Exercise HashMap

Unsorted Vector	Sorted Vector	Unsorted LinkedList	Sorted LinkedList	Hash table
O(1)	O(1)	O(n)	O(n)	O(1)
O(n)	O(n)	O(1)	O(1)	O(1)
O(n)	O(n)	O(1)	O(1)	O(1)

It is assumed that by either *adding* or *removing* an element in a LinkedList, we refer to the last element. Otherwise, the time complexity would be $O(n)$.

Comments

Vectors are great because each *look up* is performed in constant time, while *adding* and *removing* takes linear time. The latter is because we when we add a new item, we actually create a new vector and thus we have to copy all the values from the old vector to the new one. **Linked Lists** are good at *adding* and *removing* items that are either in the beginning or the end of a list. However, the *search* operation is performed in linear time as we need to go through the entire list (in general). **Hash table** aims to perform *search*, *adding* and *removing* in constant time. Occasionally, we will have to resize the table, which will be costly, but on average, we still get constant time for each operation.

StringHash class

```
1  package inda4;
2
3  import java.util.List;
4  import java.util.LinkedList;
5
6  /**
7   * A hash table of strings.
8   *
9   * @author Stefan Nilsson
10  * @version 2010-07-21
11  */
12 public class StringHash implements StringDictionary {
13     private List<String>[] table;
14
15     /**
16      * Creates a hash table with the given capacity.
17      *
18      * @throws IllegalArgumentException if capacity <= 0.
19      */
20     public StringHash(int capacity) {
21         if (capacity <= 0)
22             throw new IllegalArgumentException("capacity=" + capacity);
23
24         // We want to do the following:
25         //
26         //     table = new LinkedList<String>[capacity];
27         //
28         // However, that won't compile ("generic array creation")
29         // since Java generics and arrays don't get along very well.
30         // Instead we need to do the following:
31         //
32         //     table = new LinkedList[capacity];
33         //
34         // The above will compile, but with a warning. The proper
35         // approach is to document why the warning can be safely
```

```
36          // ignored and then suppress the warning. Thus:
37
38          /*
39           * This array will contain only LinkedList<String>
40           * instances, all created in the add method. This
41           * is sufficient to ensure type safety.
42           */
43          @SuppressWarnings("unchecked") // for this declaration only
44          List<String>[] t = new LinkedList[capacity];
45
46          table = t;
47      }
48
49      /**
50       * Adds the given string to this dictionary.
51       * Returns <code>true</code> if the dictionary
52       * did not already contain the given string.
53       *
54       * Complexity: O(1) expected time.
55       */
56      @Override
57      public boolean add(String s) {
58          int hashCode = getIndex(s);
59
60          List<String> ls = table[hashCode];
61
62          if(ls == null)
63          {
64              ls = new LinkedList<String>();
65              ls.add(s);
66              table[hashCode] = ls;
67
68              return true;
69          }
70          else
71          {
72              ls.add(s);
73              table[hashCode] = ls; // do we need this extra reference?
74
75              return false;
76          }
77      }
78
79      /**
80       * Removes the given string from this dictionary
81       * if it is present. Returns <code>true</code> if
82       * the dictionay contained the specified element.
83       *
84       * Complexity: O(1) expected time.
85       */
86      @Override
87      public boolean remove(String s) {
88
89          int hashCode = getIndex(s);
90
91          List<String> ls = table[hashCode];
```

```
92
93          if(ls == null)
94              return false;
95
96          return ls.remove(s);
97      }
98
99      /**
100      * Returns <code>true</code> if the string is
101      * in this dictionary.
102      *
103      * Complexity: O(1) expected time.
104      */
105     @Override
106     public boolean contains(String s) {
107
108         int hashCode = getIndex(s);
109
110         List<String> ls = table[hashCode];
111
112         if(ls == null)
113             return false;
114
115         return ls.contains(s);
116     }
117
118     private int abs(int x)
119     {
120         if(x < 0)
121             return -x;
122         return x;
123     }
124
125     private int getIndex(String s)
126     {
127         return abs(s.hashCode() % table.length);
128     }
129 }
```

StringHashTest class

```
1  package inda4;
2
3  import static org.junit.Assert.*;
4
5  import org.junit.Test;
6
7  public class StringHashTest {
8
9      @Test
10     public void CreateNewHashListTest() {
11         StringHash a = new StringHash(3);
12
13         try
14         {
15             StringHash b = new StringHash(-3);
```

```
16              fail("should not happen");
17
18          } catch(IllegalArgumentException e){}
19
20      }
21
22      @Test
23      public void AddNewElementTest()
24      {
25          StringHash a = new StringHash(5);
26
27          assertTrue(a.add("hi"));
28          assertFalse(a.add("hi"));
29          assertTrue(a.add("hello"));
30          assertTrue(a.add("hi3"));
31      }
32
33      @Test
34      public void ContainsElementTest()
35      {
36          StringHash a = new StringHash(5);
37
38          assertTrue(a.add("hi"));
39          assertTrue(a.add("hello"));
40          assertTrue(a.add("hi3"));
41
42          assertTrue(a.contains("hello"));
43          assertFalse(a.contains("test"));
44      }
45
46      @Test
47      public void RemoveElementTest()
48      {
49          StringHash a = new StringHash(5);
50
51          assertTrue(a.add("hi"));
52          assertTrue(a.add("hello"));
53          assertTrue(a.add("hi3"));
54
55          assertTrue(a.contains("hello"));
56          assertTrue(a.remove("hello"));
57          assertFalse(a.contains("hello"));
58      }
59
60
61  }
```