```java
/*
 * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
 * Version: Sun Oct  6 13:58:27 CEST 2024
 */

package ex04.solution.task01;

import java.util.Collection;

import ex02.solution.task01.BinarySearchTree.Entry;


public class AVLTree <K extends Comparable<? super K>, V> {

  private AVLTreeImpl<K, V> avlTreeImpl =
    new AVLTreeImpl<>();
    //new AVLTreeImplADV<K, V>("AVL-Tree Ü4"); // Show in ADV
    //new AVLTreeImplADV<K, V>("AVL-Tree Ü4", 3, 2); // Show in ADV: Fix heights

  public V put(K key, V value) {
    return avlTreeImpl.put(key, value);
  }

  public V get(K key) {
    return avlTreeImpl.get(key);
  }

  public V remove(K key) {
    return avlTreeImpl.remove(key);
  }

  public int getHeight() {
    return avlTreeImpl.getHeight();
  }

  public int size() {
    return avlTreeImpl.size();
  }

  public boolean isEmpty() {
    return avlTreeImpl.isEmpty();
  }

  public void clear() {
    avlTreeImpl.clear();
  }

  public Collection<Entry<K, V>> inorder() {
    return avlTreeImpl.inorder();
  }

  public void printInorder() {
    avlTreeImpl.printInorder();
  }

  public void print() {
    avlTreeImpl.print();
  }

  protected AVLTreeImpl<K, V> getImpl() {
    return avlTreeImpl;
  }
```

```java

  public static void main(String[] args) {

    AVLTree<Integer, String> avlTree = new AVLTree<>();

    System.out.println("Inserting 5:");
    avlTree.put(5, "Str_5");
    avlTree.print();
    System.out.println("================================");
    System.out.println("Inserting 7:");
    avlTree.put(7, "Str_7");
    avlTree.print();
    System.out.println("================================");
    System.out.println("Inserting 9: Single-Rotation");
    avlTree.put(9, "Str_9");
    avlTree.print();
    System.out.println("================================");
    System.out.println("Inserting 3:");
    avlTree.put(3, "Str_3");
    avlTree.print();
    System.out.println("================================");
    System.out.println("Inserting 1: Single-Rotation");
    avlTree.put(1, "Str_1");
    avlTree.print();
    System.out.println("================================");
    System.out.println("Inserting 4: Double-Rotation");
    avlTree.put(4, "Str_4");
    avlTree.print();
    System.out.println("================================");

  }

}

```

```

/* Session-Log:

Inserting 5:
 5 - Str_5  : h=0 ROOT
================================
Inserting 7:
 5 - Str_5  : h=1 ROOT
 7 - Str_7  : h=0 \ parent(key)=5
================================
Inserting 9: Single-Rotation
 5 - Str_5  : h=0 / parent(key)=7
 7 - Str_7  : h=1 ROOT
 9 - Str_9  : h=0 \ parent(key)=7
================================
Inserting 3:
 3 - Str_3  : h=0 / parent(key)=5
 5 - Str_5  : h=1 / parent(key)=7
 7 - Str_7  : h=2 ROOT
 9 - Str_9  : h=0 \ parent(key)=7
================================
Inserting 1: Single-Rotation
 1 - Str_1  : h=0 / parent(key)=3
 3 - Str_3  : h=1 / parent(key)=7
 5 - Str_5  : h=0 \ parent(key)=3
 7 - Str_7  : h=2 ROOT
 9 - Str_9  : h=0 \ parent(key)=7
================================
Inserting 4: Double-Rotation
 1 - Str_1  : h=0 / parent(key)=3
 3 - Str_3  : h=1 / parent(key)=5
 4 - Str_4  : h=0 \ parent(key)=3
 5 - Str_5  : h=2 ROOT
 7 - Str_7  : h=1 \ parent(key)=5
 9 - Str_9  : h=0 \ parent(key)=7
================================

*/
```

```java
/*
 * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
 * Version: Sun Oct  6 13:58:27 CEST 2024
 */

package ex04.solution.task01;

import java.lang.reflect.Array;
import java.util.Collection;
import java.util.LinkedList;
import java.util.List;

import ex02.solution.task01.BinarySearchTree;


class AVLTreeImpl<K extends Comparable<? super K>, V> extends
    BinarySearchTree<K, V> {

  /**
   * After a BST-operation, actionNode shall point to where the balance has to
   * be checked. -> rebalance() will then be called with actionNode.
   */
  protected AVLNode actionNode;


  protected class AVLNode extends BinarySearchTree<K, V>.Node {

    private int height;
    private Node parent;

    AVLNode(Entry<K, V> entry) {
      super(entry);
    }

    protected AVLNode setParent(AVLNode parent) {
      AVLNode old = avlNode(this.parent);
      this.parent = parent;
      return old;
    }

    protected AVLNode getParent() {
      return avlNode(parent);
    }

    protected int setHeight(int height) {
      int old = this.height;
      this.height = height;
      return old;
    }

    protected int getHeight() {
      return height;
    }

    @SuppressWarnings("unchecked")
    @Override
    public void setLeftChild(BinarySearchTree<K, V>.Node leftChild) {
      super.setLeftChild(leftChild);
      if (leftChild != null) {
        ((AVLNode)leftChild).setParent(this);
      }
    }

    @Override
    public AVLNode getLeftChild() {
      return avlNode(super.getLeftChild());
    }

    @SuppressWarnings("unchecked")
    @Override
    public void setRightChild(BinarySearchTree<K, V>.Node rightChild) {
      super.setRightChild(rightChild);
      if (rightChild != null) {
        ((AVLNode)rightChild).setParent(this);
      }
    }

    @Override
    public AVLNode getRightChild() {
      return avlNode(super.getRightChild());
    }

    @Override
    public String toString() {
      String result = String.format("%2d - %-6s : h=%d",
                                    getEntry().getKey(), getEntry().getValue(), height);
      if (parent == null) {
        result += " ROOT";
      } else {
        boolean left = (parent.getLeftChild() == this) ? true : false;
        result += (left ? " / " : " \\ ") + "parent(key)="
            + parent.getEntry().getKey();
      }
      return result;
    }

  } // End of class AVLNode


  protected AVLNode getRoot() {
    return avlNode(root);
  }

  public V put(K key, V value) {
    Entry<K, V> entry = find(key);
    if (entry != null) {
      // key already exists in the Tree
      return entry.setValue(value);
    }
    // key does not exist in the Tree yet
    super.insert(key, value);
    rebalance(actionNode);
    actionNode = null;
    return null;
  }

  public V get(K key) {
    Entry<K, V> entry = super.find(key);
    if (entry == null) {
      return null;
    }
    return entry.getValue();
  }
```

```java

  @Override
  protected Node insert(Node node, Entry<K, V> entry) {
    if (node != null) {
      actionNode = avlNode(node);
    }
    // calling now the BST-insert() which will do the work:
    AVLNode result = avlNode(super.insert(node, entry));
    if (node == null) {
      // In this case: result of super.insert() is the new node!
      result.setParent(actionNode);
    }
    return result;
  }

  /**
   * The height of the tree.
   *
   * @return The current height. -1 for an empty tree.
   */
  @Override
  public int getHeight() {
    return height(avlNode(root));
  }

  /**
   * Returns the height of this node.
   *
   * @param node
   * @return The height or -1 if null.
   */
  @SuppressWarnings("static-method")
  protected int height(AVLNode node) {
    return (node != null) ? node.getHeight() : -1;
  }

  /**
   * Restructures the tree with rotations.
   *
   * @param xPos
   *          The X-node.
   * @return The new root-node of this subtree.
   */
  protected AVLNode restructure(AVLNode xPos) {
    AVLNode yPos = xPos.getParent();
    AVLNode zPos = yPos.getParent();
    AVLNode newSubTreeRoot = null;
    if (yPos == zPos.getLeftChild()) {
      if (xPos == yPos.getLeftChild()) {
        newSubTreeRoot = rotateWithLeftChild(zPos);
      } else {
        newSubTreeRoot = doubleRotateWithLeftChild(zPos);
      }
    } else {
      if (xPos == yPos.getRightChild()) {
        newSubTreeRoot = rotateWithRightChild(zPos);
      } else {
        newSubTreeRoot = doubleRotateWithRightChild(zPos);
      }
    }
    return newSubTreeRoot;
  }

  protected AVLNode tallerChild(AVLNode node) {
    AVLNode result;
    if (height(node.getLeftChild()) >= height(node.getRightChild())) {
      result = node.getLeftChild();
    } else {
      result = node.getRightChild();
    }
    return result;
  }

  protected AVLNode rotateWithLeftChild(AVLNode k2) {
    AVLNode parentSubtree = k2.getParent();
    AVLNode k1 = k2.getLeftChild();
    k2.setLeftChild(k1.getRightChild());
    k1.setRightChild(k2);
    adjustSubtreeParent(k2,  k1, parentSubtree);
    return k1;
  }

  protected AVLNode doubleRotateWithLeftChild(AVLNode k3) {
    //k3.setLeftChild(rotateWithRightChild(k3.getLeftChild()));
    // -> k3.setLeftChild() is done/ensured in adjustSubtreeParent():
    rotateWithRightChild(k3.getLeftChild());
    return rotateWithLeftChild(k3);
  }

  protected AVLNode rotateWithRightChild(AVLNode k1) {
    AVLNode parentSubtree = k1.getParent();
    AVLNode k2 = k1.getRightChild();
    k1.setRightChild(k2.getLeftChild());
    k2.setLeftChild(k1);
    adjustSubtreeParent(k1, k2, parentSubtree);
    return k2;
  }

  protected AVLNode doubleRotateWithRightChild(AVLNode k3) {
    //k3.setRightChild(rotateWithLeftChild(k3.getRightChild()));
    // -> k3.setRightChild() is done/ensured in adjustSubtreeParent():
    rotateWithLeftChild(k3.getRightChild());
    return rotateWithRightChild(k3);
  }

  /**
   * Assures the connection between a restructured subtree and the parent of
   * this subtree.
   * Used after rotations.
   *
   * @param oldSubtreeRoot
   *          The old root-node of this subtree.
   * @param newSubtreeRoot
   *          The new root-node of this subtree.
   * @param parentSubtree
   *          The parent-node of this subtree.
   */
  protected void adjustSubtreeParent(AVLNode oldSubtreeRoot,
      AVLNode newSubtreeRoot, AVLNode parentSubtree) {
    if (parentSubtree != null) {
      if (parentSubtree.getLeftChild() == oldSubtreeRoot) {
        parentSubtree.setLeftChild(newSubtreeRoot);
      } else {
        parentSubtree.setRightChild(newSubtreeRoot);
      }
    } else { // newSubtreeRoot is now also the root of the whole tree
      root = newSubtreeRoot;
      newSubtreeRoot.setParent(null);
    }
  }
```

```java

  protected boolean isBalanced(AVLNode node) {
    int bf = height(node.getLeftChild()) - height(node.getRightChild());
    return (-1 <= bf) && (bf <= 1);
  }

  /**
   * Assures the balance of the tree from 'node' up to the root.
   *
   * @param node
   *          The node from where to start.
   */
  protected void rebalance(AVLNode node) {
    while (node != null) {
      setHeight(node);
      if (!isBalanced(node)) {
        AVLNode xPos = tallerChild(tallerChild(node));
        node = restructure(xPos);
        setHeight(node.getLeftChild());
        setHeight(node.getRightChild());
        setHeight(node);
      }
      node = node.getParent();
    }
  }

  /**
   * Assures the correct height for node.
   *
   * @param node
   *          The node to assure its height.
   */
  protected void setHeight(AVLNode node) {
    if (node == null) {
      return;
    }
    int heightLeftChild = height(node.getLeftChild());
    int heightRightChild = height(node.getRightChild());
    node.setHeight(1 + Math.max(heightLeftChild, heightRightChild));
  }

  /**
   * Factory-Method. Creates a new node.
   *
   * @param entry
   *          The entry to be inserted in the new node.
   * @return The new created node.
   */
  @Override
  protected Node newNode(Entry<K, V> entry) {
    return new AVLNode(entry);
  }

  public V remove(K key) {
    Entry<K, V> entry = find(key);
    if (entry == null) {
      return null;
    }
    // calling now the BST-remove(Entry) which will do the work:
    super.remove(entry);
    if (actionNode != null) {
      assureParentForChilds(actionNode);
      rebalance(actionNode);
      actionNode = null;
    }
    return entry.getValue();
  }

  @Override
  protected RemoveResult remove(Node node, Entry<K, V> entry) {
    if (node.getEntry() == entry) {
      actionNode = avlNode(node).getParent();
    }
    // calling now the BST-remove(Node, Entry) which will do the work:
    return super.remove(node, entry);
  }

  @Override
  protected Node getParentNext(Node p) {
    actionNode = avlNode(super.getParentNext(p));
    return actionNode;
  }

  @SuppressWarnings("static-method")
  protected void assureParentForChilds(AVLNode parent) {
    @SuppressWarnings("unchecked")
    AVLNode[] childs = (AVLNode[])Array.newInstance(parent.getClass(), 2);
    childs[0] = parent.getLeftChild();
    childs[1] = parent.getRightChild();
    for (AVLNode child : childs) {
      if (child != null) {
        child.setParent(parent);
      }
    }
  }

  @Override
  protected void inorder(Node node, Collection<Node> inorderList) {
    super.inorder(node, inorderList);
  }

  // Type-Casting: Node -> AVLNode (Cast-Encapsulation)
  @SuppressWarnings({ "unchecked", "static-method" })
  protected AVLNode avlNode(Node node) {
    return (AVLNode)node;
  }

  public void print() {
    List<Node> nodeList = new LinkedList<>();
    inorder(root, nodeList);
    for (Node node: nodeList) {
      System.out.println(node + "  ");
    }
  }

}

```

```java
/*
 * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
 * Version: Sun Oct  6 13:58:27 CEST 2024
 */

package ex04.solution.task01;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNull;
import static org.junit.Assert.assertTrue;

import java.util.Collection;
import java.util.Hashtable;
import java.util.LinkedList;
import java.util.Map;
import java.util.Random;

import org.junit.Before;
import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runners.MethodSorters;

import ex02.solution.task01.BinarySearchTree.Entry;


@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class AVLTreeJUnitTest {

  AVLTreeImpl<Integer, String> avlTree;

  @Before
  public void setUp() {
    avlTree = new AVLTreeImpl<>();
  }

  @Test
  public void test01Put() {
    int[] keys = { 2, 1, 3 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 ROOT",
        " 3 - Str_3  : h=0 \\ parent(key)=2",
    };
    runTest(keys, expected);
  }

  @Test
  public void test02Get() {
    int[] keys = { 2, 1, 5, 4, 3 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=2 ROOT",
        " 3 - Str_3  : h=0 / parent(key)=4",
        " 4 - Str_4  : h=1 \\ parent(key)=2",
        " 5 - Str_5  : h=0 \\ parent(key)=4",
    };
    runTest(keys, expected);
    assertEquals("Str_2", avlTree.get(2));
    assertEquals("Str_5", avlTree.get(5));
    assertNull(avlTree.get(0));
    assertNull(avlTree.get(6));
  }
```

```java

    @Test
    public void test03SingleRotationLeftInRoot() {
      int[] keys = { 1, 2, 3 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=1 ROOT",
          " 3 - Str_3  : h=0 \\ parent(key)=2",
      };
      runTest(keys, expected);
    }

    @Test
    public void test04SingleRotationLeftBelowRoot() {
      int[] keys = { 5, 6, 1, 2, 3 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=1 / parent(key)=5",
          " 3 - Str_3  : h=0 \\ parent(key)=2",
          " 5 - Str_5  : h=2 ROOT",
          " 6 - Str_6  : h=0 \\ parent(key)=5",
      };
      runTest(keys, expected);
    }

    @Test
    public void test05SingleRotationRightInRoot() {
      int[] keys = { 3, 2, 1 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=1 ROOT",
          " 3 - Str_3  : h=0 \\ parent(key)=2",
      };
      runTest(keys, expected);
    }

    @Test
    public void test06SingleRotationRightBelowRoot() {
      int[] keys = { 2, 1, 5, 4, 3 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=2 ROOT",
          " 3 - Str_3  : h=0 / parent(key)=4",
          " 4 - Str_4  : h=1 \\ parent(key)=2",
          " 5 - Str_5  : h=0 \\ parent(key)=4",
      };
      runTest(keys, expected);
    }

    @Test
    public void test07DoubleRotationLeftInRoot() {
      int[] keys = { 1, 3, 2 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=1 ROOT",
          " 3 - Str_3  : h=0 \\ parent(key)=2",
      };
      runTest(keys, expected);
    }
```

```java

    @Test
    public void test08DoubleRotationLeftBelowRoot() {
      int[] keys = { 2, 1, 3, 5, 4 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=2 ROOT",
          " 3 - Str_3  : h=0 / parent(key)=4",
          " 4 - Str_4  : h=1 \\ parent(key)=2",
          " 5 - Str_5  : h=0 \\ parent(key)=4",
      };
      runTest(keys, expected);
    }

    @Test
    public void test09DoubleRotationRightinRoot() {
      int[] keys = { 3, 1, 2 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=1 ROOT",
          " 3 - Str_3  : h=0 \\ parent(key)=2",
      };
      runTest(keys, expected);
    }

    @Test
    public void test10DoubleRotationRightBelowRoot() {
      int[] keys = { 4, 3, 5, 1, 2 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=1 / parent(key)=4",
          " 3 - Str_3  : h=0 \\ parent(key)=2",
          " 4 - Str_4  : h=2 ROOT",
          " 5 - Str_5  : h=0 \\ parent(key)=4",
      };
      runTest(keys, expected);
    }

    @Test
    public void test11MultipleSameKeys() {
      int[] keys = { 3, 1, 2 };
      String[] expected = {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_2  : h=1 ROOT",
          " 3 - Str_3  : h=0 \\ parent(key)=2",
      };
      runTest(keys, expected);
      avlTree.put(2, "Str_22");
      avlTree.put(2, "Str_23");
      expected = new String[] {
          " 1 - Str_1  : h=0 / parent(key)=2",
          " 2 - Str_23 : h=1 ROOT",
          " 3 - Str_3  : h=0 \\ parent(key)=2",
      };
      Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
      avlTree.inorder(avlTree.getRoot(), nodes);
      verify(nodes, expected);
    }
```

```java

  @Test
  public void test12RemovingCase1() {
    // Löschen Fall 1 gem. BST-Folie 12:
    Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
    int[] keys = { 6, 2, 9, 1, 4, 8 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 / parent(key)=6",
        " 4 - Str_4  : h=0 \\ parent(key)=2",
        " 6 - Str_6  : h=2 ROOT",
        " 8 - Str_8  : h=0 / parent(key)=9",
        " 9 - Str_9  : h=1 \\ parent(key)=6",
    };
    runTest(keys, expected);
    assertEquals("Str_4", avlTree.remove(4));
    expected = new String[] {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 / parent(key)=6",
        " 6 - Str_6  : h=2 ROOT",
        " 8 - Str_8  : h=0 / parent(key)=9",
        " 9 - Str_9  : h=1 \\ parent(key)=6",
    };
    avlTree.inorder(avlTree.getRoot(), nodes);
    verify(nodes, expected);
  }

  @Test
  public void test13RemovingCase2() {
    // Löschen Fall 2 gem. BST-Folie 13:
    Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
    int[] keys = { 6, 2, 9, 1, 4, 8, 5 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=2 / parent(key)=6",
        " 4 - Str_4  : h=1 \\ parent(key)=2",
        " 5 - Str_5  : h=0 \\ parent(key)=4",
        " 6 - Str_6  : h=3 ROOT",
        " 8 - Str_8  : h=0 / parent(key)=9",
        " 9 - Str_9  : h=1 \\ parent(key)=6",
    };
    runTest(keys, expected);
    assertEquals("Str_4", avlTree.remove(4));
    expected = new String[] {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 / parent(key)=6",
        " 5 - Str_5  : h=0 \\ parent(key)=2",
        " 6 - Str_6  : h=2 ROOT",
        " 8 - Str_8  : h=0 / parent(key)=9",
        " 9 - Str_9  : h=1 \\ parent(key)=6",
    };
    avlTree.inorder(avlTree.getRoot(), nodes);
    verify(nodes, expected);
  }
```

```java

  @Test
  public void test14RemovingCase3() {
    // Löschen Fall 3 gem. BST-Folie 14:
    // Hinweis: Baum entsprechend 'aufgefüllt' (wegen AVL!)
    Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
    int[] keys = { 1, -10, 4, -15, -5, 2, 9, -18, -12, -7, -3, 3, 7, 10, 6 };
    String[] expected = {
        "-18 - Str_-18 : h=0 / parent(key)=-15",
        "-15 - Str_-15 : h=1 / parent(key)=-10",
        "-12 - Str_-12 : h=0 \\ parent(key)=-15",
        "-10 - Str_-10 : h=2 / parent(key)=1",
        "-7 - Str_-7 : h=0 / parent(key)=-5",
        "-5 - Str_-5 : h=1 \\ parent(key)=-10",
        "-3 - Str_-3 : h=0 \\ parent(key)=-5",
        " 1 - Str_1  : h=4 ROOT",
        " 2 - Str_2  : h=1 / parent(key)=4",
        " 3 - Str_3  : h=0 \\ parent(key)=2",
        " 4 - Str_4  : h=3 \\ parent(key)=1",
        " 6 - Str_6  : h=0 / parent(key)=7",
        " 7 - Str_7  : h=1 / parent(key)=9",
        " 9 - Str_9  : h=2 \\ parent(key)=4",
        "10 - Str_10 : h=0 \\ parent(key)=9",
    };
    runTest(keys, expected);
    assertEquals("Str_4", avlTree.remove(4));
    expected = new String[] {
        "-18 - Str_-18 : h=0 / parent(key)=-15",
        "-15 - Str_-15 : h=1 / parent(key)=-10",
        "-12 - Str_-12 : h=0 \\ parent(key)=-15",
        "-10 - Str_-10 : h=2 / parent(key)=1",
        "-7 - Str_-7 : h=0 / parent(key)=-5",
        "-5 - Str_-5 : h=1 \\ parent(key)=-10",
        "-3 - Str_-3 : h=0 \\ parent(key)=-5",
        " 1 - Str_1  : h=3 ROOT",
        " 2 - Str_2  : h=1 / parent(key)=6",
        " 3 - Str_3  : h=0 \\ parent(key)=2",
        " 6 - Str_6  : h=2 \\ parent(key)=1",
        " 7 - Str_7  : h=0 / parent(key)=9",
        " 9 - Str_9  : h=1 \\ parent(key)=6",
        "10 - Str_10 : h=0 \\ parent(key)=9",
    };
    avlTree.inorder(avlTree.getRoot(), nodes);
    verify(nodes, expected);
  }

  @Test
  public void test15RemovingAtRoot1() {
    int[] keys = { 1, 2, 3 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 ROOT",
        " 3 - Str_3  : h=0 \\ parent(key)=2",
    };
    runTest(keys, expected);
    assertEquals("Str_1", avlTree.remove(1));
    assertEquals(2, avlTree.size());
    assertEquals("Str_3", avlTree.remove(3));
    assertEquals(1, avlTree.size());
    assertEquals("Str_2", avlTree.remove(2));
    assertEquals(0, avlTree.size());
  }

  @Test
  public void test16RemovingAtRoot2() {
    int[] keys = { 1, 2, 3 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 ROOT",
        " 3 - Str_3  : h=0 \\ parent(key)=2",
    };
    runTest(keys, expected);
    assertEquals("Str_1", avlTree.remove(1));
    assertEquals(2, avlTree.size());
    assertEquals("Str_2", avlTree.remove(2));
    assertEquals(1, avlTree.size());
    assertEquals("Str_3", avlTree.remove(3));
    assertEquals(0, avlTree.size());
  }

  @Test
  public void test17RemovingAtRoot3() {
    Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
    int[] keys = { 1, 2, 3 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 ROOT",
        " 3 - Str_3  : h=0 \\ parent(key)=2",
    };
    runTest(keys, expected);
    assertEquals("Str_2", avlTree.remove(2));
    expected = new String[] {
        " 1 - Str_1  : h=0 / parent(key)=3",
        " 3 - Str_3  : h=1 ROOT",
    };
    avlTree.inorder(avlTree.getRoot(), nodes);
    verify(nodes, expected);
    assertEquals(2, avlTree.size());
    assertEquals("Str_3", avlTree.remove(3));
    assertEquals(1, avlTree.size());
    assertEquals("Str_1", avlTree.remove(1));
    assertEquals(0, avlTree.size());
  }

  @Test
  public void test18RemovingAtRoot4() {
    Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
    int[] keys = { 3, 2, 6, 4 };
    String[] expected = {
        " 2 - Str_2  : h=0 / parent(key)=3",
        " 3 - Str_3  : h=2 ROOT",
        " 4 - Str_4  : h=0 / parent(key)=6",
        " 6 - Str_6  : h=1 \\ parent(key)=3",
    };
    runTest(keys, expected);
    assertEquals("Str_3", avlTree.remove(3));
    expected = new String[] {
        " 2 - Str_2  : h=0 / parent(key)=4",
        " 4 - Str_4  : h=1 ROOT",
        " 6 - Str_6  : h=0 \\ parent(key)=4",
    };
    avlTree.inorder(avlTree.getRoot(), nodes);
    verify(nodes, expected);
  }
```

```java

  @Test
  public void test19RemovingAtRoot5() {
    Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
    int[] keys = { 3, 2, 6, 1, 4, 7, 5 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 / parent(key)=3",
        " 3 - Str_3  : h=3 ROOT",
        " 4 - Str_4  : h=1 / parent(key)=6",
        " 5 - Str_5  : h=0 \\ parent(key)=4",
        " 6 - Str_6  : h=2 \\ parent(key)=3",
        " 7 - Str_7  : h=0 \\ parent(key)=6",
    };
    runTest(keys, expected);
    assertEquals("Str_3", avlTree.remove(3));
    expected = new String[] {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 / parent(key)=4",
        " 4 - Str_4  : h=2 ROOT",
        " 5 - Str_5  : h=0 / parent(key)=6",
        " 6 - Str_6  : h=1 \\ parent(key)=4",
        " 7 - Str_7  : h=0 \\ parent(key)=6",
    };
    avlTree.inorder(avlTree.getRoot(), nodes);
    verify(nodes, expected);
  }

  @Test
  public void test20RemovingAtRoot6() {
    int[] keys = { 1 };
    String[] expected = {
        " 1 - Str_1  : h=0 ROOT",
    };
    runTest(keys, expected);
    assertEquals(null, avlTree.remove(8888));
    assertEquals(1, avlTree.size());
    runTest(keys, expected);
    assertEquals("Str_1", avlTree.remove(1));
    assertEquals(0, avlTree.size());
  }

  @Test
  public void test21RemovingEntryNotInTree() {
    Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
    int[] keys = { 1, 2, 3 };
    String[] expected = {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 ROOT",
        " 3 - Str_3  : h=0 \\ parent(key)=2",
    };
    runTest(keys, expected);
    assertNull(avlTree.remove(4));
    expected = new String[] {
        " 1 - Str_1  : h=0 / parent(key)=2",
        " 2 - Str_2  : h=1 ROOT",
        " 3 - Str_3  : h=0 \\ parent(key)=2",
    };
    avlTree.inorder(avlTree.getRoot(), nodes);
    verify(nodes, expected);
  }

  @Test
  public void test22StressTest() {
    final int SIZE = 10000;
    Random randomGenerator = new Random(1);
    // a Map to compare:
    Map<Integer, String> map = new Hashtable<>();
    // key-Counters: count for every key how many time it was generated
    Map<Integer, Integer> keyCounters = new Hashtable<>();
    // fill the Tree
    for (int i = 0; i < SIZE; i++) {
      int key = (int) (randomGenerator.nextFloat() * SIZE / 3);
      Integer numberOfKeys = keyCounters.get(key);
      if (numberOfKeys == null) {
        numberOfKeys = 1;
      } else {
        numberOfKeys++;
      }
      keyCounters.put(key, numberOfKeys);
      avlTree.put(key, "_" + i);
      map.put(key, "_" + i);
      assertEquals(keyCounters.size(), avlTree.size());
      assertEquals(map.size(), avlTree.size());
    }
    verifyInorder();
    // remove all Keys
    Integer[] keyArr = new Integer[1];
    keyArr = map.keySet().toArray(keyArr);
    for (int key : keyArr) {
      assertEquals(map.remove(key), avlTree.remove(key));
      assertEquals(map.size(), avlTree.size());
      verifyInorder();
    }
    assertEquals(0, avlTree.size());
  }

  private void verifyInorder() {
    Collection<Entry<Integer, String>> inorderList = avlTree.inorder();
    int last = Integer.MIN_VALUE;
    for (Entry<Integer, String> entry: inorderList) {
      Integer key = entry.getKey();
      assertTrue(key.compareTo(last) >= 0);
      last = key;
    }
  }

  private void runTest(int[] keys, String[] expected) {
    for (int key : keys) {
      avlTree.put(key, "Str_" + key);
    }
    Collection<AVLTreeImpl<Integer, String>.Node> nodes = new LinkedList<>();
    avlTree.inorder(avlTree.getRoot(), nodes);
    assertEquals(expected.length, nodes.size());
    verify(nodes, expected);
  }

  private static void verify(Collection<AVLTreeImpl<Integer, String>.Node> nodes, Stri
ng[] expected) {
    int i = 0;
    for (AVLTreeImpl<Integer, String>.Node node: nodes) {
      String nodeStr = node.toString();
      String expectedStr = expected[i];
      assertEquals(expectedStr, nodeStr);
      i++;
    }
  }

}

```