```java
/*
 * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
 * Version: Mon Sep 22 15:46:11 CEST 2025
 */

package ex02.baseline.task01;

import java.util.Collection;

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

  public 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;
    }
```



```java

    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;
  }
```

```
134
135     /**
136      * Returns a collection with all entries in inorder.
137      *
138      * @return Inorder-Collection of all entries.
139      */
140     public Collection<Entry<K, V>> inorder() {
141       // TODO Implement here...
142       return null;
143     }
144
145     /**
146      * Prints the entries of the tree as a list in inorder to the console.
147      */
148     public void printInorder() {
149       // TODO Implement here...
150     }
151
152     public Entry<K, V> remove(Entry<K, V> entry) {
153       // TODO Implement here...
154       return null;
155     }
156
157     /**
158      * The height of the tree.
159      *
160      * @return The current height. -1 for an empty tree.
161      */
162     public int getHeight() {
163       // TODO Implement here...
164       return -1;
165     }
166
167     public int size() {
168       // TODO Implement here...
169       return -1;
170     }
171
172     public boolean isEmpty() {
173       // TODO Implement here...
174       return true;
175     }
```



```
176
177     public static void main(String[] args) {
178
179       // Example from lecture "Löschen (IV/IV)":
180       BinarySearchTree<Integer, String> bst = new BinarySearchTree<>();
181       //BinarySearchTree<Integer, String> bst = new BinarySearchTreeADV<>("Loeschen (IV/
    IV)");
182       //BinarySearchTree<Integer, String> bst = new BinarySearchTreeADV<>("Loeschen (IV/
    IV)", 0, 4);
183       System.out.println("Inserting:");
184       bst.insert(1, "Str1");
185       bst.printInorder();
186       bst.insert(3, "Str3");
187       bst.printInorder();
188       bst.insert(2, "Str2");
189       bst.printInorder();
190       bst.insert(8, "Str8");
191       bst.printInorder();
192       bst.insert(9, "Str9");
193       bst.insert(6, "Str6");
194       bst.insert(5, "Str5");
195       bst.printInorder();
196
197       System.out.println("Removeing 3:");
198       Entry<Integer, String> entry = bst.find(3);
199       System.out.println(entry);
200       bst.remove(entry);
201       bst.printInorder();
202
203     }
204
205     /* Session-Log:
206
207     Inserting:
208     [1/Str1]
209     [1/Str1] [3/Str3]
210     [1/Str1] [2/Str2] [3/Str3]
211     [1/Str1] [2/Str2] [3/Str3] [8/Str8]
212     [1/Str1] [2/Str2] [3/Str3] [5/Str5] [6/Str6] [8/Str8] [9/Str9]
213     Removeing 3:
214     [3/Str3]
215     [1/Str1] [2/Str2] [5/Str5] [6/Str6] [8/Str8] [9/Str9]
216
217     */
218
219
220 } // End of class BinarySearchTree
221
```

```java
/*
 * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
 * Version: Mon Sep 22 15:46:11 CEST 2025
 */

package ex02.baseline.task01;

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

import ex02.baseline.task01.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: 4.12 ms
Average entries are: 10495.0
Average height is: 30.25
In h=C*log2(n), C=h/log2(n) = 2.2646598183667286

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 successful!
Search for node 15138431: 4 nodes found!

Search for node 15138431: 0 nodes found!

*/
```

```java
/*
 * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
 * Version: Mon Sep 22 15:46:11 CEST 2025
 */

package ex02.baseline.task01;

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 ex02.baseline.task01.BinarySearchTree.Entry;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class BinarySearchTreeJUnitTest {

  BinarySearchTree<Integer, String> bst;

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

  @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());
  }
```

```
107
108   @Test
109   public void test06RemoveCase3() {
110     bst.insert(1, "String_1");
111     Entry<Integer, String> entryToRemove = bst.insert(3, "String_3");
112     bst.insert(2, "String_2");
113     bst.insert(8, "String_8");
114     bst.insert(6, "String_6");
115     bst.insert(9, "String_9");
116     bst.insert(5, "String_5");
117     assertEquals(7, bst.size());
118     assertEquals(4, bst.getHeight());
119     Entry<Integer, String> removedEntry =  bst.remove(entryToRemove);
120     assertSame(entryToRemove, removedEntry);
121     assertEquals(6, bst.size());
122     assertEquals(3, bst.getHeight());
123     bst.remove(bst.find(6));
124     assertEquals(5, bst.size());
125     assertEquals(3, bst.getHeight());
126     bst.remove(bst.find(9));
127     assertEquals(4, bst.size());
128     assertEquals(2, bst.getHeight());
129   }
130
131   @Test
132   public void test07RemoveCase3Special() {
133     bst.insert(2, "String_2");
134     bst.insert(1, "String_1");
135     bst.insert(3, "String_3.1");
136     bst.insert(3, "String_3.2");
137     Collection<Entry<Integer, String>> col;
138     col = bst.findAll(3);
139     assertEquals(2, col.size());
140     Entry<Integer, String> removedEntry = bst.remove(bst.find(2));
141     assertNotNull(removedEntry);
142     assertEquals("String_2", removedEntry.getValue());
143     col = bst.findAll(3);
144     assertEquals(2, col.size());
145   }
146
147   @Test
148   public void test09StressTest() {
149     final int SIZE = 10000;
150     Random randomGenerator = new Random(1);
151     List<Entry<Integer, String>> entriesList = new LinkedList<>();
152     // key-Counters: count for every key how many time it was generated
153     Map<Integer, Integer> keyCounters = new HashMap<>();
154     // fill the Tree
155     for (int i = 0; i < SIZE; i++) {
156       int key = (int) (randomGenerator.nextFloat() * SIZE / 3);
157       Integer numberOfKeys = keyCounters.get(key);
158       if (numberOfKeys == null) {
159         numberOfKeys = 1;
160       } else {
161         numberOfKeys++;
162       }
163       keyCounters.put(key, numberOfKeys);
164       Entry<Integer, String> entry = bst.insert(key, "String_" + i);
165       entriesList.add(entry);
166       assertEquals(i + 1, bst.size());
167     }
168     // verify the number of entries per key
169     for (Map.Entry<Integer, Integer> keyEntry : keyCounters.entrySet()) {
170       int key = keyEntry.getKey();
171       int numberOfKeys = keyEntry.getValue();
172       assertEquals(numberOfKeys, bst.findAll(key).size());
173     }
```



```
174
175     // remove all entries
176     int size = bst.size();
177     for (Entry<Integer, String> entry : entriesList) {
178       Entry<Integer, String> deletedEntry = bst.remove(entry);
179       assertSame(entry, deletedEntry);
180       assertEquals(--size, bst.size());
181     }
182   }
183
184 }
185
```

```java
/*
 * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
 * Version: Mon Sep 22 15:46:11 CEST 2025
 */

package ex02.baseline.task01;

import ch.hsr.adv.commons.core.logic.domain.styles.ADVStyle;
import ch.hsr.adv.commons.core.logic.util.ADVException;
import ch.hsr.adv.commons.tree.logic.domain.ADVBinaryTreeNode;
import ch.hsr.adv.lib.bootstrapper.ADV;
import ch.hsr.adv.lib.tree.logic.binarytree.BinaryTreeModule;

@SuppressWarnings("unchecked")
public class BinarySearchTreeADV<K extends Comparable<? super K>, V>
    extends BinarySearchTree<K, V> {

  protected BinaryTreeModule advTree;

  protected class NodeADV extends BinarySearchTree<K, V>.Node
      implements ADVBinaryTreeNode<String> {

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

    @Override
    public String getContent() {
      return getEntry().getKey() + " / " + getEntry().getValue();
    }

    @Override
    public ADVStyle getStyle() {
      return null;
    }

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

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

  } // class BinaryTreeTestADV.NodeADV

  public BinarySearchTreeADV(String sessionName) {
    this(sessionName, -1, -1);
  }

  public BinarySearchTreeADV(String sessionName,
                             int maxLeftHeight, int maxRightHeight) {
    advTree = new BinaryTreeModule(sessionName);
    if ((maxLeftHeight != -1) && (maxLeftHeight != -1)) {
      advTree.setFixedTreeHeight(maxLeftHeight, maxRightHeight);
    }
    try {
      ADV.launch(null);
    } catch (ADVException e) {
      e.printStackTrace();
      System.exit(1);
    }
  }

  @Override
  protected Node newNode(Entry<K, V> entry) {
    return new NodeADV(entry);
  }
```



```java

  @Override
  public Entry<K, V> insert(K key, V value) {
    Entry<K, V> newEntry = super.insert(key, value);
    displayOnADV("insert(" + key + "," + value + ")");
    return newEntry;
  }

  @Override
  public Entry<K, V> remove(Entry<K, V> entry) {
    Entry<K, V> deletedEntry = super.remove(entry);
    displayOnADV("remove(" + entry + ")");
    return deletedEntry;
  }

  protected void displayOnADV(String advMessage) {
    advTree.setRoot((NodeADV) root);
    try {
      ADV.snapshot(advTree, "\n" + advMessage);
    } catch (ADVException e) {
      e.printStackTrace();
      System.exit(2);
    }
  }

}
```

```
1  /*
2   * OST - Uebungen 'Algorithmen & Datenstrukturen (AlgDat)'
3   * Version: Mon Sep 22 15:46:11 CEST 2025
4   */
5
6  package ex02.baseline.task01;
7
8  public class BinarySearchTreeTestADV {
9
10   public static void main(String[] args) {
11
12     BinarySearchTree<Integer, String> bts =
13         new BinarySearchTreeADV<>("Deleting internal node");
14         //new BinarySearchTreeADV<>("Deleting internal node", 0, 4);
15
16     // Example from script: deleting internal node (slide 14):
17     int[] iarr = { 1, 3, 2, 8, 6, 9, 5 };
18     for (int i : iarr) {
19       bts.insert(i, "Str" + i);
20     }
21     bts.remove(bts.find(3));
22
23   }
24
25 }
26
```