# CMPSCI 187: Programming With Data Structures

Lecture #28: Binary Search Trees
21 November 2011

## Binary Search Trees

- The Idea of a Binary Search Tree
- The `BinarySearchTreeADT` Interface
- The `LinkedBinarySearchTree` Class
- Adding an Element
- Removing An Element
- Array Implementations of Binary Search Trees

## The Idea of a Binary Search Tree

- We want to be able to determine quickly (ideally in O(log n) time) whether a particular element is in our collection. Binary search did this for an array. But at the same time, we would like to be able to insert and delete elements without time-consuming shifts.

- In a **binary search tree**, every node has an element from an ordered base type. If a node n has element x, every element in the left subtree of n is less than x and every element in the right subtree is greater than x.

- Finding an element (or determining that it is not there) means following a path from the root to a leaf in the worst case. The time to do this is proportional to the depth of the tree. If the tree is balanced, this will be O(log n). In CMPSCI 311 (and a bit next lecture) we'll see ways to keep a tree balanced.

- But there is no reason that inserts and deletes should leave a balanced tree.

## The `BinarySearchTreeADT` Interface

- Once we know that our binary tree is going to store elements from an ordered type, and use the binary search tree rules, we can set up the interface.
- We don't restrict T here, but we will cast `T` elements into `Comparable<T>`. If T only implements `Comparable<? super T>`, this won't work.
- The balanced binary tree classes will implement this same interface.

```
public interface BinarySearchTreeADT<T>
                 extends BinaryTreeADT<T> {
   public void addElement (T element);
   public T removeElement (T targetElement);
   public void removeAllOccurrrences (T targetElement);
   public T removeMin( );
   public T removeMax( );
   public T findMin( );
   public T findMax( );
```

## The `LinkedBinarySearchTree` Class

- We will inherit from the `LinkedBinaryTree` class, so that we have fields `count` and `root`, two constructors that we may use without changes, and the four iterator methods. We have a `find` method as well, but we don't want it -- it checks every element in the worst case, but we hope to use the ordering to do better.

```
public class LinkedBinarySearchTree<T>
             extends LinkedBinaryTree<T>
             implements BinarySearchTreeADT <T> {
   public LinkedBinarySearchTree ( ) {super( );}
   public LinkedBinarySearchTree (T element) {super(element);}

// add, remove, find methods to follow
```

## Adding an Element: The Idea

- Our problem is to find a legal place to add a new leaf, containing the new element, in place of an existing null pointer.
- Although this code does not use recursion, the idea is recursive.  To insert the new node into the subtree under node x, we compare the element against x and then insert the element into either the left or the right subtree of x.
- The base case is when we insert into an empty tree, and just make a new root containing the new element.
- What happens if the element is equal to an existing one?  The test says to go to the left subtree of x if the new element is *smaller* than x's element.  If it is equal, it goes into the right subtree.  Of course if we didn't want to store duplicates, we could abort the add process when we discover the equality.

## Adding an Element: Code

- Note the cast on line 3, which will fail if T is not an ordered class.

```
public void addElement (T element) {
   BinaryTreeNode<T> temp = new BinaryTreeNode<T> (element);
   Comparable<T> comp = (Comparable<T>) element;
   if (isEmpty( )) root = temp;
   else {BinaryTreeNode<T> current = root;
      boolean added = false;
      while (!added) {
         if (comp.compareTo (current.element) < 0) {
            if (current.left == null) {
               current.left = temp; added = true;}
            else current = current.left;}
         else if (current.right == null) {
               current.right = temp; added = true;}
         else current = current.right;}}
   count++;}
```

## Removing an Element: The Idea

- The easy case is when the element to be removed is at a leaf -- we just remove it and nothing else needs to happen.
- If the element to be removed is a unary node, we just promote its only child into its position. Because everything satisfied the BST rules before, it still does.
- If the element to be removed has two children, we have to move either the largest element in the left subtree or the smallest element in the right subtree into its position. (It's arbitrary, we choose the latter.) But that element might not be a leaf, so we might not be done. Fortunately, it cannot have a left child (why?) and so we can promote its right child into its position.

## Removing an Element: Code

- This is not all the code -- we need a method to fill in holes from deletions, and the search method won't fit on one page.

```
public T removeElement (T target) throws ElementNotFoundException {
   T result = null;
   Comparable<T> comp = (Comparable<T>) target;
   if (!isEmpty( ))
      if (comp.equals (root.element)) {
         result = root.element;
         root = replacement (root);
         count--;}
      else {BinaryTreeNode<T> current, parent = root;
         boolean found = false;
         if (comp.compareTo(root) < 0) current = root.left;
         else current = root.right;
         while (current != null && !found) { // see next page
         }
         if (!found) throw new ElementNotFoundException( );}
   return result;}
```

## Removing an Element: More Code

- If we stumble across the target, we set `found` to be true, otherwise we move the variable `current` down the tree.

- Here and on the last page, we need a method to fill in holes, which is on the next page.

```
while (current != null) && !found)
   if (target.equals (current.element)) {
      found = true; count--; result = current.element;
      if (current == parent.left)
         parent.left = replacement (current);
      else parent.right = replacement (current);}
   else {parent = current;
      if (comp.compareTo (current.element) < 0)
         current = current.left;
      else current = current.right;}
```

## Removing an Element: Still More Code

- This method finds a node that can safely be moved into the hole at “node”. In the general case this is the leftmost descendant of node’s right child.

```
protected BinaryTreeNode<T> replacement (BinaryTreeNode<T> node) {
   BinaryTreeNode<T> result = null;
   if ((node.left == null) && (node.right == null)) result = null;
   else if ((node.left != null) && (node.right == null)
      result = node.left;
   else if ((node.left == null) && (node.right != null)
      result = node.right;
   else {BinaryTreeNode<T> current = node.right, parent = node;
      while (current.left != null) {
         parent = current; current = current.left;}
      if (node.right == current) current.left = node.left;
      else {parent.left = current.right;
         current.right = node.right;
         current.left = node.left;}
      result = current;}
    return result;}
```

## Other Operations in `LinkedBinarySearchTree`

- To remove all occurrences of an element, we just keep removing them until there are no more, catching the final exception if needed.
- To remove the minimum element, we first find it by taking left children as long as they exist.   If the minimum is a leaf, we just remove it -- if it is an internal node (with no left child) we remove it and promote its right child into its position.
- The `removeMax`,  `findMin`, and `findMax` operations are similar.

## Array Implementations of Binary Search Trees

- Remember that we have two ways to implement a binary tree with an array. The first uses arithmetic operations to give each array index a parent (unless it is 0, the root), a left child, and a right child. The second replaces the pointers of the linked implementation with `int` variables storing an index.

- If we allow our tree to become unbalanced in the first version, we will expand our array to include room for a complete binary tree. Any indices that don't correspond to nodes actually in the tree will store null entries. This could use a lot of extra space if the tree is very unbalanced.

- L&C, in Section 10.3, have an array-based search tree that does some massive shifting when an element is removed. We won't look at this code.

- The other idea can be implemented with no wasted space if we keep a parent pointer for each node. Every add uses the next new array location, and on every remove we put the last entry into the hole created, updating the pointers.