CmpSci 187: Programming with Data Structures
Spring 2015

Lecture #17, Implementing Binary Search Trees

John Ridgway

April 2, 2015

1 Implementing Binary Search Trees

Review: The BST Interface

- Binary search trees store objects in nodes, arranged according to the BST Rule — everything in x's left subtree is $\leq x$ and everything in its right subtree is $\geq x$.
- The binary search interface is very similar to the list interface.
- Remember that we have three ways to iterate through a tree, using the different iterators.

Clicker Question #1

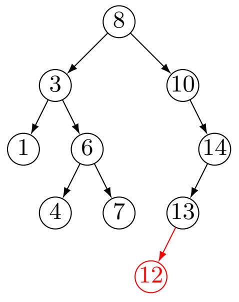
If I add a node with value 12 to this BST, without changing the links of any other nodes, where must the new node go?

A. a right child of 7

B. a left child of 10

C. a left child of 13

D. it's not possible to add it



Code for `BSTInterface`

```
import java.util.Iterator;

public interface
    BSTInterface<T extends Comparable<T>>
{
  boolean isEmpty();
```

```
    int size();
    boolean contains(T element);
    boolean remove(T element);
    T get(T element);
    void add(T element);
    Iterator<T> preOrderIterator();
    Iterator<T> inOrderIterator();
    Iterator<T> postOrderIterator();
  }
```

Review: The `BSTNode` Class

(Getters and setters omitted.)

```
  public class BSTNode<T> {
    private T data;
    private BSTNode<T> left;
    private BSTNode<T> right;

    public BSTNode(T data, BSTNode<T> left,
                   BSTNode<T> right)
    {
      this.data = data;
      this.left = left;
      this.right = right;
    }
  }
```

Review: The `BinarySearchTree` Header

```
  public class
      BinarySearchTree<T extends Comparable<T>>
      implements BSTInterface<T>
  {
    private BSTNode<T> root;

    public boolean isEmpty() {
      return root == null;
    }
```

Review: Recursive `size` Method

We use a helper method, which clearly is correct on empty trees, and which uses a recursive definition to calculate the size of a nonempty tree from the size of its subtrees.

```
    public int size() {
      return subtreeSize(root);
    }
    private int subtreeSize(BSTNode<T> node) {
      if (node == null) {
        return 0;
      } else {
        return 1 + subtreeSize(node.getLeft())
                 + subtreeSize(node.getRight());
      }
    }
```

Review: Recursive `get`

```
public T get(T t) {
  return getFromTree(t, root);
}

private T getFromTree(T t,
                         BSTNode<T> node) {
  if (node == null) return null;
  if (t.compareTo(node.getData()) < 0)
    return getFromTree(t, node.getLeft());
  if (t.compareTo(node.getData()) == 0)
    return node.getData();
  return getFromTree(t, node.getRight());
}
```

What's Left to Implement?

- We've written two of the main observer methods of our `BinarySearchTree` class, `size`, and `get`, and hinted at `contains`.
- We still need to write the two main transformer methods, `add` and `remove`. Here our main problem will be to maintain the BST rule at every node as we change the tree structure.
- We also need to write the three iterators for the three different orders on the nodes.

The `add` Method

- We use yet another recursive helper method to insert elements. The `addTo` 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 and return the new version of the subtree."

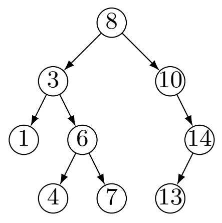
Code for the `add` Method

```
public void add(T t) {
  root = addTo(t, root);
}

private BSTNode<T> addTo(T t, BSTNode<T> node)
{
  if (node == null) {
    return new BSTNode<T>(t, null, null); }
  if (t.compareTo(node.getData()) <= 0) {
    node.setLeft(addTo(t, node.getLeft()));
  } else {
    node.setRight(addTo(t, node.getRight()));
  }
  return node;
}
```

Clicker Question #2

Suppose that this BST was built by successively adding each of the elements, starting from an empty tree. Which of these might have been the order in which they were added?



A. 8, 10, 14, 3, 6, 7, 1, 4, 13

B. 8, 3, 14, 1, 4, 13, 6, 10, 7

C. 8, 3, 6, 10, 1, 14, 13, 7, 4

D. either a or c is possible

The `remove` Method

- Removing a node will be considerably more complicated because we have to preserve the BST property everywhere.
- Once again we'll use a recursive helper `removeFrom`.
- Before we look at the code for the `removeFrom` method, let's look less formally at the logic for removing a node.

Logic for Removing

- The easiest case is when the node to be removed is a leaf. We just delete it.
- The next easiest case is when the node to be removed has only one child. We replace the node with its child, whether left or right.
- Removing a node that has two children is more complicated. We will do it by finding the in-order predecessor of the target node, moving its data 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.

Clicker Question #3

Let x be any node of a BST, with two children, and let y be its in-order predecessor. Which of these statements is *not* true?

A. y cannot possibly have a right child.

B. y must be the left child of x.

C. y must be a member of x's left subtree.

D. y is either the left child of x or y is a right child.

Methods for Removing

- We will use four methods in all. The actual `remove` method calls a recursive helper method `removeFrom` that returns the new version of the tree rooted at its parameter.
- `removeFrom` uses two other methods:
  - `getLargest` returns the data stored in the right-most node in a sub-tree; and
  - `removeLargest` removes the right-most node of the sub-tree and returns its (possibly modified) root.

Code for the `remove` Method

```
public boolean remove(T t) {
  boolean result = contains(t);
  if (result) {
    root = removeFrom(t, root);
  }
  return result;
}
```

Code for the `removeFrom` Method

```
private BSTNode<T> removeFrom(T t,
                              BSTNode<T> node) {
  if (t.compareTo(node.getData()) < 0) {
    node.setLeft(removeFrom(t, node.getLeft()));
  } else if (t.compareTo(node.getData()) > 0) {
    node.setRight(removeFrom(t, node.getRight()));
  } else if (node.getLeft() == null) {
    return node.getRight();
  } else if (node.getRight() == null) {
    return node.getLeft();
  } else { // neither child is null
    node.setData(getLargest(node.getLeft()));
    node.setLeft(removeLargest(node.getLeft()));
  }
  return node;  }
```

`getLargest` and `removeLargest`

```
private T getLargest(BSTNode<T> node) {
  if (node.getRight() == null) {
    return node.getData();
  } else {
    return getLargest(node.getRight());
  } }

private BSTNode<T> removeLargest(BSTNode<T> node)
{
  if (node.getRight() == null) {
    return node.getLeft();
  } else {
    node.setRight(removeLargest(node.getRight()));
    return node;
  } }
```

Clicker Question #4

This method returns the data from what node?

```
private T getLargest(BSTNode<T> node) {
  if (node.getRight() == null) {
    return node.getData();
  } else {
    return getLargest(node.getRight());
  } }
```

A. `node`

B. the leftmost node in the tree of `node`'s sibling

C. the parent of `node`

D. the rightmost node in the tree of `node`

The Three Iterators

The iterators are all similar; we create a queue and enqueue all of the elements in the appropriate order, then return an iterator on the queue.

```
public Iterator<T> inOrderIterator() {
  Queue<T> queue = new LinkedQueue<T>();
  inOrderTraverse(queue, root);
  return queue.iterator();
}
private void inOrderTraverse(Queue<T> queue,
                             BSTNode<T> node) {
  if (node == null) return;
  inOrderTraverse(queue, node.getLeft());
  queue.enqueue(node.getData());
  inOrderTraverse(queue, node.getRight()); }
```

Comparing Trees and Lists

- It's natural to compare the performance of BST's and linear lists on the same tasks. There's a problem, though, with our worst-case analysis. In the worst case, when every node has at most one child, a BST is essentially a linear list.
- Searching into a BST takes a worst-case time equal to the height of the tree. If the tree is balanced, this height is $O(\log n)$ for an n-node tree, but for an unbalanced tree it could be as large as $O(n)$.
- DJW have a chart on page 579 comparing balanced BST's, array-based lists, and linked lists.
- The only BST operations that take more than $O(\log n)$ are the three iterator methods; they copy all data into a queue.
- The array-based sorted list can perform `get` in $O(\log n)$, but `add` and `remove` each take $O(n)$ because of the need to move data. The linked list takes $O(1)$ to process an `add` or `remove` operation, but $O(n)$ to find where to do it.

How to Balance a Tree?

- How can we balance a tree? In CMPSCI 311 you will learn about self-balancing trees, in at least one of the several versions.
- These trees always have a height of $O(\log n)$ when their size is n.
- Adding and removing take $O(\log n)$ time, because there may be a restructuring step of $O(\log n)$ time after each such move to restore the balance.

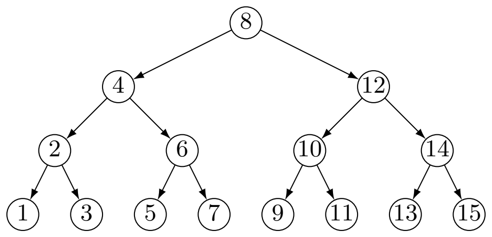
Wrong Ways to Balance

- DJW consider only a simpler method of rebalancing that takes $O(n)$ time whenever we choose to do it.
- This is to copy the entire tree into an array of elements, then add each element of the array in turn back into a new tree. We naturally form our array by using one of our three traversal methods.
- If we use in-order, the array becomes a sorted list. Unfortunately, adding the elements in sorted order produces a very unbalanced tree.
- If we form the array using pre-order, and then load the array back into the BST, it turns out that the new tree is identical to the old one.
- What happens if we form the array from the post-order traversal, and then load it back in?

An Example

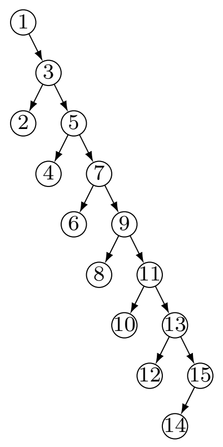
- Consider a perfectly balanced BST of Integers with 15 nodes with values 1 through 15.
- The post-order traversal of this tree visits the nodes in order 1, 3, 2, 5, 7, 6, 4, 9, 11, 10, 13, 15, 14, 12, 8.
- If we insert the nodes into a new BST in this order, we get a tree of height 9 instead of the original height 3.

A Perfectly Balanced BST



Its post-order traversal:
1, 3, 2, 5, 7, 6, 4, 9, 11, 10, 13, 15, 14, 12, 8

Adding them in Post-Order



Original in post-order:
1, 3, 2, 5, 7,
6, 4, 9, 11, 10,
13, 15, 14, 12, 8

Balancing a BST Effectively

- The right idea here is to study how we want the new tree to be arranged. Suppose we have used in-order to form the array, so that the array is sorted.
- If the tree is balanced, with an equal number of nodes on each side, the root node will be the middle element of the array. Its left child should be about 1/4 of the way along, and its right child 3/4 of the way, and so forth. This should suggest a recursion.

Code to Balance a BST - Part I

```
public void balance() {
  T[] values = (T[]) new Comparable[size()];
  Iterator<T> iterator = inOrderIterator();
  int index = 0;
  while (iterator.hasNext()) {
    values[index] = iterator.next();
    index += 1;
  }
  root = null;
  insertTree(values, 0, index - 1);
}
```

Code to Balance a BST - Part II

```
private void insertTree(T[] values, int low,
                        int high)
{
  if (low == high) {
    add(values[low]);
  } else if (low + 1 == high) {
    add(values[low]);
    add(values[high]);
  } else {
    int mid = (low + high)/2;
    add(values[mid]);
    insertTree(values, low, mid - 1);
    insertTree(values, mid + 1, high);
  }
}
```