• 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?

---

**Next to add:**

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

---

**Animation:**

[https://www.cs.usfca.edu/~qalles/visualization/BST.html](https://www.cs.usfca.edu/~qalles/visualization/BST.html)

On what does the shape of a binary search tree depend?
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? 69?

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

package javafoundations;

public interface BinarySearchTree<T extends Comparable<T>> extends BinaryTree<T> {
    public void add(T element);
    public T find(T target);
    public T findMin();
    public T findMax();
    public T remove(T target);
}

//*****************************************************************************/
// BinarySearchTree.java Java Foundations
/*****************************************************************************/
// Defines the interface to a binary search tree.
/*****************************************************************************/
package javafoundations;

public class BSTNode<T extends Comparable<T>> extends BTNode<T>
{
    public BSTNode(T element)
    {
        super(element);
    }
}

public void add(T item)
{
    if (item.compareTo(element) < 0)
        if (left == null)
            left = new BSTNode(item);
        else // Add recursively
            ((BSTNode)left).add(item);
    else // item >= element, go right
        if (right == null)
            right = new BSTNode(item);
        else // Add recursively
            ((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 2
        else // Situation 3
        {
            result = getSuccessor();
            result.left = left;
            result.right = right;
        }
    }
    return result;
}
else
    if (target.compareTo(element) < 0)
        if (left != null)
            left = ((BSTNode)left).remove(target);
    else // target > element, look to the right to remove
        if (right != null)
            right = ((BSTNode)right).remove(target);
    
    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;
}
The find and add operations of a balanced tree of n nodes have an efficiency of \( O(\log_2 n) \).

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