CSE 373

Binary search trees; tree height and balance
read: Weiss Ch. 4, section 4.1 - 4.3

slides created by Marty Stepp
http://www.cs.washington.edu/373/

© University of Washington, all rights reserved.
Trees

- **tree**: A directed, acyclic structure of linked nodes.
  - *directed*: Has one-way links between nodes.
  - *acyclic*: No path wraps back around to the same node twice.

- **binary tree**: One where each node has at most two children.

**Recursive definition**: A tree is either:
- empty (*null*), or
- a *root* node that contains:
  - *data*,
  - a *left* subtree, and
  - a *right* subtree.
  
  -(The left and/or right subtree could be empty.)
Tree terminology

- **node**: an object containing a data value and left/right children
  - **root**: topmost node of a tree
  - **leaf**: a node that has no children
  - **branch**: any internal node; neither the root nor a leaf
  - **parent**: a node that refers to this one
  - **child**: a node that this node refers to
  - **sibling**: a node with a common

- **subtree**: the smaller tree of nodes on the left or right of the current node

- **height**: length of the longest path from the root to any node

- **level** or **depth**: length of the path from a root to a given node

```
1
  /  \
 /    \
2     3
   /  \
  /    \
4     5
   /  \
5     6
```

height = 3
Binary search trees

- **binary search tree ("BST"):** a binary tree where each non-empty node $R$ has the following properties:
  - every element of $R$'s left subtree contains data "less than" $R$'s data,
  - every element of $R$'s right subtree contains data "greater than" $R$'s,
  - $R$'s left and right subtrees are also binary search trees.

- BSTs store their elements in sorted order, which is helpful for searching/sorting tasks.
BST examples

• Which of the trees shown are legal binary search trees?
public class TreeSet<E extends Comparable<E>> implements Set<E> {
    private TreeNode root; // null for an empty tree

    public TreeSet() {
        root = null;
    }

    ...

    private class TreeNode {
        private E data;
        private TreeNode left;
        private TreeNode right;
        ...
    }
}
Searching a BST

• Describe an algorithm for searching a binary search tree.
  ▪ Try searching for the value 31, then 6.

• What is the maximum number of nodes you would need to examine to perform any search?

```
    18
   /   \
  12   35
 /     /  \
4     22  58
|     /   /  \
-2   15  31  40
    /   /   /  \
   7   13  16  87
```

Template for tree methods

```java
public type name(parameters) {
    name(root, parameters);
}

private type name(TreeNode node, parameters) {
    ...
}
```

- Tree methods are often implemented recursively
  - with a public/private pair
  - the private version accepts the root node to process
The contains method

// Returns whether this BST contains the given integer.
public boolean contains(E value) {
    return contains(root, value);
}

private boolean contains(TreeNode node, E value) {
    if (node == null) {
        return false;  // base case: not found here
    } else {
        int comp = node.data.compareTo(value);
        if (comp == 0) {
            return true;  // base case: found here
        } else if (comp > 0) {
            return contains(node.left, value);
        } else {  // comp < 0
            return contains(node.right, value);
        }
    }
}

Adding to a BST

- Suppose we want to add new values to the BST below.
  - Where should the value 14 be added?
  - Where should 3 be added? 7?
  - If the tree is empty, where should a new value be added?

- What is the general algorithm?
Adding exercise

- Draw what a binary search tree would look like if the following values were added to an initially empty tree in this order:
The \( x = \text{change}(x) \) pattern

- Methods that modify a tree should have the following pattern:
  - input (parameter): old state of the node
  - output (return): new state of the node

- In order to actually change the tree, you must reassign:

```java
node = change(node, parameters);
node.left = change(node.left, parameters);
node.right = change(node.right, parameters);
overallRoot = change(overallRoot, parameters);
```
The add method

// Adds the given value to this BST in sorted order.
public void add(E value) {
    root = add(root, value);
}

private TreeNode add(TreeNode node, E value) {
    if (node == null) {
        node = new TreeNode(value);
    } else {
        int comp = node.data.compareTo(value);
        if (comp > 0) {
            node.left = add(node.left, value);
        } else if (comp < 0) {
            node.right = add(node.right, value);
        } // else a duplicate; do nothing
    }

    return node;
}
Removing from a BST

• How can we remove a value from a BST in such a way as to maintain proper BST ordering?

- `tree.remove(73);`
- `tree.remove(29);`
- `tree.remove(87);`
- `tree.remove(55);`
Cases for removal 1

1. a leaf:
2. a node with a left child only:
3. a node with a right child only:

replace with null
replace with left child
replace with right child

tree.remove(-3);
tree.remove(29);
tree.remove(55);
tree.remove(29);
tree.remove(42);
Cases for removal 2

4. a node with both children: replace with \textbf{min from right}
   \begin{itemize}
   \item (replacing with \textbf{max from left} would also work)
   \end{itemize}

\begin{verbatim}
    tree.remove(55);
\end{verbatim}
The remove method

// Removes the given value from this BST, if it exists.
public void remove(E value) {
    root = remove(root, value);
}

private TreeNode remove(TreeNode node, E value) {
    if (node == null) {
        return null;
    } else {
        int comp = root.data.compareTo(value);
        if (comp > 0) {
            root.left = remove(root.left, value);
        } else if (comp < 0) {
            root.right = remove(root.right, value);
        } else {
            // comp == 0; remove this node
            if (root.right == null) {
                return root.left;  // replace w/ L
            } else if (root.left == null) {
                return root.right;  // replace w/ R
            } else {
                // both children; replace w/ min from R
                root.data = getMin(root.right);
                root.right = remove(root.right, root.data);
            }
        }
    }
    return root;
}
Searching BSTs

- The BSTs below contain the same elements.
  - What orders are "better" for searching?
**Trees and balance**

- **balanced tree**: One whose subtrees differ in height by at most 1 and are themselves balanced.
  - A balanced tree of \(N\) nodes has a height of \(\sim \log_2 N\).
  - A very unbalanced tree can have a height close to \(N\).

- The runtime of adding to / searching a BST is closely related to height.
- Some tree collections (e.g. `TreeSet`) contain code to balance themselves as new nodes are added.

![Balanced Tree Diagram]
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 $\log_7 \approx 3$
- Perfectly balanced
Mostly balanced tree

- Same Values: 2, 8, 14, 15, 18, 20, 21
  - Order added: 20, 8, 21, 18, 14, 15, 2
- Somewhat balanced; height 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
Some height numbers

- **Observation:** The shallower the BST the better.
  - Average case height is $O(\log N)$
  - Worst case height is $O(N)$
  - Simple cases such as adding $(1, 2, 3, ..., N)$, or the opposite order, lead to the worst case scenario: height $O(N)$.

- For binary tree of height $h$:
  - max # of leaves: $2^{h-1}$
  - max # of nodes: $2^h - 1$
  - min # of leaves: 1
  - min # of nodes: $h$
Calculating tree height

- Height is max number of nodes in path from root to any leaf.
  - height(null) = 0
  - height(a leaf) = ?
  - height(A) = ?

  *Hint:* it's recursive!

- height(a leaf) = 1
- height(A) = 1 + max(height(A.left), height(A.right))