```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 David P. and Zach B.
 *
 */
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";

        boolean found = false;

        // If the tree is not empty, proceed to check
        if (t.size() > 0) {
            BinaryTree<T> left = t.newInstance();
            BinaryTree<T> right = t.newInstance();
            T root = t.root();
            t.disassemble(left, right);
            // Compare the element with the root
            int compareResult = x.compareTo(root);
            if (compareResult == 0) {
                found = true; // Element found
            } else if (compareResult < 0) {
                found = isInTree(left, x); // Search in the left subtree
            } else {
                found = isInTree(right, x); // Search in the right subtree
            }
            // Reassemble the tree after checking
            t.assemble(root, left, right);
        }

        return found;
    }

    /**
     * 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";
        // If the tree is empty, insert the element as the root
        if (t.size() == 0) {
            t.assemble(x, t.newInstance(), t.newInstance());
        } else {
            BinaryTree<T> left = t.newInstance();
            BinaryTree<T> right = t.newInstance();
            T root = t.root();
            t.disassemble(left, right);
            // Compare the element to be inserted with the root
            int compareResult = x.compareTo(root);
            // Insert into the appropriate subtree
            if (compareResult < 0) {
                insertInTree(left, x);
            } else {
                insertInTree(right, x);
            }
```

```java
            // Reassemble the tree with the updated subtree

            t.assemble(root, 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";

        T smallest;
        BinaryTree<T> left = t.newInstance();
        BinaryTree<T> right = t.newInstance();
        T root = t.root();
        t.disassemble(left, right);

        // If the left subtree is empty, the root is the smallest element
        if (left.size() == 0) {
            smallest = root;
            t.transferFrom(right);
        } else {
            // Recursively find the smallest element in the left subtree
            smallest = removeSmallest(left);
            t.assemble(root, left, right);
        }

        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)";

        // type of variable to hold removed element
        T removed;
        // create new instances of left and right subtrees
        BinaryTree<T> left = t.newInstance();
        BinaryTree<T> right = t.newInstance();

        // get root of tree
        T root = t.root();
        t.disassemble(left, right);

        int compareResult = x.compareTo(root);
        // if element is equal to the root
        if (compareResult == 0) {
            removed = root;
            // if right subtree is empty, transfer left to the tree.
            if (right.size() == 0) {
                t.transferFrom(left);
            } else if (left.size() == 0) {
                // if left subtree is empty, transfer the right subtree to the tree.
                t.transferFrom(right);
                // if both are not empty
            } else {
                // remove smallest element from right subtree
                T smallestInRight = removeSmallest(right);
                t.assemble(smallestInRight, left, right);
            }
            // If the element to be removed is less than the root
        } else if (compareResult < 0) {
            // recursively remove element from left tree
            removed = removeFromTree(left, x);
            // reassemble tree
            t.assemble(root, left, right);
        } else {
            // recursively remove element from right tree.
            removed = removeFromTree(right, x);
            t.assemble(root, left, right);
        }

        return removed;
    }

    /**
     * Creator of initial representation.
     */
    private void createNewRep() {
        this.tree = new BinaryTree1<>();
    }

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

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

        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";

        return removeFromTree(this.tree, x);
    }

    @Override
    public final T removeAny() {
        assert this.size() > 0 : "Violation of: this /= empty_set";

        return removeSmallest(this.tree);
    }

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

        return isInTree(this.tree, x);
    }

    @Override
    public final int size() {

        return this.tree.size();

    }

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

}

```