Homework 7: AVLs

Graded

Student
Vidit Dharmendra Pokharna

Total Points
93 / 100 pts

Autograder Score
98.0 / 100.0

Failed Tests
Checkstyle (8/10)

Question 2

Feedback & Manual Grading **-5** / 0 pts

✔ **– 5 pts** Efficiency 1

[-2] checkstyle errors
[-5] efficiency sortedInBetween: should not traverse each side of the tree each time, only the side that contains data in between the bounds

Great work :) -Isabelle □‿□

Autograder Results

Autograder Output

If you're seeing this message, everything compiled and ran properly!
-CS1332 TAs

Checkstyle (8/10)

AVL.java:
line: 110 - Unused Javadoc tag. [JavadocMethodCheck]
line: 220, column65 - Expected @param tag for 'dummy'. [JavadocMethodCheck]

Submitted Files

AVL.java

Download

```java
import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.NoSuchElementException;

/**
 * Your implementation of an AVL.
 *
 * @author Vidit Pokharna
 * @version 1.0
 * @userid vpokharna3
 * @GTID 903772087
 *
 * Collaborators: LIST ALL COLLABORATORS YOU WORKED WITH HERE
 *
 * Resources: LIST ALL NON-COURSE RESOURCES YOU CONSULTED HERE
 */
public class AVL<T extends Comparable<? super T>> {

    // Do not add new instance variables or modify existing ones.
    private AVLNode<T> root;
    private int size;

    /**
     * Constructs a new AVL.
     *
     * This constructor should initialize an empty AVL.
     *
     * Since instance variables are initialized to their default values, there
     * is no need to do anything for this constructor.
     */
    public AVL() {
        // DO NOT IMPLEMENT THIS CONSTRUCTOR!
    }

    /**
     * Constructs a new AVL.
     *
     * This constructor should initialize the AVL with the data in the
     * Collection. The data should be added in the same order it is in the
     * Collection.
     *
     * @param data the data to add to the tree
     * @throws java.lang.IllegalArgumentException if data or any element in data
     *                                            is null
     */
```

```
47      public AVL(Collection<T> data) {
48          if (data == null) {
49              throw new IllegalArgumentException("List of data is null so unable to add to tree");
50          }
51          for (T t : data) {
52              if (t == null) {
53                  throw new IllegalArgumentException("Unable to add null data to tree");
54              }
55              add(t);
56          }
57      }
58
59      /**
60       * Adds the element to the tree.
61       *
62       * Start by adding it as a leaf like in a regular BST and then rotate the
63       * tree as necessary.
64       *
65       * If the data is already in the tree, then nothing should be done (the
66       * duplicate shouldn't get added, and size should not be incremented).
67       *
68       * Remember to recalculate heights and balance factors while going back
69       * up the tree after adding the element, making sure to rebalance if
70       * necessary.
71       *
72       * Hint: Should you use value equality or reference equality?
73       *
74       * @param data the data to add
75       * @throws java.lang.IllegalArgumentException if data is null
76       */
77      public void add(T data) {
78          if (data == null) {
79              throw new IllegalArgumentException("Unable to add null data to tree");
80          }
81          root = addHelper(root, data);
82      }
83
84      /**
85       * Helper method used for adding node to AVL
86       *
87       * @param node node within the tree
88       * @param data data to compare with
89       * @return root of the tree with data added
90       */
91      private AVLNode<T> addHelper(AVLNode<T> node, T data) {
92          if (node == null) {
93              size++;
94              return new AVLNode<T>(data);
95          }
```

```
96          int compareValue = node.getData().compareTo(data);
97          if (compareValue > 0) {
98              node.setLeft(addHelper(node.getLeft(), data));
99          } else if (compareValue < 0) {
100             node.setRight(addHelper(node.getRight(), data));
101         }
102         updateHeight(node);
103         return balance(node);
104     }
105
106     /**
107      * Helper method used for setting balance factor and height
108      *
109      * @param node node to set BF and height
110      * @return balance factor of the node being checked
111      */
112     private void updateHeight(AVLNode<T> node) {
113         int leftHeight = -1;
114         int rightHeight = -1;
115         if (node.getLeft() != null) {
116             leftHeight = node.getLeft().getHeight();
117         }
118         if (node.getRight() != null) {
119             rightHeight = node.getRight().getHeight();
120         }
121         node.setHeight(Math.max(leftHeight, rightHeight) + 1);
122         node.setBalanceFactor(leftHeight - rightHeight);
123     }
124
125     /**
126      * Helper method to rebalance the tree, calling the different rotations when necessary
127      *
128      * @param node root of the tree
129      * @return root of the tree after rebalancing has been done
130      */
131     private AVLNode<T> balance(AVLNode<T> node) {
132         if (node.getBalanceFactor() == -2) {
133             if (node.getRight().getBalanceFactor() > 0) {
134                 node.setRight(rightRotate(node.getRight()));
135             }
136             node = leftRotate(node);
137         } else if (node.getBalanceFactor() == 2) {
138             if (node.getLeft().getBalanceFactor() < 0) {
139                 node.setLeft(leftRotate(node.getLeft()));
140             }
141             node = rightRotate(node);
142         }
143         return node;
144     }
```

```
145
146     /**
147      * Left rotation of the tree
148      *
149      * @param node root of the tree
150      * @return node that replaces after rotation
151      */
152     private AVLNode<T> leftRotate(AVLNode<T> node) {
153         AVLNode<T> replace = node.getRight();
154         node.setRight(replace.getLeft());
155         replace.setLeft(node);
156         updateHeight(node);
157         updateHeight(replace);
158         return replace;
159     }
160
161     /**
162      * Right rotation of the tree
163      *
164      * @param node root of the tree
165      * @return node that replaces after rotation
166      */
167     private AVLNode<T> rightRotate(AVLNode<T> node) {
168         AVLNode<T> replace = node.getLeft();
169         node.setLeft(replace.getRight());
170         replace.setRight(node);
171         updateHeight(node);
172         updateHeight(replace);
173         return replace;
174     }
175
176     /**
177      * Removes and returns the element from the tree matching the given
178      * parameter.
179      *
180      * There are 3 cases to consider:
181      * 1: The node containing the data is a leaf (no children). In this case,
182      * simply remove it.
183      * 2: The node containing the data has one child. In this case, simply
184      * replace it with its child.
185      * 3: The node containing the data has 2 children. Use the predecessor to
186      * replace the data, NOT successor. As a reminder, rotations can occur
187      * after removing the predecessor node.
188      *
189      * Remember to recalculate heights and balance factors while going back
190      * up the tree after removing the element, making sure to rebalance if
191      * necessary.
192      *
193      * Do not return the same data that was passed in. Return the data that
```

```
194      * was stored in the tree.
195      *
196      * Hint: Should you use value equality or reference equality?
197      *
198      * @param data the data to remove
199      * @return the data that was removed
200      * @throws java.lang.IllegalArgumentException if data is null
201      * @throws java.util.NoSuchElementException   if the data is not found
202      */
203     public T remove(T data) {
204         if (data == null) {
205             throw new IllegalArgumentException("Unable to remove null data to tree");
206         }
207         AVLNode<T> dummy = new AVLNode<>(null);
208         root = removeHelper(root, dummy, data);
209         size--;
210         return dummy.getData();
211     }
212 
213     /**
214      * Helper method used for removing node from tree
215      *
216      * @param node node within the tree
217      * @param data data to compare with
218      * @return data matching parameter in get method
219      */
220     private AVLNode<T> removeHelper(AVLNode<T> node, AVLNode<T> dummy, T data) {
221         if (node == null) {
222             throw new NoSuchElementException("The data has not been found");
223         }
224         if (node.getData().equals(data)) {
225             dummy.setData(node.getData());
226             if (node.getRight() == null && node.getLeft() == null) {
227                 return null;
228             } else if (node.getRight() == null) {
229                 return node.getLeft();
230             } else if (node.getLeft() == null) {
231                 return node.getRight();
232             } else {
233                 AVLNode<T> dummy2 = new AVLNode<>(node.getData());
234                 node.setLeft(rPred(node.getLeft(), dummy2));
235                 node.setData(dummy2.getData());
236             }
237         } else if (data.compareTo(node.getData()) < 0) {
238             node.setLeft(removeHelper(node.getLeft(), dummy, data));
239         } else if (data.compareTo(node.getData()) > 0) {
240             node.setRight(removeHelper(node.getRight(), dummy, data));
241         }
242         updateHeight(node);
```

```
243         return balance(node);
244     }
245 
246     /**
247      * Helper method used for removing predecessor and placing into removed node
248      *
249      * @param node node within the tree
250      * @param parent data to compare with
251      * @return data matching parameter in get method
252      */
253     private AVLNode<T> rPred(AVLNode<T> node, AVLNode<T> parent) {
254         if (node.getRight() == null) {
255             parent.setData(node.getData());
256             return node.getLeft();
257         }
258         node.setRight(rPred(node.getRight(), parent));
259         updateHeight(node);
260         return balance(node);
261     }
262 
263     /**
264      * Returns the element from the tree matching the given parameter.
265      *
266      * Hint: Should you use value equality or reference equality?
267      *
268      * Do not return the same data that was passed in. Return the data that
269      * was stored in the tree.
270      *
271      * @param data the data to search for in the tree
272      * @return the data in the tree equal to the parameter
273      * @throws java.lang.IllegalArgumentException if data is null
274      * @throws java.util.NoSuchElementException   if the data is not in the tree
275      */
276     public T get(T data) {
277         if (data == null) {
278             throw new IllegalArgumentException("Unable to get null data from tree");
279         }
280         return getHelper(root, data);
281     }
282 
283     /**
284      * Helper method used for getting node from tree
285      *
286      * @param node node within the tree
287      * @param data data to compare with
288      * @return data matching parameter in get method
289      */
290     private T getHelper(AVLNode<T> node, T data) {
291         if (node == null) {
```

```
292             throw new NoSuchElementException("The data is not in the tree");
293         }
294         int compareValue = node.getData().compareTo(data);
295         if (compareValue > 0) {
296             return getHelper(node.getLeft(), data);
297         } else if (compareValue < 0) {
298             return getHelper(node.getRight(), data);
299         } else if (compareValue == 0) {
300             return node.getData();
301         }
302         return node.getData();
303     }
304 
305     /**
306      * Returns whether or not data matching the given parameter is contained
307      * within the tree.
308      *
309      * Hint: Should you use value equality or reference equality?
310      *
311      * @param data the data to search for in the tree.
312      * @return true if the parameter is contained within the tree, false
313      * otherwise
314      * @throws java.lang.IllegalArgumentException if data is null
315      */
316     public boolean contains(T data) {
317         if (data == null) {
318             throw new IllegalArgumentException("Unable to find null data in tree");
319         }
320         return containsHelper(root, data);
321     }
322 
323     /**
324      * Helper method used for checking if data in some node is in tree
325      *
326      * @param node node within the tree
327      * @param data data to compare with
328      * @return whether data is in tree
329      */
330     private boolean containsHelper(AVLNode<T> node, T data) {
331         if (node == null) {
332             return false;
333         }
334         int compareValue = node.getData().compareTo(data);
335         if (compareValue > 0) {
336             return containsHelper(node.getLeft(), data);
337         } else if (compareValue < 0) {
338             return containsHelper(node.getRight(), data);
339         } else if (compareValue == 0) {
340             return true;
```

```
341         }
342         return false;
343     }
344 
345     /**
346      * Returns the height of the root of the tree.
347      *
348      * Should be O(1).
349      *
350      * @return the height of the root of the tree, -1 if the tree is empty
351      */
352     public int height() {
353         if (root == null) {
354             return -1;
355         } else {
356             return root.getHeight();
357         }
358     }
359 
360     /**
361      * Clears the tree.
362      *
363      * Clears all data and resets the size.
364      */
365     public void clear() {
366         root = null;
367         size = 0;
368     }
369 
370     /**
371      * Returns the data on branches of the tree with the maximum depth. If you
372      * encounter multiple branches of maximum depth while traversing, then you
373      * should list the remaining data from the left branch first, then the
374      * remaining data in the right branch. This is essentially a preorder
375      * traversal of the tree, but only of the branches of maximum depth.
376      *
377      * This must be done recursively.
378      *
379      * Your list should not have duplicate data, and the data of a branch should be
380      * listed in order going from the root to the leaf of that branch.
381      *
382      * Should run in worst case O(n), but you should not explore branches that
383      * do not have maximum depth. You should also not need to traverse branches
384      * more than once.
385      *
386      * Hint: How can you take advantage of the balancing information stored in
387      * AVL nodes to discern deep branches?
388      *
389      * Example Tree:
```

```
390      *                    10
391      *                  /    \
392      *                 5      15
393      *               /  \    /  \
394      *              2    7  13   20
395      *             / \  / \   \ / \
396      *            1  4 6  8  14 17 25
397      *           /       \       \
398      *          0         9       30
399      *
400      * Returns: [10, 5, 2, 1, 0, 7, 8, 9, 15, 20, 25, 30]
401      *
402      * @return the list of data in branches of maximum depth in preorder
403      * traversal order
404      */
405     public List<T> deepestBranches() {
406         List<T> list = new ArrayList<T>();
407         rDeepBranch(root, list);
408         return list;
409     }
410
411     /**
412      * Recursive method to traverse the avl tree
413      *
414      * @param node the node that the recursive method will take to traverse the bst
415      * @param list list that will be added to
416      */
417     private void rDeepBranch(AVLNode<T> node, List<T> list) {
418         if (node == null) {
419             return;
420         } else {
421             list.add(node.getData());
422             if (node.getLeft() != null) {
423                 int difference = node.getHeight() - node.getLeft().getHeight();
424                 if (difference == 1 || difference == 0) {
425                     rDeepBranch(node.getLeft(), list);
426                 }
427             }
428             if (node.getRight() != null) {
429                 int difference = node.getHeight() - node.getRight().getHeight();
430                 if (difference == 1 || difference == 0) {
431                     rDeepBranch(node.getRight(), list);
432                 }
433             }
434         }
435     }
436
437     /**
438      * Returns a sorted list of data that are within the threshold bounds of
```

```
     * data1 and data2. That is, the data should be > data1 and < data2.
     *
     * This must be done recursively.
     *
     * Should run in worst case O(n), but this is heavily dependent on the
     * threshold data. You should not explore branches of the tree that do not
     * satisfy the threshold.
     *
     * Example Tree:
     *                       10
     *                   /       \
     *                  5         15
     *                /   \     /    \
     *               2     7   13    20
     *              / \   / \    \  / \
     *             1   4 6   8   14 17  25
     *            /         \         \
     *           0           9         30
     *
     * sortedInBetween(7, 14) returns [8, 9, 10, 13]
     * sortedInBetween(3, 8) returns [4, 5, 6, 7]
     * sortedInBetween(8, 8) returns []
     *
     * @param data1 the smaller data in the threshold
     * @param data2 the larger data in the threshold
     * @return a sorted list of data that is > data1 and < data2
     * @throws IllegalArgumentException if data1 or data2 are null
     * or if data1 > data2
     */
    public List<T> sortedInBetween(T data1, T data2) {
        if (data1 == null || data2 == null) {
            throw new IllegalArgumentException("The data given is null");
        } else if (data1.compareTo(data2) > 0) {
            throw new IllegalArgumentException("1st data input is greater than 2nd data input");
        }
        List<T> list = new ArrayList<T>();
        rSortBetween(root, list, data1, data2);
        return list;
    }

    /**
     * Recursive method to traverse the avl tree
     *
     * @param curr the node that the recursive method will take to traverse the bst
     * @param list the list of nodes forming the preorder traversal
     * @param data1 lower bound
     * @param data2 upper bound
     */
    private void rSortBetween(AVLNode<T> curr, List<T> list, T data1, T data2) {
```

```
488         if (curr != null) {
489             rSortBetween(curr.getLeft(), list, data1, data2);
490             if (curr.getData().compareTo(data1) > 0 && curr.getData().compareTo(data2) < 0) {
491                 list.add(curr.getData());
492             }
493             rSortBetween(curr.getRight(), list, data1, data2);
494         }
495     }
496 
497     /**
498      * Returns the root of the tree.
499      *
500      * For grading purposes only. You shouldn't need to use this method since
501      * you have direct access to the variable.
502      *
503      * @return the root of the tree
504      */
505     public AVLNode<T> getRoot() {
506         // DO NOT MODIFY THIS METHOD!
507         return root;
508     }
509 
510     /**
511      * Returns the size of the tree.
512      *
513      * For grading purposes only. You shouldn't need to use this method since
514      * you have direct access to the variable.
515      *
516      * @return the size of the tree
517      */
518     public int size() {
519         // DO NOT MODIFY THIS METHOD!
520         return size;
521     }
522 }
523 
```