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

BST.java

Download

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

/**
 * Your implementation of a BST.
 *
 * @author Vidit Pokharna
 * @version 1.0
 * @userid vpokharna3
 * @GTID 903772087
 *
 * Collaborators:
 *
 * Resources:
 */
public class BST<T extends Comparable<? super T>> {

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

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

    /**
     * Constructs a new BST.
     *
     * This constructor should initialize the BST with the data in the
     * Collection. The data should be added in the same order it is in the
     * Collection.
     *
```

```java
     * Hint: Not all Collections are indexable like Lists, so a regular for loop
     * will not work here. However, all Collections are Iterable, so what type
     * of loop would work?
     *
     * @param data the data to add
     * @throws java.lang.IllegalArgumentException if data or any element in data
     *                                            is null
     */
    public BST(Collection<T> data) {
        if (data == null || data.contains(null)) {
            throw new IllegalArgumentException("The collection is either null or contains a null value");
        }
        for (T t : data) {
            add(t);
        }
    }

    /**
     * Adds the data to the tree.
     *
     * This must be done recursively.
     *
     * The data becomes a leaf in the tree.
     *
     * Traverse the tree to find the appropriate location. If the data is
     * already in the tree, then nothing should be done (the duplicate
     * shouldn't get added, and size should not be incremented).
     *
     * Must be O(log n) for best and average cases and O(n) for worst case.
     *
     * @param data the data to add
     * @throws java.lang.IllegalArgumentException if data is null
     */
    public void add(T data) {
        if (data == null) {
            throw new IllegalArgumentException("The data is either null or contains a null value");
        }
        root = rAdd(root, data);
    }

    /**
     * Private recursive method used for adding values to the tree
     * @param curr dummy variable to represent a node
     * @param data the data to add
     * @return the node that will become the root
     */
    private BSTNode<T> rAdd(BSTNode<T> curr, T data) {
        if (curr == null) {
            curr = new BSTNode<T>(data);
```

```java
            size++;
            return curr;
        } else if (curr.getData().compareTo(data) > 0) {
            curr.setLeft(rAdd(curr.getLeft(), data));
        } else if (curr.getData().compareTo(data) < 0) {
            curr.setRight(rAdd(curr.getRight(), data));
        }
        return curr;
    }

    /**
     * Removes and returns the data from the tree matching the given parameter.
     *
     * This must be done recursively.
     *
     * There are 3 cases to consider:
     * 1: The node containing the data is a leaf (no children). In this case,
     * simply remove it.
     * 2: The node containing the data has one child. In this case, simply
     * replace it with its child.
     * 3: The node containing the data has 2 children. Use the successor to
     * replace the data. You MUST use recursion to find and remove the
     * successor (you will likely need an additional helper method to
     * handle this case efficiently).
     *
     * Do not return the same data that was passed in. Return the data that
     * was stored in the tree.
     *
     * Hint: Should you use value equality or reference equality?
     *
     * Must be O(log n) for best and average cases and O(n) for worst case.
     *
     * @param data the data to remove
     * @return the data that was removed
     * @throws java.lang.IllegalArgumentException if data is null
     * @throws java.util.NoSuchElementException   if the data is not in the tree
     */
    public T remove(T data) {
        if (data == null) {
            throw new IllegalArgumentException("The data provided has a null value");
        }
        BSTNode<T> dummy = new BSTNode<T>(null);
        root = rRemove(root, data, dummy);
        size--;
        return dummy.getData();
    }

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

```java
     *
     * @param data the data to check for removal
     * @param curr the node that is starting the traversal to find the node to remove
     * @param dummy a node that will hold the data to remove
     * @return the node that must be removed
     */
    private BSTNode<T> rRemove(BSTNode<T> curr, T data, BSTNode<T> dummy) {
        if (curr == null) {
            throw new NoSuchElementException("the data cannot be found in the tree");
        } else {
            if (curr.getData().compareTo(data) > 0) {
                curr.setLeft(rRemove(curr.getLeft(), data, dummy));
                return curr;
            } else if (curr.getData().compareTo(data) < 0) {
                curr.setRight(rRemove(curr.getRight(), data, dummy));
                return curr;
            } else if (curr.getData().compareTo(data) == 0) {
                dummy.setData(curr.getData());
                if (curr.getLeft() == null && curr.getRight() == null) {
                    return null;
                } else if (curr.getLeft() == null) {
                    return curr.getRight();
                } else if (curr.getRight() == null) {
                    return curr.getLeft();
                } else {
                    BSTNode<T> dummy2 = new BSTNode<T>(null);
                    curr.setRight(remSuccessor(curr.getRight(), dummy2));
                    curr.setData(dummy2.getData());
                    return curr;
                }
            }
        }
        return null;
    }

    /**
     * Private recursive method used for replacing the current node with the predecessor
     * @param node the node that is starting the traversal to find the node to remove
     * @param dummy a node that will hold the data to remove
     * @return the predecessor
     */
    private BSTNode<T> remSuccessor(BSTNode<T> node, BSTNode<T> dummy) {
        if (node.getLeft() == null) {
            dummy.setData(node.getData());
            return node.getRight();
        } else {
            node.setLeft(remSuccessor(node.getLeft(), dummy));
            return node;
        }
```

```java
    }

    /**
     * Returns the data from the tree matching the given parameter.
     *
     * This must be done recursively.
     *
     * Do not return the same data that was passed in. Return the data that
     * was stored in the tree.
     *
     * Hint: Should you use value equality or reference equality?
     *
     * Must be O(log n) for best and average cases and O(n) for worst case.
     *
     * @param data the data to search for
     * @return the data in the tree equal to the parameter
     * @throws java.lang.IllegalArgumentException if data is null
     * @throws java.util.NoSuchElementException   if the data is not in the tree
     */
    public T get(T data) {
        if (data == null) {
            throw new IllegalArgumentException("The data is either null or contains a null value");
        } else if (!contains(data)) {
            throw new NoSuchElementException("The BST does not contain the data");
        }
        return rGet(data, root).getData();
    }

    /**
     * Private recursive method used for checking if values are in the tree
     * @param data the data to add
     * @param curr dummy variable to represent a node
     * @return the current node, for which the data matches to
     */
    private BSTNode<T> rGet(T data, BSTNode<T> curr) {
        if (curr.getData().compareTo(data) > 0) {
            return rGet(data, curr.getLeft());
        } else if (curr.getData().compareTo(data) < 0) {
            return rGet(data, curr.getRight());
        }
        return curr;
    }

    /**
     * Returns whether or not data matching the given parameter is contained
     * within the tree.
     *
     * This must be done recursively.
     *
```

```
     * Hint: Should you use value equality or reference equality?
     *
     * Must be O(log n) for best and average cases and O(n) for worst case.
     *
     * @param data the data to search for
     * @return true if the parameter is contained within the tree, false
     * otherwise
     * @throws java.lang.IllegalArgumentException if data is null
     */
    public boolean contains(T data) {
        if (data == null) {
            throw new IllegalArgumentException("the provided data has the value of null");
        }
        return rContains(data, root);
    }

    /**
     * Private recursive method used for checking if values are in the tree
     * @param data the data to add
     * @param curr dummy variable to represent a node
     * @return whether the bst contains the data
     */
    private boolean rContains(T data, BSTNode<T> curr) {
        if (curr == null) {
            return false;
        } else if (curr.getData().equals(data)) {
            return true;
        } else if (curr.getData().compareTo(data) > 0) {
            return rContains(data, curr.getLeft());
        } else if (curr.getData().compareTo(data) < 0) {
            return rContains(data, curr.getRight());
        }
        return false;
    }

    /**
     * Generate a pre-order traversal of the tree.
     *
     * This must be done recursively.
     *
     * Must be O(n).
     *
     * @return the preorder traversal of the tree
     */
    public List<T> preorder() {
        List<T> list = new ArrayList<>();
        if (root == null) {
            return list;
        }
```

```java
        rPreorder(root, list);
        return list;
    }

    /**
     * Recursive method to traverse the bst in preorder
     *
     * @param curr the node that the recursive method will take to traverse the bst
     * @param list the list of nodes forming the preorder traversal
     */
    private void rPreorder(BSTNode<T> curr, List<T> list) {
        if (curr != null) {
            list.add(curr.getData());
            rPreorder(curr.getLeft(), list);
            rPreorder(curr.getRight(), list);
        }
    }

    /**
     * Generate an in-order traversal of the tree.
     *
     * This must be done recursively.
     *
     * Must be O(n).
     *
     * @return the inorder traversal of the tree
     */
    public List<T> inorder() {
        List<T> list = new ArrayList<>();
        if (root == null) {
            return list;
        }
        rInorder(root, list);
        return list;
    }

    /**
     * Recursive method to traverse the bst in inorder
     *
     * @param curr the node that the recursive method will take to traverse the bst
     * @param list the list of nodes forming the inorder traversal
     */
    private void rInorder(BSTNode<T> curr, List<T> list) {
        if (curr != null) {
            rInorder(curr.getLeft(), list);
            list.add(curr.getData());
            rInorder(curr.getRight(), list);
        }
    }
```

```java

    /**
     * Generate a post-order traversal of the tree.
     *
     * This must be done recursively.
     *
     * Must be O(n).
     *
     * @return the postorder traversal of the tree
     */
    public List<T> postorder() {
        List<T> list = new ArrayList<>();
        if (root == null) {
            return list;
        }
        rPostorder(root, list);
        return list;
    }

    /**
     * Recursive method to traverse the bst in postorder
     *
     * @param curr the node that the recursive method will take to traverse the bst
     * @param list the list of nodes forming the postorder traversal
     */
    private void rPostorder(BSTNode<T> curr, List<T> list) {
        if (curr != null) {
            rPostorder(curr.getLeft(), list);
            rPostorder(curr.getRight(), list);
            list.add(curr.getData());
        }
    }

    /**
     * Generate a level-order traversal of the tree.
     *
     * This does not need to be done recursively.
     *
     * Hint: You will need to use a queue of nodes. Think about what initial
     * node you should add to the queue and what loop / loop conditions you
     * should use.
     *
     * Must be O(n).
     *
     * @return the level order traversal of the tree
     */
    public List<T> levelorder() {
        List<T> list = new ArrayList<>();
        if (root == null) {
```

```
            return list;
        }
        Queue<BSTNode<T>> queue = new LinkedList<BSTNode<T>>();
        queue.add(root);
        while (!queue.isEmpty()) {
            BSTNode temp = queue.poll();
            list.add((T) temp.getData());
            if (temp.getLeft() != null) {
                queue.add(temp.getLeft());
            }
            if (temp.getRight() != null) {
                queue.add(temp.getRight());
            }
        }
        return list;
    }

    /**
     * Returns the height of the root of the tree.
     *
     * This must be done recursively.
     *
     * A node's height is defined as max(left.height, right.height) + 1. A
     * leaf node has a height of 0 and a null child has a height of -1.
     *
     * Must be O(n).
     *
     * @return the height of the root of the tree, -1 if the tree is empty
     */
    public int height() {
        if (root == null) {
            return -1;
        }
        return rHeight(root);
    }

    /**
     * Private recursive method to return heights throughout the tree
     * @param curr the node that will begin the traversal to determine the height
     * @return the height of the root
     */
    private int rHeight(BSTNode<T> curr) {
        int left = curr.getLeft() != null ? rHeight(curr.getLeft()) : -1;
        int right = curr.getRight() != null ? rHeight(curr.getRight()) : -1;
        return Math.max(left, right) + 1;
    }

    /**
     * Clears the tree.
```

```
     *
     * Clears all data and resets the size.
     *
     * Must be O(1).
     */
    public void clear() {
        root = null;
        size = 0;
    }

    /**
     * Finds and retrieves the k-largest elements from the BST in sorted order,
     * least to greatest.
     *
     * This must be done recursively.
     *
     * In most cases, this method will not need to traverse the entire tree to
     * function properly, so you should only traverse the branches of the tree
     * necessary to get the data and only do so once. Failure to do so will
     * result in an efficiency penalty.
     *
     * EXAMPLE: Given the BST below composed of Integers:
     *
     *                50
     *              /    \
     *            25      75
     *           /  \
     *          12   37
     *         /  \    \
     *        10  15    40
     *           /
     *          13
     *
     * kLargest(5) should return the list [25, 37, 40, 50, 75].
     * kLargest(3) should return the list [40, 50, 75].
     *
     * Should have a running time of O(log(n) + k) for a balanced tree and a
     * worst case of O(n + k), with n being the number of data in the BST
     *
     * @param k the number of largest elements to return
     * @return sorted list consisting of the k largest elements
     * @throws java.lang.IllegalArgumentException if k < 0 or k > size
     */
    public List<T> kLargest(int k) {
        if (k < 0 || k > size) {
            throw new IllegalArgumentException("the k value provided is not valid");
        }
        List<T> list = inorder();
        list = list.subList(size - k, size);
```

```java
        return list;
    }


    /**
     * Returns the root of the tree.
     *
     * For grading purposes only. You shouldn't need to use this method since
     * you have direct access to the variable.
     *
     * @return the root of the tree
     */
    public BSTNode<T> getRoot() {
        // DO NOT MODIFY THIS METHOD!
        return root;
    }

    /**
     * Returns the size of the tree.
     *
     * For grading purposes only. You shouldn't need to use this method since
     * you have direct access to the variable.
     *
     * @return the size of the tree
     */
    public int size() {
        // DO NOT MODIFY THIS METHOD!
        return size;
    }
}
```