CSC 143 Java

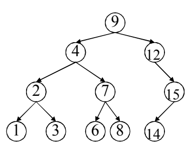
Binary Search Trees

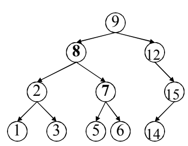
Binary Search Trees

- Issue: improve efficiency of BT contains() [O(N)]
  - Recall: binary search was an improvement over linear search because data was in order
- Solution: order the nodes in the tree so that, given a node with 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 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 Set's add and contains operations
  - TreeSet!

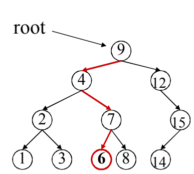
```
public class SimpleTreeSet<E> implements Set<E> {
    private BTNode<E> root;        // bound to root node, or null if empty
    public SimpleTreeSet( ) {
        root = null;
    }
    // rest of class definition
}
```


contains for a BST

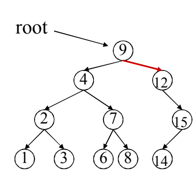
- For a general binary tree, contains had to search both subtrees
  - Like linear search, must visit every element
- With BSTs, need to only search one subtree
  - All small elements to the left, all large elements to the right
  - Search either left or right subtree, based on comparison between elem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

Examples

contains(6, root)



contains(10, root)



Code for *contains* (in TreeSet)

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


Cost of TreeSet *contains*

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

Exercise

- **Return the smallest element in a BST**
- Return null if tree is empty

```
private E smallest(BTNode<E> r) {

}
```


boolean add (Object o)

- 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 the duplicate add
- Recursive case
  - Compare value to current node's value
  - If value < current node 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:
- 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:

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) {
  int oldSize = size;
  root = addToSubtree(root, elem);    // add elem to tree
  return oldSize < size;
}
```

Code for *addToSubtree*

```
/** Add elem to tree rooted at r.  Return (possibly new) tree containing elem, and set
   treeChanged = true if the node was actually added */

private BTNode addToSubtree(BTNode<E> n, E elem) {
   if (n == null) {                        // adding to empty tree
      size++;
      return new BTNode(elem, null, null);
   }
   int comp = ((Comparable)elem).compareTo(n.item);
   if (comp == 0) { }                      // do nothing; elem already in tree
   else if (comp < 0)                      // add to left subtree
      n.left = addToSubtree(n.left, elem);
   else                                    // add to right subtree
      n.right = addToSubtree(n.right, elem);

   return n;      // this tree has been modified to contain elem
}
```


Cost of insert

- Cost at each node: O(1)
- How many recursive calls (i.e., total cost)?
  - Proportional to height of tree
  - Best case?
  - Worst case?

Analysis of Binary Search Tree

- Cost of operations is proportional to height of tree
- Height can be as bad as O(n) if tree is a long chain (list)
- 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 balanced
  - Cost of find, insert and other operations are O(log n)

Remove Algorithm

- First find the node (call it N) to delete.
  - Use find algorithm.
- If N is a leaf, just delete it.
- If N has just one child, have N's parent bypass N and connect to N's child. If N has two children:
  - Replace N's item with the *smallest item K of the right subtree*
    - We've seen this algorithm
  - (Recursively) delete the node that had K (this node is now useless)

Code for Remove

Use two mutually recursive methods:

- BTNode<E> **deleteItem**(BTNode<E> t, Object o);
  - find and remove the node containing o from the tree t
  - return a new tree that is the same as t, but with o removed
- BTNode<E> **removeNode**(BTNode<E> t);
  - remove this node
    precondition: t != NULL
  - returns the root node of the resulting tree

Finding the Node

```
private BTNode<E> deleteItem (BTNode<E> t, Object o)   {
    if ( t == null) return t;
   else {
         int comp = ((Comparable)o).compareTo(t.item);
         if (comp == 0)                 // found it
                   t = removeNode(t);
         else if (comp > 0)
                   t.right = deleteItem(t.right, o);
         else
                   t.left = deleteItem(t.left, o);
    }
    return t;
    }
}
```

Deleting the Node

```
private BTNode<E> removeNode (BTNode<E> t)  {
   if (t.left == null && t.right == null)
            return null;          // no children
   else if ( t.left == null )
            return t.right;
   else if ( t.right == null )
            return t.left;
   else {                         // 2 children case
            E me = smallest(t.right);
            t.item = me;
            t.right = deleteItem(t.right, me);
            // or a deleteLeftMost() method
   }
   return t;
}
```


Summary

- A binary search tree is a good general implementation of a Set. Also good for a Map.
  - Cost of contains and add are O(log n) assuming balanced tree
  - Good properties depend on the tree being balanced
- Open issues (or why take a data structures course?)
  - How do you keep the tree balanced as items are added and removed?
  - How about an Iterator?
  - Other structures (2 -3 trees, red-black trees, graphs)