CS 240 Final Part 3: Part 1

Eli Zupke

December 5, 2017

The GitHub folder can be found at https://github.com/Trainzack/CS240/tree/master/Final%20Part%203.

Note: Some code did not have iterators until I modified them during the final!

Contents

1 src/IteratorOfIterators.java

```
1  import java.util.Iterator;
2  import java.util.NoSuchElementException;
3
4  /**
5   *
6   * Iterates over an array of Iterators, and collates the results.
7   * Example:
8   *  Iterating over IA and IB, each with three items, yields:
9   *      A1 B1 A2 B2 A3 B3
10  *  In that order
11  * @author eli
12  *
13  * @param <T> What type of objects the iterator's iterators are iterating over
14  */
15 public class IteratorOfIterators<T> implements Iterator<T> {
16
17     // This array contains the iterators that we will be iterating over.
18     private Iterator<T>[] its;
19
20     // This is the index of the iterator in its that is due to be given next.
21     private int nextIndex;
22
23     /**
24      * Creates a new Iterator of Iterators
```

```
     * @param its The array of iterators that we should iterate over.
     */
    public IteratorOfIterators(Iterator<T>[] its) {
        this.its = its;
        nextIndex = 0;
    }


    @Override
    public boolean hasNext() {

        // Return false only if the next iterator is empty.
        // Note: assumes all iterators have the same number of items.
        if (!its[nextIndex].hasNext()) {
            return false;
        }

        return true;
    }

    @Override
    public T next() {

        if (!hasNext()) {
            throw new NoSuchElementException();
        }

        // This is the object that we are returning.
        T next = its[nextIndex].next();

        // Go forward, or wrap around if we've reached the end.
        nextIndex = (nextIndex + 1) % its.length;

        return next;
    }

    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }

}
```

2 src/Test.java

```
import java.util.Iterator;

/**
 * This is a class designed to test Person.java and IteratorOfIterators.java
 *
 * To test Person.java, it runs the two methods provided
 *
 * To test IteratorOfIterators.java, it supplies that class with three separate
```

```
   iterators, each with the same number of elements.
 *  It then iterates over them, and prints out the result.
 *
 * @author eli
 *
 */
public class Test {

    /**
     * @param args
     */
    public static void main(String[] args) {

        final int LENGTH = 3;

        System.out.println("Person Test 1");
        System.out.println();
        test1();
        System.out.println();
        System.out.println("Person Test 2");
        System.out.println();
        try {
            // Test2 should end in a RuntimeException.
            test2();
            System.out.println("Uh-Oh! We were expecting a runtime exception!");
        } catch (RuntimeException e) {
            e.printStackTrace();
            System.err.println("(We were expecting that!)");
        }
        System.out.println();
        System.out.println();

        System.out.println("Iterators of Iterators:");


        // These are the three data structures that we will iterate over.
        VectorStack<String> stack = new VectorStack<>();
        DoubleLinkedList<String> list = new DoubleLinkedList<>();
        SortedDictionaryStaticArray<Integer, String> dict = new
    SortedDictionaryStaticArray<>(LENGTH);

        // We need to fill up the data structures, so we add some labled and
    numbered strings to them.
        for (int i = 0; i < LENGTH; i++) {
            stack.push("S" + i);
            list.add("L" + i);
            // The key of the dictionary is an integer, so that it will sort to
    the right place.
            dict.add(new Integer(i), "D" + i);
        }

        // There is probably a better way to get rid of these warnings, but I am
    at a loss.
        //  This is okay
        @SuppressWarnings("rawtypes")
        Iterator[] its = new Iterator[] {stack.getIterator(), list.getIterator(),
    dict.getValueIterator()};

        //  This is okay because we just declared it earlier, so we know they will
```

```
 match.
62          @SuppressWarnings("unchecked")
63          IteratorOfIterators<String> t = new IteratorOfIterators<>(its);
64
65          while (t.hasNext()) {
66              System.out.print(t.next() + " ");
67          }
68
69      }
70
71      /**This is a sample test main() for Person. It should output:
72
73          == The wall of Kim ==
74          I agree
75          Friends are awesome
76          Only Kim can read this
77          == The wall of Pat ==
78          I agree
79          Friends are awesome
80
81       ***********************************************/
82
83      public static void test1() {
84
85          Person first = new Person("Kim");
86          Person second = new Person("Pat");
87          first.post("Only Kim can read this");
88
89          first.meet(second);
90          second.post("Friends are awesome");
91          first.post("I agree");
92
93          first.listMessages();
94          second.listMessages();
95
96      }
97
98      /*********************************************
99
100         This is a sample test main() for Person. It should output:
101
102         false
103         true
104         true
105
106         and then throw a RuntimeException (see the comments).
107
108      ***********************************************/
109
110     public static void test2() {
111
112         Person first = new Person("Kim");
113         Person second = new Person("Pat");
114
115         System.out.println(first.knows(second));   // should print "false"
116
117         first.meet(second);
118
119         System.out.println(first.knows(second));   // should print "true"
```

```
        System.out.println(second.knows(first));   // should print "true"

        first.knows(first);                        // should throw a RuntimeException
    }
}
```

3 src/VectorStack.java

```
import java.util.EmptyStackException;
import java.util.Iterator;
import java.util.NoSuchElementException;
import java.util.Vector;

/**
   A stack data structure implemented via Vectors.
   @author Eli Zupke
   @version 1.0
 */
public class VectorStack<T> implements StackInterface<T>
{

    private Vector<T> stack;

    public VectorStack() {
        stack = new Vector<T>();
    }

    /** Adds a new entry to the top of this stack.
       @param newEntry  An object to be added to the stack. */
    public void push(T newEntry) {
        stack.add(newEntry);
    }

    /** Removes and returns this stack's top entry.
       @return  The object at the top of the stack.
       @throws  EmptyStackException if the stack is empty before the operation. */
    public T pop() {

        if (isEmpty()) {
            throw new EmptyStackException();
        }
        else {
            // Grab the last element of the stack, remove it from the stack, and
    return it
            return stack.remove(stack.size() - 1);
        }

    }

    /** Retrieves this stack's top entry.
       @return  The object at the top of the stack.
       @throws  EmptyStackException if the stack is empty. */
    public T peek() {
```

```
45
46          if (isEmpty()) {
47              throw new EmptyStackException();
48          }
49          else {
50              // Grab the last element of the stack, and return it
51              return stack.get(stack.size() - 1);
52          }
53      }
54
55      /** Detects whether this stack is empty.
56         @return  True if the stack is empty. */
57      public boolean isEmpty() {
58
59          return stack.isEmpty();
60
61      }
62
63      /** Removes all entries from this stack. */
64      public void clear() {
65
66          stack.clear();
67
68      }
69
70      /**
71       * Returns an iterator that iterates over the stack's values. Removal is not
        supported.
72       * (This functionality was added during the final!)
73       * @return An iterator of type T that iterates over the stack's values
74       */
75      public Iterator<T> getIterator() {
76          return new VectorStackIterator();
77      }
78
79
80      private class VectorStackIterator implements Iterator<T> {
81
82          // index is the index of the value we just gave.
83          private int index = -1;
84
85          @Override
86          public boolean hasNext() {
87
88              return index < stack.size() - 1;
89          }
90
91          @Override
92          public T next() {
93              if (!hasNext()) {
94                  throw new NoSuchElementException();
95              }
96
97              index ++;
98              return stack.get(index);
99
100         }
101
102         @Override
```

```
103         public void remove() {
104             throw new UnsupportedOperationException();
105         }
106     }
107 }
```

4 src/DoubleLinkedList.java

```
1  import java.util.Iterator;
2  import java.util.NoSuchElementException;
3
4
5  public class DoubleLinkedList<T> {
6
7      // Point to the front and back of the list, respectively
8      DoubleNode<T> front;
9      DoubleNode<T> back;
10
11     // Keep track of how long the list is for time's sake.
12     int size = 0;
13
14     public DoubleLinkedList() {
15
16         // The single header node
17         front = null;
18         back = null;
19
20     }
21
22
23
24     public void add(T item) {
25
26         DoubleNode<T> newNode = new DoubleNode<>(item);
27
28         //special case: the list is empty
29         if (front == null) {
30             front = newNode;
31             back = newNode;
32             size = 1;
33             return;
34         }
35         // Link in the new node to the list
36         back.setNextNode(newNode);
37         newNode.setPreviousNode(back);
38         back = newNode;
39
40         // Increment the size counter
41         size++;
42
43     }
44
45
```

```
46      public void add(T item, int index) throws IndexOutOfBoundsException {
47
48          if (index < 0 || index > size) {
49              throw new IndexOutOfBoundsException();
50          }
51
52          DoubleNode<T> newNode = new DoubleNode<T>(item);
53
54          if (isEmpty()) {
55              // special case; add to start and end of list;
56              front = newNode;
57              back = newNode;
58
59
60          } else if (index == 0) {
61              // special case; add to start of list
62              newNode.setNextNode(front);
63              front.setPreviousNode(newNode);
64              front = newNode;
65
66          } else if (index == size) {
67              // special case; add to end of list
68              newNode.setPreviousNode(back);
69              back.setNextNode(newNode);
70              back = newNode;
71
72          } else {
73
74              // This is the node at the index where we will be adding.
75              // its index will increase by one.
76              DoubleNode<T> postNode = getNodeAtIndex(index);
77
78              DoubleNode<T> preNode = postNode.getPreviousNode();
79
80              // Link the new node into the list.
81              preNode.setNextNode(newNode);
82              newNode.setPreviousNode(preNode);
83
84              postNode.setPreviousNode(newNode);
85              newNode.setNextNode(postNode);
86
87          }
88
89          size++;
90      }
91
92
93      public T remove(int index) {
94          ensureNotEmpty();
95          if (index < 0 || index >= size) {
96              throw new IndexOutOfBoundsException();
97          }
98
99          T data = null;
100
101         if (size() == 1) {
102             // special case: remove from front and back
103             data = front.getData();
104             front = null;
```

```
105             back = null;
106
107         } else if (index == 0) {
108             // special case: remove from front
109             data = front.getData();
110             DoubleNode<T> newFront = front.getNextNode();
111
112             newFront.setPreviousNode(null);
113             front = newFront;
114
115         } else if (index == size - 1) {
116             // special case: remove from back
117             data = back.getData();
118             DoubleNode<T> newBack = back.getPreviousNode();
119
120             newBack.setNextNode(null);
121             back = newBack;
122         } else {
123
124             // This is the node at the index where we will be removing.
125             DoubleNode<T> exNode = getNodeAtIndex(index);
126
127             data = exNode.getData();
128
129             // These are the nodes around the node we will be removing.
130             DoubleNode<T> postNode = exNode.getNextNode();
131             DoubleNode<T> preNode = exNode.getPreviousNode();
132
133             // Test.printArray(toArray());
134
135             // Link the new node into the list.
136             preNode.setNextNode(postNode);
137             postNode.setPreviousNode(preNode);
138
139         }
140
141         size--;
142         return data;
143     }
144
145
146     public T remove() {
147         // TODO: replace with more efficient implementation
148         return remove(size()-1);
149     }
150
151
152     public void clear() {
153
154         DoubleNode<T> curNode = front;
155
156         // Go through the list, and remove one of each node's connections
157         while (curNode != null) {
158             curNode.setPreviousNode(null);
159             curNode = curNode.getNextNode();
160         }
161
162         // Now, every node is only connected one way, so when the ends are
163         // de-referenced, the rest will be too.
```

```
164         front = null;
165         back = null;
166 
167         size = 0;
168     }
169 
170 
171     public T view(int index) {
172 
173         return getNodeAtIndex(index).getData();
174     }
175 
176 
177     public boolean contains(T item) {
178 
179         DoubleNode<T> curNode = front;
180 
181         // Go through each node. If we see the right data, return true.
182         // If we get to the end, return false.
183         while (curNode != null) {
184             if (curNode.getData() == item) {
185                 return true;
186             }
187             curNode = curNode.getNextNode();
188         }
189 
190 
191         return false;
192     }
193 
194 
195     public int size() {
196         return size;
197     }
198 
199 
200     public boolean isEmpty() {
201         return (size == 0);
202     }
203 
204 
205     public T[] toArray() {
206 
207         // If the array is empty, just return null.
208         if (isEmpty()) return null;
209 
210         @SuppressWarnings("unchecked")
211         T[] outArray = (T[])new Object[size]; // Unchecked cast
212 
213         DoubleNode<T> curNode = front;
214 
215         // Go through the list and copy the contents of each node to the array
216         for(int i = 0; i < size; i++) {
217             outArray[i] = curNode.getData();
218             curNode = curNode.getNextNode();
219         }
220 
221         if (curNode != null) {
222             throw new IllegalStateException("Size of list wasn't correctly
```

```
    maintained!");
        }

        return outArray;
    }


    public T replace(int index, T newItem) {

        // Find the node that contains the data we are replacing
        DoubleNode<T> replacedNode = getNodeAtIndex(index);

        // Grab the data, so that we can return it later
        T data = replacedNode.getData();

        // Replace the data in the node and return the data
        replacedNode.setData(newItem);
        return data;
    }

    /**
     * Returns the node at the given index
     * @param index The index to find the node at
     * @return The given index
     * @throws IndexOutOfBoundsException if the index is not in the list.
     */
    private DoubleNode<T> getNodeAtIndex(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException();
        }

        DoubleNode<T> curNode = front;
        // Go through the array until we get to the node at the given index
        for(int i = 0; i < index; i++) {
            if (curNode == null) {
                // The index was too large!
                throw new IndexOutOfBoundsException();
            }
            curNode = curNode.getNextNode();
        }
        return curNode;
    }

    /**
     * Test to see if the queue is not empty.
     * If it is empty, throw an EmptyQueueException
     */
    private void ensureNotEmpty() {
        if (isEmpty()) {
            throw new EmptyQueueException();
        }
    }


    /**
     * Returns an iterator that iterates over the stack's values. Removal is not
    supported.
     * (This functionality was added during the final!)
     * @return An iterator of type T that iterates over the stack's values
```

```
280      */
281     public Iterator<T> getIterator() {
282         return new thisIt();
283     }
284 
285 
286     private class thisIt implements Iterator<T> {
287 
288         // The node we just gave
289         DoubleNode<T> prevNode = null;
290         // The node we are about to give
291         DoubleNode<T> curNode = front;
292 
293 
294         @Override
295         public boolean hasNext() {
296             return curNode != null;
297         }
298 
299         @Override
300         public T next() {
301             if (!hasNext()) {
302                 throw new NoSuchElementException();
303             }
304             prevNode = curNode;
305             curNode = curNode.getNextNode();
306             return prevNode.getData();
307 
308         }
309 
310         @Override
311         public void remove() {
312             throw new UnsupportedOperationException();
313         }
314     }
315 
316 
317 }
```

5 src/SortedDictionaryStaticArray.java

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

```java
public class SortedDictionaryStaticArray<K extends Comparable<? super K>, V>
    implements DictionaryInterface<K, V> {

    // Used to keep track of where the last element of the dictionary is stored.
    private int end;
    private int capacity;

    // 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.
    private K[] keyArray;
    private V[] valueArray;

    /**
     * Creates a new sorted dictionary via static array.
     * @param capacity The maximum number of key-value pairs in this dictionary.
     */
    public SortedDictionaryStaticArray(int capacity) {

        this.capacity = capacity;

        // The dictionary starts at zero, so start the end variable pointing at -1
     (empty)
        end = -1;

        // Instantiate the arrays for both the keys and values.

        @SuppressWarnings("unchecked")
        K[] tempKeyArray = (K[])new Comparable[capacity]; // Unchecked cast
        keyArray = tempKeyArray;

        @SuppressWarnings("unchecked")
        V[] tempValueArray = (V[])new Comparable[capacity]; // Unchecked cast
        valueArray = tempValueArray;

    }

    @Override
    public V add(K key, V value) {
        // These store keys and values in the event that we need to add the key in
     the middle of the array.
        K curKey = null;
        V curValue = null;

        // Go down the array until we get to a value greater than the one we're
    adding, then move the rest down
        int i = 0;
        // System.out.println("Inserting " + value.toString());
        //Test.printArray(keyArray); Test.printArray(valueArray);

        for (; i < getSize() + 1; i++) {
            if (keyArray[i] == key) {
                // We already have the key, it seems.

                // Hold on to the old value, replace it with the new one, then
    return it.
                V returnValue = valueArray[i];
                valueArray[i] = value;

                return returnValue;
```

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

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

```
180     }
181
182     private class StaticArrayIterator<I> implements Iterator<I> {
183
184         // Whether this is an iterator of keys (if true) or values (if false)
185         boolean key;
186
187         // index is the index of the value we just gave.
188         private int index = -1;
189
190         // whether there is an element we can remove.
191         boolean canRemove = false;
192
193         StaticArrayIterator(boolean _key) {
194             super();
195             key = _key;
196         }
197
198         @Override
199         public boolean hasNext() {
200             return index < end;
201         }
202
203         // Because I will always equal K or V, and we know which one it will equal
    , we can do this cast.
204         @SuppressWarnings("unchecked")
205         @Override
206         public I next() {
207             if (!hasNext()) {
208                 throw new NoSuchElementException();
209             }
210             index++;
211             canRemove = true;
212             if (key) {
213                 return (I)keyArray[index];
214             } else {
215                 return (I)valueArray[index];
216             }
217
218         }
219
220         @Override
221         public void remove() {
222             if (!canRemove) {
223                 throw new IllegalStateException();
224             }
225             canRemove = false;
226
227             SortedDictionaryStaticArray.this.remove(keyArray[index]);
228             // Because we removed an element, we need to move our index backwards
229             index--;
230
231         }
232     }
233
234
235     @Override
236     public boolean isEmpty() {
237         return getSize() == 0;
```

```
238     }
239
240     @Override
241     public int getSize() {
242         // The size of the dictionary is always equal to the position of the end
    index plus 1.
243         return end + 1;
244     }
245
246     @Override
247     public void clear() {
248
249         // Dereference everything in both arrays
250         for (int i = 0; i < capacity; i++) {
251             keyArray[i] = null;
252             valueArray[i] = null;
253         }
254
255         // Move the end index back to before the start of the array.
256         end = -1;
257     }
258
259     /**
260      * Test to see if we have room to add an element.
261      * Throws an IndexOutOfBoundsException if the array is full.
262      */
263     private void ensureCapacity() {
264         if (end + 1 >= capacity) {
265             throw new IndexOutOfBoundsException("Max array size reached!");
266         }
267     }
268 }
```