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/src.

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

Contents

1 src/Person.java

```
1  import java.util.Iterator;
2
3  /*******************************************************************************
4    Winter 2017 CS 240 Programming Exam : Person
5
6   Author: Eli
7
8   Dependencies: Stack, Queue, Dictionary
9
10  Description:  Models a person, a list of messages that they can
11                read, and a list of their friends, so that when you
12                post a message, all your friends can read it too.
13
14 *******************************************************************************/
15
16 public class Person {
17
18
19     public SingleLinkedDataQueue<Person> friends;
20
21     public LinkedDataStack<String> messages;
22
23     public String name;
24
25     // Create a new Person with this name.
26     public Person(String name) {
```

```
27          friends = new SingleLinkedDataQueue<>();
28
29          messages = new LinkedDataStack<>();
30
31          this.name = name;
32      }
33
34      // Make these two people become friends with each other.
35      // Throw an exception if you try to meet yourself.
36      // We are allowed to assume we didn't meet this person yet.
37      public void meet(Person otherPerson) {
38
39          if (otherPerson == this) {
40              throw new RuntimeException("Stop meeting yourself!");
41          }
42
43          friends.enqueue(otherPerson);
44          otherPerson.friends.enqueue(this);
45
46      }
47
48      // Are these two people friends?
49      // Throw an exception if you ask about knowing yourself.
50      public boolean knows(Person otherPerson) {
51
52          if (otherPerson == this) {
53              throw new RuntimeException("Stop meeting yourself!");
54          }
55
56          Iterator<Person> freindIterator = friends.getIterator();
57
58          while (freindIterator.hasNext()) {
59              if (freindIterator.next() == otherPerson) return true;
60          }
61
62          return false;
63
64      }
65
66      // Post a message to my list and the lists of all my friends
67      public void post(String message) {
68
69          messages.push(message);
70
71          Iterator<Person> freindIterator = friends.getIterator();
72
73          while (freindIterator.hasNext()) {
74              freindIterator.next().messages.push(message);
75          }
76
77
78      }
79
80      // Print a header, then all messages this Person can read, newest first
81      public void listMessages() {
82         System.out.println("== The wall of " + name + " ==");
83
84         while (!messages.isEmpty()) {
85             System.out.println(messages.pop());
```

```
        }

    }
}
```

2 src/Test.java

```
import java.util.Iterator;


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();
            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:");

        VectorStack<String> stack = new VectorStack<>();
        DoubleLinkedList<String> list = new DoubleLinkedList<>();
        SortedDictionaryStaticArray<Integer, String> dict = new
    SortedDictionaryStaticArray<>(LENGTH);

        for (int i = 0; i < LENGTH; i++) {
            stack.push("S" + i);
            list.add("L" + i);
            dict.add(new Integer(i), "D" + i);
        }

        Iterator[] its = {stack.getIterator(), list.getIterator(), dict.
    getValueIterator()};

        IteratorOfIterators<String> t = new IteratorOfIterators<>(its);

        while (t.hasNext()) {
```

```
45              System.out.print(t.next() + " ");
46          }
47
48      }
49
50      /**This is a sample test main() for Person. It should output:
51
52          == The wall of Kim ==
53          I agree
54          Friends are awesome
55          Only Kim can read this
56          == The wall of Pat ==
57          I agree
58          Friends are awesome
59
60       ***********************************************/
61
62      public static void test1() {
63
64          Person first = new Person("Kim");
65          Person second = new Person("Pat");
66          first.post("Only Kim can read this");
67
68          first.meet(second);
69          second.post("Friends are awesome");
70          first.post("I agree");
71
72          first.listMessages();
73          second.listMessages();
74
75      }
76
77      /*********************************************
78
79          This is a sample test main() for Person. It should output:
80
81          false
82          true
83          true
84
85          and then throw a RuntimeException (see the comments).
86
87       *********************************************/
88
89      public static void test2() {
90
91          Person first = new Person("Kim");
92          Person second = new Person("Pat");
93
94          System.out.println(first.knows(second));   // should print "false"
95
96          first.meet(second);
97
98          System.out.println(first.knows(second));   // should print "true"
99          System.out.println(second.knows(first));   // should print "true"
100
101         first.knows(first);                    // should throw a RuntimeException
102     }
103 }
```

3 src/SingleLinkedDataQueue.java

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

/**
   A queue implemeneted with single linked data
   @author Eli Zupke
   @version 1.0
 */
public class SingleLinkedDataQueue<T> implements QueueInterface<T>
{

    Node<T> top;


    public SingleLinkedDataQueue() {

        top = null;
    }

    /** Adds a new entry to the back of this queue.
      @param newEntry  An object to be added. */
    public void enqueue(T newEntry) {

        // Create the new node that we will add
        Node<T> addedNode = new Node<T>(newEntry);

        // Add this node to the end of the queue and set it to the top.
        addedNode.setNextNode(top);
        top = addedNode;
    }

    /** Removes and returns the entry at the front of this queue.
      @return  The object at the front of the queue.
      @throws  EmptyQueueException if the queue is empty before the operation. */
    public T dequeue() {
        if (isEmpty()) {
            throw new EmptyQueueException();
        } else {
            Node<T> prevNode = null;
            Node<T> thisNode = top;

            // We need to loop through the enitre queue to find the top.
            while(true) {
                if (thisNode.getNextNode() == null) {
                    // We have found the front of the queue
                    // We remove this node from the chain by setting the previous node's next node to null
                    if (prevNode == null) {
                        // This is a special case; we must set top to null
                        top = null;
                    } else {
                        prevNode.setNextNode(null);
                    }

                    return thisNode.getData();
                } else {
```

```
                    // We are still in the middle of the queue; we must go further
                    prevNode = thisNode;
                    thisNode = thisNode.getNextNode();
                }
            }
        }
    }

    /**  Retrieves the entry at the front of this queue.
      @return  The object at the front of the queue.
      @throws  EmptyQueueException if the queue is empty. */
    public T getFront() {
        if (isEmpty()) {
            throw new EmptyQueueException();
        } else {
            Node<T> thisNode = top;

            // Loop through the entire queue
            while(true) {
                if (thisNode.getNextNode() == null) {
                    // We have found the front of the queue
                    return thisNode.getData();
                } else {
                    // We must continue to the next node
                    thisNode = thisNode.getNextNode();
                }
            }
        }
    }



    /** Detects whether this queue is empty.
      @return  True if the queue is empty, or false otherwise. */
    public boolean isEmpty() {

        return (top == null);

    }

    /** Removes all entries from this queue. */
    public void clear() {
        // We can clear the queue by dereferencing the top node.
        top = null;
    }



    /**
     * 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
     */
    public Iterator<T> getIterator() {
        return new SingledLinkedDataQueueIterator();
    }


```

```
114     private class SingledLinkedDataQueueIterator implements Iterator<T> {
115 
116         // The node we just gave
117         Node<T> prevNode = null;
118         // The node we are about to give
119         Node<T> curNode = top;
120 
121 
122         @Override
123         public boolean hasNext() {
124             return curNode != null;
125         }
126 
127         @Override
128         public T next() {
129             if (!hasNext()) {
130                 throw new NoSuchElementException();
131             }
132             prevNode = curNode;
133             curNode = curNode.getNextNode();
134             return prevNode.getData();
135 
136         }
137 
138         @Override
139         public void remove() {
140             throw new UnsupportedOperationException();
141         }
142     }
143 
144 
145 
146 } // end QueueInterface
```

4 src/LinkedDataStack.java

```
1  import java.util.EmptyStackException;
2  import java.util.Iterator;
3  import java.util.NoSuchElementException;
4  
5  
6  /**
7     An implementation of the Stack data type using linked data
8     @author Eli Zupke
9     @version 1.0
10  */
11 public class LinkedDataStack<T> implements StackInterface<T>
12 {
13     // This is a reference to the node at the top of the stack
14     Node<T> top;
15 
16     public LinkedDataStack() {
17 
```

```
18          top = null;
19
20      }
21
22
23      public void push(T newEntry) {
24
25          Node<T> newTop = new Node<T>(newEntry);
26          newTop.setNextNode(top);
27          top = newTop;
28      }
29
30      /** Removes and returns this stack's top entry.
31         @return  The object at the top of the stack.
32         @throws  EmptyStackException if the stack is empty before the operation. */
33      public T pop() {
34
35          if (isEmpty()) {
36              throw new EmptyStackException();
37          } else {
38
39              // We get the data of the top node to be returned later and move the
    top pointer down a level in the stack.
40              T poppedData = top.getData();
41              top = top.getNextNode();
42              return poppedData;
43          }
44
45      }
46
47      /** Retrieves this stack's top entry.
48         @return  The object at the top of the stack.
49         @throws  EmptyStackException if the stack is empty. */
50      public T peek() {
51
52          if (isEmpty()) {
53              throw new EmptyStackException();
54          } else {
55              return top.getData();
56          }
57
58      }
59
60      /** Detects whether this stack is empty.
61         @return  True if the stack is empty. */
62      public boolean isEmpty() {
63          return (top == null);
64      }
65
66      /** Removes all entries from this stack. */
67      public void clear() {
68
69          // Dereferences the top element of the stack, which will ultimately remove
     the entire stack from memory.
70          top = null;
71      }
72
73      /**
74       * 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
     */
    public Iterator<T> getIterator() {
        return new LinkedDataStackIterator();
    }


    private class LinkedDataStackIterator implements Iterator<T> {

        // The node we just gave
        Node<T> prevNode = null;
        // The node we are about to give
        Node<T> curNode = top;


        @Override
        public boolean hasNext() {
            return curNode != null;
        }

        @Override
        public T next() {
            if (!hasNext()) {
                throw new NoSuchElementException();
            }
            prevNode = curNode;
            curNode = curNode.getNextNode();
            return prevNode.getData();

        }

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

}
```