CSCI 136
Data Structures & Advanced Programming

Lecture 26

Spring 2018

Profs Bill & Jon

Administrative Details

- Lab TA coverage
  - This weekend Emily is filling in Saturday 11-1pm and 5-7pm
  - Shanti volunteered to move her Friday hours to Saturday 5-7pm. Is that an improvement?
- Pre-registration info session:
  - Friday @2:30pm
- Sigma Xi talks this week:
  - ~~Thursday +~~ Friday at 4:15pm

Last Time

- Heaps
  - Implementation details
  - Top-down (percolateUp) vs. Bottom-up (pushDownRoot) heapify
  - HeapSort
  - Skew Heaps (details in book)

Today

- Revisiting Search
  - OrderedVector implementation
  - *Begin* Binary Search Trees

Search

- Some data structures we have discussed do not support searching:
  - Linear: Queues and Stacks
  - PriorityQueue: Heaps
- How fast can we search (`get(E value)`) in:

Array/Vector	`O(n)`
Linked List	`O(n)`
OrderedVector	$O(\log_2 n)$

OderedVector

- What data structure did people use in Lab 8?
  - Did anyone use OrderedVector?

- Does not actually implement `get(E obj)`!?!
  - `ForGETtableOrderedVector.java`

Improving on OrderedVector

- The OrderedVector class allows `O(`$\log_2 n$`)` time searching over `n` comparable objects
  - add() and remove(), though, take `O(n)` time in the worst case (and on average)

- Goal: improve update times without sacrificing the `O(`$\log_2 n$`)` search time

Binary Trees and Orders

- Binary trees suggest multiple orderings of their elements (pre-/in-/post-/level-orders)
- In particular, in-order traversal suggests a natural way to hold comparable items
  - For each node **`v`** in tree
    - All values in left subtree of **`v`** are ≤ **`v`**
    - All values in right subtree of **`v`** are ≥ **`v`**
- This leads us to...

Binary Search Trees

- Binary search trees maintain a *total* ordering among elements
- Definition: A BST is either:
  - Empty
  - A tree where (1) root value is greater than or equal to all values in left subtree, and less than or equal to all values in right subtree; (2) left and right subtrees are also BSTs
- Examples:

  data = { 3, 9, 2, 4, 5, 5, 0, 6 }

BST Observations

- The same data can be represented by many BST shapes
- Observations:
  - Searching for a value in a BST takes time proportional to the height of the tree
  - We want a tree to be as shallow as possible
    - A full or complete tree has height $\log_2 n$

(Really) Consider Binary Search

- In one sentence, what is the “essence” of the binary search strategy?
  - Rule out half of the list with each comparison

- arr = { -9, 17, 45, 46, 70, 101, 136 }

  - What order do we compare arr’s elements?
  - How can we build a BST that mimics that search order?

Taking a Step Back

- Balance is critical to performance, but we will initially ignore balance as we work through the BST operations
- We will discuss (briefly?) methods to keep our trees balanced at the end of the unit
    - Like everything, details are in the book!

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`

BST Implementation

- 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
  - What about the constructor that doesn't?

BST Implementation: locate

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

BST Implementation: locate

```
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 root's subtree
    else
        return locate(child, value) //keep looking
```

BST Implementation: locate

- What about this line?

  child ← 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....

The code : locate

```
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(val)` returns either a node containing `v` or a node where `v` can be added as a child
- `locate(val)` 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...

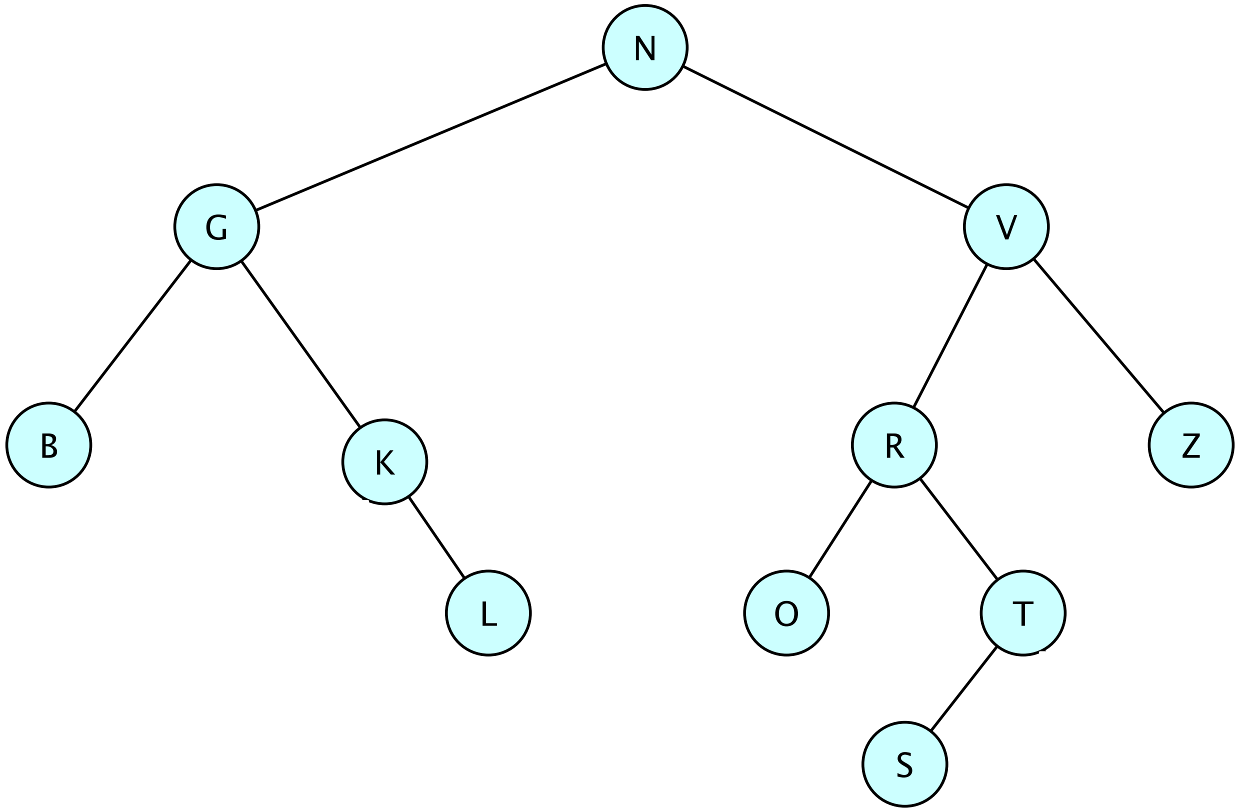
Contains

```
public boolean contains(E value){
   if (root.isEmpty()) return false;

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

   return value.equals(possibleLocation.value());
}
```

Binary Search Tree Add

- Remember!!! A binary tree is a binary *search* tree if it is:
  - Empty, or
  - All nodes in the left subtree are less than or equal to the root, all nodes in the right subtree are greater than or equal to the root, and the left and right subtrees are binary search trees.
- In our implementation, right subtrees only hold values that are *strictly greater than* the root
  - Why?

Add: Repeated Nodes



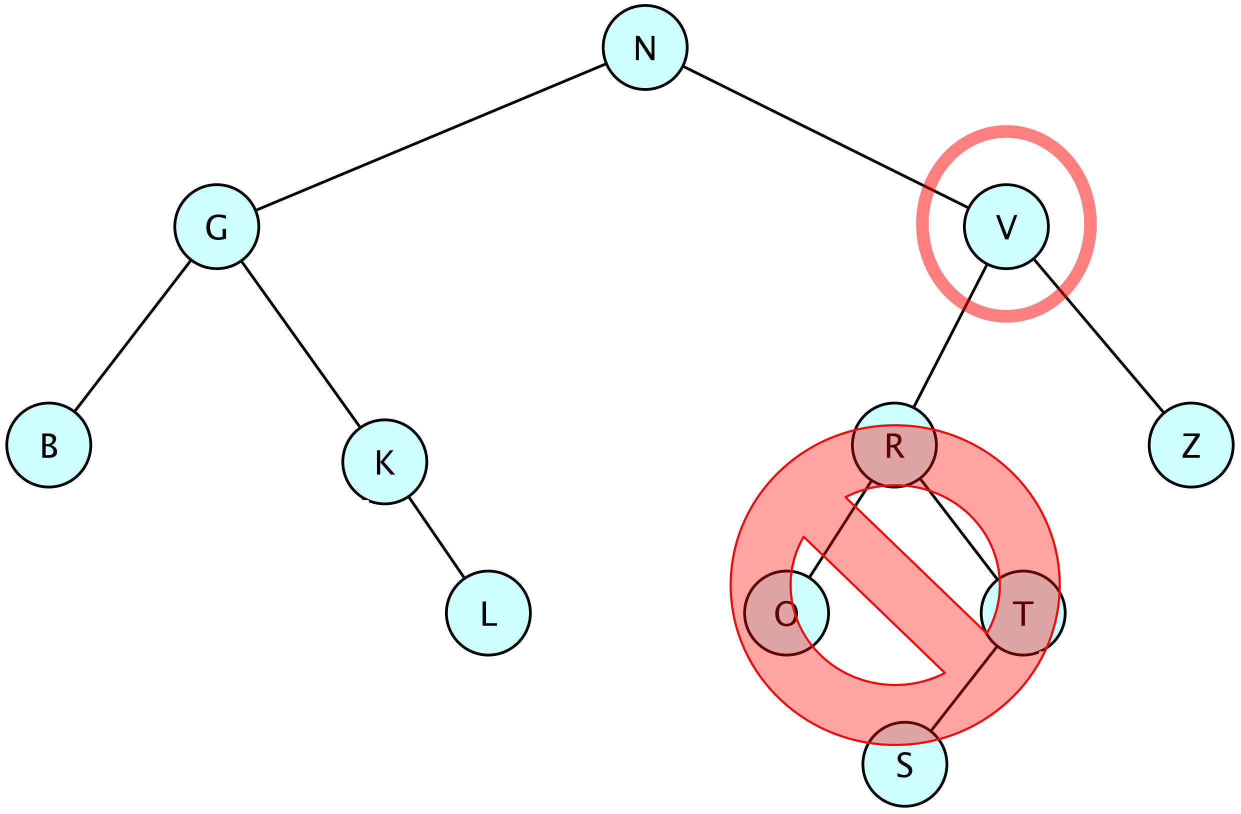
Where would a new K be added?
A new V?

First (Bad) Attempt: add(E 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(value,nodeValue) > 0)
            insertLocation.setRight(newNode); // value > nodeValue
        else
            insertLocation.setLeft(newNode); // value <= nodeValue
    }
    count++;
}
```

Problem: If duplicate values are allowed in the BST, the left subtree might not be empty when setLeft is called

How to Add Duplicate Values



How to perform: bst.add("v") ???

locate("v").setLeft(new BinaryTree ("v")); ???