

CS240: Homework 5

Eli Zupke

November 12, 2017

The GitHub folder can be found at [URL](#).

Contents

| | | |
|-------------------|--|--------------------|
| 1 | src/cs240/Test.java | 1 |
| 2 | src/cs240/DictionaryInterface.java | 5 |
| 3 | src/cs240/SortedDictionaryStaticArray.java | 6 |
| 4 | src/cs240/SortedDictionaryLinkedData.java | 11 |

1 src/cs240/Test.java

```
1 package cs240;
2
3 import java.util.Iterator;
4
5 public class Test {
6
7     private final static int TEST_SIZE = 10;
8
9     public static void main(String args[]) {
10
11         DictionaryInterface<Integer, String> dict = new
SortedDictionaryStaticArray<>(TEST_SIZE);
12         dictionaryTest(dict, "Static Array Dict");
13         dict = new SortedDictionaryLinkedData<>();
14         dictionaryTest(dict, "Linked Data Dict");
15
16     }
17
18     public static void dictionaryTest(DictionaryInterface<Integer, String> dict,
String listName) {
19
20         System.out.println("=====");
21         System.out.println("Testing " + listName);
22         System.out.println("=====");
23
24         Integer[] keyData = new Integer[TEST_SIZE];
25         String[] valueDataA = new String[TEST_SIZE];
26         String[] valueDataB = new String[TEST_SIZE];
```

```

27
28     for (int i = 0; i < TEST_SIZE; i++) {
29         keyData[i] = new Integer(i);
30         valueDataA[i] = "A:" + i;
31         valueDataB[i] = "B:" + i;
32     }
33
34     // Add items to the dictionary
35
36     boolean valueCheck = true;
37     boolean containmentCheck = true;
38     boolean sizeCheck = true;
39     boolean replacementCheck = true;
40     boolean removalCheck = true;
41     boolean emptyCheck = true;
42
43     boolean iteratorHasNext = true;
44     boolean iteratorNext = true;
45     boolean iteratorRemove = true;
46
47     // The dictionary should be empty
48     emptyCheck = booleanLatch(emptyCheck, dict.isEmpty());
49     // Add the even numbers to the dictionary
50     for (int i = 0; i < TEST_SIZE; i+= 2) {
51
52         // System.out.println("Adding " + i);
53         String v = dict.add(keyData[i], valueDataA[i]);
54
55         // We are replacing nothing, so this should always return null.
56         replacementCheck = booleanLatch(replacementCheck, v == null);
57
58         // Check to make sure the size of the dictionary is correct
59         sizeCheck = booleanLatch(sizeCheck, dict.getSize() == (i / 2) + 1);
60
61         // The dictionary should not be empty
62         emptyCheck = booleanLatch(emptyCheck, !dict.isEmpty());
63
64         for (int j = 0; j <= i; j += 2) {
65             // Ensure that the dictionary contains the things we put in
66             containmentCheck = booleanLatch(containmentCheck, dict.contains(
keyData[j]));
67             // Ensure that the value is correct
68             valueCheck = booleanLatch(valueCheck, dict.getValue(keyData[j]) ==
valueDataA[j]);
69         }
70
71         // Ensure that the dictionary does not contain the things we didn't
put in
72         for (int j = i + 2; j < TEST_SIZE; j += 2) {
73             if (dict.contains(keyData[j])) {
74                 containmentCheck = false;
75             }
76         }
77     }
78
79
80     // Add all of the numbers to the dictionary, but with the B values.
81     for (int i = 0; i < TEST_SIZE; i++) {
82         // System.out.println("=====");

```

```

83         // System.out.println("Adding " + i);
84         String v = dict.add(keyData[i], valueDataB[i]);
85
86         // dict.getSize(); // only do this because of testing side-effect
87
88         if (i % 2 == 0) {
89             // We are replacing the values we entered earlier
90             replacementCheck = booleanLatch(replacementCheck, v == valueDataA[
i]);
91         } else {
92             // We are replacing nothing, so this should always return null.
93             replacementCheck = booleanLatch(replacementCheck, v == null);
94         }
95
96
97         for (int j = 1; j <= i; j += 2) {
98             // Ensure that the dictionary contains the odd numbers we put in
99             containmentCheck = booleanLatch(containmentCheck, dict.contains(
keyData[j]));
100             // Ensure that the value is correct
101             valueCheck = booleanLatch(valueCheck, dict.getValue(keyData[j]) ==
valueDataB[j]);
102         }
103
104         // Ensure that the dictionary does not contain the things we didn't
put in
105         for (int j = i + 2; j < TEST_SIZE; j += 2) {
106             booleanLatch(containmentCheck, !dict.contains(keyData[j]));
107         }
108     }
109
110
111     Iterator<Integer> iK = dict.getKeyIterator();
112     Iterator<String> iV = dict.getValueIterator();
113     if (iK == null || iV == null) {
114         iteratorHasNext = false;
115         iteratorNext = false;
116         iteratorRemove = false;
117     } else {
118         for (int i = 0; i < TEST_SIZE; i++) {
119             iteratorHasNext = booleanLatch(iteratorHasNext, iK.hasNext() && iV
.hasNext());
120             Integer k = iK.next();
121             String v = iV.next();
122             iteratorNext = booleanLatch(iteratorNext, k == keyData[k] && v ==
valueDataB[i]);
123         }
124         iteratorHasNext = booleanLatch(iteratorHasNext, !iK.hasNext() && !iV.
hasNext());
125     }
126
127     for (int i = 0; i < TEST_SIZE; i++) {
128         String v = dict.remove(keyData[i]);
129         removalCheck = booleanLatch(removalCheck, v == valueDataB[i] && dict.
getSize() == (TEST_SIZE - i - 1));
130     }
131
132     // The dictionary should be empty
133     emptyCheck = booleanLatch(emptyCheck, dict.isEmpty());

```

```

134
135     // Fill up the dictionary again.
136     for (int i = 0; i < TEST_SIZE; i++) {
137         dict.add(keyData[i], valueDataA[i]);
138     }
139
140     iK = dict.getKeyIterator();
141
142     for (int i = TEST_SIZE; i > 0; i--) {
143         iK.next();
144         iK.remove();
145         iteratorRemove = booleanLatch(iteratorRemove, dict.getSize() == i - 1)
146     };
147     // Fill up the dictionary again.
148     for (int i = 0; i < TEST_SIZE; i++) {
149         dict.add(keyData[i], valueDataB[i]);
150     }
151
152     iV = dict.getValueIterator();
153
154     for (int i = TEST_SIZE; i > 0; i--) {
155         iV.next();
156         iV.remove();
157         iteratorRemove = booleanLatch(iteratorRemove, dict.getSize() == i - 1)
158     };
159
160
161     printTestResult(removalCheck, listName, "Removal Check");
162     printTestResult(emptyCheck, listName, "Empty Check");
163     printTestResult(sizeCheck, listName, "Size Check");
164     printTestResult(valueCheck, listName, "Value Check");
165     printTestResult(containmentCheck, listName, "Containment Check");
166     printTestResult(iteratorHasNext, listName, "Iterator Check (Has Next)");
167     printTestResult(iteratorNext, listName, "Iterator Check (Next)");
168     printTestResult(iteratorRemove, listName, "Iterator Check (Remove)");
169 }
170
171
172 /**
173  * Print the results of a test
174  * @param result Whether the test passed
175  * @param name The name of the test
176  */
177 public static void printTestResult(boolean result, String listName, String
name) {
178     if (result) System.out.println(listName + " passed " + name + ".");
179     else System.err.println(listName + " failed " + name + "!");
180 }
181
182 /**
183  * Returns the value of a flag such that the value of the flag is set to false
184  * if the new data is set to false,
185  * but is never returned to true.
186  * @param flag The flag whose value we are returning
187  * @param newData The new data we are getting
188  * @return The new value of the flag
189  */

```

```

189     private static boolean booleanLatch(boolean flag, boolean newData) {
190         if (flag && newData) {
191             return true;
192         }
193         return false;
194     }
195
196     /**
197      * Take an array of objects, and print them out nicely.
198      * @param l The array to print
199      */
200     public static void printArray(Object[] l) {
201
202         System.out.print("[");
203         for (int i = 0; i < l.length; i++) {
204             System.out.print(l[i]);
205             if (i+1 < l.length) {
206                 System.out.print(", ");
207             }
208         }
209         System.out.println("]");
210     }
211
212
213
214 }

```

2 src/cs240/DictionaryInterface.java

```

1 package cs240;
2
3 import java.util.Iterator;
4 /**
5  * An interface for a dictionary with distinct search keys.
6  * @author Frank M. Carrano
7  * @author Timothy M. Henry
8  * @version 4.0
9  */
10 public interface DictionaryInterface<K, V>
11 {
12     /** Adds a new entry to this dictionary. If the given search key already
13      * exists in the dictionary, replaces the corresponding value.
14      * @param key An object search key of the new entry.
15      * @param value An object associated with the search key.
16      * @return Either null if the new entry was added to the dictionary
17      * or the value that was associated with key if that value
18      * was replaced. */
19     public V add(K key, V value);
20
21     /** Removes a specific entry from this dictionary.
22      * @param key An object search key of the entry to be removed.
23      * @return Either the value that was associated with the search key
24      * or null if no such object exists. */

```

```

25     public V remove(K key);
26
27     /** Retrieves from this dictionary the value associated with a given
28         search key.
29         @param key  An object search key of the entry to be retrieved.
30         @return  Either the value that is associated with the search key
31                 or null if no such object exists. */
32     public V getValue(K key);
33
34     /** Sees whether a specific entry is in this dictionary.
35         @param key  An object search key of the desired entry.
36         @return  True if key is associated with an entry in the dictionary. */
37     public boolean contains(K key);
38
39     /** Creates an iterator that traverses all search keys in this dictionary.
40         @return  An iterator that provides sequential access to the search
41                 keys in the dictionary. */
42     public Iterator<K> getKeyIterator();
43
44     /** Creates an iterator that traverses all values in this dictionary.
45         @return  An iterator that provides sequential access to the values
46                 in this dictionary. */
47     public Iterator<V> getValueIterator();
48
49     /** Sees whether this dictionary is empty.
50         @return  True if the dictionary is empty. */
51     public boolean isEmpty();
52
53     /** Gets the size of this dictionary.
54         @return  The number of entries (key-value pairs) currently
55                 in the dictionary. */
56     public int getSize();
57
58     /** Removes all entries from this dictionary. */
59     public void clear();
60 } // end DictionaryInterface

```

3 src/cs240/SortedDictionaryStaticArray.java

```

1  package cs240;
2
3  import java.util.Iterator;
4  import java.util.NoSuchElementException;
5
6
7  /**
8   * Implements the Sorted Dictionary ADT using a fixed size array. Keys in this
9   * dictionary are sorted ascendingly.
10  *
11  * @author Eli Zupke
12  *
13  * @param <K> The type that will be used as keys in this dictionary
14  * @param <V> The type that will be used as values in this dictionary
15  */

```

```

14 public class SortedDictionaryStaticArray<K extends Comparable<? super K>, V>
    implements DictionaryInterface<K, V> {
15
16     // Used to keep track of where the last element of the dictionary is stored.
17     private int end;
18     private int capacity;
19
20     // These two arrays hold the keys and the values. The corresponding value of
    each key will be the entry in the value array with the same index.
21     private K[] keyArray;
22     private V[] valueArray;
23
24     /**
25      * Creates a new sorted dictionary via static array.
26      * @param capacity The maximum number of key-value pairs in this dictionary.
27      */
28     public SortedDictionaryStaticArray(int capacity) {
29
30         this.capacity = capacity;
31
32         // The dictionary starts at zero, so start the end variable pointing at -1
    (empty)
33         end = -1;
34
35         // Instantiate the arrays for both the keys and values.
36         @SuppressWarnings("unchecked")
37         K[] tempKeyArray = (K[])new Comparable[capacity]; // Unchecked cast
38         keyArray = tempKeyArray;
39
40         @SuppressWarnings("unchecked")
41         V[] tempValueArray = (V[])new Comparable[capacity]; // Unchecked cast
42         valueArray = tempValueArray;
43
44     }
45
46     @Override
47     public V add(K key, V value) {
48         // These store keys and values in the event that we need to add the key in
    the middle of the array.
49         K curKey = null;
50         V curValue = null;
51
52         // Go down the array until we get to a value greater than the one we're
    adding, then move the rest down
53         int i = 0;
54         // System.out.println("Inserting " + value.toString());
55         //Test.printArray(keyArray); Test.printArray(valueArray);
56
57         for (; i < getSize() + 1; i++) {
58             if (keyArray[i] == key) {
59                 // We already have the key, it seems.
60
61                 // Hold on to the old value, replace it with the new one, then
    return it.
62                 V returnValue = valueArray[i];
63                 valueArray[i] = value;
64
65                 return returnValue;
66             } else if (keyArray[i] == null) {

```

```

67         // System.out.println("TEST END");
68         // We got to the end of the array, so let's place it at the end!
69         valueArray[i] = value;
70         keyArray[i] = key;
71         end++;
72         // We've added the element, so leave.
73         return null;
74     } else if (keyArray[i].compareTo(key) > 0) {
75         // System.out.println("TEST GREATER");
76         // We have found where to place our key, so let's do it!
77         K tempKey = keyArray[i];
78         V tempValue = valueArray[i];
79
80         keyArray[i] = key;
81         valueArray[i] = value;
82
83         curKey = tempKey;
84         curValue = tempValue;
85
86         // Since we now know that we need to expand the array, but don't
know whether we have enough room, let's check
87         ensureCapacity();
88         end++;
89         break;
90     }
91 }
92 // If we get here, then we know that we went through the last else if,
93 // and we still need to move the remaining values over one index.
94
95 for (i += 1; i < getSize(); i++) {
96     // Move the next group of values
97     K tempKey = keyArray[i];
98     V tempValue = valueArray[i];
99
100    keyArray[i] = curKey;
101    valueArray[i] = curValue;
102
103    curKey = tempKey;
104    curValue = tempValue;
105 }
106
107 return null;
108 }
109
110 @Override
111 public V remove(K key) {
112
113     V value = null;
114
115     // Declare the index variable outside the loop, so we can continue where
we left off in the next one
116     int i = 0;
117
118     // Find the key, store its value, and stop the loop.
119     // If it gets to the end, then the next loop will not be entered, and we
will return null.
120     for (; i <= end; i++) {
121         if (keyArray[i] == key) {
122             value = valueArray[i];

```



```

123         break;
124     }
125 }
126
127 // If we didn't find the key, then we can stop now
128 if (value == null) {
129     return null;
130 }
131
132 // Otherwise, move the rest of the values back.
133 for (; i < end; i++) {
134     // Move the next group of values
135     keyArray[i] = keyArray[i+1];
136     valueArray[i] = valueArray[i+1];
137 }
138 keyArray[end] = null;
139 valueArray[end] = null;
140 // Finally, reduce the end index by one.
141 end--;
142
143 return value;
144 }
145
146 @Override
147 public V getValue(K key) {
148
149     // Sequential search the key array for the key we are looking for.
150     for (int i = 0; i < capacity; i++) {
151         if (keyArray[i] == key) {
152             // We found what we're looking for.
153             return valueArray[i];
154         }
155     }
156     // We couldn't find the key we were looking for.
157     return null;
158 }
159
160 @Override
161 public boolean contains(K key) {
162
163     // Sequential search the key array for the key we are looking for.
164     for (int i = 0; i < capacity; i++) {
165         if (keyArray[i] == key) {
166             return true;
167         }
168     }
169     return false;
170 }
171
172 @Override
173 public Iterator<K> getKeyIterator() {
174     return new StaticArrayIterator<K>(true);
175 }
176
177 @Override
178 public Iterator<V> getValueIterator() {
179
180     return new StaticArrayIterator<V>(false);
181 }

```

```

182
183 private class StaticArrayIterator<I> implements Iterator<I> {
184
185     // Whether this is an iterator of keys (if true) or values (if false)
186     boolean key;
187
188     // index is the index of the value we just gave.
189     private int index = -1;
190
191     // whether there is an element we can remove.
192     boolean canRemove = false;
193
194     StaticArrayIterator(boolean _key) {
195         super();
196         key = _key;
197     }
198
199     @Override
200     public boolean hasNext() {
201         return index < end;
202     }
203
204     // Because I will always equal K or V, and we know which one it will equal
205     // , we can do this cast.
206     @SuppressWarnings("unchecked")
207     @Override
208     public I next() {
209         if (!hasNext()) {
210             throw new NoSuchElementException();
211         }
212         index++;
213         canRemove = true;
214         if (key) {
215             return (I)keyArray[index];
216         } else {
217             return (I)valueArray[index];
218         }
219     }
220
221     @Override
222     public void remove() {
223         if (!canRemove) {
224             throw new IllegalStateException();
225         }
226         canRemove = false;
227
228         SortedDictionaryStaticArray.this.remove(keyArray[index]);
229         // Because we removed an element, we need to move our index backwards
230         index--;
231     }
232 }
233
234
235
236 @Override
237 public boolean isEmpty() {
238     return getSize() == 0;
239 }

```

```

240
241     @Override
242     public int getSize() {
243         // The size of the dictionary is always equal to the position of the end
244         // index plus 1.
245         return end + 1;
246     }
247
248     @Override
249     public void clear() {
250         // Dereference everything in both arrays
251         for (int i = 0; i < capacity; i++) {
252             keyArray[i] = null;
253             valueArray[i] = null;
254         }
255
256         // Move the end index back to before the start of the array.
257         end = -1;
258     }
259
260     /**
261     * Test to see if we have room to add an element.
262     * Throws an IndexOutOfBoundsException if the array is full.
263     */
264     private void ensureCapacity() {
265         if (end + 1 >= capacity) {
266             throw new IndexOutOfBoundsException("Max array size reached!");
267         }
268     }
269 }

```

4 src/cs240/SortedDictionaryLinkedData.java

```

1 package cs240;
2
3 import java.util.Iterator;
4 import java.util.NoSuchElementException;
5
6 /**
7  * An implementation of the Sorted Dictionary ADT using linked data
8  * @author Eli Zupke
9  *
10  * @param <K> The type that will be used as keys in this dictionary
11  * @param <V> The type that will be used as values in this dictionary
12  */
13 public class SortedDictionaryLinkedData<K extends Comparable<? super K>, V>
14     implements DictionaryInterface<K, V> {
15
16     // This is the first node in the dictionary.
17     private Node front;
18
19     private class Node {

```

```

19
20     K key;
21     V value;
22     Node next;
23
24     public Node(K _key, V _value) {
25         key = _key;
26         value = _value;
27         next = null;
28     }
29 }
30
31 @Override
32 public V add(K key, V value) {
33
34     // Special case. This node simply becomes the front.
35     if (front == null) {
36         front = new Node(key, value);
37         // we replaced nothing, so return null.
38         return null;
39     }
40
41     // Special case. This node becomes the front.
42     if (key.compareTo(front.key) < 0) {
43         Node newNode = new Node(key, value);
44         newNode.next = front;
45         front = newNode;
46         return null;
47     }
48
49     Node prevNode = null;
50     Node curNode = front;
51
52     while (curNode != null) {
53         if (curNode.key == key) {
54             // The key we're adding already exists in our dictionary, so go
55             // ahead and replace the value.
56             V oldValue = curNode.value;
57             curNode.value = value;
58             return oldValue;
59         } else if (key.compareTo(curNode.key) < 0) {
60             // We need to insert the key here.
61             Node newNode = new Node(key, value);
62             newNode.next = curNode;
63             prevNode.next = newNode;
64             return null;
65         }
66
67         prevNode = curNode;
68         curNode = curNode.next;
69     }
70     // We have got to the end of the linked data, but we still haven't added
71     // the new pair.
72     // Therefore, we add it to the end.
73     Node newNode = new Node(key, value);
74     prevNode.next = newNode;
75     return null;
76 }

```

```

76  @Override
77  public V remove(K key) {
78
79      // Special case: the node we want to remove is the front
80      if (front != null && front.key == key) {
81          V oldValue = front.value;
82          front = front.next;
83          return oldValue;
84      }
85
86      Node prevNode = null;
87      Node curNode = front;
88      while (curNode != null) {
89          if (curNode.key == key) {
90              // We've found the key in our dictionary
91              V oldValue = curNode.value;
92              prevNode.next = curNode.next;
93              return oldValue;
94          }
95          prevNode = curNode;
96          curNode = curNode.next;
97      }
98      return null;
99  }
100
101  @Override
102  public V getValue(K key) {
103
104      Node curNode = front;
105
106      // Sequential search the list until we get to the node that we need.
107      while (curNode != null) {
108          if (curNode.key == key) {
109              return curNode.value;
110          }
111          curNode = curNode.next;
112      }
113
114      return null;
115  }
116
117  @Override
118  public boolean contains(K key) {
119
120      Node curNode = front;
121
122      while (curNode != null) {
123          if (curNode.key == key)
124              return true;
125          curNode = curNode.next;
126      }
127
128      return false;
129  }
130
131
132  @Override
133  public Iterator<K> getKeyIterator() {
134      return new StaticArrayKeyIterator();

```

```

135     }
136
137     private class StaticArrayKeyIterator implements Iterator<K> {
138
139         // The node we just gave
140         Node prevNode = null;
141         // The node we are about to give
142         Node curNode = front;
143
144
145         @Override
146         public boolean hasNext() {
147             return curNode != null;
148         }
149
150         @Override
151         public K next() {
152             if (!hasNext()) {
153                 throw new NoSuchElementException();
154             }
155             prevNode = curNode;
156             curNode = curNode.next;
157             return prevNode.key;
158         }
159     }
160
161     @Override
162     public void remove() {
163         if (prevNode == null) {
164             throw new IllegalStateException();
165         }
166         SortedDictionaryLinkedData.this.remove(prevNode.key);
167     }
168 }
169
170
171 @Override
172 public Iterator<V> getValueIterator() {
173
174     return new StaticArrayValueIterator();
175 }
176
177 private class StaticArrayValueIterator implements Iterator<V> {
178
179     // The node we just gave
180     Node prevNode = null;
181     // The node we are about to give
182     Node curNode = front;
183
184
185     @Override
186     public boolean hasNext() {
187         return curNode != null;
188     }
189
190     @Override
191     public V next() {
192         if (!hasNext()) {
193             throw new NoSuchElementException();

```

```

194         }
195         prevNode = curNode;
196         curNode = curNode.next;
197         return prevNode.value;
198     }
199
200
201     @Override
202     public void remove() {
203         if (prevNode == null) {
204             throw new IllegalStateException();
205         }
206         SortedDictionaryLinkedData.this.remove(prevNode.key);
207     }
208 }
209
210
211 @Override
212 public boolean isEmpty() {
213     return front == null;
214 }
215
216 @Override
217 public int getSize() {
218     int size = 0;
219
220     Node curNode = front;
221
222     // System.out.println("=====");
223     while (curNode != null) {
224         // System.out.println(curNode.value);
225         size++;
226         curNode = curNode.next;
227     }
228
229     return size;
230 }
231
232 @Override
233 public void clear() {
234     front = null;
235 }
236
237
238 }

```