CSE 143 Notes 5/17/06

Binary Search Trees

Cost of *contains*

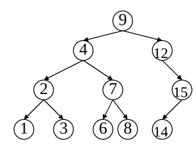
- Review: in a binary tree, *contains* is O(N)
- Can we do better than O(N)?
- Turn to previous experience for inspiration...
  - Why was binary search so much better than linear search?
  - Can we apply the same idea to trees?

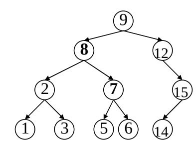
Binary Search Trees

- 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 < *v*
  - All nodes in its right subtree contain values > *v*
- A binary tree with these properties is called a *binary search tree* (BST)
- Notes:
  - Can also define a BST using >= and <= instead of >, <
    - This would allow duplicate values in the tree
  - In Java, if the values are not primitive types, they must implement comparable interface (i.e., provide compareTo)

Examples(?)

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





Implementing a Set with a BST

- Can exploit properties of BSTs to have fast, divide-and-conquer implementations of add and contains
  - e.g., a tree-based set – a collection of items
  - A tree set can be represented by a pointer to the root node of a binary search tree, or null if the set is empty

```
public class IntSet {
    private TreeNode root;          // root node, or null if empty
    public IntSet( ) { root = null; }
    // size() as for unordered binary tree
    ...
}
```


contains for a BST

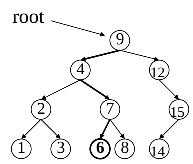
- For a general binary tree, contains had to search both subtrees
  - Like linear search
- With BSTs, need only to search one subtree
  - All small elements to the left, all large elements to the right
  - Search either left or right subtree, based on comparison between item and value at the root of the (sub-)tree
  - Like binary search

Code for *contains* (in IntSet)

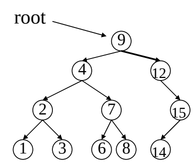
```
/** Return whether n is in this set */
public boolean contains(int n) {
    return contains(root, n);
}
// Return whether n is in (sub-)tree with root r
private boolean contains(TreeNode r, int n) {
    if (r == null) {
        return ___________________ ;
    } else {
        if (n == r.data) { return ___________________ ; }            // found it!
        else if (n < r.data)   { return __________________________ ; }  // search left
        else /* n > r.data */ { return __________________________ ; }  // 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:

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 (for sets; for other collections, can have a convention about how to allow for duplicate values)
- 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

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

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 IntSet)

```
/** Ensure that n is in the set. */
public void add(int n) {
    root = add(root, n);  // add n to tree if not present
}
/** Add n to tree rooted at r.  Return (possibly new) tree containing n. */
private TreeNode add(TreeNode r, int n) {




}
```


Code for *add*

```
/** Add n to tree rooted at r.  Return (possibly new) tree containing n. */
private TreeNode add(TreeNode r, int n) {
    if (r == null) {                        // adding to empty tree
        return new TreeNode(n, null, null);
    }
    if (n < r.data) {                       // add to left subtree
        r.left = add(r.left, n);
    } else if n > r.data) {                 // add to right subtree
        r.right = add(r.right, n);
    } // otherwise n == r.data, no change needed
    return r;     // return reference to this (possibly modified) tree to caller
}
```


Cost of *add*

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

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 *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
- Not covered (or, why take a data structures course?)
  - How are other operations implemented (e.g. iterator, remove)?
  - How do you keep the tree balanced as items are added and removed?