The Ohio State University

PROJECT 4: SET ON BINARY SEARCH TREES

Daniil Gofman

Ansh Pachauri

SW 2: Dev & Dsgn

Paolo Bucci

Yiyang Chen

Shivam Gupta

September 29, 2023

```java
import java.util.Iterator;

import components.binarytree.BinaryTree;
import components.binarytree.BinaryTree1;
import components.set.Set;
import components.set.SetSecondary;

/**
 * {@code Set} represented as a {@code BinaryTree} (maintained as a binary
 * search tree) of elements with implementations of primary methods.
 *
 * @param <T>
 *            type of {@code Set} elements
 * @mathdefinitions <pre>
 * IS_BST(
 *   tree: binary tree of T
 *  ): boolean satisfies
 *  [tree satisfies the binary search tree properties as described in the
 *   slides with the ordering reported by compareTo for T, including that
 *   it has no duplicate labels]
 * </pre>
 * @convention IS_BST($this.tree)
 * @correspondence this = labels($this.tree)
 *
 * @author Daniil Gofman, Ansh Pachauri
 *
 */
public class Set3a<T extends Comparable<T>> extends SetSecondary<T> {

    /*
     * Private members -------------------------------------------------------
     */

    /**
     * Elements included in {@code this}.
     */
    private BinaryTree<T> tree;

    /**
     * Returns whether {@code x} is in {@code t}.
     *
     * @param <T>
     *            type of {@code BinaryTree} labels
     * @param t
     *            the {@code BinaryTree} to be searched
     * @param x
     *            the label to be searched for
     * @return true if t contains x, false otherwise
     * @requires IS_BST(t)
     * @ensures isInTree = (x is in labels(t))
     */
    private static <T extends Comparable<T>> boolean isInTree(BinaryTree<T> t,
            T x) {
        assert t != null : "Violation of: t is not null";
        assert x != null : "Violation of: x is not null";
        // initialize variables
        boolean result = false;
        // check if tree is empty
        if (t.size() > 0) {
            BinaryTree<T> left = t.newInstance();
            BinaryTree<T> right = t.newInstance();
            T root = t.disassemble(left, right);
            // compare x with the root and check for x in the
            // appropriate node
            if (root.equals(x)) {
                result = true;
            } else if (root.compareTo(x) < 0) {
                result = isInTree(right, x);
            } else {
                result = isInTree(left, x);
            }
            // reassemble tree
            t.assemble(root, left, right);
        }
        // return if x is in the tree
        return result;
    }

    /**
     * Inserts {@code x} in {@code t}.
     *
     * @param <T>
     *            type of {@code BinaryTree} labels
     * @param t
     *            the {@code BinaryTree} to be searched
     * @param x
     *            the label to be inserted
     * @aliases reference {@code x}
     * @updates t
     * @requires IS_BST(t) and x is not in labels(t)
     * @ensures IS_BST(t) and labels(t) = labels(#t) union {x}
     */
    private static <T extends Comparable<T>> void insertInTree(BinaryTree<T> t,
            T x) {
        assert t != null : "Violation of: t is not null";
        assert x != null : "Violation of: x is not null";
        // Initialize variables
        BinaryTree<T> left = t.newInstance();
        BinaryTree<T> right = t.newInstance();
        // Check if the tree is empty
        if (t.size() > 0) {
            T root = t.disassemble(left, right);
            // Check if the root is greater than,
            // Or less than x and add to the appropriate node.
            if (root.compareTo(x) > 0) {
                insertInTree(left, x);
            } else {
                insertInTree(right, x);
            }
            // Reassemble the tree
            t.assemble(root, left, right);
            // If the tree is empty
        } else {
            // Reassemble the tree with x as the root
            t.assemble(x, left, right);
        }

    }

    /**
     * Removes and returns the smallest (left-most) label in {@code t}.
     *
     * @param <T>
     *            type of {@code BinaryTree} labels
     * @param t
     *            the {@code BinaryTree} from which to remove the label
     * @return the smallest label in the given {@code BinaryTree}
     * @updates t
     * @requires IS_BST(t) and |t| > 0
     * @ensures <pre>
     * IS_BST(t)  and  removeSmallest = [the smallest label in #t]  and
     *  labels(t) = labels(#t) \ {removeSmallest}
     * </pre>
     */
    private static <T> T removeSmallest(BinaryTree<T> t) {
        assert t != null : "Violation of: t is not null";
        assert t.size() > 0 : "Violation of: |t| > 0";

        // Initialize variables to store left and right children
        BinaryTree<T> left = t.newInstance();
        BinaryTree<T> right = t.newInstance();
        // Get root of the tree
        T root = t.disassemble(left, right);
        T smallest = root;
        if (left.size() > 0) {
            smallest = removeSmallest(left);
            t.assemble(root, left, right);
        } else {
            t.transferFrom(right);
        }
        // Return the smallest element
        return smallest;
    }

    /**
     * Finds label {@code x} in {@code t}, removes it from {@code t}, and
     * returns it.
     *
     * @param <T>
     *            type of {@code BinaryTree} labels
     * @param t
     *            the {@code BinaryTree} from which to remove label {@code x}
     * @param x
     *            the label to be removed
     * @return the removed label
     * @updates t
     * @requires IS_BST(t) and x is in labels(t)
     * @ensures <pre>
     * IS_BST(t)  and  removeFromTree = x  and
     *  labels(t) = labels(#t) \ {x}
     * </pre>
     */
    private static <T extends Comparable<T>> T removeFromTree(BinaryTree<T> t,
            T x) {
        assert t != null : "Violation of: t is not null";
        assert x != null : "Violation of: x is not null";
        assert t.size() > 0 : "Violation of: x is in labels(t)";

        // Create new binary trees for the left and right subtrees.
        BinaryTree<T> right = t.newInstance();
        BinaryTree<T> left = t.newInstance();

        // Disassemble the original tree 't' and get the root element.
        T root = t.disassemble(left, right);

        // Create a variable to store the element that will be removed.
        T remove = root;

        // Check if the root element is equal to the element 'x'.
        if (root.equals(x)) {
            // If 'x' is found at the root, handle the removal case.
            // If the right subtree is not empty, find the smallest element
            // in the right subtree and set it as the new root.
            if (right.size() > 0) {
                root = removeSmallest(right);
                t.assemble(root, left, right);
            } else {
                // If the right subtree is empty, transfer the left subtree to 't'.
                t.transferFrom(left);
            }
        } else {
            // If 'x' is not equal to the root element, recursively search for 'x'
            // in either the left or right subtree based on the comparison.

            if (root.compareTo(x) < 0) {
                // If 'x' is greater than the root element, search in the right subtree.
                remove = removeFromTree(right, x);
            } else {
                // If 'x' is less than the root element, search in the left subtree.
                remove = removeFromTree(left, x);
            }

            // Assemble the tree after the recursive call.
            t.assemble(root, left, right);
        }

        // Return the element that was removed.
        return remove;
    }

    /**
     * Creator of initial representation.
     */
    private void createNewRep() {

        // Create new representation
        this.tree = new BinaryTree1<T>();
    }

    /*
     * Constructors -----------------------------------------------------------
     */

    /**
     * No-argument constructor.
     */
    public Set3a() {

        // Default constructor
        this.createNewRep();
    }

    /*
     * Standard methods
-------------------------------------------------------
     */

    @SuppressWarnings("unchecked")
    @Override
    public final Set<T> newInstance() {
        try {
            return this.getClass().getConstructor().newInstance();
        } catch (ReflectiveOperationException e) {
            throw new AssertionError(
                    "Cannot construct object of type " + this.getClass());
        }
    }

    @Override
    public final void clear() {
        this.createNewRep();
    }

    @Override
    public final void transferFrom(Set<T> source) {
        assert source != null : "Violation of: source is not null";
        assert source != this : "Violation of: source is not this";
        assert source instanceof Set3a<?> : ""
                + "Violation of: source is of dynamic type Set3<?>";
        /*
         * This cast cannot fail since the assert above would have stopped
         * execution in that case: source must be of dynamic type Set3a<?>,
and
         * the ? must be T or the call would not have compiled.
         */
        Set3a<T> localSource = (Set3a<T>) source;
        this.tree = localSource.tree;
        localSource.createNewRep();
    }

    /*
     * Kernel methods
---------------------------------------------------------
     */

    @Override
    public final void add(T x) {
        assert x != null : "Violation of: x is not null";
        assert !this.contains(x) : "Violation of: x is not in this";

```

```java
        // Inserts element x into a BST
        insertInTree(this.tree, x);
    }

    @Override
    public final T remove(T x) {
        assert x != null : "Violation of: x is not null";
        assert this.contains(x) : "Violation of: x is in this";

        // Remove and return element from the tree
        return removeFromTree(this.tree, x);
    }

    @Override
    public final T removeAny() {
        assert this.size() > 0 : "Violation of: this /= empty_set";
        // Removes the smallest term from the tree
        return removeSmallest(this.tree);
    }

    @Override
    public final boolean contains(T x) {
        assert x != null : "Violation of: x is not null";
        // checks if x is in the tree
        return isInTree(this.tree, x);
    }

    @Override
    public final int size() {
        // return the size of the tree
        return this.tree.size();
    }

    @Override
    public final Iterator<T> iterator() {
        return this.tree.iterator();
    }

}
```