17.1 – Binary Search Trees (BSTs)

- Write the inorder traversal of this tree.
  What do you observe?
- A search tree is a tree whose elements are organized to facilitate finding a particular element.
- A binary search tree is a binary tree that, for each node \( n \):
  - the left subtree of \( n \) contains elements less than the element stored in \( n \)
  - the right subtree of \( n \) contains elements greater than or equal to the element stored in \( n \)
- Binary search trees must hold comparable data. Why?
- How do you search for an element?

17.1 – Adding an Element to a BST

Next to add:

```
+-----+   +-----+
| 88  |   | 77  |
|     |   |     |
```

```
+-----+   +-----+
| 24  |   | 77  |
|     |   |     |
```

```
+-----+   +-----+
| 24  |   | 82  |
|     |   |     |
```

```
+-----+   +-----+
| 17  |   | 58  |
|     |   |     |
```

```
+-----+   +-----+
| 17  |   | 97  |
|     |   |     |
```

```
+-----+   +-----+
| 40  |   | 97  |
|     |   |     |
```

Animation: [http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html](http://www.cs.jhu.edu/~goodrich/dsa/trees/btree.html)

On what does the shape of a binary search tree depend?

17.1 – Degenerate Tree

- A grossly unbalanced tree, with some long paths
- When does it occur?
- Why is it undesirable?

17.1 – Removing an Element from a BST

- Removing a target in a BST is not as simple as that for linear data structures.
- After removing the element, the resulting tree must still be valid.
- What if you remove 88? 51? 60?
17.1 – After the Root Node is Removed

* Draw Tree (2 valid configurations)

17.2 – BST Implementation

The BinarySearchTree interface class adds support for
add, remove, find, findMin, and findMax

17.2 – javafoundations.BinarySearchTree

```java
public interface BinarySearchTree<T extends Comparable<T>> extends BinaryTree<T> {
    // Adds the specified element to the tree.
    public void add(T element);
    // Finds and returns the element in the tree matching the
    // specified target. Overrides the find method of BinaryTree.
    public T find(T target);
    // Returns the minimum value in the binary search tree.
    public T findMin();
    // Returns the maximum value in the binary search tree.
    public T findMax();
    // Removes and returns the specified element from the tree.
    public T remove(T target);
}
```

17.2 – javafoundations.BSTNode

```java
public class BSTNode<T extends Comparable<T>> extends BTNode<T> {
    // Creates a new tree node with the specified data.
    public BSTNode(T element) {
        super(element);
    }
    // (more...)
}
```
17.2 – javafoundations.BSTNode

Next to add:
77
24
82
17
58
97
40

public void add(T item) {
    if (item.compareTo(element) < 0)
        if (left == null)
            left = new BSTNode (item);
        else
            ((BSTNode)left).add (item);
    else // item >= element, go right
        if (right == null)
            right = new BSTNode (item);
        else
            ((BSTNode)right).add (item);
}

public BSTNode<T> find (T target) {
    BSTNode<T> result = null;
    if (target.compareTo(element) == 0)
        result = this;
    else {
        if (target.compareTo(element) < 0)
            if (left != null)
                result = ((BSTNode)left).find (target);
        else
            if (right != null)
                result = ((BSTNode)right).find (target);
    }
    return result;
}

public BSTNode<T> remove(T target) {
    BSTNode<T> result = this;
    if (target.compareTo(element) == 0)
        if (left == null && right == null)
            result = null; // Situation 1
        else if (left != null && right == null)
            result = (BSTNode)left; // Situation 2
        else if (left == null && right != null)
            result = (BSTNode)right; // Situation 3
        else
            result = getSuccessor();
    else
        if (target.compareTo(element) < 0)
            if (left != null)
                result = ((BSTNode)left).remove(target); // Situation 1
        else // target > element, look to the right to remove
            if (right != null)
                result = ((BSTNode)right).remove(target); // Situation 3
    return result;
}

protected BSTNode<T> getSuccessor() {
    BSTNode<T> successor = (BSTNode)right;
    while (successor.getLeft() != null)
        successor = (BSTNode)successor.getLeft();
    ((BSTNode)right).remove (successor.getElement());
    return successor;
}
```java
package javafoundations;
import javafoundations.*;
import javafoundations.exceptions.*;
public class LinkedBinarySearchTree<T extends Comparable<T>>
    extends LinkedBinaryTree<T>
    implements BinarySearchTree<T> {
    //  Creates an empty binary search tree.
    public LinkedBinarySearchTree() {
        super();
    }
    //  Creates a binary search tree with the specified element at its
    //  root.
    public LinkedBinarySearchTree(T element) {
        root = new BSTNode<T>(element);
    }
    //  Adds the specified element to this binary search tree.
    public void add(T item) {
        if (root == null) {
            root = new BSTNode<T>(item);
        } else {
            ((BSTNode)root).add(item);
        }
    }
    //  Removes and returns the element matching the specified target
    //  from this binary search tree. Throws an ElementNotFoundException
    //  if the target is not found.
    public T remove(T target) {
        BSTNode<T> node = null;
        if (root != null) {
            node = ((BSTNode)root).find(target);
            if (node == null) {
                throw new ElementNotFoundException("Remove operation failed. " + "No such element in tree.");
            }
            root = ((BSTNode)root).remove(target);
            return node.getElement();
        }
    }
    public T findMin() {
        return root.min().getElement();
    }
    public T findMax() {
        return root.max().getElement();
    }
}
```

---

17.3 – Balanced Binary Search Trees

- The find and add operations of a balanced tree of n nodes have an efficiency of \( O(\log_2 n) \) \( = \) length of the longest path
- The more degenerate a tree becomes, the find and add operations approach \( O(n) \)
- Our BST implementation does not guarantee a balanced tree
- The shape of a BST is determined by the order which elements are added to the tree
- Other types of trees exist to ensure that they stay balanced
- They include AVL trees and red/black trees. See animation at
  - [http://www.cs.jhu.edu/~goodrich/dsa/trees/avltree.html](http://www.cs.jhu.edu/~goodrich/dsa/trees/avltree.html)
If the imbalance is caused by a long path in the left subtree of the right child of the root we can address it by
- performing a right rotation around the offending subtree
- and then performing a left rotation around the root

If the imbalance is caused by a long path in the right subtree of the left child of the root we can address it by
- performing a left rotation around the offending subtree
- and then performing a right rotation around the root