

Homework 4: Binary Search Trees

● Graded

Student

Vidit Dharmendra Pokharna

Total Points

94 / 100 pts

Autograder Score

99.0 / 100.0

Failed Tests

Generics problems (-1/0)

Question 2

Feedback & Manual Grading

■ -5 / 0 pts

✓ - 5 pts Efficiency 1

💬 [-1] generics: missing generic line 395
[-5] k-largest efficiency: constructing in order traversal then taking sublist traverses more elements than what is strictly necessary

Great work :) -Isabelle ☺☺

Autograder Results

Autograder Output

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

Generics problems (-1/0)

Generics problem (line 395)

Submitted Files

```
1  import java.util.ArrayList;
2  import java.util.Collection;
3  import java.util.List;
4  import java.util.NoSuchElementException;
5  import java.util.Queue;
6  import java.util.LinkedList;
7
8  /**
9   * Your implementation of a BST.
10  *
11  * @author Vidit Pokharna
12  * @version 1.0
13  * @userid vpokharna3
14  * @GTID 903772087
15  *
16  * Collaborators:
17  *
18  * Resources:
19  */
20  public class BST<T extends Comparable<? super T>> {
21
22      /**
23       * Do not add new instance variables or modify existing ones.
24       */
25      private BSTNode<T> root;
26      private int size;
27
28      /**
29       * Constructs a new BST.
30       *
31       * This constructor should initialize an empty BST.
32       *
33       * Since instance variables are initialized to their default values, there
34       * is no need to do anything for this constructor.
35       */
36      public BST() {
37          // DO NOT IMPLEMENT THIS CONSTRUCTOR!
38      }
39
40      /**
41       * Constructs a new BST.
42       *
43       * This constructor should initialize the BST with the data in the
44       * Collection. The data should be added in the same order it is in the
45       * Collection.
46       *
```

```

47  * Hint: Not all Collections are indexable like Lists, so a regular for loop
48  * will not work here. However, all Collections are Iterable, so what type
49  * of loop would work?
50  *
51  * @param data the data to add
52  * @throws java.lang.IllegalArgumentException if data or any element in data
53  *           is null
54  */
55  public BST(Collection<T> data) {
56      if (data == null || data.contains(null)) {
57          throw new IllegalArgumentException("The collection is either null or contains a null value");
58      }
59      for (T t : data) {
60          add(t);
61      }
62  }
63
64  /**
65   * Adds the data to the tree.
66   *
67   * This must be done recursively.
68   *
69   * The data becomes a leaf in the tree.
70   *
71   * Traverse the tree to find the appropriate location. If the data is
72   * already in the tree, then nothing should be done (the duplicate
73   * shouldn't get added, and size should not be incremented).
74   *
75   * Must be O(log n) for best and average cases and O(n) for worst case.
76   *
77   * @param data the data to add
78   * @throws java.lang.IllegalArgumentException if data is null
79   */
80  public void add(T data) {
81      if (data == null) {
82          throw new IllegalArgumentException("The data is either null or contains a null value");
83      }
84      root = rAdd(root, data);
85  }
86
87  /**
88   * Private recursive method used for adding values to the tree
89   * @param curr dummy variable to represent a node
90   * @param data the data to add
91   * @return the node that will become the root
92   */
93  private BSTNode<T> rAdd(BSTNode<T> curr, T data) {
94      if (curr == null) {
95          curr = new BSTNode<T>(data);

```

```

96         size++;
97         return curr;
98     } else if (curr.getData().compareTo(data) > 0) {
99         curr.setLeft(rAdd(curr.getLeft(), data));
100    } else if (curr.getData().compareTo(data) < 0) {
101        curr.setRight(rAdd(curr.getRight(), data));
102    }
103    return curr;
104 }

105 /**
106  * Removes and returns the data from the tree matching the given parameter.
107  *
108  * This must be done recursively.
109  *
110  * There are 3 cases to consider:
111  * 1: The node containing the data is a leaf (no children). In this case,
112  * simply remove it.
113  * 2: The node containing the data has one child. In this case, simply
114  * replace it with its child.
115  * 3: The node containing the data has 2 children. Use the successor to
116  * replace the data. You MUST use recursion to find and remove the
117  * successor (you will likely need an additional helper method to
118  * handle this case efficiently).
119  *
120  * Do not return the same data that was passed in. Return the data that
121  * was stored in the tree.
122  *
123  * Hint: Should you use value equality or reference equality?
124  *
125  * Must be O(log n) for best and average cases and O(n) for worst case.
126  *
127  * @param data the data to remove
128  * @return the data that was removed
129  * @throws java.lang.IllegalArgumentException if data is null
130  * @throws java.util.NoSuchElementException if the data is not in the tree
131  */
132
133 public T remove(T data) {
134     if (data == null) {
135         throw new IllegalArgumentException("The data provided has a null value");
136     }
137     BSTNode<T> dummy = new BSTNode<T>(null);
138     root = rRemove(root, data, dummy);
139     size--;
140     return dummy.getData();
141 }

142 /**
143  * Private recursive method used for removing a certain node from the bst

```

```

145 *
146 * @param data the data to check for removal
147 * @param curr the node that is starting the traversal to find the node to remove
148 * @param dummy a node that will hold the data to remove
149 * @return the node that must be removed
150 */
151 private BSTNode<T> rRemove(BSTNode<T> curr, T data, BSTNode<T> dummy) {
152     if (curr == null) {
153         throw new NoSuchElementException("the data cannot be found in the tree");
154     } else {
155         if (curr.getData().compareTo(data) > 0) {
156             curr.setLeft(rRemove(curr.getLeft(), data, dummy));
157             return curr;
158         } else if (curr.getData().compareTo(data) < 0) {
159             curr.setRight(rRemove(curr.getRight(), data, dummy));
160             return curr;
161         } else if (curr.getData().compareTo(data) == 0) {
162             dummy.setData(curr.getData());
163             if (curr.getLeft() == null && curr.getRight() == null) {
164                 return null;
165             } else if (curr.getLeft() == null) {
166                 return curr.getRight();
167             } else if (curr.getRight() == null) {
168                 return curr.getLeft();
169             } else {
170                 BSTNode<T> dummy2 = new BSTNode<T>(null);
171                 curr.setRight(remSuccessor(curr.getRight(), dummy2));
172                 curr.setData(dummy2.getData());
173                 return curr;
174             }
175         }
176     }
177     return null;
178 }
179
180 /**
181  * Private recursive method used for replacing the current node with the predecessor
182  * @param node the node that is starting the traversal to find the node to remove
183  * @param dummy a node that will hold the data to remove
184  * @return the predecessor
185  */
186 private BSTNode<T> remSuccessor(BSTNode<T> node, BSTNode<T> dummy) {
187     if (node.getLeft() == null) {
188         dummy.setData(node.getData());
189         return node.getRight();
190     } else {
191         node.setLeft(remSuccessor(node.getLeft(), dummy));
192         return node;
193     }

```

```

194     }
195
196     /**
197     * Returns the data from the tree matching the given parameter.
198     *
199     * This must be done recursively.
200     *
201     * Do not return the same data that was passed in. Return the data that
202     * was stored in the tree.
203     *
204     * Hint: Should you use value equality or reference equality?
205     *
206     * Must be O(log n) for best and average cases and O(n) for worst case.
207     *
208     * @param data the data to search for
209     * @return the data in the tree equal to the parameter
210     * @throws java.lang.IllegalArgumentException if data is null
211     * @throws java.util.NoSuchElementException if the data is not in the tree
212     */
213     public T get(T data) {
214         if (data == null) {
215             throw new IllegalArgumentException("The data is either null or contains a null value");
216         } else if (!contains(data)) {
217             throw new NoSuchElementException("The BST does not contain the data");
218         }
219         return rGet(data, root).getData();
220     }
221
222     /**
223     * Private recursive method used for checking if values are in the tree
224     * @param data the data to add
225     * @param curr dummy variable to represent a node
226     * @return the current node, for which the data matches to
227     */
228     private BSTNode<T> rGet(T data, BSTNode<T> curr) {
229         if (curr.getData().compareTo(data) > 0) {
230             return rGet(data, curr.getLeft());
231         } else if (curr.getData().compareTo(data) < 0) {
232             return rGet(data, curr.getRight());
233         }
234         return curr;
235     }
236
237     /**
238     * Returns whether or not data matching the given parameter is contained
239     * within the tree.
240     *
241     * This must be done recursively.
242     *

```

```

243 * Hint: Should you use value equality or reference equality?
244 *
245 * Must be O(log n) for best and average cases and O(n) for worst case.
246 *
247 * @param data the data to search for
248 * @return true if the parameter is contained within the tree, false
249 * otherwise
250 * @throws java.lang.IllegalArgumentException if data is null
251 */
252 public boolean contains(T data) {
253     if (data == null) {
254         throw new IllegalArgumentException("the provided data has the value of null");
255     }
256     return rContains(data, root);
257 }
258
259 /**
260  * Private recursive method used for checking if values are in the tree
261  * @param data the data to add
262  * @param curr dummy variable to represent a node
263  * @return whether the bst contains the data
264  */
265 private boolean rContains(T data, BSTNode<T> curr) {
266     if (curr == null) {
267         return false;
268     } else if (curr.getData().equals(data)) {
269         return true;
270     } else if (curr.getData().compareTo(data) > 0) {
271         return rContains(data, curr.getLeft());
272     } else if (curr.getData().compareTo(data) < 0) {
273         return rContains(data, curr.getRight());
274     }
275     return false;
276 }
277
278 /**
279  * Generate a pre-order traversal of the tree.
280  *
281  * This must be done recursively.
282  *
283  * Must be O(n).
284  *
285  * @return the preorder traversal of the tree
286  */
287 public List<T> preorder() {
288     List<T> list = new ArrayList<>();
289     if (root == null) {
290         return list;
291     }

```

```

292     rPreorder(root, list);
293     return list;
294 }
295
296 /**
297  * Recursive method to traverse the bst in preorder
298  *
299  * @param curr the node that the recursive method will take to traverse the bst
300  * @param list the list of nodes forming the preorder traversal
301  */
302 private void rPreorder(BSTNode<T> curr, List<T> list) {
303     if (curr != null) {
304         list.add(curr.getData());
305         rPreorder(curr.getLeft(), list);
306         rPreorder(curr.getRight(), list);
307     }
308 }
309
310 /**
311  * Generate an in-order traversal of the tree.
312  *
313  * This must be done recursively.
314  *
315  * Must be O(n).
316  *
317  * @return the inorder traversal of the tree
318  */
319 public List<T> inorder() {
320     List<T> list = new ArrayList<>();
321     if (root == null) {
322         return list;
323     }
324     rInorder(root, list);
325     return list;
326 }
327
328 /**
329  * Recursive method to traverse the bst in inorder
330  *
331  * @param curr the node that the recursive method will take to traverse the bst
332  * @param list the list of nodes forming the inorder traversal
333  */
334 private void rInorder(BSTNode<T> curr, List<T> list) {
335     if (curr != null) {
336         rInorder(curr.getLeft(), list);
337         list.add(curr.getData());
338         rInorder(curr.getRight(), list);
339     }
340 }

```



```

341
342 /**
343  * Generate a post-order traversal of the tree.
344  *
345  * This must be done recursively.
346  *
347  * Must be O(n).
348  *
349  * @return the postorder traversal of the tree
350  */
351 public List<T> postorder() {
352     List<T> list = new ArrayList<>();
353     if (root == null) {
354         return list;
355     }
356     rPostorder(root, list);
357     return list;
358 }
359
360 /**
361  * Recursive method to traverse the bst in postorder
362  *
363  * @param curr the node that the recursive method will take to traverse the bst
364  * @param list the list of nodes forming the postorder traversal
365  */
366 private void rPostorder(BSTNode<T> curr, List<T> list) {
367     if (curr != null) {
368         rPostorder(curr.getLeft(), list);
369         rPostorder(curr.getRight(), list);
370         list.add(curr.getData());
371     }
372 }
373
374 /**
375  * Generate a level-order traversal of the tree.
376  *
377  * This does not need to be done recursively.
378  *
379  * Hint: You will need to use a queue of nodes. Think about what initial
380  * node you should add to the queue and what loop / loop conditions you
381  * should use.
382  *
383  * Must be O(n).
384  *
385  * @return the level order traversal of the tree
386  */
387 public List<T> levelorder() {
388     List<T> list = new ArrayList<>();
389     if (root == null) {

```

```

390     return list;
391 }
392 Queue<BSTNode<T>> queue = new LinkedList<BSTNode<T>>();
393 queue.add(root);
394 while (!queue.isEmpty()) {
395     BSTNode temp = queue.poll();
396     list.add((T) temp.getData());
397     if (temp.getLeft() != null) {
398         queue.add(temp.getLeft());
399     }
400     if (temp.getRight() != null) {
401         queue.add(temp.getRight());
402     }
403 }
404 return list;
405 }
406
407 /**
408  * Returns the height of the root of the tree.
409  *
410  * This must be done recursively.
411  *
412  * A node's height is defined as max(left.height, right.height) + 1. A
413  * leaf node has a height of 0 and a null child has a height of -1.
414  *
415  * Must be O(n).
416  *
417  * @return the height of the root of the tree, -1 if the tree is empty
418  */
419 public int height() {
420     if (root == null) {
421         return -1;
422     }
423     return rHeight(root);
424 }
425
426 /**
427  * Private recursive method to return heights throughout the tree
428  * @param curr the node that will begin the traversal to determine the height
429  * @return the height of the root
430  */
431 private int rHeight(BSTNode<T> curr) {
432     int left = curr.getLeft() != null ? rHeight(curr.getLeft()) : -1;
433     int right = curr.getRight() != null ? rHeight(curr.getRight()) : -1;
434     return Math.max(left, right) + 1;
435 }
436
437 /**
438  * Clears the tree.

```

```

439 *
440 * Clears all data and resets the size.
441 *
442 * Must be O(1).
443 */
444 public void clear() {
445     root = null;
446     size = 0;
447 }
448
449 /**
450 * Finds and retrieves the k-largest elements from the BST in sorted order,
451 * least to greatest.
452 *
453 * This must be done recursively.
454 *
455 * In most cases, this method will not need to traverse the entire tree to
456 * function properly, so you should only traverse the branches of the tree
457 * necessary to get the data and only do so once. Failure to do so will
458 * result in an efficiency penalty.
459 *
460 * EXAMPLE: Given the BST below composed of Integers:
461 *
462 *         50
463 *       /  \
464 *      25   75
465 *     /  \
466 *    12  37
467 *   / \  \
468 *  10 15 40
469 *   /
470 *  13
471 *
472 * kLargest(5) should return the list [25, 37, 40, 50, 75].
473 * kLargest(3) should return the list [40, 50, 75].
474 *
475 * Should have a running time of O(log(n) + k) for a balanced tree and a
476 * worst case of O(n + k), with n being the number of data in the BST
477 *
478 * @param k the number of largest elements to return
479 * @return sorted list consisting of the k largest elements
480 * @throws java.lang.IllegalArgumentException if k < 0 or k > size
481 */
482 public List<T> kLargest(int k) {
483     if (k < 0 || k > size) {
484         throw new IllegalArgumentException("the k value provided is not valid");
485     }
486     List<T> list = inorder();
487     list = list.subList(size - k, size);

```

```
488     return list;
489 }
490
491
492 /**
493  * Returns the root of the tree.
494  *
495  * For grading purposes only. You shouldn't need to use this method since
496  * you have direct access to the variable.
497  *
498  * @return the root of the tree
499  */
500 public BSTNode<T> getRoot() {
501     // DO NOT MODIFY THIS METHOD!
502     return root;
503 }
504
505 /**
506  * Returns the size of the tree.
507  *
508  * For grading purposes only. You shouldn't need to use this method since
509  * you have direct access to the variable.
510  *
511  * @return the size of the tree
512  */
513 public int size() {
514     // DO NOT MODIFY THIS METHOD!
515     return size;
516 }
517 }
518
```