```java
import java.util.Comparator;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

public class ScapegoatTree<Key extends Comparable<Key>, Value> {

    static private class Node<Key, Value> {
        Key key;
        Value value;
        Node<Key, Value> left, right, parent;

        Node(Key key, Value value) {
            this.key = key;
            this.value = value;
            this.parent = null;
        }
    }

    private Node<Key, Value> root;
    private final Comparator<Key> comparator;
    private int size;
    private int maxSize;
    private final double alpha = 0.57;

    // constructor
    // @param key type, value type
    public ScapegoatTree() {
        root = null;
        comparator = new Comparator<Key>() {
            @Override
            public int compare(Key o1, Key o2) {
                return o1.compareTo(o2);
            }
        };
        size = 0;
        maxSize = 0;
    }

    // constructor
    // @param key type, value type
    public ScapegoatTree(Comparator<Key> keyComparator) {
        root = null;
        comparator = keyComparator;
        size = 0;
        maxSize = 0;
    }

    /* size
       @param  void
       @return size of tree is zero
    */
    public boolean isEmpty(){
        return size() == 0;
    }

    public void clear(){
        root = null;
        size = 0;
        maxSize = 0;
    }

    // public interface for tree size
    // @param  void
    // @return size of tree
    public int size() {
        return size(this.root);
    }

    // public interface for tree size
    // @param  Node node
    // @return size of subtree rooted at node
    private int size(Node<Key, Value> node) {
        if (node == null)
            return 0;
        else
            return 1 + size(node.left) + size(node.right);
    }

    // membership test
    public boolean contains(Key key) {
        return get(key) != null;
    }

    // public interface for get
    // @param  void
    // @return node.value containing key
    public Value get(Key key) {
        Node<Key, Value> node = get(this.root, key);
        if(node != null)
            return node.value;
        return null;
    }

    // recursive implementation for get
    private Node<Key, Value> get(Node<Key, Value> node, Key key) {
        if (node == null)
            return null;
        int compareResult = comparator.compare(key, node.key);
        if(compareResult < 0)
            return get(node.left, key);
        else if (compareResult > 0)
            return get(node.right, key);
        else {
            return node;
        }
    }

    // non-recursive insert
    public void insert(Key key, Value value) {
        if (root == null) {
            root = new Node<>(key, value);
            size++;
        } else {
            Node<Key, Value> node = root;
            Node<Key, Value> parent = node.parent;
            boolean isLeftChild = false;
            while (true) {
                if (node == null) {
                    Node<Key, Value> newNode = new Node<>(key, value);
                    newNode.parent = parent;
                    this.size++;
                    this.maxSize = Math.max(size, maxSize);
                    if (isLeftChild) {
                        newNode.parent.left = newNode;
                    } else {
                        newNode.parent.right = newNode;
                    }
                    if (isDeepNode(newNode)) {
                        // new node is in its place but tree needs to be rebalanced
                        // from an ancestor of node called scapegoat
                        Node<Key, Value> scapegoat = findScapegoat(newNode);
                        Node<Key, Value> scapegoatParent = scapegoat.parent;
                        boolean scapegoatOnParentsLeft = scapegoatParent != null &&
scapegoatParent.left == scapegoat;
                        // connect root of balanced tree to parent
                        Node<Key, Value> balancedTreeRoot =
buildTree(flattenTree(scapegoat), 0, size(scapegoat) - 1);
                        if (scapegoatParent != null) {
                            if (scapegoatOnParentsLeft) {
                                scapegoatParent.left = balancedTreeRoot;
                            } else {
                                scapegoatParent.right = balancedTreeRoot;
                            }
                        }
                        // if parent is null then scapegoat must be root
                        if (scapegoat == root) {
                            root = balancedTreeRoot;
                        }
                        maxSize = size;
                    }
                    return;
                }
                int compareResult = comparator.compare(key, node.key);
                if (compareResult == 0) {
                    node.value = value;
                    if (parent != null) {
                        if (isLeftChild) {
                            node.parent.left = node;
                        } else {
                            node.parent.right = node;
                        }
                    }
                    return;
                } else {
                    parent = node;
                    if (compareResult < 0) {
                        node = node.left;
                        isLeftChild = true;
                    } else {
                        node = node.right;
                        isLeftChild = false;
                    }
                }
            }
        }
    }
    public void insertWithoutRebalancing(Key key, Value value) {
        root = insertWithoutRebalancing(root, key, value);
    }

    private Node<Key,Value> insertWithoutRebalancing(Node<Key,Value> node, Key key,
Value value) {
        if(node == null) {
            node = new Node<>(key, value);
            this.size++;
            this.maxSize = Math.max(size, maxSize);
            return node;
        }
        int compareResult = comparator.compare(key, node.key);
        if(compareResult == 0){
            node.value = value;
            this.maxSize = Math.max(size, maxSize);
        }
        else if(compareResult < 0) {
            node.left = insertWithoutRebalancing(node.left, key, value);
            node.left.parent = node;
        }
        else {
            node.right = insertWithoutRebalancing(node.right, key, value);
            node.right.parent = node;
        }
        return node;
    }

    public void deleteWithoutRebalancing(Key key) {
        root = delete(root, key);
    }

    public void delete(Key key) {
        root = delete(root, key);
        if(size<alpha*maxSize) {
            // rebuild tree completely
            root = buildTree(flattenTree(root), 0, size);
            maxSize = size;
        }
    }


    private Node<Key,Value> findScapegoat(Node<Key,Value> node) {
        int size = 1;
        int height = 0;
        int subtreeSize = 0;
        while (node.parent != null) {
            height++;
            subtreeSize = 1 + size + size(getSibling(node));
            if (height > (int) Math.floor(Math.log(subtreeSize) /
Math.log(1.0/alpha))) {
                return node.parent;
            }
            node = node.parent;
            size = subtreeSize;
        }
        return root;
    }

    private Node<Key, Value> getSibling(Node<Key,Value> node) {
        if (node.parent != null) {
            if (node.parent.left == node) {
                return node.parent.right;
            }
            else {
                return node.parent.left;
            }
        }
        return null;
    }

    private Node<Key, Value> getSiblingByKey(Node<Key,Value> node) {
        if (node.parent != null) {
            int compareResult = comparator.compare(node.key, node.parent.key);


    private Node<Key,Value> delete(Node<Key,Value> node, Key key) {
        if(node == null) {
            return null;
        }
        int compareResult = comparator.compare(key, node.key);
        if(compareResult < 0) {
            node.left = delete(node.left, key);
            }
        else {
            if(compareResult > 0)  {
                node.right = delete(node.right, key);
            }
            else {
                // node to be deleted. need to adjust parent pointer!
                // three cases: 1. no left child
                //              2. no right child
                //              3 two children
                if(node.left == null) {
                if(node.right != null) {
                        node.right.parent = node.parent;
                    }
                    size--;
                    return node.right;
                }
                else {
                    if(node.right == null) {
                        node.left.parent = node.parent;
                        size--;
                        return node.left;
                    }
                    else {
                        // two children
                        Node<Key,Value> successor = minimum(node.right);
                        node.key = successor.key;
                        node.value = successor.value;
                        node.right = delete(node.right, node.key);
                        size--;
                    }
                }
            }
        }
        return node;
    }
```

```
            if(compareResult < 0)
                return node.parent.right;
            else
                return node.parent.left;
            }
        return null;
    }

    private List<Node<Key,Value>> flattenTree(Node<Key,Value> node) {
        List<Node<Key,Value>> nodes = new ArrayList<>();
        Stack<Node<Key,Value>> stack = new Stack<>();
        // flatten tree without recursion with in-order walk
        // using stack described in GalparinRivest93
        Node<Key,Value> currentNode = node;
        while(true) {
            if(currentNode != null){
                stack.push(currentNode);
                currentNode = currentNode.left;
            }
            else {
                if(!stack.isEmpty()) {
                    currentNode = stack.pop();
                    nodes.add(currentNode);
                    currentNode = currentNode.right;
                }
                else {
                    return nodes;
                }
            }
        }
    }

    private int getNodeHeight(Node<Key,Value> node) {
        if (node == null) {
            return -1;
        }
        else {
            if (node.parent == null) {
                return 0;
            }
            else {
                return getNodeHeight(node.parent) + 1;
            }
        }
    }


    // returns root of a balanced tree built from ordered list of nodes
    private Node<Key,Value> buildTree(List<Node<Key,Value>> nodes, int start, int
end) {
        int median = (int) Math.ceil(((double)(start + end)) / 2.0);
        if (start > end) {
            return null;
        }
        // median becomes root of subtree instead of scapegoat
        Node<Key,Value> node = nodes.get(median);
        // clear parent to avoid loops
        node.parent = null;
        // recursively get left and right nodes
        Node<Key,Value> leftNode = buildTree(nodes, start, median - 1);
        node.left = leftNode;
        if (leftNode != null) {
            leftNode.parent = node;
        }
        Node<Key,Value> rightNode = buildTree(nodes, median + 1, end);
        node.right = rightNode;
        if (rightNode != null) {
            rightNode.parent = node;
        }
        return node;
    }

    private boolean isDeepNode(Node<Key,Value> node) {
        int height = getNodeHeight(node);
        int h_alpha = (int) Math.floor(Math.log(size) / Math.log(1.0/alpha));
        return height > h_alpha;
    }

    public int depth() {
        return depth(root)-1;
    }

    private int depth(Node<Key,Value> node) {
        if(node == null)
            return 0;
        if(node.left == null) {
            if(node.right == null) {
                return 1;
            }
            else {
                return depth(node.right) + 1;
            }
        }
        else if(node.right == null) {
            return depth(node.left) + 1;
        }
        else {
            return Math.max(depth(node.left), depth(node.right))+1;
        }
    }

    // do not have generic minimum to return a "not found" for generic types
    public Value minimum() {
        Node<Key,Value> min = minimum(root);
        if(min != null) {
            return min.value;
        }
        else {
            throw new NullPointerException("empty tree does not have a minimum");
        }
    }

    private Node<Key,Value> minimum(Node<Key,Value> node) {
        while(node.left != null)
            node = node.left;
        return node;
    }

    // do not have generic minimum to return a "not found" for generic types
```

```
public Value maximum() {
    Node<Key,Value> max = maximum(root);
    if(max != null) {
        return max.value;
    }
    else {
        throw new NullPointerException("empty tree does not have a maximum");
    }
}

private Node<Key,Value> maximum(Node<Key,Value> node) {
    while(node.right != null)
        node = node.right;
    return node;
}

public List<Node<Key,Value>> inOrder() {
    List<Node<Key,Value>> nodes = new ArrayList<>();
    inOrder(root, nodes);
    return nodes;
}

private void inOrder(Node<Key,Value> node, List<Node<Key,Value>> nodes) {
    if(node != null) {
        inOrder(node.left, nodes);
        nodes.add(node);
        System.out.print(node.value + " ");
        inOrder(node.right, nodes);
    }
}

public List<Node<Key,Value>> preOrder() {
    List<Node<Key,Value>> nodes = new ArrayList<>();
    preOrder(root, nodes);
    return nodes;
}

private void preOrder(Node<Key,Value> node, List<Node<Key,Value>> nodes) {
    if(node != null) {
        nodes.add(node);
        System.out.print(node.value + " ");
        preOrder(node.left, nodes);
        preOrder(node.right, nodes);
    }
}

public List<Node<Key,Value>> postOrder() {
    List<Node<Key,Value>> nodes = new ArrayList<>();
    postOrder(root, nodes);
    return nodes;
}

private void postOrder(Node<Key,Value> node, List<Node<Key,Value>> nodes) {
    if(node != null) {
        postOrder(node.left, nodes);
        postOrder(node.right, nodes);
        nodes.add(node);
        System.out.print(node.value + " ");
    }
    }

    private Node<Key,Value> findMin(Node<Key,Value> node) {
        while (node.left != null) {
            node = node.left;
        }
        return node;
    }

    private Node<Key,Value> findNode(Key key) {
        Node<Key,Value> node = root;
        while (node != null) {
            int cmp = key.compareTo(node.key);
            if (cmp < 0) {
                node = node.left;
            } else if (cmp > 0) {
                node = node.right;
            } else {
                return node;
            }
        }
        return null;
    }
    private void buildString(Node<Key, Value> node, StringBuilder sb) {
        sb.append(node.key);
        if (node.left != null || node.right != null) {
            sb.append("(");
            if (node.left != null) {
                buildString(node.left, sb);
            } else {
                sb.append(" ");
            }
            sb.append(",");
            if (node.right != null) {
                buildString(node.right, sb);
            } else {
                sb.append(" ");
            }
            sb.append(")");
        }
    }

}



    public Key successor(Key key) {
        Node<Key,Value> node = findNode(key);
        if (node == null) {
            return null;
        }
        if (node.right != null) {
            Node<Key,Value> successorNode = findMin(node.right);
            return successorNode.key;
        } else {
            Node<Key,Value> parent = node.parent;
            while (parent != null && node == parent.right) {
                node = parent;
                parent = parent.parent;
            }
            return (parent != null) ? parent.key : null;
        }
    }
```

```
public Key predecessor(Key key) {
    Node<Key,Value> node = findNode(key);
    if (node == null) {
        return key;
    }
    if (node.left != null) {
        Node<Key,Value> maxNode = node.left;
        while (maxNode.right != null) {
            maxNode = maxNode.right;
        }
        return maxNode.key;
    } else {
        Node<Key,Value> cur = root;
        Node<Key,Value> prev = null;
        while (cur != node) {
            if (comparator.compare(node.key, cur.key) < 0) {
                cur = cur.left;
```

```java
            } else {
                prev = cur;
                cur = cur.right;
            }
        }
        if (prev != null) {
            return prev.key;
        } else {
            return key;
        }
    }
}

// Java String is immutable!

@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    if (root != null) {
        buildString(root, sb);
    }
    return sb.toString();
}
```