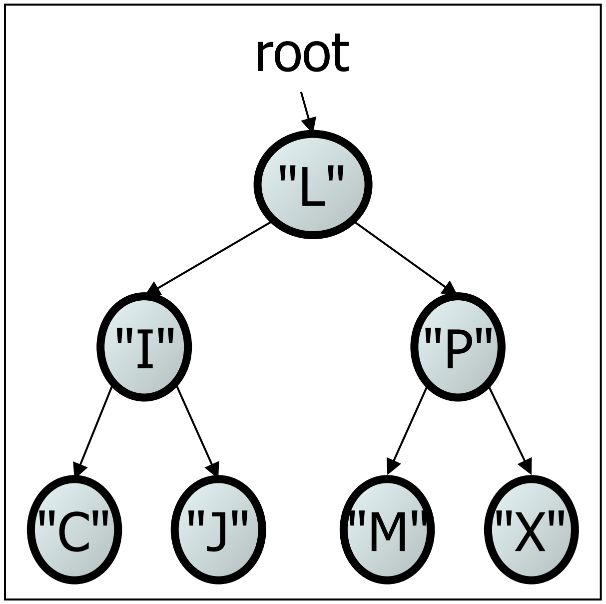
```
public interface StringSet {
    public boolean add(String value);

    public boolean contains(String value);

    public void print();

    public boolean remove(String value);

    public int size();
}
```

`StringTreeSet` class

```
// A StringTreeSet represents a Set of Strings.
public class StringTreeSet {
    private StringTreeNode root;   // null for an empty set

    methods
}
```

- Client code talks to the `StringTreeSet`, not to the node objects inside it

- Methods of the `StringTreeSet` create and manipulate the nodes, their data and links between them



Set implementation: contains (search)

```
public boolean contains(String value) {
    return contains(root, value);
}

private boolean contains(StringTreeNode node, String value) {
    if (node == null) {
        return false;                              // not in set
    } else if (node.data.compareTo(value) == 0) {
        return true;                               // found!
    } else if (node.data.compareTo(value) > 0) {
        return contains(node.left, value);         // search left
    } else {
        return contains(node.right, value);        // search right
    }
}
```

Set implementation: insert

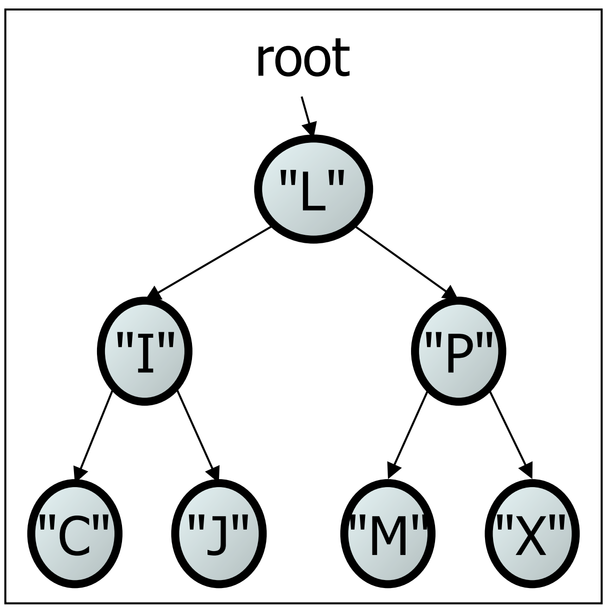
- Starts like `contains`
  - Trace out path where node should be
- Add node as new leaf
  - Don't change any other nodes or references
  - Correct place to maintain binary search tree property

Set implementation: insert

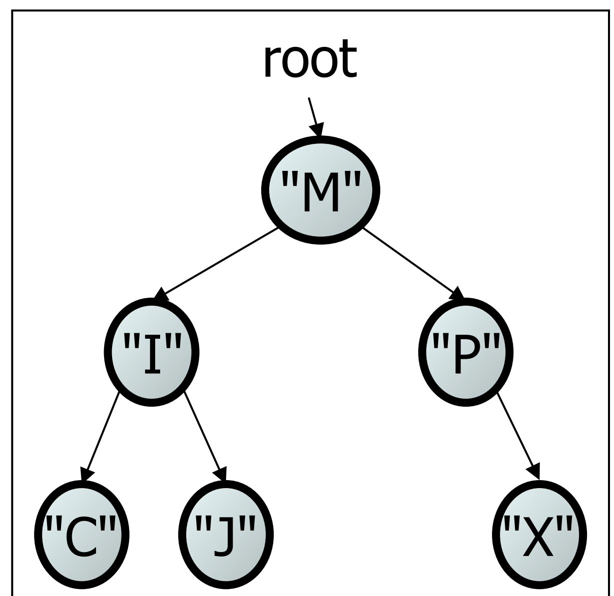
```
public boolean add(String value) {
    int oldSize = size();
    this.root = add(root, value);
    return oldSize != size();
}

private StringTreeNode add(StringTreeNode node, String value) {
    if (node == null) {
        node = new StringTreeNode(value);
        numElements++;
    } else if (node.data.compareTo(value) == 0) {
      return node;
    } else if (node.data.compareTo(value) > 0) {
        node.left = add(node.left, value);
    } else { node.right = add(node.right, value); }
    return node;
}
```

Set implementation: remove

- Possible states for the node to be removed:
  - a leaf: replace with null
  - a node with a left child only: replace with left child
  - a node with a right child only: replace with right child
  - a node with both children: replace with min value from right




```
set.remove("L");
```



Set implementation: remove

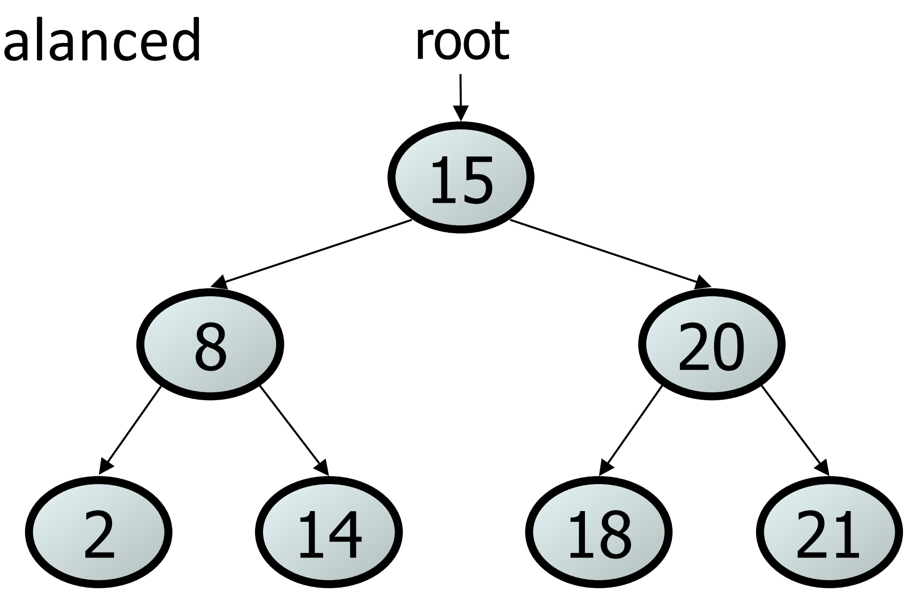
```
public boolean remove(String value) {
    int oldSize = numElements;
    root = remove(root, value);
    return oldSize > numElements;
}
protected StreeNode remove(StreeNode node, String value) {
    if (node == null) { return node;
    } else if (node.data.compareTo(value) < 0) { node.right = remove(node.right, value);
    } else if (node.data.compareTo(value) > 0) { node.left = remove(node.left, value);
    } else {
        if (node.right != null && node.left != null) {
            node.data = getMinValue(node.right);
            node.right = remove(node.right, node.data);
        } else if (node.right != null) {
            node = node.right;
            numElements--;
        } else {
            node = node.left;
            numElements--;
        }
    }
    return node;
}
```

Evaluate Set as BST

- Space used
  - Overhead of two references per entry
  - BST adds nodes as needed; no excess capacity
- Runtime
  - `add`, `contains` take time proportional to tree height
  - height expected to be O(log N)

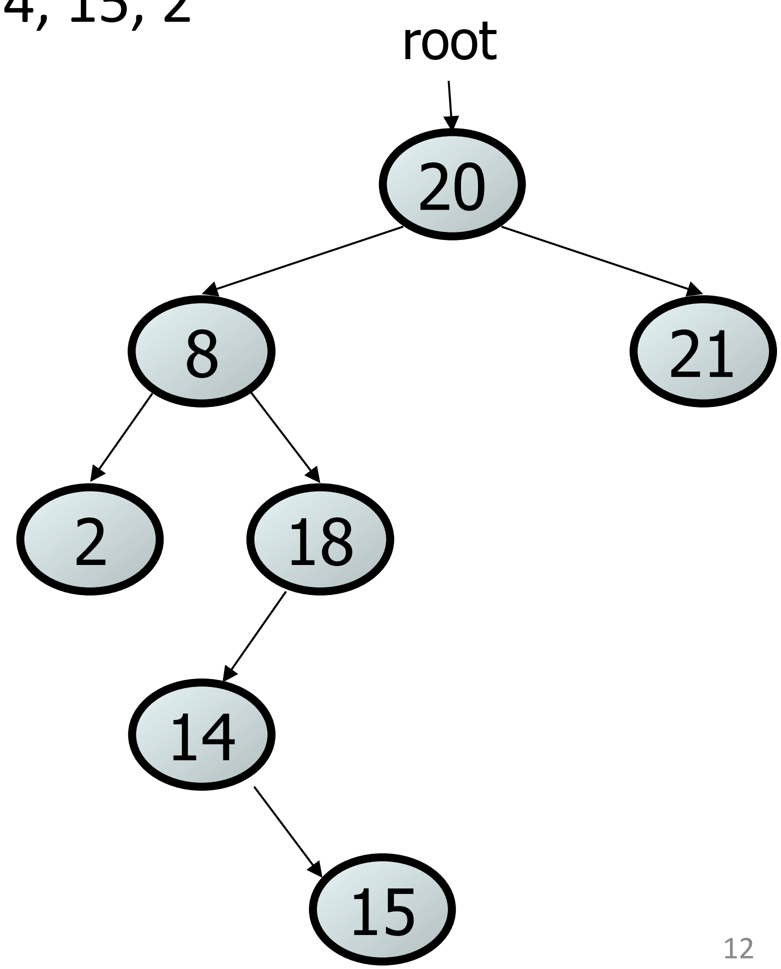
A Balanced Tree

- Values: 2 8 14 15 18 20 21
  - Order added: 15, 8, 2, 20, 21, 14, 18
- Different tree structures possible
  - Depends on order inserted
- 7 nodes, expected height log 7 ≈ 3
- Perfectly balanced



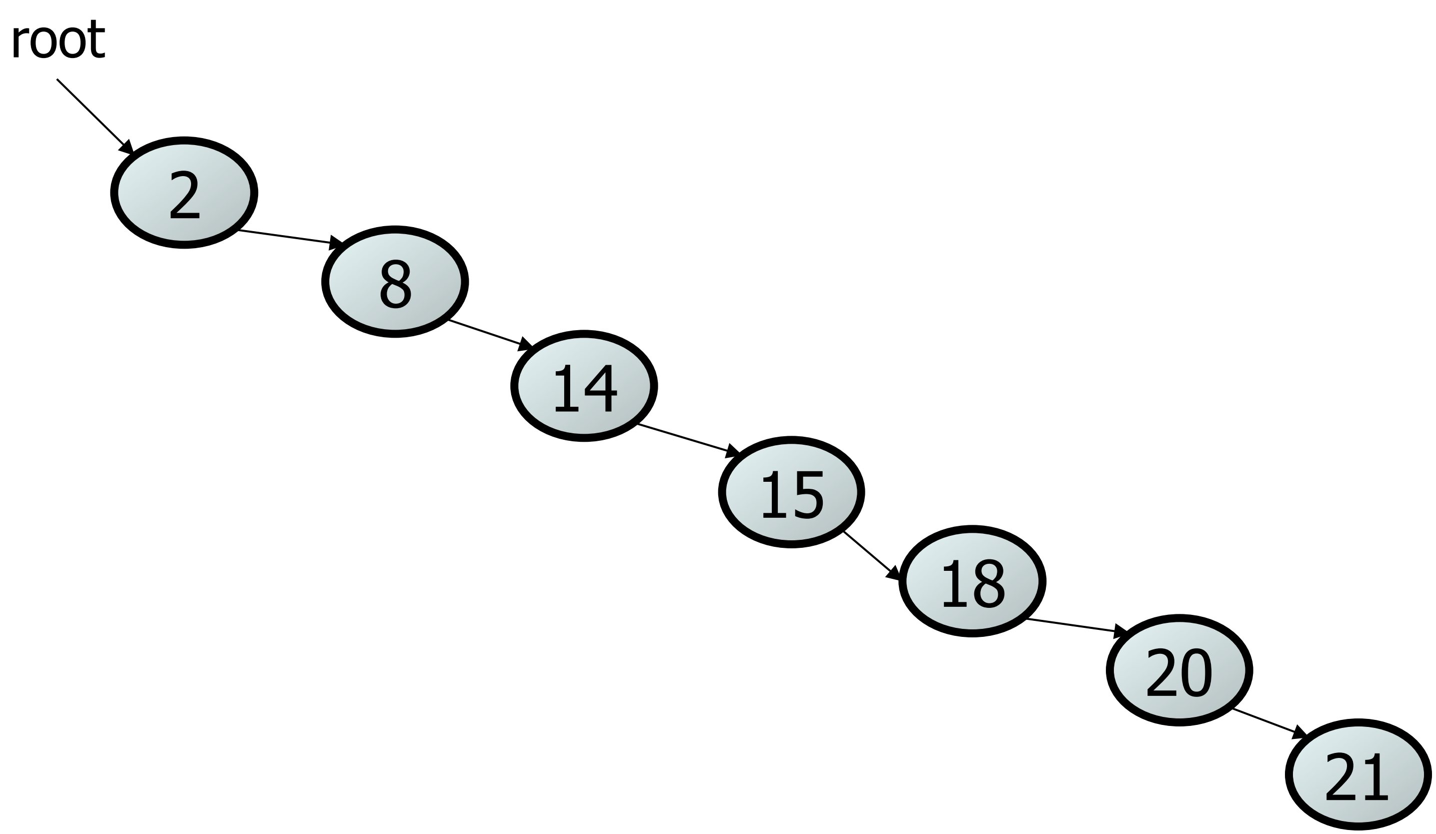
Mostly Balanced Tree

- Same Values: 2 8 14 15 18 20 21
  - Order added: 20, 8, 21, 18, 14, 15, 2
- Mostly balanced, height 4/5



Degenerate Tree

- Same Values: 2 8 14 15 18 20 21
  - Order added: 2, 8, 14, 15, 18, 20, 21
- Totally unbalanced, height 7



Binary Trees: Some Numbers

Recall: height of a tree = length of longest path from the root to a leaf.

For binary tree of height *h*:

- max # of leaves: 2^h
- max # of nodes: $2^{(h+1)} - 1$
- min # of leaves: 1
- min # of nodes: $h + 1$

We're not going to do better than log(n) height, and we need something to keep us away from n.

Implementing Set ADT (Revisited)

	Insert	Remove	Search
Unsorted array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted array	$\Theta(\log(n)+n)$	$\Theta(\log(n) + n)$	$\Theta(\log(n))$
Linked list	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
BST (if balanced)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

AVL Tree Motivation

Observation: the shallower the BST the better

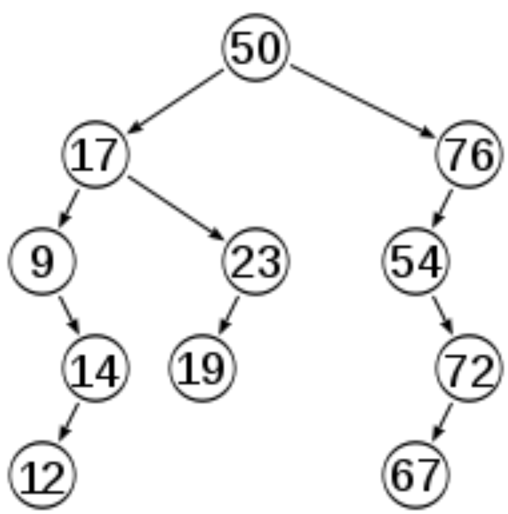
- For a BST with n nodes
  - Average case height is $\Theta(\log n)$
  - Worst case height is $\Theta(n)$
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario: height $\Theta(n)$

Strategy: Don't let the tree get lopsided

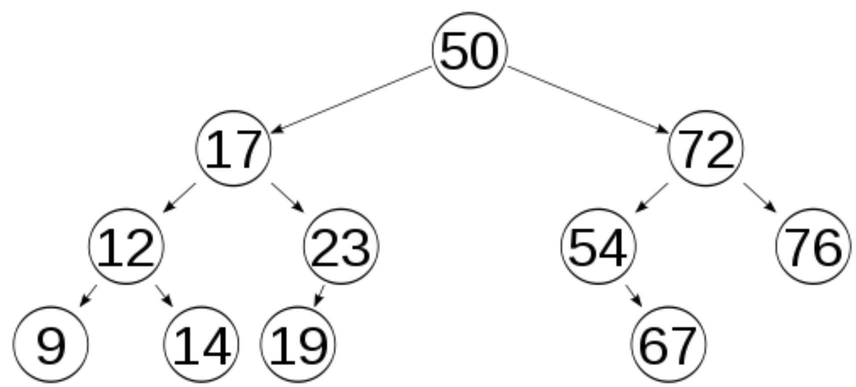
- Constantly monitor balance for each subtree
- Rebalance subtree before going too far astray

Balanced Tree

- **Balanced Tree**: a tree in which heights of subtrees are approximately equal



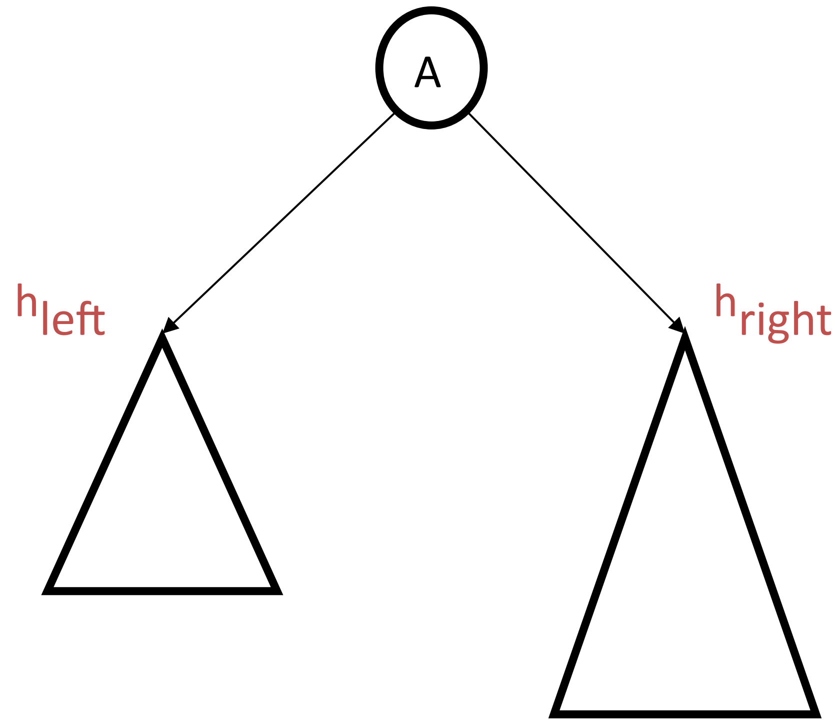
unbalanced tree



balanced tree

Tree height calculation

- Height is max number of edges from root to leaf
  - height(null) = -1
  - height(1) = 0
  - height(A)?
    - Hint: it's recursive!



Tree balance and height

(a) The balanced tree has a height of: ___________

(b) The unbalanced tree has a height of:___________