CMPSCI 187: Programming With Data Structures

Lecture #27: Implementing Binary Search Trees
David Mix Barrington
14 November 2012

Implementing Binary Search Trees

- Review: The BST Interface and Class Header
- Size: Recursive and Iterative Approaches
- Contains: A Recursive Helper Method
- Add: Inserting at a Leaf
- Remove: Replacing the Missing Node
- The Three Iterators

Review: The BST Interface

- The binary search interface is very similar to the list interface.
- Remember that we have three ways to iterate through a tree, which are referenced by the parameters for the `reset` and `getNext` methods.

```
public interface BSTInterface<T extends Comparable<T>> {
   static final int INORDER = 1,PREORDER = 2, POSTORDER = 3;
   boolean isEmpty( );
   int size( );
   boolean contains (T element);
   boolean remove (T element);
   T get (T element);
   void add (T element);
   int reset (int orderType);
   T getNext (int orderType);}
```

Review: The BSTNode Class and the Header

- BST nodes are much like LLNodes, the header is much like the RefLists.

```
public class BSTNode<T extends Comparable<T>> {
       protected T info;
       protected BSTNode left;
       protected BSTNode right;
       public BSTNode (T info) {
          this.info = info; left = right = null;}
       } // also has getters and setters

public class BinarySearchTree<T extends Comparable<T>>
             implements BSTInterface<T> {
   protected BSTNode root;
   boolean found; // used by remove
   protected LinkUnbndQueue<T> inOrderQueue; // for traversal
   protected LinkUnbndQueue<T> preOrderQueue;  // ditto
   protected LinkUnbndQueue<T> postOrderQueue; // ditto
   public BinarySearchTree( ) {
      root = null;}
   public boolean isEmpty( ) {
      return (root == null);}
```

Size: Recursive and Iterative Approaches

- We could keep a count of the nodes in the tree and update it on any insert of delete. But it's useful to have the size of any **subtree** available, and we can calculate that (including the size of the root's subtree, which is the whole tree) with a recursive **helper method**.

- Clearly the size of a node's subtree is 1 plus the size of its left subtree plus the size of its right subtree. But computing this while avoiding null pointer exceptions is a bit subtle. The easiest thing is to have recSize be prepared for an empty tree:

```
public int size( ) {return recSize (root);}

private int recSize (BSTNode<T> tree) {
   if (tree == null)
      return 0;
   return recSize(tree.getLeft()) +
          recSize (tree.getRight()) + 1;}
```

An Iterative Approach to Finding Size

- We can visit all the nodes of the tree by what amounts to a depth-first search with a stack. Since each node will go onto the stack exactly once, we just keep a counter and (as on Midterm #2) increment it for each node pushed.

- This method is more complicated than the recursive one and not faster.

```
public int size( ) {
   int count = 0;
   if (root != null) {
      LinkedStack<BSTNode<T>> hold = new LinkedStack<BSTNode<T>>;
      BSTNode<T> currNode;
      hold.push (root);
      while (!hold.isEmpty()) {
         currNode = hold.top( ); hold.pop( ); count++;
         if (currNode.getLeft( ) != null)
            hold.push (currNode.getLeft( ));
         if (currNode.getRight( )) != null)
            hold.push (currNode.getRight( ));}}
   return count;}
```

Contains: A Recursive Helper Method

- We know how to search a BST -- if the element we seek is at the current node we have found it, and if not we know that we have to search either the left or the right subtree. The job that we want to make recursive is "determine whether this element is in the subtree under this node". The base cases are a null node (false) and a node with the given element (true).

- The `get` method (and `recGet`) has exactly the same logic but different base cases -- we either return null if we fail or return the element that we found.

```
public boolean contains (T element) {
   return recContains (element, root);}

private boolean recContains (T element, BSTNode<T> tree) {
   if (tree == null) return false;
   if (element.compareTo(tree.getInfo( )) < 0)
      return recContains (element, tree.getLeft( ));
   if (element.compareTo(tree.getInfo( )) > 0)
      return recContains (element, tree.getRight( ));
   return true;}
```

Add: Inserting at a Leaf

- We use yet another recursive helper method to insert elements. The `recAdd` method takes a subtree as a parameter, in the form of its root node. It returns the new version of the same subtree, by returning a pointer to *its* root node.

- The recursive job is "insert this element into the correct position within the subtree given by this node -- return the new version of the subtree". The base case is when the subtree is empty. Note that new equal elements are inserted to the left of the existing equal elements.

```
public void add (T element) {root = recAdd (element, root);}

private BSTNode<T> recAdd (T element, BSTNode<T> tree) {
   if (tree == null)
      tree = new BSTNode<T> (element);
   else if (element.compareTo(tree.getInfo( )) <= 0)
      tree.setLeft(recAdd(element, tree.getLeft( )));
   else tree.setRight(recAdd(element, tree.getRight( )));
   return tree;}
```

Remove: Replacing the Missing Node

- Before we look at the code for the recRemove record, let's look at the logic for removing a node.
- The easiest case is when the node to be removed is a leaf. We just delete it, and the resulting tree is still a valid BST.
- The next easiest case is when the node to be removed is a single parent. We replace the node with its only child, whether left or right, and we again have a valid BST.
- Removing a parent of two children is more complicated. We will do it by finding the inorder predecessor of the target node, moving its info to the target node, and then removing the predecessor node. We do this last removal recursively, but actually the predecessor can't have a right child (why?).

Four Methods to Remove a Node

- Here are three of the four methods we need. We have a helper `recRemove` that replaces the subtree with a new one with the element removed. Once we find the node, `removeNode` deletes the predecessor and moves its info.

```
public boolean remove (T element) {
   root = recRemove(element, root); return found;}

private BSTNode<T> recRemove (T element, BSTNode<T> tree) {
   if (tree == null) found = false;
   else if (element.compareTo(tree.getInfo( )) < 0)
      tree.setLeft(recRemove(element, tree.getLeft( )));
   else if (element.compareTo(tree.getInfo( )) > 0)
      tree.setRight(recRemove(element, tree.getRight( )));
   else {tree = removeNode(tree); found = true;}
   return true;}

private T getPredecessor(BSTNode<T> tree) {
// returns info of rightmost node in tree
  while (tree.getRight( ) != null) tree = tree.getRight( );
  return tree.getInfo( );}
```

The removeNode Method

- If the tree has one child, we return it so it will go into the tree in place of the removed node. (If it has no children we'll return null.) Otherwise we find the rightmost node in the left subtree, recursively remove it, and move its info into the current node replacing the removed info.

```
private BSTNode<T> removeNode (BSTNode<T> tree) {
// returns root of this tree with its root node removed or replaced
   T data;
   if (tree.getLeft( ) == null)
      return tree.getRight( );
   if (tree.getRight( ) == null)
      return tree.getLeft( );
   data = getPredecessor (tree.getLeft( ));
   tree.setInfo (data);
   tree.setLeft (recRemove (data, tree.getLeft( )));
   return tree;}
```

The Three Iterators

- Remember that we have a queue for each of the order types. When we reset for one of the types, this queue is replaced with a fresh one and a recursive method is called to put the contents of each node of the tree, in the correct order, into the queue. Then the getNext( ) method only has to dequeue the next node in the queue to get the next node in the traversal.

```
public int reset (int orderType) {
   int numNodes = size( );
   if (orderType == INORDER) {
      inOrderQueue = new LinkedUnbndQueue<T> (numNodes);
      inOrder(root);}
   //... for PREORDER, POSTORDER

public T getNext (int orderType)
   if (orderType == INORDER)
      return inOrderQueue.dequeue( );
   //... for PREORDER, POSTORDER
```

Recursive Traversal Methods

- Now we have three clean methods to copy the info to the three queues.

```
private void inOrder (BSTNode<T> tree) {
   if (tree != null) {
      inOrder(tree.getLeft( ));
      inOrderQueue.enqueue(tree.getInfo( ));
      inOrder(tree.getRight( ));}}

private void preOrder (BSTNode<T> tree) {
   if (tree != null) {
      preOrderQueue.enqueue(tree.getInfo( ));
      preOrder(tree.getLeft( ));
      preOrder(tree.getRight( ));}}

private void postOrder (BSTNode<T> tree) {
   if (tree != null) {
      postOrder(tree.getLeft( ));
      postOrder(tree.getRight( ));
      postOrderQueue.enqueue(tree.getInfo( ));}}
```