```
import java.util.Formatter;

/** A Binary search tree. The parameter KEY gives the type of
 *  key (label).
 *  @author P. N. Hilfinger */
public class BST<Key extends Comparable<Key>> {

    /** A new BST with label LAB and children LEFT and RIGHT. */
    public BST(Key lab, BST<Key> left, BST<Key> right) {
        _label = lab;
        _left = left;
        _right = right;
    }

    /** A BST with label LAB and no children. */
    public BST(Key lab) {
        this(lab, null, null);
    }

    /** Return my label. */
    public Key label() {
        return _label;
    }

    /** Return my left child. */
    public BST<Key> left() {
        return _left;
    }

    /** Return my right child. */
    public BST<Key> right() {
        return _right;
    }

    /** Set label() to L. */
    public void setLabel(Key L) {
        _label = L;
    }

    /** Set left() to T. */
    public void setLeft(BST<Key> T) {
        _left = T;
    }

    /** Set right to T. */
    public void setRight(BST<Key> T) {
        _left = T;
    }

    /** My label. */
    private Key _label;
    /** My children. */
    private BST<Key> _left, _right;


    /** Return node in T containing L, or null if none. KEY is
     *  label type. */
    public static <Key extends Comparable<Key>>
        BST<Key> find(BST<Key> T, Key L) {
        if (T == null) {
            return T;
        }
        if (L.compareTo(T.label()) == 0) {
            return T;
        } else if (L.compareTo(T.label()) < 0) {
            return find(T.left(), L);
        } else {
            return find(T.right(), L);
        }
    }

    /** Insert L in T, replacing existing
     *  value if present, and returning
     *  new tree. KEY is type of tree label. */
    public static <Key extends Comparable<Key>>
        BST<Key> insert(BST<Key> T, Key L) {
        if (T == null) {
            return new BST<Key>(L);
        } else if (L.compareTo(T.label()) == 0) {
            T.setLabel(L);
        } else if (L.compareTo(T.label()) < 0) {
            T.setLeft(insert(T.left(), L));
        } else {
            T.setRight(insert(T.right(), L));
        }
        return T;
    }

    /** Remove KEY value L from T, returning new tree. */
    public static <Key extends Comparable<Key>>
        BST<Key> remove(BST<Key> T, Key L) {
        if (T == null) {
            return null;
        } else if (L.compareTo(T.label()) == 0) {
            if (T.left() == null) {
                return T.right();
            } else if (T.right() == null) {
                return T.left();
            } else {
                Key smallest = minVal(T.right());  // ??
                T.setRight(remove(T.right(), smallest));
                T.setLabel(smallest);
            }
        } else if (L.compareTo(T.label()) < 0) {
            T.setLeft(remove(T.left(), L));
        } else {
            T.setRight(remove(T.right(), L));
        }
        return T;
    }

    /** Return the minimum KEY value in T.  T must not be null. */
    public static <Key extends Comparable<Key>>
        Key minVal(BST<Key> T) {

        while (T.left() != null) {
            T = T.left();
        }
        return T.label();
    }

    private void pTree(BST<Key> T, int indent, Formatter out) {
        for (int i = 0; i < indent; i += 1) {
            out.format(" ");
        }
        if (T == null) {
            out.format("()");
        }
        out.format("(%s", T.label());
        if (T.left() == null && T.right() == null) {
            out.format(")");
        } else {
            out.format("%n");
            pTree(T.left(), indent + 4, out);
            out.format("%n");
            pTree(T.right(), indent + 4, out);
        }
        out.format(")");
    }

    @Override
    public String toString() {
        Formatter out = new Formatter();
        pTree(this, 0, out);
        return out.toString();
    }

}
```