DICTIONARYSTRINGTRIES SOLUCIÓN

```java
/***********************************************************
*******************
 * Student's name:
 * Student's group:
 * Identity number (DNI if Spanish/passport if Erasmus):
 *
 * Data Structures. Grados en Informática. UMA.

***********************************************************
******************/

import dataStructures.dictionary.AVLDictionary;
import dataStructures.dictionary.Dictionary;
import dataStructures.list.LinkedList;
import dataStructures.list.List;
import dataStructures.tuple.Tuple2;

import java.util.Iterator;
import java.util.Objects;

//import static dataStructures.searchTree.AVL.searchRec;

public class DictionaryStringTrie<V> {
  protected static class Node<V> {
    V value;
    Dictionary<Character, Node<V>> children;

    Node() {
      this.value = null;
      this.children = new AVLDictionary<>();
    }
  }

  protected Node<V> root;


/***********************************************************
*******************
   * DO NOT WRITE ANY CODE ABOVE

***********************************************************
******************/

  // | = Exercise a - constructor
  public DictionaryStringTrie() {
      root = null;
  }
  // | = Exercise b - isEmpty
  public boolean isEmpty() {
      return root == null;
  }

  /*
  Devuelve 0 si value es null, 1 en caso contrario
   */

  // | = Exercise c - sizeValue
  protected static <V> int sizeValue(V value) {
      return value == null ? 0 : 1;
  }

  /*
    Devuelve el número total de nodos que contiene el Trie
   */

  // | = Exercise d - size
  public int size() {
      return size(root);
  }

  protected static <V> int size(Node<V> n) {
      if(n==null) return 0;
      int suma = 0;
      if(n.children!=null){
          for(Character c : n.children.keys())
suma+=size(n.children.valueOf(c));
      }return 1 + suma;
  }

  /*
  Devuelve el nodo hijo a través de un carácter c, o null
si este no se encuentra definido
   */

  // | = Exercise e - childOf
  protected static <V> Node<V> childOf(char c, Node<V>
node) {
      if(node==null) return null;
      return node.children.valueOf(c);
  }

  /*
  Devuelve el valor de la palabra pasada por parámetro, o
null si no se encuentra
   */

  // | = Exercise f - search
  public V search(String str) {
    return search(str,root);
  }

  protected static <V> V search(String str, Node<V> node) {
      return node!=null ? (str.isEmpty() ? node.value :
search(str.substring(1), childOf(str.charAt(0),node))) :
null;
  }

  /*
  Inserta un String, o actualiza el valor si ya está la
palabra
   */
  // | = Exercise g - insert
  public void insert(String str, V value) {
      root = insert(str,value,root);
  }

  protected static <V> Node<V> insert(String str, V value,
Node<V> node) {
      if(node==null && str.isEmpty()){
          node = new Node<>();
          node.value = value;
      }else if(node==null){
          node = new Node<>();

node.children.insert(str.charAt(0),insert(str.substring(1),
value,null));
      }else if(str.isEmpty()){
          node.value = value;
      }else{
          Node<V> hijo = childOf(str.charAt(0),node);
          if(hijo==null)
node.children.insert(str.charAt(0),insert(str.substring(1),
value,null));
          else{

node.children.insert(str.charAt(0),insert(str.substring(1),
value,hijo));
          }
      }
    return node;
  }


/***********************************************************
*******************
   * ONLY FOR PART TIME STUDENTS

***********************************************************
```

```
*****************/

  public String toString() {
    StringBuilder sb = new StringBuilder();
    if (root != null) {
      sb.append(root.getClass().getSimpleName());
      sb.append(' ');
      sb.append(root.value);
      sb.append('\n');
      toString(sb,1, root);
    }
    return sb.toString();
  }

  private static <V> void toString(StringBuilder sb, int n,
Node<V> node) {
    for (Tuple2<Character, Node<V>> par :
node.children.keysValues()) {
      char c = par._1();
      Node<V> child = par._2();
      tabulate(sb, n);
      sb.append(c);
      sb.append(" -> ");
      sb.append(node.getClass().getSimpleName());
      sb.append(' ');
      sb.append(child.value);
      sb.append('\n');
      toString(sb, n + 1, child);
    }
  }

  private static void tabulate(StringBuilder sb, int n) {
    for (int i = 0; i < 6*n; i++) {
      sb.append(' ');
    }
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return
false;
    DictionaryStringTrie<?> that =
(DictionaryStringTrie<?>) o;
    return equals(root, that.root);
  }

  private static <V> boolean equals(Node<V> node, Node<?>
that) {
    if (node == that) return true;
    if(!Objects.equals(node.value, that.value))
```

```
        return false;
      // same values
      for(char c : node.children.keys())
        if(!that.children.isDefinedAt(c))
          return false;
      for(char c : that.children.keys())
        if(!node.children.isDefinedAt(c))
          return false;
      // same keys
      for(Tuple2<Character, Node<V>> t :
node.children.keysValues()) {
        char c = t._1();
        Node<V> child = t._2();
        if(!equals(child, that.children.valueOf(c)))
          return false;
      }
      // same associations
      return true;
    }

    public static DictionaryStringTrie<Integer> sampleTrie()
{
      // bat -> 0  be -> 1  bed -> 2  cat -> 3  to -> 4  toe
-> 5
      DictionaryStringTrie<Integer> trie = new
DictionaryStringTrie<>();
      Node<Integer> n0 = new Node<>();
      Dictionary<Character, Node<Integer>> d0 = n0.children;
      Node<Integer> n1 = new Node<>();
      Dictionary<Character, Node<Integer>> d1 = n1.children;
      Node<Integer> n2 = new Node<>();
      Dictionary<Character, Node<Integer>> d2 = n2.children;
      Node<Integer> n3 = new Node<>();
      Dictionary<Character, Node<Integer>> d3 = n3.children;
      Node<Integer> n4 = new Node<>();
      Dictionary<Character, Node<Integer>> d4 = n4.children;
      Node<Integer> n5 = new Node<>();
      Dictionary<Character, Node<Integer>> d5 = n5.children;
      Node<Integer> n6 = new Node<>();
      Dictionary<Character, Node<Integer>> d6 = n6.children;
      Node<Integer> n7 = new Node<>();
      Dictionary<Character, Node<Integer>> d7 = n7.children;
      Node<Integer> n8 = new Node<>();
      Node<Integer> n9 = new Node<>();
      Node<Integer> n10 = new Node<>();
      Node<Integer> n11 = new Node<>();
      d0.insert('b',n1);
      d0.insert('c',n2);
      d0.insert('t',n3);
      d1.insert('a',n4);
      d1.insert('e',n5);
      n4.value = 4;
      Node<Integer> n5 = new Node<>();
      Node<Integer> n6 = new Node<>();
      Node<Integer> n7 = new Node<>();
      Node<Integer> n8 = new Node<>();
      n8.value = 5;
      n0.children.insert('a', n1);
      n1.children.insert('b', n2);
      n1.children.insert('c', n5);
      n2.children.insert('c', n3);
      n2.children.insert('d', n4);
      n5.children.insert('d', n6);
      n6.children.insert('e', n7);
      n7.children.insert('f', n8);
      trie.root = n0;
      return trie;
    }

    public static DictionaryStringTrie<Integer> sampleTrie3()
  {
      // abcd -> 1
      DictionaryStringTrie<Integer> trie = new
  DictionaryStringTrie<>();
      Node<Integer> n0 = new Node<>();
      Node<Integer> n1 = new Node<>();
      Node<Integer> n2 = new Node<>();
      Node<Integer> n3 = new Node<>();
      Node<Integer> n4 = new Node<>();
      n4.value = 1;
      n0.children.insert('a', n1);
      n1.children.insert('b', n2);
      n2.children.insert('c', n3);
      n3.children.insert('d', n4);
      trie.root = n0;
      return trie;
    }

    public static DictionaryStringTrie<Integer> sampleTrie4()
  {
      // abcd -> 1  def -> 2
      DictionaryStringTrie<Integer> trie = new
  DictionaryStringTrie<>();
      Node<Integer> n0 = new Node<>();
      Node<Integer> n1 = new Node<>();
      Node<Integer> n2 = new Node<>();
      Node<Integer> n3 = new Node<>();
      Node<Integer> n4 = new Node<>();
      n4.value = 1;
      Node<Integer> n5 = new Node<>();
      Node<Integer> n6 = new Node<>();
      Node<Integer> n7 = new Node<>();
    n7.value = 2;
    n0.children.insert('a', n1);
    n0.children.insert('d', n5);
    n1.children.insert('b', n2);
    n2.children.insert('c', n3);
    n3.children.insert('d', n4);
    n5.children.insert('e', n6);
    n6.children.insert('f', n7);
    trie.root = n0;
    return trie;
  }
}
```

```
  // | = Exercise b - isEmpty
  public boolean isEmpty() {
      return root == null;
  }

  /*
  Devuelve 0 si value es null, 1 en caso contrario
   */

  // | = Exercise c - sizeValue
  protected static <V> int sizeValue(V value) {
      return value == null ? 0 : 1;
  }

  /*
    Devuelve el número total de nodos que contiene el Trie
   */

  // | = Exercise d - size
  public int size() {
      return size(root);
  }

  protected static <V> int size(Node<V> n) {
      if(n==null) return 0;
      int suma = 0;
      if(n.children!=null){
          for(Character c : n.children.keys())
suma+=size(n.children.valueOf(c));
      }return 1 + suma;
  }

  /*
  Devuelve el nodo hijo a través de un carácter c, o null
si este no se encuentra definido
   */

  // | = Exercise e - childOf
  protected static <V> Node<V> childOf(char c, Node<V>
node) {
      if(node==null) return null;
      return node.children.valueOf(c);
  }

  /*
  Devuelve el valor de la palabra pasada por parámetro, o
null si no se encuentra
   */

  // | = Exercise f - search
  public V search(String str) {
    return search(str,root);
  }

  protected static <V> V search(String str, Node<V> node) {
      return node!=null ? (str.isEmpty() ? node.value :
search(str.substring(1), childOf(str.charAt(0),node))) :
null;
  }

  /*
  Inserta un String, o actualiza el valor si ya está la
palabra
   */
  // | = Exercise g - insert
  public void insert(String str, V value) {
      root = insert(str,value,root);
  }

  protected static <V> Node<V> insert(String str, V value,
Node<V> node) {
      if(node==null && str.isEmpty()){
          node = new Node<>();
          node.value = value;
      }else if(node==null){
          node = new Node<>();

node.children.insert(str.charAt(0),insert(str.substring(1),
value,null));
      }else if(str.isEmpty()){
          node.value = value;
      }else{
          Node<V> hijo = childOf(str.charAt(0),node);
          if(hijo==null)
node.children.insert(str.charAt(0),insert(str.substring(1),
value,null));
          else{

node.children.insert(str.charAt(0),insert(str.substring(1),
value,hijo));
          }
      }
    return node;
  }


/************************************************************
*******************
   * ONLY FOR PART TIME STUDENTS

************************************************************
```

```
*****************/

  public String toString() {
    StringBuilder sb = new StringBuilder();
    if (root != null) {
      sb.append(root.getClass().getSimpleName());
      sb.append(' ');
      sb.append(root.value);
      sb.append('\n');
      toString(sb,1, root);
    }
    return sb.toString();
  }

  private static <V> void toString(StringBuilder sb, int n,
Node<V> node) {
    for (Tuple2<Character, Node<V>> par :
node.children.keysValues()) {
      char c = par._1();
      Node<V> child = par._2();
      tabulate(sb, n);
      sb.append(c);
      sb.append(" -> ");
      sb.append(node.getClass().getSimpleName());
      sb.append(' ');
      sb.append(child.value);
      sb.append('\n');
      toString(sb, n + 1, child);
    }
  }

  private static void tabulate(StringBuilder sb, int n) {
    for (int i = 0; i < 6*n; i++) {
      sb.append(' ');
    }
  }

  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null || getClass() != o.getClass()) return
false;
    DictionaryStringTrie<?> that =
(DictionaryStringTrie<?>) o;
    return equals(root, that.root);
  }

  private static <V> boolean equals(Node<V> node, Node<?>
that) {
    if (node == that) return true;
    if(!Objects.equals(node.value, that.value))
```

```
        return false;
      // same values
      for(char c : node.children.keys())
        if(!that.children.isDefinedAt(c))
          return false;
      for(char c : that.children.keys())
        if(!node.children.isDefinedAt(c))
          return false;
      // same keys
      for(Tuple2<Character, Node<V>> t :
node.children.keysValues()) {
        char c = t._1();
        Node<V> child = t._2();
        if(!equals(child, that.children.valueOf(c)))
          return false;
      }
      // same associations
      return true;
    }

    public static DictionaryStringTrie<Integer> sampleTrie()
{
      // bat -> 0  be -> 1  bed -> 2  cat -> 3  to -> 4  toe
-> 5
      DictionaryStringTrie<Integer> trie = new
DictionaryStringTrie<>();
      Node<Integer> n0 = new Node<>();
      Dictionary<Character, Node<Integer>> d0 = n0.children;
      Node<Integer> n1 = new Node<>();
      Dictionary<Character, Node<Integer>> d1 = n1.children;
      Node<Integer> n2 = new Node<>();
      Dictionary<Character, Node<Integer>> d2 = n2.children;
      Node<Integer> n3 = new Node<>();
      Dictionary<Character, Node<Integer>> d3 = n3.children;
      Node<Integer> n4 = new Node<>();
      Dictionary<Character, Node<Integer>> d4 = n4.children;
      Node<Integer> n5 = new Node<>();
      Dictionary<Character, Node<Integer>> d5 = n5.children;
      Node<Integer> n6 = new Node<>();
      Dictionary<Character, Node<Integer>> d6 = n6.children;
      Node<Integer> n7 = new Node<>();
      Dictionary<Character, Node<Integer>> d7 = n7.children;
      Node<Integer> n8 = new Node<>();
      Node<Integer> n9 = new Node<>();
      Node<Integer> n10 = new Node<>();
      Node<Integer> n11 = new Node<>();
      d0.insert('b',n1);
      d0.insert('c',n2);
      d0.insert('t',n3);
      d1.insert('a',n4);
      d1.insert('e',n5);
      n4.value = 4;
      Node<Integer> n5 = new Node<>();
      Node<Integer> n6 = new Node<>();
      Node<Integer> n7 = new Node<>();
      Node<Integer> n8 = new Node<>();
      n8.value = 5;
      n0.children.insert('a', n1);
      n1.children.insert('b', n2);
      n1.children.insert('c', n5);
      n2.children.insert('c', n3);
      n2.children.insert('d', n4);
      n5.children.insert('d', n6);
      n6.children.insert('e', n7);
      n7.children.insert('f', n8);
      trie.root = n0;
      return trie;
    }

    public static DictionaryStringTrie<Integer> sampleTrie3()
  {
      // abcd -> 1
      DictionaryStringTrie<Integer> trie = new
  DictionaryStringTrie<>();
      Node<Integer> n0 = new Node<>();
      Node<Integer> n1 = new Node<>();
      Node<Integer> n2 = new Node<>();
      Node<Integer> n3 = new Node<>();
      Node<Integer> n4 = new Node<>();
      n4.value = 1;
      n0.children.insert('a', n1);
      n1.children.insert('b', n2);
      n2.children.insert('c', n3);
      n3.children.insert('d', n4);
      trie.root = n0;
      return trie;
    }

    public static DictionaryStringTrie<Integer> sampleTrie4()
  {
      // abcd -> 1  def -> 2
      DictionaryStringTrie<Integer> trie = new
  DictionaryStringTrie<>();
      Node<Integer> n0 = new Node<>();
      Node<Integer> n1 = new Node<>();
      Node<Integer> n2 = new Node<>();
      Node<Integer> n3 = new Node<>();
      Node<Integer> n4 = new Node<>();
      n4.value = 1;
      Node<Integer> n5 = new Node<>();
      Node<Integer> n6 = new Node<>();
      Node<Integer> n7 = new Node<>();
```

```
        n7.value = 2;
        n0.children.insert('a', n1);
        n0.children.insert('d', n5);
        n1.children.insert('b', n2);
        n2.children.insert('c', n3);
        n3.children.insert('d', n4);
        n5.children.insert('e', n6);
        n6.children.insert('f', n7);
        trie.root = n0;
        return trie;
    }
}
```