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 suggestions but you should try to take advantage of
  - Abstract base classes/inheritance
  - Data structures you've learned

2

Last Time

- Finished array-based heaps
- Some heapsort observations
- Skew heaps

Today's Outline

- Binary search trees (Ch 14)
  - Overview
  - Definition
  - Some Applications
  - The *locate* method
  - Further Implementation
  - Tree balancing to maintain small height
  - Partial taxonomy of balanced tree species

Search

- Some data structures we have discussed do not support searching:
  - Queue, Stack, PriorityQueue, Heap
- How fast can we search (`get(E value)`) in:
  - Array/Vector
    - O(n)
  - Linked List
    - O(n)
  - OrderedVector
    - O(log n)

Improving on OrderedVector

- The OrderedVector class provides O(log n) time searching for a group of 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 n) search time

Binary Trees and Orders

- Binary trees impose multiple orderings on 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 root value is greater than or equal to all values in left subtree, and less than or equal to all values in right subtree; 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
  - Additions to a BST happen at nodes missing at least one child
  - 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`

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

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 whose root is *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

```
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?

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

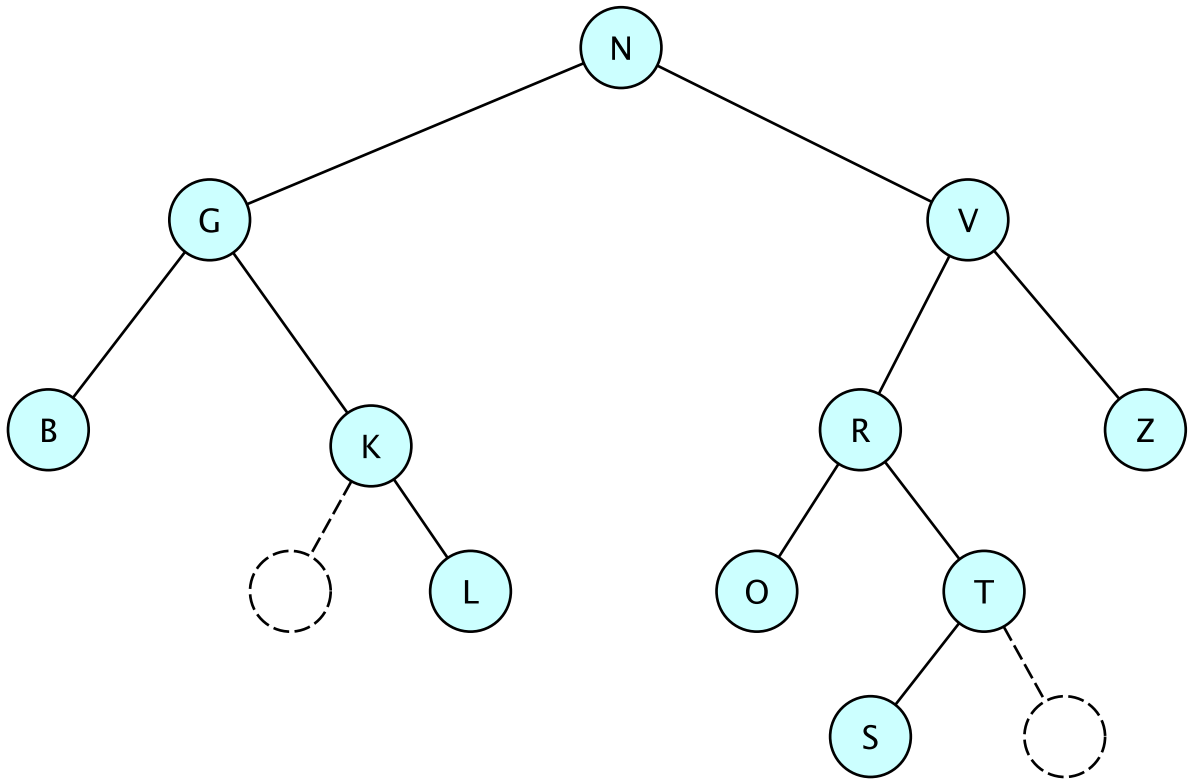
```
public boolean contains(E value){
   if (root.isEmpty()) return false;

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

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

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(nodeValue,value) < 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

Add: Repeated Nodes



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