```java
/**
 * Generic Binary Search Tree ADT.
 *
 * @author Son Nguyen (Kyrie)
 * @version 6/3/2020
 */
package proj5;
public class BinarySearchTree<T extends Comparable<T
>>
{
    private BSTNode<T> root;

    /**
     * Default constructor
     */
    public BinarySearchTree() {
        root=null;
    }

    /**
     * inserts newNode into tree rooted at
startingNode.
     * Returns root of that tree with newNode
inserted.
     *
     * @param startingNode
     * @param newNode
     * @return root of tree with node inserted
     */
    private BSTNode<T> insert(BSTNode<T> startingNode
, BSTNode<T> newNode) {
        if (startingNode == null) {
            return newNode;
        }
        // pretend that key has compareTo method
        else if (((T) startingNode.key).compareTo((T
) newNode.key) > 0) {
            // newNode goes on left
            startingNode.llink = insert(startingNode.
llink,newNode);
            return startingNode;
        }
        else {
            // newNode goes on right
```

```
39              startingNode.rlink = insert(startingNode.
   rlink,newNode);
40              return startingNode;
41          }
42      }
43
44      /**
45       * inserts an object into BST
46       * @param item object to insert
47       */
48      public void insert(T item) {
49          BSTNode<T> newNode = new BSTNode<T>(item);
50          root=insert(root,newNode);
51      }
52
53      /**
54       * get the min key of a subroot
55       * @param subroot the subroot to get min key
56       * @return the min value of this subroot
57       */
58      private T minValue(BSTNode<T> subroot) {
59          T toReturn = (T) subroot.key;
60          while (subroot.llink != null) {
61              toReturn = (T) subroot.llink.key;
62              subroot = subroot.llink;
63          }
64          return toReturn;
65      }
66
67      /**
68       * get the max key of a subroot
69       * @param subroot the subroot to get max key
70       * @return the max value of this subroot
71       */
72      private T maxValue(BSTNode<T> subroot) {
73          T toReturn = (T) subroot.key;
74          while (subroot.rlink != null) {
75              toReturn = (T) subroot.rlink.key;
76              subroot = subroot.rlink;
77          }
78          return toReturn;
79      }
80
81      /**
```

```
 82      * deletes victim from tree rooted at subroot
 83      *
 84      * @param subroot
 85      * @param victim
 86      * @return pointer to same part of tree but with victim removed, null otherwise
 87      * POST CONDITION: the BSTNode holding the victim is deleted from the subroot
 88      * or its key is replaced by the minimum value in the right branch
 89      */
 90     private BSTNode<T> delete(BSTNode<T> subroot, T victim) {
 91         if (subroot == null) {
 92             return subroot;
 93         }
 94         else if (victim.compareTo((T) subroot.key) < 0) {
 95             subroot.llink = delete(subroot.llink, victim);
 96         }
 97         else if (victim.compareTo((T) subroot.key) > 0) {
 98             subroot.rlink = delete(subroot.rlink, victim);
 99         }
100         else {
101             if (subroot.isLeaf()) {
102                 return null;
103             }
104             else if (subroot.hasLeftChildOnly()) {
105                 return subroot.llink;
106             }
107             else if (subroot.hasRightChildOnly()) {
108                 return subroot.rlink;
109             }
110             else {
111                 subroot.key = minValue(subroot.rlink);
112                 subroot.rlink = delete(subroot.rlink, (T) subroot.key);
113             }
114         }
115         return subroot;
```

```
116     }
117
118     /**
119      * delete an item for the tree, do nothing if
    the item is not found
120      * @param victim the item to be deleted
121      */
122     public void delete(T victim) {
123         root = delete(root, victim);
124     }
125
126     /**
127      * search for an item in a subtree
128      * @param subroot
129      * @param item
130      * @return the node containing the item, null if
     the object is not in the tree
131      */
132     private BSTNode<T> search(BSTNode<T> subroot, T
    item) {
133         if (subroot != null) {
134             if (item.compareTo((T) subroot.key) < 0
    ) {
135                 return search(subroot.llink, item);
136             } else if (item.compareTo((T) subroot.
    key) > 0) {
137                 return search(subroot.rlink, item);
138             }
139         }
140         return subroot;
141     }
142
143     /**
144      * search for an item in the tree
145      * @param item the item to be searched
146      * @return the item's information stored in the
    tree,
147      * null if cannot find the item
148      */
149     public T search(T item) {
150         BSTNode<T> returnNode = search(root, item);
151         if (returnNode != null) {
152             return (T) returnNode.key;
153         }
```

```
154         return null;
155     }
156 
157 //  /**
158 //   * NOTE: CRAPPY METHOD!  I wish I had toString
    ...
159 //   *
160 //   * Recursive helper method of print.
161 //   * Uses inorder tree traversal algorithm.
162 //   * @param N subroot of tree to print
163 //   */
164 //  private void printNode(BSTNode<T> N) {
165 //      if (N != null) {      // do nothing if the
    node is null
166 //          System.out.print("(");
167 //          printNode(N.llink);
168 //          System.out.print("  " + N.key + "  ");
169 //          printNode(N.rlink);
170 //          System.out.print(")");
171 
172 //      }
173 //  }
174 //
175 //
176 //  /**
177 //   * NOTE: CRAPPY METHOD!  I wish I had toString
    ...
178 //   *
179 //   * prints a parenthesized version
180 //   * of the tree showing the subtree structure
181 //   */
182 //  public void print() {
183 //      printNode(root);
184 //      System.out.println();
185 //  }
186 
187     /**
188      * @param subroot the branch
189      * @return string representation of a branch
190      */
191     private String branchString(BSTNode<T> subroot
    ) {
192         String toReturn = "";
193         if (subroot != null) {
```

```
194             if (subroot.llink != null) {
195                 toReturn += branchString(subroot.
    llink);
196             }
197             toReturn += subroot;
198             if (subroot.rlink != null) {
199                 toReturn += branchString(subroot.
    rlink);
200             }
201         }
202         return toReturn;
203     }
204 
205     /**
206      * @return string representation of the tree
207      */
208     public String toString() {
209         return branchString(root);
210     }
211 }
212 
```