Costliness of \textit{contains}

- Review: in a binary tree, \textit{contains} is $O(N)$
- \textit{contains} may be a frequent operation in an application
- Can we do better than $O(N)$?
- Turn to list searching for inspiration:
  - Why was binary search so much better than linear search?
  - Can we apply the same idea to trees?

\textbf{Binary Search Trees}

- Idea: order the nodes in the tree so that, given that a node contains a value $v$,
  - All nodes in its left subtree contain values $\leq v$
  - All nodes in its right subtree contain values $\geq v$
- A binary tree with these properties is called a \textit{binary search tree} (BST)

\textbf{Examples(?)}

- Are these are binary search trees? Why or why not?

\textbf{Implementing a Set with a BST}

- Can exploit properties of BSTs to have fast, divide and conquer implementations of Set’s add and contains operations
- TreeSet!
- A TreeSet can be represented by a pointer to the root node of a binary search tree, or null if no elements yet
  ```java
  public class SimpleTreeSet implements Set {
    private BTNode root; // root node, or null if none
  }
  ```

\textbf{contains for a BST}

- Original contains had to search both subtrees
  - Like linear search
- With BSTs, can only search one subtree!
  - All small elements to the left, all large elements to the right
  - Search either left or right subtree, based on comparison between elem and value at root of tree
  - Like binary search
Code for `contains` (in TreeSet)

```java
/** Return whether elem is in set */
public boolean contains(Object elem) {
    return subtreeContains(root, (Comparable)elem);
}
// Return whether elem is in (sub-)tree with root n
private boolean subtreeContains(BTNode n, Comparable elem) {
    if (n == null) {
        return false;
    } else {
        int comp = elem.compareTo(n.item);
        if (comp == 0) { return true; } // found it!
        else if (comp < 0) { return subtreeContains(n.left, elem); } // search left
        else { return subtreeContains(n.right, elem); } // search right
    }
}
```

Examples

```
contains(6)  contains(10)
```

Cost of BST `contains`

- Work done at each node:
- Number of nodes visited (depth of recursion):
- Total cost:

```
Cost of BST contains
• Work done at each node:
• Number of nodes visited (depth of recursion):
• Total cost:
```

Example

```
• Add 8, 10, 5, 1, 7, 11 to an initially empty BST, in that order:
```

Add

- Must preserve BST invariant: insert new element in correct place in BST
- Two base cases
  - Tree is empty: create new node which becomes the root of the tree
  - If node contains the value, found it; suppress duplicate add
- Recursive case
  - Compare value to current node’s value
  - If value < current node’s value, add to left subtree recursively
  - Otherwise, add to right subtree recursively

Example (2)

- What if we change the order in which the numbers are added?
- Add 1, 5, 7, 8, 10, 11 to a BST, in that order (following the algorithm):
**Code for add (in TreeSet)**

```java
/** Ensure that elem is in the set. Return true if elem was added, false otherwise. */
public boolean add(Object elem) {
    try {
        BTNode newRoot = addToSubtree(root, (Comparable)elem); // add elem to tree
        root = newRoot; // update root to point to new root node
        return true; // return true (tree changed)
    } catch (DuplicateAdded e) { // detected a duplicate addition
        return false; // return false (tree unchanged)
    }
}
```

**Code for addToSubtree**

```java
/** Add elem to tree rooted at n. Return (possibly new) tree containing elem, or throw DuplicateAdded if elem already was in tree */
private BTNode addToSubtree(BTNode n, Comparable elem) throws DuplicateAdded {
    if (n == null) { return new BTNode(elem, null, null); } // adding to empty tree
    int comp = elem.compareTo(n.item);
    if (comp == 0) { throw new DuplicateAdded(); } // elem already in tree
    if (comp < 0) { // add to left subtree
        BTNode newSubtree = addToSubtree(n.left, elem);
        n.left = newSubtree; // update left subtree
    } else /* comp > 0 */ { // add to right subtree
        BTNode newSubtree = addToSubtree(n.right, elem);
        n.right = newSubtree; // update right subtree
    }
    return n; // this tree has been modified to contain elem
}
```

**Cost of add**

- Cost at each node:
  - How many recursive calls?
    - Proportional to height of tree
  - Best case?
  - Worst case?

**A Challenge: iterator**

- How to return an iterator that traverses the sorted set in order?
  - Need to iterate through the items in the BST, from smallest to largest
  - Problem: how to keep track of position in tree where iteration is currently suspended
    - Need to be able to implement next(), which advances to the correct next node in the tree
    - Solution: keep track of a path from the root to the current node
      - Still some tricky code to find the correct next node in the tree

**Another Challenge: remove**

- Algorithm: find the node containing the element value being removed, and remove that node from the tree
  - Removing a leaf node is easy: replace with an empty tree
  - Removing a node with only one non-empty subtree is easy: replace with that subtree
  - How to remove a node that has two non-empty subtrees?
    - Need to pick a new element to be the new root node, and adjust at least one of the subtrees
    - E.g., remove the largest element of the left subtree (will be one of the easy cases described above), make that the new root

**Analysis of Binary Search Tree Operations**

- Cost of operations is proportional to height of tree
  - Best case: tree is balanced
    - Depth of all leaf nodes is roughly the same
    - Height of a balanced tree with $n$ nodes is $-\log_2 n$
  - If tree is unbalanced, height can be as bad as the number of nodes in the tree
    - Tree becomes just a linear list
Summary

• A binary search tree is a good general implementation of a set, if the elements can be ordered
  • Both contains and add benefit from divide-and-conquer strategy
  • No sliding needed for add
  • Good properties depend on the tree being roughly balanced

• Open issues (or, why take a data structures course?)
  • How are other operations implemented (e.g. iterator, remove)?
  • Can you keep the tree balanced as items are added and removed?