```java
/**
 * Represents a Binary Search Tree that stores elements.
 *
 * @author Aaron G. Cass and Chris Fernandes and Blair Hagen
 * @version 5/26/16
 *
 * As a student at Union College, I am part of a community that values
intellectual effort, curiosity and discovery. I understand that in
order to truly claim my educational and academic achievements, I am
obligated to act with academic integrity. Therefore, I affirm that I
will carry out my academic endeavors with full academic honesty, and I
rely on my fellow students to do the same.
 *
 * @param <Element> The kind of element that is stored. Must be
comparable
 * to Elements.
 */
public class BST<Element extends Comparable<Element>>
{
    private BSTNode<Element> root;

    /**
     * Constructs an empty Binary Search Tree.
     */
    public BST()
    {
        root = null;
    }

    /**
     * Searches for an element in the tree.
     *
     * @param toFind
     *            the element to search for
     * @return true iff the element is in the tree
     */
    public boolean search(Element toFind)
    {
        return search(root, toFind);
    }

    /**
     * Searches for an element in a subtree of the tree.
     *
     * @param subtreeRoot
     *             the root of the subtree in which to search
     * @param toFind
     *             the element to search for
     * @return true iff the element is in the give subtree
     */
    private boolean search(BSTNode<Element> subtreeRoot, Element
toFind)
    {
     if (subtreeRoot == null) {
            return false;
        } else if (subtreeRoot.key.compareTo(toFind) == 0) {
            return true;
        } else if (subtreeRoot.key.compareTo(toFind) > 0) {
            return search(subtreeRoot.llink, toFind);
        } else {
            return search(subtreeRoot.rlink, toFind);
        }
    }

    /**
     * Adds an element to the tree.
     *
     *
     * @param toAdd
     *             the element to add.
     */
    public void add(Element toAdd)
    {
     BSTNode<Element> newNode = new BSTNode<Element>(toAdd);
     root = add(root, newNode);
    }

    /**
     * Adds newNode to the subtree that subtreeRoot points at.
     * After the new node is added to the tree, the method
     * returns the root of the subtree.
     *
     * Recursive defn of a binary search tree: A binary tree is either
an empty tree or a single
     * node that points to one or two nodes.
     *
     * BASE CASE: GET TO AN EMPTY TREE
      * PSEUDOCODE:
```

```
     * to return a pointer to a tree with newNode at the end, you
either:
     * return newNode if the subtreeRoot points to an empty tree OR
     * return subtreeRoot, whose next pointer points to a Tree where
newNode is inserted
     * at the end.
     *
     *
     * @param subtreeRoot
     *                           the root of the subtree to which to
add.  If
     *                      null, the subtree is empty
     * @param newNode
     *                           the node to add to the subtree.
     * @return the root of the subtree after newNode has been added.
     */
    private BSTNode<Element> add(BSTNode<Element> subtreeRoot,
BSTNode<Element> newNode)
    {
     if (subtreeRoot == null) {
           return newNode;
     }
     else if (newNode.key.compareTo(subtreeRoot.key) <= 0){
           subtreeRoot.llink = add(subtreeRoot.llink, newNode) ;
           return subtreeRoot;
     }
     else {
           subtreeRoot.rlink = add(subtreeRoot.rlink, newNode);
           return subtreeRoot;
     }
    }

}
```

```
/**
 * The linked list class gives you access to the beginning of a linked
 * list through a private instance variable called firstNode.  This
class
 * should contain all of the methods for general manipulation of
```

```
linked lists:
 * traversal, insertion, deletion, searching, etc.
 *
 * @author Chris Fernandes and Blair Hagen
 * @version 5/26/16
 */
public class LinkedList
{
     private int length;          // number of nodes
     private ListNode firstNode;  // pointer to first node

     public LinkedList()
     {
          length=0;
          firstNode=null;
     }

     /** insert newNode at end of the LL that begins at startOfSublist
      *
      * BASE CASE: FIND THE LAST NODE
      * PSEUDOCODE:
      * To insert newNode at the tail of the LL, you either:
      * assign newNode to startOfSublist.next if startOfSublist points
to last node OR
      * insert newNode at the tail of the LL that starts at
startOfSublist.next
      *
      * @param startOfSublist
      *                           pointer to start of list that newNode
should be inserted into
      * @param newNode
      *                           node to insert
      */
     private void insertAtTailPast(ListNode startOfSublist, ListNode
newNode)
     {
          if (startOfSublist.next == null) {
               startOfSublist.next = newNode;
          }
          else {
               insertAtTailPast(startOfSublist.next,newNode);
          }
     }
```

```java
    /** insert new data at end of list
     *
     * @param newData int to insert
     */
    public void insertAtTailPast(int newData) {
        ListNode newNode = new ListNode(newData);
        if (getLength() == 0) {
            firstNode=newNode;
        }
        else {
            insertAtTailPast(firstNode,newNode);
        }
        length++;
    }

    /**
     * insert newNode at end of the LL that begins at startOfSublist
     * When done, return a pointer to the head of this linked list
     *
     * Recursive defn of a linked list: an LL is either an empty list
OR
     * a single node that points to an LL
     *
     * BASE CASE: GET TO AN EMPTY LIST
     * PSEUDOCODE:
     * To return a pointer to an LL where newNode is inserted at the
end, you either:
     * return newNode, if startOfSublist points to an empty list OR
     * return startOfSublist, whose next pointer now points at an LL
where newNode
     * is inserted at the end
     *
     * @param startOfSublist pointer to LL in which to insert
     * @param newNode the new node to insert
     * @return a pointer to a linked list where newNode has been
inserted at the end
     */
    private ListNode insertAtTail(ListNode startOfSublist, ListNode
newNode)
    {
        if (startOfSublist == null) {
            return newNode;
        }
        else {
            startOfSublist.next = 
insertAtTail(startOfSublist.next, newNode);
            return startOfSublist;
        }
    }

    /** insert new data at end of list (DO NOT ALTER THIS METHOD)
     *
     * @param newData int to insert
     */
    public void insertAtTail(int newData) {
        ListNode newNode = new ListNode(newData);
        firstNode = insertAtTail(firstNode,newNode);
        length++;
    }

    /** return linked list as printable string
     *
     */
    public String toString()
    {
        String toReturn="(";
        ListNode runner;
        runner=firstNode;
        while (runner!=null)
        {
            toReturn = toReturn + runner;  //call node's toString 
automatically
            runner=runner.next;
            if (runner!=null)
            {
                toReturn = toReturn + ", ";
            }
        }
        toReturn = toReturn + ")";
        return toReturn;
    }

    /**
     * getter
     *
     * @return number of nodes in LL
     */
    public int getLength() { return length; }
```

```
}
```