```java
import java.util.Iterator;


/**
 * Simple but somewhat efficient implementation of IDnaStrand. \ This
 * implementation uses StringBuilders to represent genomic/DNA data.
 *
 * @author ola
 * @date January 2008, modified and commented September 2008
 * @date October 2011, made myInfo a StringBuilder rather than a String
 * @date October 2011, modified to add new methods and remove old ones
 * @date October 2016, updated to implement new interface
 */

public class StringBuilderStrand implements IDnaStrand {

        private StringBuilder myInfo;
        private int myAppends;

        /**
         * Create a strand representing s. No error checking is done to see if s
         * represents valid genomic/DNA data.
         *
         * @param s
         *            is the source of cgat data for this strand
         */

        public StringBuilderStrand(String s) {
                initialize(s);
        }

        @Override
        public IDnaStrand cutAndSplice(String enzyme, String splicee) {
                int pos = 0;
                int start = 0;
                StringBuilder search = myInfo;
                boolean first = true;
                IDnaStrand ret = null;

                // code identical to StringStrand, both String and StringBuilder
                // support .substring and .indexOf

                while ((pos = search.indexOf(enzyme, start)) ≥ 0) {
                        if (first) {
                                ret = getInstance(search.substring(start, pos));
                                first = false;
                        } else {
                                ret.append(search.substring(start, pos));

                        }
                        start = pos + enzyme.length();
                        ret.append(splicee);
                        pos++;
                }

                if (start < search.length()) {
                        // NOTE: This is an important special case! If the enzym
e
                        // is never found, return an empty String.
                        if (ret ≡ null) {
                                ret = getInstance("");
                        } else {
                                ret.append(search.substring(start));
                        }
                }
                return ret;
        }

        /**
```

```java
         * Initialize this strand so that it represents the value of source. No
         * error checking is performed.
         *
         * @param source
         *            is the source of this enzyme
         */
        @Override
        public void initialize(String source) {
                myInfo = new StringBuilder(source);
                myAppends = 0;
        }

        /**
         * @return number of base-pairs in this strand
         */
        @Override
        public long size() {
                return myInfo.length();
        }

        @Override
        public String toString() {
                return myInfo.toString();
        }

        /**
         * Simply append a strand of dna data to this strand. No error checking
is
         * done. This method isn't efficient; it doesn't use a StringBuilder or
a
         * StringBuffer.
         *
         * @param dna
         *            is the String appended to this strand
         */
        public IDnaStrand append(String dna) {
                myInfo.append(dna);
                myAppends++;
                return this;
        }

        public IDnaStrand reverse() {
                StringBuilder copy = new StringBuilder(myInfo);
                StringBuilderStrand ss = new StringBuilderStrand("replace");
                copy.reverse();
                ss.myInfo = copy;
                return ss;
        }

        @Override
        public String getStats() {
                return String.format("# appends = %d", myAppends);
        }

        public char charAt(int index) {
                return myInfo.charAt(index);
        }

        @Override
        public IDnaStrand getInstance(String source) {
                return new StringBuilderStrand(source);
        }
}
```

```
import java.util.*;

/**
 * Simple binary search tree implementation of a set. Operations are O(log n)
 * in average case and O(n) in the worst case for unbalanced trees.
 * @author Owen Astrachan
 */


public class BSTSet<E extends Comparable<E>> implements ISimpleSet<E> {

    private class TreeNode {
        E info;

        TreeNode left;
        TreeNode right;
        TreeNode parent;

        TreeNode(E element, TreeNode lptr, TreeNode rptr, TreeNode p) {
            info = element;
            left = lptr;
            right = rptr;
            parent = p;
        }
    }

    private int mySize;

    private TreeNode myRoot;

    public BSTSet() {
        mySize = 0;
        myRoot = null;
    }

    public int size() {
        return mySize;
    }

    public boolean add(E element) {
        if (myRoot ≡ null) {
            myRoot = new TreeNode(element, null, null, null);
            mySize++;
            return true;
        }
        TreeNode root = myRoot;

        while (root ≠ null) {
            int comp = root.info.compareTo(element);
            if (comp ≡ 0)
                return false;
            if (comp > 0) {
                if (root.left ≡ null) {
                    root.left = new TreeNode(element, null, null, root);
                    mySize++;
                    return true;
                } else {
                    root = root.left;
                }
            } else {
                if (root.right ≡ null) {
                    root.right = new TreeNode(element, null, null, root);
                    mySize++;
                    return true;
                } else {
                    root = root.right;
                }
```

```
            }
        }
        // can never reach here
        return false;

    }

    public boolean remove(E element) {
        TreeNode root = myRoot;
        while (root ≠ null) {
            int comp = root.info.compareTo(element);
            if (comp ≡ 0) {
                mySize--;
                remove(root);
                return true;
            } else if (comp > 0) {
                root = root.left;
            } else {
                root = root.right;
            }
        }
        return false;
    }

    private void remove(TreeNode root) {
        if (root.left ≡ null ∧ root.right ≡ null) {
            // removing leaf
            if (root.parent ≡ null) { // removing root?
                myRoot = null; // tree now empty
            } else {
                if (root.parent.left ≡ root) {
                    root.parent.left = null;
                } else {
                    root.parent.right = null;
                }
            }
        } else if (root.left ≡ null ∨ root.right ≡ null) {
            // one child, not two
            TreeNode child = root.left;  // only child is left?
            if (root.left ≡ null) {     // nope, it’s right
                child = root.right;
            }
            if (root.parent ≡ null) {   // new root
                myRoot = child;
            } else if (root.parent.left ≡ root) {
                root.parent.left = child;
            } else {
                root.parent.right = child;
            }
            child.parent = root.parent;
        } else {                          // removing node with two children
            TreeNode successor = root.right;
            if (successor.left ≡ null) {
                root.info = successor.info;
                root.right = successor.right;
                if (successor.right ≠ null) {
                    successor.right.parent = root;
                }
            } else {
                // immediate right child of removed node has a left child
                while (successor.left ≠ null) {
                    successor = successor.left;
                }
                root.info = successor.info;
                successor.parent.left = successor.right;
                if (successor.right ≠ null) {
                    successor.right.parent = successor.parent;
                }
            }
        }
    }

    private TreeNode successor(TreeNode t) {
        if (t ≡ null)
            return null; // no successor
        else if (t.right ≠ null) {
            t = t.right;
            while (t.left ≠ null) {
                t = t.left;
            }
            return t;
        } else {
            TreeNode parent = t.parent;
            while (parent ≠ null ∧ parent.right ≡ t) {
                t = parent;
                parent = t.parent;
            }
            return parent;
        }
    }

    public boolean contains(E element) {
        TreeNode root = myRoot;
        while (root ≠ null) {
            int comp = root.info.compareTo(element);
            if (comp ≡ 0)
                return true;
            else if (comp > 0) {
                root = root.left;
            } else {
                root = root.right;
            }
        }
        return false;
    }

    public Iterator<E> iterator() {
        return new TreeIterator(myRoot);
    }

    private class TreeIterator implements Iterator<E> {

        private TreeNode myCurrent;

        private TreeNode myPrevious;

        public TreeIterator(TreeNode root) {
            while (root.left ≠ null) {
                root = root.left;
            }
            myCurrent = root;
            myPrevious = null;
        }

        public boolean hasNext() {
            return myCurrent ≠ null;
        }

        public E next() {
            E data = myCurrent.info;
            myPrevious = myCurrent;
            myCurrent = successor(myCurrent);
            return data;
        }

        public void remove() {
            if (myPrevious ≡ null) {
                throw new IllegalStateException(
```

```
                        "cannot remove, no valid next call");
            }
            BSTSet.this.remove(myPrevious);
            myPrevious = null;
            mySize--;
        }
    }

}
```