Lab: Binary Search Trees

In this lab, you will need the Display class, TreeUtil class and the TreeNode class from your Binary Trees lab.

Exercise: Searching and Inserting

Download BSTUtilities.java. Implement the `contains` and `insert` methods, which should each run in O(*h*) time, where *h* is the height of the tree. (Note that `insert` should ignore attempts to add duplicate elements.) To help you debug your work (and to add some fun to the lab), be sure to call `display`'s `visit` method to light up the path your search takes. Then go and test that these two methods work correctly together.

```
public static boolean contains(TreeNode t, Comparable x, TreeDisplay display)
public static TreeNode insert(TreeNode t, Comparable x, TreeDisplay display)
```

Exercise: Deleting

Deleting from a binary search tree is a valuable and challenging exercise (even though it won't appear on the AP test). We'll break down the problem into two methods.

```
public static TreeNode delete(TreeNode t, Comparable x, TreeDisplay display)
private static TreeNode deleteNode(TreeNode t, TreeDisplay display)
```

The `delete` method finds the node in tree `t` that contains the value `x`, and then calls `deleteNode` to perform the actual deletion. We'll start by implementing the more difficult `deleteNode` method, which removes the value in a node `t` and returns a pointer to the resulting tree.

When we remove a value from the tree, we need to make sure that the remaining tree is still a valid binary search tree (with all values still in ascending order). It helps to consider three cases. In all three, we are deleting node *h*, and we need to determine *h*'s successor—the next node in ascending order.

Case 1: *h* has no successor

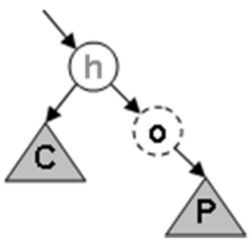
We simply return a pointer to the left subtree.

h has no successor

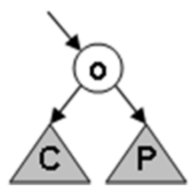
h has been deleted

Case 2: *h*'s successor is its right child

We need to "splice out" the node containing the successor, and replace *h* with the value from the successor.



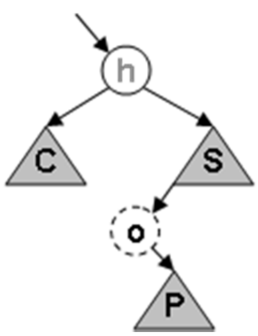
h's successor is its right child



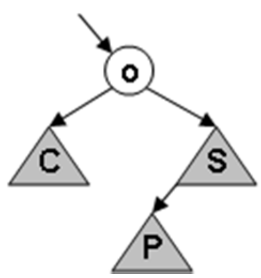
h has been deleted

Case 3: *h*'s successor is not its right child

Again, we "splice out" the successor, and replace *h* with this value.



h's successor is not its right child



h has been deleted

Once we've written `deleteNode`, writing the `delete` method is fairly straightforward. You may want to call `display`'s `visit` method from each of these, and test a variety of cases.

Exercise: `MyTreeSet<E>`

Download MyTreeSet.java, which will support the following standard `Set<E>` operations:

```
•  int size()
•  boolean contains(Object obj)
•  boolean add(E obj)           // if obj is not present in this set, adds obj and
                                // returns true; otherwise returns false
•  boolean remove(Object obj)   // if obj is present in this set, removes obj and
                                // returns true; otherwise returns false
```

Internally, `MyTreeSet<E>` stores its data in a binary search tree. In addition, it should hold onto a single `TreeDisplay` object to display its contents at all times. It should also remember the number of elements in the tree, so that it can report its size in constant time. The `contains`, `add`, and `remove` methods should run in O(*h*) time, where *h* is the height of the tree.

Call your `BSTUtilities` methods, rather than re-implementing them! Then download TreeSetTester.java to test your work.

Additional Credit (Above 95%)

+2%: Add a method called `iterator`, which takes no arguments and returns an `Iterator`. The `Iterator` should return all values in the `MyTreeSet` in ascending order. You must do this by manipulating only the pointers within tree nodes. You may not create another data structure that stores the values contained in `MyTreeSet.` For example, you may not create a linked list or ArrayList representation by traversing the tree. (You may create additional data structures that hold pointers.)

+3%: `MyTreeMap<K, V>`: We can also use a binary search tree to make a map. Create a `MyTreeMap<K, V>` class, which should support the following operations:

```
•  int size()
•  boolean containsKey(Object key)
•  V put(K key, V value)           // associates key with value
                                   // returns the value formerly associated with key
                                   // or null if key was not present
•  V get(Object key)               // returns the value associated with key
                                   // or null if there is no associated value
•  V remove(Object key)         // removes and returns the value associated with key;
                                // returns null if there is no associated value
```