

CSE 143

Binary Search Trees

3/8/2005

(c) 2001-5, University of Washington

21-1

Costliness of *contains*

- Review: in a binary tree, *contains* is $O(N)$ (worst case)
- *contains* may be a frequent operation in an application
- 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?

3/8/2005

(c) 2001-5, University of Washington

21-2

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 \geq and \leq instead of $>$, $<$
This implies there could be duplicate values in the tree
 - In Java, if the values are not primitive types, they must implement interface *comparable* (i.e., provide *compareTo*)

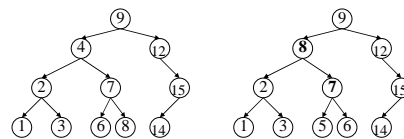
3/8/2005

(c) 2001-5, University of Washington

21-3

Examples(?)

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



3/8/2005

(c) 2001-5, University of Washington

21-4

Implementing a Set with a BST

- Can exploit properties of BSTs to have fast, divide-and-conquer implementations of *add* and *contains*
 - 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() { root = null; }
    // size as for BinTree
    ...
}
```

3/8/2005

(c) 2001-5, University of Washington

21-5

contains for a BST

- For a general binary tree, *contains* had to search both subtrees
 - Like linear search
- With BSTs, need to 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 item and value at the root of the (sub-)tree
 - Like binary search

3/8/2005

(c) 2001-5, University of Washington

21-6

Code for *contains* (in TreeSet)

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

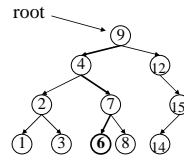
3/8/2005

(c) 2001-5, University of Washington

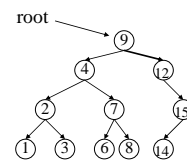
21-7

Examples

contains(6)



contains(10)



3/8/2005

(c) 2001-5, University of Washington

21-8

Cost of BST *contains*

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

3/8/2005

(c) 2001-5, University of Washington

21-9

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

3/8/2005

(c) 2001-5, University of Washington

21-10

Example

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

3/8/2005

(c) 2001-5, University of Washington

21-11

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):

3/8/2005

(c) 2001-5, University of Washington

21-12

Code for *add* (in TreeSet)

```
/** Ensure that item is in the set. */
public void add(Object item) {
    root = addToSubtree(root, (Comparable) item); // add item to tree
}
/** Add item to tree rooted at r. Return (possibly new) tree containing item. */
private BTNode addToSubtree(BTNode r, Comparable item) {
    ...
}
```

3/8/2005

(c) 2001-5, University of Washington

21-13

Code for *addToSubtree*

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

3/8/2005

(c) 2001-5, University of Washington

21-14

Cost of *add*

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

3/8/2005

(c) 2001-5, University of Washington

21-15

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

3/8/2005

(c) 2001-5, University of Washington

21-16

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?

3/8/2005

(c) 2001-5, University of Washington

21-17