CSC 143

Binary Search Trees

Costliness of *contains*

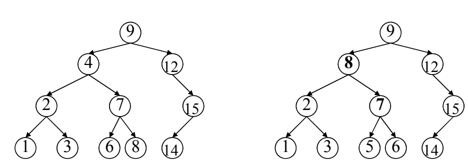
- Review: in a binary tree, *contains* is O(N)
- *contains* may be a frequent operation in an application
- Can we do better than O(N)?
- Turn to list searching for inspiration...
  - Why was binary search so much better than linear search?
  - Can we apply the same idea to trees?

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)

Examples

- Are these are binary search trees? Why or why not?
- Tree on the left: yes, tree on the right: no (since 7 < 8)



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 of no elements yet

```
public class SimpleTreeSet implements Set {
    private BTNode root;        // root node, or null if none
    public SimpleTreeSet( ) { this.root = null; }
    // size as for BinTree
    …
}
```

5

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

6

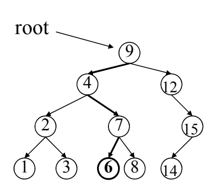
Code for *contains* (in TreeSet)

```
/** Return whether elem is in set */
/** Type E must implement Comparable */
public boolean contains(E elem) {
    return subtreeContains(root, elem);
}
// Return whether elem is in (sub-)tree with root n
private boolean subtreeContains(BTNode n, E 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 /* comp > 0 */ { return subtreeContains(n. right, elem); }  // search right
    }
}
```

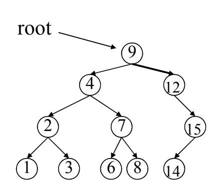
7

Examples

contains(6)



contains(10)



8

Cost of BST *contains*

- Work done at each node: O(1)
- Number of nodes visited (depth of recursion): O(log N) if the tree is balanced (= for any node, the difference in height between the left and right subtrees is at most 1).
  It could be O(N) is the tree looks like a linked list!
- Total cost: O(log N) with a balanced tree

9

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

10

Example

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

11

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):

12

Code for *add* (in TreeSet)

```
/** Ensure that elem is in the set.  Return true if elem was added, false otherwise. */
public boolean add(E elem) {
    try {
        BTNode newRoot = addToSubtree(root, 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)
    }
}
/** 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, E elem) throws DuplicateAdded {
    … }
```


Code for *addToSubtree*

```
/** 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, E 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: O(1) (or O(cost of compareTo))
- How many recursive calls?
  - Proportional to height of tree
  - Best case? O(log N) if adding a new element to a balanced tree
  - Worst case? O(N) if adding a new element to tree that looks like a linked list.

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 $\sim\log_2 n$
- 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?