5.7 DictBinTree class

```
1 /**
2  * Created by Patrick Jakobsen(pajak16) Gabriel Jadderson(gajad16) on 09/04/2017.
3  *
4  * @author pajak16, gajad1
5  */
6 public class DictBinTree implements Dict
7 {
8
9     Node root;
10    int treeSize;
11
12    /**
13     * Constructor
14     */
15    public DictBinTree()
16    {
17        //The tree starts out empty.
18        root = null;
19    }
20
21    /**
22     * Inserts a key in the dictionary.
23     *
24     * @param k is the key to insert into the dictionary.
25     */
26    @Override
27    public void insert(int k)
28    {
29        //y keeps track of the previous node checked.
30        Node y = null;
31        //x is the starting outset.
32        Node x = root;
33
34        //If x is null, then we reached an empty spot that the new node can be
      inserted on. The node will belong to the last node traversed.
35        while (x != null)
36        {
37            //Keep track of the previously checked node at all times.
38            y = x;
39
40            //Figure out in which direction the new node should be inserted.
41            if (k < x.data)
42            {
43                x = x.left;
44            } else
45            {
46                x = x.right;
47            }
48        }
49
50        //Here the node is inserted somewhere.
51        //If y is null, that means that root is null as well, otherwise y would
      have become root.
52        if (y == null)
53        {
54            //There is no root, so make the inserted value root.
55            root = new Node(k);
56        } else if (k < y.data)
57        { //Otherwise, place the new node either left or right depending on the
      relation between the new key and the key in y.
58            y.left = new Node(k);
59        } else
60        {
61            y.right = new Node(k);
62        }
63
64        //The tree is one larger now. This is kept track of for orderedTraversal.
65        treeSize++;
66    }
67
68    /**
69     * This method creates a sorted array of all the keys in the dictionary and
      returns it. The array is sorted in nondescending order.
70     *
71     * @return The method returns an int[] of the keys sorted in nondescending
      order.
72     */
```

```
73      @Override
74      public int[] orderedTraversal()
75      {
76          //Result to be returned.
77          int[] result = new int[treeSize];
78
79          //The function that is used to traverse the tree recursively.
80          if (root != null)
81          {
82              treewalk(root, result);
83          }
84          //Returns the resulting int[] with the keys.
85          return result;
86      }
87
88      //This variable is used as a marker for inserting the elements in the array in
         treewalk.
89      static int mark = 0;
90
91      /**
92       * This method recursively traverses the tree and inserts the contents of its
        right subtree in the array, then its own element and then the contents of its
        right subtree. This is done using the static mark variable in the class, which
        makes this method not thread safe, even with synchronized blocks.
93       *
94       * @param start The Node object to work on in the current call.
95       * @param array The array to fill elements into.
96       */
97      private void treewalk(Node start, int[] array)
98      {
99
100         //If there is a left subtree, then those elements are all smaller than the
         key in this Node. Traverse that subtree first.
101         if (start.left != null)
102         {
103             treewalk(start.left, array);
104         }
105
106         //System.out.println(start.data);
107
108         //All the smaller elements have already been added to the array. Because
        the elements in the right subtree are all larger than the key in this Node,
        this is the next key to be added to the list.
109         array[mark] = start.data;
110
111         //System.out.println("array contains here: " + array[mark] + "and the mark
         is: " + mark);
112
113         //Increment the location at which the next element will be inserted.
114         mark++;
115
116         //Insert the remaining elements.
117         if (start.right != null)
118         {
119             treewalk(start.right, array);
120         }
121
122     }
123
124     /**
125      * This method takes searches for a key in the dictionary and return true if
        it exists and false if it doesn't.
126      *
127      * @param k is the key to search for.
128      * @return If the key exists in the tree, it returns true, otherwise it
        returns false.
129      */
130     @Override
131     public boolean search(int k)
132     {
133         //Start searching at the root.
134         Node x = root;
135
136         //If the Node that is being searched is null, a dead end has been reached.
         If k == data, then a Node that fulfills the criteria has been found.
137         while (x != null && k != x.data)
138         {
139             //In here is is known that x is not null.
140             //If k is smaller than the key in the Node, then search to the left,
        otherwise search to the right.
141             if (k < x.data)
142             {
```

```
143                 x = x.left;
144             } else
145             {
146                 x = x.right;
147             }
148         }
149 
150         //Here the outcome is determined. If the above ended with null, then k was never found, and hence it isn't in the tree and false is returned. If x != null, then k was found in some Node, and true is returned.
151         if (x != null)
152         {
153             return true;
154         } else
155         {
156             return false;
157         }
158     }
159 
160 }
161 
162 /**
163  * Node class to represent a node in the binary tree.
164  */
165 class Node
166 {
167     Node left;
168     Node right;
169     int data;
170 
171     public Node(int data)
172     {
173         this.left = null;
174         this.right = null;
175         this.data = data;
176     }
177 }
```