```
/*
 * HSR − Uebungen ’Algorithmen & Datenstrukturen 2’
 * Version: Mon Sep 24 12:08:35 CEST 2018
 */

package uebung02.as.aufgabe01;

import java.util.Collection;
import java.util.LinkedList;

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

  protected Node root;

  public static class Entry<K, V> {

    private K key;
    private V value;

    public Entry(K key, V value) {
      this.key = key;
      this.value = value;
    }

    protected K setKey(K key) {
      K oldKey = this.key;
      this.key = key;
      return oldKey;
    }

    public K getKey() {
      return key;
    }

    public V setValue(V value) {
      V oldValue = this.value;
      this.value = value;
      return oldValue;
    }

    public V getValue() {
      return value;
    }

    @Override
    public String toString() {
      StringBuilder result = new StringBuilder();
      result.append("[").append(key).append("/").append(value).append("]");
      return result.toString();
    }

  } // End of class Entry

  protected class Node {

    private Entry<K, V> entry;
    private Node leftChild;
    private Node rightChild;

    public Node(Entry<K, V> entry) {
      this.entry = entry;
    }

    public Node(Entry<K, V> entry, Node leftChild, Node rightChild) {
      this.entry = entry;
      this.leftChild = leftChild;
      this.rightChild = rightChild;
    }
```

```

    public Entry<K, V> getEntry() {
      return entry;
    }

    public Entry<K, V> setEntry(Entry<K, V> entry) {
      Entry<K, V> oldEntry = entry;
      this.entry = entry;
      return oldEntry;
    }

    public Node getLeftChild() {
      return leftChild;
    }

    public void setLeftChild(Node leftChild) {
      this.leftChild = leftChild;
    }

    public Node getRightChild() {
      return rightChild;
    }

    public void setRightChild(Node rightChild) {
      this.rightChild = rightChild;
    }

  } // End of class Node

  public Entry<K, V> insert(K key, V value) {
    // TODO Implement here...
    return null;
  }

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

  public void clear() {
    // TODO Implement here...
  }

  public Entry<K, V> find(K key) {
    // TODO Implement here...
    return null;
  }

  /**
   * Returns a collection with all entries with key.
   *
   * @param key
   *          The key to be searched.
   * @return Collection of all entries found. An empty collection is returned if
   *         no entry with key is found.
   */
  public Collection<Entry<K, V>> findAll(K key) {
    // TODO Implement here...
    return null;
  }
```

```java

  /**
   * Returns a collection with all entries in inorder.
   *
   * @return Inorder−Collection of all entries.
   */
  public Collection<Entry<K, V>> inorder() {
    // TODO Implement here...
    return null;
  }

  /**
   * Prints the entries of the tree as a list in inorder to the console.
   */
  public void printInorder() {
    // TODO Implement here...
  }

  public Entry<K, V> remove(Entry<K, V> entry) {
    // TODO Implement here...
    return null;
  }

  /**
   * The height of the tree.
   *
   * @return The actual height. −1 for an empty tree.
   */
  public int getHeight() {
    // TODO Implement here...
    return −1;
  }

  public int size() {
    // TODO Implement here...
    return −1;
  }

  public boolean isEmpty() {
    // TODO Implement here...
    return true;
  }
```

```java

  public static void main(String[] args) {

    // Example from lecture "L?schen (IV/IV)":
    BinarySearchTree<Integer, String> bst = new BinarySearchTree<>();
    //BinarySearchTree<Integer, String> bst = new BinarySearchTreeGVS<>();
    System.out.println("Inserting:");
    bst.insert(1, "Str1");
    bst.printInorder();
    bst.insert(3, "Str3");
    bst.printInorder();
    bst.insert(2, "Str2");
    bst.printInorder();
    bst.insert(8, "Str8");
    bst.printInorder();
    bst.insert(9, "Str9");
    bst.insert(6, "Str6");
    bst.insert(5, "Str5");
    bst.printInorder();

    System.out.println("Removeing 3:");
    Entry<Integer, String> entry = bst.find(3);
    System.out.println(entry);
    bst.remove(entry);
    bst.printInorder();

    //if (bst instanceof BinarySearchTreeGVS) {
    //  ((BinarySearchTreeGVS<Integer, String>)bst).gvsTree.disconnect();
    //}

  }

  /* Session−Log:

  Inserting:
  [1/Str1]
  [1/Str1] [3/Str3]
  [1/Str1] [2/Str2] [3/Str3]
  [1/Str1] [2/Str2] [3/Str3] [8/Str8]
  [1/Str1] [2/Str2] [3/Str3] [5/Str5] [6/Str6] [8/Str8] [9/Str9]
  Removeing 3:
  [3/Str3]
  [1/Str1] [2/Str2] [5/Str5] [6/Str6] [8/Str8] [9/Str9]

  */


} // End of class BinarySearchTree

```

```java
/*
 * HSR − Uebungen ’Algorithmen & Datenstrukturen 2’
 * Version: Mon Sep 24 12:08:35 CEST 2018
 */

package uebung02.as.aufgabe01;

import java.util.Iterator;
import java.util.Random;

import uebung02.as.aufgabe01.BinarySearchTree.Entry;

public class BinarySearchTreeTest {

  private static Random randomGenerator = new Random(1);

  private static BinarySearchTree<Integer, String> generateTree(int nodes) {
    int key;
    BinarySearchTree<Integer, String> ret = new BinarySearchTree<>();
    for (int i = 0; i < nodes; i++) {
      key = randomGenerator.nextInt() * Integer.MAX_VALUE;
      ret.insert(key, "String_" + i);
    }
    return ret;
  }

  public static void main(String[] args) {
    System.out.println("BINARY TREE TEST");
    System.out
        .println("Please be patient, the following operations may take some time...");
    final int TESTRUNS = 100;
    final int BEGINSIZE = 10000;
    final int VARYSIZE = 10;
    long startTime = System.currentTimeMillis();

    BinarySearchTree<Integer, String> bst = new BinarySearchTree<>();
    double avgHeight = 0;
    double avgEntries = 0;
    double avgTime = 0;
    for (int i = 0; i < TESTRUNS; i++) {
      startTime = System.currentTimeMillis();
      bst = generateTree(BEGINSIZE + i * VARYSIZE);
      avgTime += System.currentTimeMillis() − startTime;
      avgHeight += bst.getHeight();
      avgEntries += BEGINSIZE + i * VARYSIZE;
    }
    avgTime /= TESTRUNS;
    avgEntries /= TESTRUNS;
    avgHeight /= TESTRUNS;
    System.out.println("Test successful, results are as follows:");
    System.out.println("Average time for generation is: " + avgTime + "ms");
    System.out.println("Average entries are: " + avgEntries);
    System.out.println("Average height is: " + avgHeight);
    System.out.println("In h=C*log2(n), C=h/log2(n) = " + avgHeight
        / (Math.log(avgEntries) / Math.log(2)));
    System.out.println();
```

```java

    bst = generateTree(20);
    int search = 15138431;
    Entry<Integer, String> searchResult;
    bst.insert(search, "String_" + search);
    searchResult = bst.find(search);
    if (searchResult == null) {
      System.err.println("Search for node " + search + " failed!");
    } else {
      System.out.println("Search for node " + search + " successful!");
    }
    System.out.println();
    bst.insert(search, "String_" + search);
    bst.insert(search, "String_" + search);
    bst.insert(search, "String_" + search);
    Iterator<Entry<Integer, String>> it = bst.findAll(search).iterator();
    int count = 0;
    while (it.hasNext()) {
      count++;
      it.next();
      System.out.println("Search for node " + search + " successful!");
    }
    System.out.println("Search for node " + search + ": " + count
        + " nodes found!");
    System.out.println();
    it = bst.findAll(search).iterator();
    count = 0;
    while (it.hasNext()) {
      bst.remove(it.next());
    }

    it = bst.findAll(search).iterator();
    count = 0;
    while (it.hasNext()) {
      count++;
      it.next();
      System.out.println("Search for node " + search + " successful!");
    }
    System.out.println("Search for node " + search + ": " + count
        + " nodes found!");
  }

}

/* Session−Log:

BINARY TREE TEST
Please be patient, the following operations may take some time...
Test successful, results are as follows:
Average time for generation is: 9.07ms
Average entries are: 10495.0
Average height is: 30.81
In h=C*log2(n), C=h/log2(n) = 2.306584099301782

Search for node 15138431 successful!

Search for node 15138431 successful!
Search for node 15138431 successful!
Search for node 15138431 successful!
Search for node 15138431 successful!
Search for node 15138431: 4 nodes found!

Search for node 15138431: 0 nodes found!

*/
```

```java
/*
 * HSR − Uebungen ’Algorithmen & Datenstrukturen 2’
 * Version: Mon Sep 24 12:08:35 CEST 2018
 */

package uebung02.as.aufgabe01;

import static org.junit.Assert.*;

import java.util.Collection;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
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 uebung02.as.aufgabe01.BinarySearchTree.Entry;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class BinarySearchTreeJUnitTest {

  BinarySearchTree<Integer, String> bst;

  @Before
  public void setUp() {
    bst = new BinarySearchTree<Integer, String>();
  }

  @Test
  public void test01EmptySizeInsertClear() {
    assertTrue(bst.isEmpty());
    assertEquals(0, bst.size());
    bst.insert(1, "String_1");
    assertEquals(1, bst.size());
    assertFalse(bst.isEmpty());
    bst.insert(2, "String_2");
    assertEquals(2, bst.size());
    bst.insert(2, "String_2");
    assertEquals(3, bst.size());
    bst.clear();
    assertTrue(bst.isEmpty());
    assertEquals(0, bst.size());
  }

  @Test
  public void test02Find() {
    Entry<Integer, String> entry;
    entry = bst.find(1);
    assertNull(entry);
    Entry<Integer, String> insertedEntry = bst.insert(1, "String_1");
    entry = bst.find(1);
    assertNotNull(entry);
    assertEquals(Integer.valueOf(1), entry.getKey());
    assertEquals("String_1", entry.getValue());
    assertSame(insertedEntry, entry);
  }
```

```java

  @Test
  public void test03FindAll() {
    Collection<Entry<Integer, String>> col;
    col = bst.findAll(1);
    assertEquals(0, col.size());
    bst.insert(1, "String_1");
    col = bst.findAll(2);
    assertEquals(0, col.size());
    bst.insert(2, "String_2");
    col = bst.findAll(2);
    assertEquals(1, col.size());
    bst.insert(2, "String_2");
    col = bst.findAll(2);
    assertEquals(2, col.size());
  }

  @Test
  public void test04GetHeight() {
    assertEquals(−1, bst.getHeight());
    bst.insert(1, "String_1");
    assertEquals(0, bst.getHeight());
    bst.insert(2, "String_2");
    assertEquals(1, bst.getHeight());
  }

  @Test
  public void test05Remove() {
    Entry<Integer, String> entry = new Entry<>(1, "String_1");
    entry = bst.remove(entry);
    assertNull(entry);
    final Entry<Integer, String> entry1 = bst.insert(1, "String_1");
    entry = bst.remove(entry1);
    assertSame(entry, entry1);
    assertEquals(0, bst.size());
    final Entry<Integer, String> entry1a = bst.insert(1, "String_1a");
    final Entry<Integer, String> entry1b = bst.insert(1, "String_1b");
    assertEquals(2, bst.size());
    entry = bst.remove(entry1a);
    assertSame(entry1a, entry);
    assertEquals(1, bst.size());
    entry = bst.remove(entry1b);
    assertSame(entry1b, entry);
    assertEquals(0, bst.size());
  }

  @Test
  public void test06RemoveCase3() {
    bst.insert(1, "String_1");
    Entry<Integer, String> entryToRemove = bst.insert(3, "String_3");
    bst.insert(2, "String_2");
    bst.insert(8, "String_8");
    bst.insert(6, "String_6");
    bst.insert(9, "String_9");
    bst.insert(5, "String_5");
    assertEquals(7, bst.size());
    assertEquals(4, bst.getHeight());
    Entry<Integer, String> removedEntry =  bst.remove(entryToRemove);
    assertSame(entryToRemove, removedEntry);
    assertEquals(6, bst.size());
    assertEquals(3, bst.getHeight());
    bst.remove(bst.find(6));
    assertEquals(5, bst.size());
    assertEquals(3, bst.getHeight());
    bst.remove(bst.find(9));
    assertEquals(4, bst.size());
    assertEquals(2, bst.getHeight());
  }

  @Test
  public void test07RemoveCase3Special() {
    bst.insert(2, "String_2");
    bst.insert(1, "String_1");
    bst.insert(3, "String_3.1");
    bst.insert(3, "String_3.2");
    Collection<Entry<Integer, String>> col;
    col = bst.findAll(3);
    assertEquals(2, col.size());
    Entry<Integer, String> removedEntry = bst.remove(bst.find(2));
    assertNotNull(removedEntry);
    assertEquals("String_2", removedEntry.getValue());
    col = bst.findAll(3);
    assertEquals(2, col.size());
  }

  @Test
  public void test09StressTest() {
    final int SIZE = 10000;
    Random randomGenerator = new Random();
    List<Entry<Integer, String>> entriesList = new LinkedList<>();
    // key-Counters: count for every key how many time it was generated
    Map<Integer, Integer> keyCounters = new HashMap<>();
    // 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);
      Entry<Integer, String> entry = bst.insert(key, "String_" + i);
      entriesList.add(entry);
      assertEquals(i + 1, bst.size());
    }
    // verify the number of entries per key
    for (Map.Entry<Integer, Integer> keyEntry : keyCounters.entrySet()) {
      int key = keyEntry.getKey();
      int numberOfKeys = keyEntry.getValue();
      assertEquals(numberOfKeys, bst.findAll(key).size());
    }
```

```java

    // remove all entries
    int size = bst.size();
    for (Entry<Integer, String> entry : entriesList) {
      Entry<Integer, String> deletedEntry = bst.remove(entry);
      assertSame(entry, deletedEntry);
      assertEquals(--size, bst.size());
    }
  }

}

```