CSCI 136
Data Structures &
Advanced Programming

Lecture 24
Fall 2017
Instructor: Bills
Administrative Details

- Lab 9 today!
- You can work with a partner
- Bring a design to lab
- You can deviate from our plan but you should try to take advantage of
  - Abstract base classes/inheritance
  - Data structures you’ve learned
Last Time

- Heapsort
- Skew Heaps
- Binary search trees (Ch 14)
  - Overview
  - Definition
  - Some Applications
Today’s Outline

• Binary search trees (Ch 14)
  • The locate method
  • Further Implementation
  • Tree balancing to maintain small height
  • Partial taxonomy of balanced tree species
Binary Search Trees

• Binary search trees maintain a total ordering among elements (assumes comparability)

• Definition: A BST $T$ is either:
  • Empty
  • Has root $r$ with subtrees $T_L$ and $T_R$ such that
    • All nodes in $T_L$ have smaller value than $r$
    • All nodes in $T_R$ have larger value than $r$
    • $T_L$ and $T_R$ are also BSTs
BST Observations

• The same data can be represented by many BST shapes
• Searching for a value in a BST takes time proportional to the height of the tree
  • Reminder: trees have height, nodes have depth
• Additions to a BST happen at nodes missing at least one child (*a constraint!*)
• Removing from a BST can involve *any* node
BST Operations

• BSTs will implement the OrderedStructure Interface
  • add(E item)
  • contains(E item)
  • get(E item)
  • remove(E item)
  • Runtime of above operations?
    • All $O(h)$ – where $h$ is the tree height
  • iterator()
    • This will provide an in-order traversal
The BST holds the following items:

- BinaryTree root: the root of the tree
- BinaryTree EMPTY: a static empty BinaryTree
  - To use for all empty nodes of tree
- int count: the number of nodes in the BST
- Comparator<E> ordering: for comparing nodes
  - Note: E must implement Comparable

Two constructors: One takes a Comparator
BST Implementation: locate

- Several methods search the tree
  - add, remove, contains
- We factor out common code: locate method
- *protected* locate(BinaryTree<E> node, E v)
  - Returns a BinaryTree<E> n in the subtree with root node such that either
    - n has its value equal to v, or
    - v is not in this subtree and n is the node whose child v should be
- How would we implement locate()?
BST Implementation: locate

```java
BinaryTree locate(BinaryTree root, E value) {
    if (root's value equals value) return root;
    child ← child of root that should hold value;
    if (child is empty tree, return root;
        // value not in subtree based at root;
    else // keep looking
        return locate(child, value);
}
```
BST Implementation: locate

- What about this line?

\[ \text{child} \leftarrow \text{child of root that should hold value} \]

- If the tree can have multiple nodes with same value, then we need to be careful

- Convention: During add operation, only move to right subtree if value to be added is greater than value at node

- We’ll look at add later

- Let’s look at locate now....
protected BinaryTree<E> locate(BinaryTree<E> root, E value) {
    E rootValue = root.value();
    BinaryTree<E> child;

    // found at root: done
    if (rootValue.equals(value)) return root;

    // look left if less-than, right if greater-than
    if (ordering.compare(rootValue, value) < 0)
        child = root.right();
    else
        child = root.left();

    // no child there: not in tree, return this node, else keep searching
    if (child.isEmpty()) return root;
    else
        return locate(child, value);
}
Other core BST methods

• locate(v) returns either a node containing v or a node where v can be added as a child
• locate() is used by
  • public boolean contains(E value)
  • public E get(E value)
  • public void add(E value)
  • Public void remove(E value)

• Some of these also use another utility method
  • protected BT predecessor(BT root)

• Let’s look at contains() first...
public boolean contains(E value) {
    if (root.isEmpty()) return false;

    BinaryTree<E> possibleLocation = locate(root, value);

    return value.equals(possibleLocation.value());
}
public void add(E value) {
    BinaryTree<E> newNode = new BinaryTree<E>(value, EMPTY, EMPTY);
    if (root.isEmpty()) root = newNode;
    else {
        BinaryTree<E> insertLocation = locate(root, value);
        E nodeValue = insertLocation.value();
        if (ordering.compare(nodeValue, value) < 0)
            insertLocation.setRight(newNode);
        else
            insertLocation.setLeft(newNode);
    }
    count++;
}
Add: Repeated Nodes

Where would a new K be added?  
A new V?
Add Duplicate to Predecessor

• If insertLocation has a left child then
  • Find insertLocation’s predecessor
  • Add repeated node as right child of predecessor
  • Predecessor will be in insertLocation’s left sub-tree
    • Do you believe me?
Corrected Version: add(E value)

```java
BinaryTree<E> newNode = new BinaryTree<E>(value, EMPTY, EMPTY);
if (root.isEmpty()) root = newNode;
else {
    BinaryTree<E> insertLocation = locate(root, value);
    E nodeValue = insertLocation.value();
    if (ordering.compare(nodeValue, value) < 0)
        insertLocation.setRight(newNode);
    else
        if (insertLocation.left().isEmpty())
            insertLocation.setLeft(newNode);
        else
            // if value is in tree, we insert just before
            // predecessor(insertLocation).setRight(newNode);
}
count++;
```
How to Find Predecessor

Where would a new K be added? A new V?
protected BinaryTree<E> predecessor(BinaryTree<E> root) {
    Assert.pre(!root.isEmpty(), "Root has predecessor");
    Assert.pre(!root.left().isEmpty(), "Root has left child.");

    BinaryTree<E> result = root.left();

    while (!result.right().isEmpty())
        result = result.right();

    return result;
}
Removal

• Removing the root is a (not so) special case
• Let’s figure that out first
  • If we can remove the root, we can remove any element in a BST in the same way
    • Do you believe me?
• We need to implement:
  • public E remove(E item)
  • protected BT removeTop(BT top)
Case 1: No left binary tree

\[ x \rightarrow x.\text{right} \quad \rightarrow \quad \text{return} \quad x.\text{right} \]
Case 2: No right binary tree

```
x
x.left
```

return

```
x.left
```
Case 3: Left has no right subtree
Case 4: General Case (HARD!)

- Consider BST requirements:
  - Left subtree must be $\leq$ root
  - Right subtree must be $>\ root$
- Strategy: replace the root with the largest value that is less than or equal to it
  - $\text{predecessor(root)}$: rightmost left descendant
- This may require reattaching the predecessor’s left subtree!
Case 4: General Case (HARD!)

Replace root with predecessor(root), then patch up the remaining tree
Case 4: General Case (HARD!)

Replace root with predecessor(root), then patch up the remaining tree
RemoveTop(topNode)

Detach left and right sub-trees from root (i.e. topNode)
If either left or right is empty, return the other one
If left has no right child
   make right the right child of left then return left
Otherwise find largest node C in left
   // C is the right child of its own parent P
   // C is the predecessor of right (ignoring topNode)
Detach C from P; make C’s left child the right child of P
Make C new root with left and right as its sub-trees
But What About Height?

- Can we design a binary search tree that is always “shallow”?
- Yes! In many ways. Here’s one
- AVL trees
  - Named after its two inventors, G.M. Adelson-Velsky and E.M. Landis, who published a paper about AVL trees in 1962 called "An algorithm for the organization of information"
• The *balance factor* of a node is the height of its right subtree minus the height of its left subtree. A node with balance factor 1, 0, or -1 is considered *balanced*.

• A node with any other balance factor is considered *unbalanced* and requires rebalancing the tree.
Single Rotation

Unbalanced trees can be rotated to achieve balance.
Single Right Rotation
Double Rotation
AVL Tree Facts

• A tree that is AVL except at root, where root balance factor equals $\pm 2$ can be rebalanced with at most 2 rotations.

• `add(v)` requires at most $O(\log n)$ balance factor changes and one (single or double) rotation to restore AVL structure.

• `remove(v)` requires at most $O(\log n)$ balance factor changes and (single or double) rotations to restore AVL structure.
AVL Trees: One of Many

- There are many strategies for tree balancing to preserve $O(\log n)$ height, including:
  - AVL Trees: guaranteed $O(\log n)$ height
  - Red-black trees: guaranteed $O(\log n)$ height
  - B-trees (not binary): guaranteed $O(\log n)$ height
    - 2-3 trees, 2-3-4 trees, red-black 2-3-4 trees, ...
  - Splay trees: Amortized $O(\log n)$ time operations
  - Randomized trees: $O(\log n)$ expected height