

Section B

Q1	6 /5
Q2	8 /10
Q3	10 /10
Q4	10 /10
Q5	10 /10
Q6	2 /5

Compilation penalty

Style penalty (capped at -3) 0

Total for Section B 48 / 50

Section B**Tests (1/1)****jdk21**

Username: jlk21

Compilation: 1 / 1

Model Answer's Tests - AuxiliaryCollectionImplTest: 16 / 16

Model Answer's Tests - ChaosMonkeyTest: 1 / 1

Model Answer's Tests - MemoryImplTest: 11 / 11

No Google style violations - excellent!

Style penalty (capped at -3): 0

Note: if it is below the cap, your total style penalty could be higher if the marker has stylistic concerns that go beyond what Checkstyle identifies automatically.

Section B	AuxiliaryCollectionImplFineGrainedSync.java (1/2)	j1k21	Section B	AuxiliaryCollectionImplFineGrainedSync.java (2/2)	j1k21
1:	<code>package</code> datastructures;		82:	<code>((LockableDoublyLinkedNode) tail.getPrevious()).unlock();</code>	
2:			83:		
3:	<code>import</code> java.util.ArrayList;		84:	<code>return true;</code>	
4:	<code>import</code> java.util.List;		85:	<code>}</code>	
5:	<code>import</code> java.util.concurrent.atomic.AtomicInteger;		86:		
6:			87:	<i>/**</i>	
7:	<code>@SuppressWarnings({"rawtypes", "unchecked"})</code>		88:	<i>* Searches for a node in the Collection.</i>	
8:	<code>public class</code> AuxiliaryCollectionImplFineGrainedSync<K, V> <code>implements</code> AuxiliaryCollectionI<K, V> {		89:	<i>* </i>	
9:			90:	<i>* @param key The key of the DoublyLinkedNode to look for.</i>	
10:	<code>private</code> LockableDoublyLinkedNode<K, V> head;		91:	<i>* @return Returns a DoublyLinkedNode which has a key equal the input key or null if such a node</i>	
11:	<code>private</code> LockableDoublyLinkedNode<K, V> tail;		92:	<i>* was not found.</i>	
12:	<code>private</code> AtomicInteger size;		93:	<i>*/</i>	
13:			94:	<code>@Override</code>	
14:	<code>public</code> AuxiliaryCollectionImplFineGrainedSync() {		95:	<code>public</code> DoublyLinkedNode<K, V> find(K key) {	
15:	head = <code>new</code> LockableDoublyLinkedNode<>(<code>null</code> , <code>null</code>);		96:	DoublyLinkedNode<K, V> curr = head;	
16:	tail = <code>new</code> LockableDoublyLinkedNode<>(<code>null</code> , <code>null</code>);		97:		
17:	head.setNext(tail);		98:	<code>while</code> (curr != tail) {	
18:	tail.setPrevious(head);		99:	<code>if</code> (curr.getKey() == key) {	
19:	size = <code>new</code> AtomicInteger(0);		100:	<code>return</code> curr;	
20:	}		101:	}	
21:			102:	curr = curr.getNext();	
22:	<i>/**</i>		103:	<code>}</code>	
23:	<i>* Removes the given node from the Collection. It assumes that the input is either null or a node</i>		104:		
24:	<i>* which exists in the Collection.</i>		105:	<code>return null;</code>	
25:	<i>* </i>		106:	<code>}</code>	
26:	<i>* @param toRemove The DoublyLinkedNode to remove.</i>		107:		
27:	<i>* @return Returns false if the input is null or true if the removal is successful.</i>		108:	<i>/**</i>	
28:	<i>*/</i>		109:	<i>* The size of the Collection.</i>	
29:	<code>@Override</code>		110:	<i>* </i>	
30:	<code>public boolean</code> remove(DoublyLinkedNode toRemove) {		111:	<i>* @return Returns an int representing the number of items currently in the Collection.</i>	
31:	((LockableDoublyLinkedNode) toRemove).lock();		112:	<i>*/</i>	
32:	((LockableDoublyLinkedNode) toRemove.getPrevious()).lock();		113:	<code>public int</code> size() {	
33:	((LockableDoublyLinkedNode) toRemove.getNext()).lock();		114:	<code>return</code> size.get();	
34:			115:	}	
35:	<code>if</code> (toRemove == <code>null</code>) {		116:		
36:	((LockableDoublyLinkedNode) toRemove).unlock();		117:	<i>/**</i>	
37:	((LockableDoublyLinkedNode) toRemove.getPrevious()).unlock();		118:	<i>* Returns the head of the Collection.</i>	
38:	((LockableDoublyLinkedNode) toRemove.getNext()).unlock();		119:	<i>* </i>	
39:	<code>return false;</code>		120:	<i>* @return A DoublyLinkedNode which is currently the head of the Collection, or null if the</i>	
40:	}		121:	<i>* Collection is empty.</i>	
41:			122:	<i>*/</i>	
42:	toRemove.getPrevious().setNext(toRemove.getNext());		123:	<code>public</code> DoublyLinkedNode<K, V> getHead() {	
43:	toRemove.getNext().setPrevious(toRemove.getPrevious());		124:	<code>return this</code> .head;	
44:	size.decrementAndGet();		125:	}	
45:			126:		
46:	((LockableDoublyLinkedNode) toRemove).unlock();		127:	<i>/**</i>	
47:	((LockableDoublyLinkedNode) toRemove.getPrevious()).unlock();		128:	<i>* Traverses the Collection and stores the keys of all items currently in the Collection.</i>	
48:	((LockableDoublyLinkedNode) toRemove.getNext()).unlock();		129:	<i>* </i>	
49:			130:	<i>* @return Returns a List of keys of all the items currently in the Memory.</i>	
50:	<code>return true;</code>		131:	<i>*/</i>	
51:	}		132:	<code>@Override</code>	
52:			133:	<code>public</code> List<K> allKeysInOrder() {	
53:	<i>/**</i>		134:	var collector = <code>new</code> ArrayList<K>();	
54:	<i>* Appends the given node to the end of the Collection.</i>		135:	var curr = head.getNext();	
55:	<i>* </i>		136:		
56:	<i>* @param toAdd The DoublyLinkedNode to be appended.</i>		137:	<code>while</code> (curr != tail) {	
57:	<i>* @return Returns false if the node cannot be added or true otherwise.</i>		138:	collector.add(curr.getKey());	
58:	<i>*/</i>		139:	curr = curr.getNext();	
59:	<code>@Override</code>		140:	}	
60:	<code>public boolean</code> append(DoublyLinkedNode toAdd) {		141:		
61:	((LockableDoublyLinkedNode) toAdd).lock();		142:	<code>return</code> collector;	
62:	tail.lock();		143:	}	
63:	((LockableDoublyLinkedNode) tail.getPrevious()).lock();		144:	}	
64:					
65:	<code>if</code> (toAdd == <code>null</code>) {				
66:	((LockableDoublyLinkedNode) toAdd).unlock();				
67:	tail.unlock();				
68:	((LockableDoublyLinkedNode) tail.getPrevious()).unlock();				
69:	<code>return false;</code>				
70:	}				
71:					
72:	toAdd.setPrevious(tail.getPrevious());				
73:	toAdd.setNext(tail);				
74:					
75:	tail.getPrevious().setNext(toAdd);				
76:	tail.setPrevious(toAdd);				
77:					
78:	size.incrementAndGet();				
79:					
80:	((LockableDoublyLinkedNode) toAdd).unlock();				
81:	tail.unlock();				

Section B

AuxiliaryCollectionImpl.java (1/2)

j1k21

Section B

AuxiliaryCollectionImpl.java (2/2)

j1k21

```

1: package datastructures;
2:
3: import java.util.ArrayList;
4: import java.util.List;
5:
6: /*
7:  NOTE: I have chosen to use dummy nodes for head and tail
8:  for my implementation of AuxiliaryCollectionImpl to simplify
9:  bookkeeping and avoid edge cases, such as when the list is
10: empty or only has one element. As a result of this design
11: decision, the following provided methods have been modified
12: to achieve the expected test results:
13: - The constructor
14: - find
15: - allKeysInOrder
16:
17: The commented code is from my initial attempt of implementing
18: the class without using dummy nodes.
19: */
20:
21: @SuppressWarnings({"rawtypes", "unchecked"})
22: public class AuxiliaryCollectionImpl<K, V> implements AuxiliaryCollectionI<K, V> {
23:
24:     private DoublyLinkedListNode<K, V> head;
25:     private DoublyLinkedListNode<K, V> tail;
26:     private int size;
27:
28:     public AuxiliaryCollectionImpl() {
29:         head = new DoublyLinkedListNode<>(null, null);
30:         tail = new DoublyLinkedListNode<>(null, null);
31:         head.setNext(tail);
32:         tail.setPrevious(head);
33:         size = 0;
34:     }
35:
36:     /**
37:      * Removes the given node from the Collection. It assumes that the input is either null or a node
38:      * which exists in the Collection.
39:      *
40:      * @param toRemove The DoublyLinkedListNode to remove.
41:      * @return Returns false if the input is null or true if the removal is successful.
42:      */
43:     @Override
44:     public boolean remove(DoublyLinkedListNode toRemove) {
45:         if (toRemove == null) {
46:             return false;
47:         }
48:         toRemove.getPrevious().setNext(toRemove.getNext());
49:         toRemove.getNext().setPrevious(toRemove.getPrevious());
50:         size--;
51:         // if (toRemove.getPrevious() != null) {
52:         //     toRemove.getPrevious().setNext(toRemove.getNext());
53:         // }
54:         // if (toRemove.getNext() != null) {
55:         //     toRemove.getNext().setPrevious(toRemove.getPrevious());
56:         // }
57:         // size--;
58:         // head = toRemove.getNext();
59:         // if (size <= 0) {
60:         //     tail = null;
61:         // }
62:         return true;
63:     }
64:
65:     /**
66:      * Appends the given node to the end of the Collection.
67:      *
68:      * @param toAdd The DoublyLinkedListNode to be appended.
69:      * @return Returns false if the node cannot be added or true otherwise.
70:      */
71:     @Override
72:     public boolean append(DoublyLinkedListNode toAdd) {
73:         if (toAdd == null) {
74:             return false;
75:         }
76:         toAdd.setPrevious(tail.getPrevious());
77:         toAdd.setNext(tail);
78:         tail.getPrevious().setNext(toAdd);
79:         tail.setPrevious(toAdd);
80:         size++;
81:         // if (head == null && tail == null) {

```

you need to handle the head and tail OR

8/10

5/5

```

82:     //     // There are no nodes in the list, i.e. size = 0
83:     //     head = toAdd;
84:     // } else {
85:     //     // There is at least one node in the list, i.e. size >= 1
86:     //     toAdd.setPrevious(tail);
87:     //     tail.setNext(toAdd);
88:     // }
89:     // tail = toAdd;
90:     return true;
91: }
92:
93: /**
94:  * Searches for a node in the Collection.
95:  *
96:  * @param key The key of the DoublyLinkedListNode to look for.
97:  * @return Returns a DoublyLinkedListNode which has a key equal the input key or null if such a node
98:  *         was not found.
99:  */
100: @Override
101: public DoublyLinkedListNode<K, V> find(K key) {
102:     DoublyLinkedListNode<K, V> curr = head;
103:
104:     while (curr != tail) {
105:         if (curr.getKey() == key) {
106:             return curr;
107:         }
108:         curr = curr.getNext();
109:     }
110:
111:     return null;
112: }
113:
114: /**
115:  * The size of the Collection.
116:  *
117:  * @return Returns an int representing the number of items currently in the Collection.
118:  */
119: public int size() {
120:     return size;
121: }
122:
123: /**
124:  * Returns the head of the Collection.
125:  *
126:  * @return A DoublyLinkedListNode which is currently the head of the Collection, or null if the
127:  *         Collection is empty.
128:  */
129: public DoublyLinkedListNode<K, V> getHead() {
130:     return this.head;
131: }
132:
133: /**
134:  * Traverses the Collection and stores the keys of all items currently in the Collection.
135:  *
136:  * @return Returns a List of keys of all the items currently in the Memory.
137:  */
138: @Override
139: public List<K> allKeysInOrder() {
140:     var collector = new ArrayList<K>();
141:     var curr = head.getNext();
142:
143:     while (curr != tail) {
144:         collector.add(curr.getKey());
145:         curr = curr.getNext();
146:     }
147:
148:     return collector;
149: }
150: }

```

```
1: package datastructures;
2:
3: import java.util.concurrent.locks.Lock;
4: import java.util.concurrent.locks.ReentrantLock;
5:
6: public class LockableDoublyLinkedListNode<K, V> extends DoublyLinkedListNode<K, V> {
7:     private final Lock lock;
8:
9:     public LockableDoublyLinkedListNode(K key, V value) {
10:         super(key, value);
11:         lock = new ReentrantLock();
12:     }
13:
14:     public void lock() {
15:         lock.lock();
16:     }
17:
18:     public void unlock() {
19:         lock.unlock();
20:     }
21: }
```

```
1: package datastructures;
2:
3: import java.util.List;
4: import java.util.Optional;
5:
6: public class MemoryImplCoarseGrainedSync<K, V> extends MemoryImpl<K, V> {
7:     public MemoryImplCoarseGrainedSync(int capacity) {
8:         super(capacity);
9:     }
10:
11:     @Override
12:     public synchronized Optional<V> read(K key) {
13:         return super.read(key);
14:     }
15:
16:     @Override
17:     public synchronized boolean write(K key, V value) {
18:         return super.write(key, value);
19:     }
20:
21:     @Override
22:     public synchronized int size() {
23:         return super.size();
24:     }
25:
26:     @Override
27:     public synchronized List<K> allMemKeysInOrder() {
28:         return super.allMemKeysInOrder();
29:     }
30:
31:     @Override
32:     public synchronized int getCapacity() {
33:         return super.getCapacity();
34:     }
35: }
36:
37: /*
38:  Fine-grained implementation:
39:
40:  For the implementation of the MemoryI interface, the Map data structure
41:  should be replaced with a thread-safe/atomic map data structure to avoid
42:  race conditions when accessing the map.
43:
44:  For the implementation of the AuxiliaryCollectionI interface,
45:  the following changes to the class design are proposed:
46:
47:  1. Instead of using an int for storing the size, an AtomicInteger
48:  should be used instead.
49:
50:  2. A lock should be added to every DoublyLinkedListNode, and corresponding
51:  public methods, e.g. lock() and unlock(), should be implemented to
52:  allow the AuxiliaryCollectionI implementation to lock and unlock the nodes.
53:  Alternatively, add a lockable subclass of DoublyLinkedListNode as an inner class
54:  of the AuxiliaryCollectionI implementation.
55:
56:  The following changes to the method implementations are proposed:
57:
58:  1. append: acquire locks on tail and tail.getPrevious() before reassigning
59:  pointers. Release the locks before returning from the method.
60:
61:  2. remove: acquire three locks in total: toRemove, toRemove.getPrevious(),
62:  and toRemove.getNext(). Release the locks before returning from the method.
63:
64:  3. read: acquire a lock on the map object before checking if the key
65:  is in the map. If the key is in the map, obtain the value of the node with
66:  the key and store it in a variable, then release the lock. Otherwise,
67:  release the lock and return from the method immediately. Since the implementation
68:  of AuxiliaryCollectionI is thread-safe, the call to update (private method in my
69:  implementation of MemoryImpl) should be thread-safe as well.
70:
71:  4. write: acquire a lock on the map object before accessing it. Release the lock
72:  right after the last access.
73:
74:  This implementation should entirely respect the LRU cache policy, as only one thread
75:  is allowed to access the tail end of the AuxiliaryCollectionI at any given moment.
76:  */
```

10/10

2/3

what order?

what order?

Section B

MemoryImplFineGrainedSync.java (1/2)

j1k21

Section B

MemoryImplFineGrainedSync.java (2/2)

j1k21

```
1: package datastructures;
2:
3: import java.util.HashMap;
4: import java.util.List;
5: import java.util.Optional;
6: import java.util.concurrent.locks.Lock;
7: import java.util.concurrent.locks.ReentrantLock;
8:
9: public class MemoryImplFineGrainedSync<K, V> implements MemoryI<K, V> {
10:     private static class LockableMap<K, V> extends HashMap<K, V> {
11:         private final Lock lock;
12:
13:         public LockableMap(int capacity) {
14:             super();
15:             lock = new ReentrantLock();
16:         }
17:
18:         public void lock() {
19:             lock.lock();
20:         }
21:
22:         public void unlock() {
23:             lock.unlock();
24:         }
25:     }
26:
27:     private final int capacity;
28:     LockableMap<K, DoublyLinkedListNode<K, V>> map;
29:     AuxiliaryCollectionImplFineGrainedSync list;
30:
31:     public MemoryImplFineGrainedSync(int capacity) {
32:         map = new LockableMap<>(capacity);
33:         list = new AuxiliaryCollectionImplFineGrainedSync<>();
34:
35:         this.capacity = capacity;
36:     }
37:
38:     /**
39:      * The size of the LRU memory.
40:      *
41:      * @return Returns an int equal to the number of items currently in the LRU memory.
42:      */
43:     public int size() {
44:         return map.size();
45:     }
46:
47:     /**
48:      * The maximum capacity this Memory can hold.
49:      *
50:      * @return Returns an int equal to the maximum number of items that can be held in this
51:      *         Memory.
52:      */
53:     @Override
54:     public int getCapacity() {
55:         return this.capacity;
56:     }
57:
58:     /**
59:      * Reads an item with the given key from the LRU memory.
60:      *
61:      * @param key The key of the item to read.
62:      * @return Returns an empty Optional for invalid keys or the value of the item upon a successful
63:      *         read.
64:      */
65:     @Override
66:     public Optional<V> read(K key) {
67:         map.lock();
68:         if (!map.containsKey(key)) {
69:             map.unlock();
70:             return Optional.empty();
71:         }
72:         Optional<V> value = Optional.of(map.get(key).getValue());
73:         map.unlock();
74:         update(key);
75:         return value;
76:     }
77:
78:     /**
79:      * Adds a new item with key and value to the LRU memory. Allows overwriting existing keys.
80:      *
81:      * @param key The key of the item
```

```
82:      * @param value The item's value
83:      * @return Returns true when completed.
84:      */
85:     @Override
86:     public boolean write(K key, V value) {
87:         if (key == null) {
88:             return false;
89:         }
90:
91:         map.lock();
92:
93:         if (map.containsKey(key)) {
94:             map.get(key).setValue(value);
95:             update(key);
96:             map.unlock();
97:             return true;
98:         }
99:         if (size() >= capacity) {
100:             DoublyLinkedListNode<K, V> lruNode = list.getHead().getNext();
101:             list.remove(lruNode);
102:             map.remove(lruNode.getKey());
103:         }
104:         DoublyLinkedListNode<K, V> node = new DoublyLinkedListNode<>(key, value);
105:         map.put(key, node);
106:
107:         map.unlock();
108:
109:         list.append(node);
110:         return true;
111:     }
112:
113:     /**
114:      * Finds the keys of all items currently in Memory.
115:      *
116:      * @return Returns a List of keys of all the items currently in Memory.
117:      */
118:     @Override
119:     public List<K> allMemKeysInOrder() {
120:         return list.allKeysInOrder();
121:     }
122:
123:     /**
124:      * Updates the node with the specified key such that it becomes
125:      * the most recently used.
126:      *
127:      * @param key The key of the node to be updated.
128:      */
129:     private void update(K key) {
130:         if (!map.containsKey(key)) {
131:             return;
132:         }
133:         DoublyLinkedListNode<K, V> node = map.get(key);
134:         list.remove(node);
135:         list.append(node);
136:     }
137: }
```

```
1: package datastructures;
2:
3: import java.util.HashMap;
4: import java.util.List;
5: import java.util.Optional;
6:
7: /*
8:  NOTE: I have chosen to use sentinel nodes for head and tail
9:  for my implementation of AuxiliaryCollectionImpl.
10: */
11: @SuppressWarnings({"rawtypes", "unchecked"})
12: public class MemoryImpl<K, V> implements MemoryI<K, V> {
13:
14:     private final int capacity;
15:     HashMap<K, DoublyLinkedNode<K, V>> map;
16:     AuxiliaryCollectionImpl list;
17:
18:     public MemoryImpl(int capacity) {
19:         map = new HashMap<>(capacity);
20:         list = new AuxiliaryCollectionImpl<>();
21:
22:         this.capacity = capacity;
23:     }
24:
25:     /**
26:      * The size of the LRU memory.
27:      *
28:      * @return Returns an int equal to the number of items currently in the LRU memory.
29:      */
30:     public int size() {
31:         return map.size();
32:     }
33:
34:     /**
35:      * The maximum capacity this Memory can hold.
36:      *
37:      * @return Returns an int equal to the maximum number of items that can be held in this
38:      *         Memory.
39:      */
40:     @Override
41:     public int getCapacity() {
42:         return this.capacity;
43:     }
44:
45:     /**
46:      * Reads an item with the given key from the LRU memory.
47:      *
48:      * @param key The key of the item to read.
49:      * @return Returns an empty Optional for invalid keys or the value of the item upon a successful
50:      *         read.
51:      */
52:     @Override
53:     public Optional<V> read(K key) {
54:         if (!map.containsKey(key)) {
55:             return Optional.empty();
56:         }
57:         update(key);
58:         return Optional.of(map.get(key).getValue());
59:     }
60:
61:     /**
62:      * Adds a new item with key and value to the LRU memory. Allows overwriting existing keys.
63:      *
64:      * @param key The key of the item
65:      * @param value The item's value
66:      * @return Returns true when completed.
67:      */
68:     @Override
69:     public boolean write(K key, V value) {
70:         if (key == null) {
71:             return false;
72:         }
73:         if (map.containsKey(key)) {
74:             map.get(key).setValue(value);
75:             update(key);
76:             return true;
77:         }
78:         if (size() >= capacity) {
79:             DoublyLinkedNode<K, V> lruNode = list.getHead().getNext();
80:             list.remove(lruNode);
81:             map.remove(lruNode.getKey());
```

```
82:     }
83:     DoublyLinkedNode<K, V> node = new DoublyLinkedNode<>(key, value);
84:     map.put(key, node);
85:     list.append(node);
86:     return true;
87: }
88:
89: /**
90:  * Finds the keys of all items currently in Memory.
91:  *
92:  * @return Returns a List of keys of all the items currently in Memory.
93:  */
94: @Override
95: public List<K> allMemKeysInOrder() {
96:     return list.allKeysInOrder();
97: }
98:
99: /**
100:  * Updates the node with the specified key such that it becomes
101:  * the most recently used.
102:  *
103:  * @param key The key of the node to be updated.
104:  */
105: private void update(K key) {
106:     if (!map.containsKey(key)) {
107:         return;
108:     }
109:     DoublyLinkedNode<K, V> node = map.get(key);
110:     list.remove(node);
111:     list.append(node);
112: }
113: }
```

Section B	Output (1/2)	j1k21	Section B	Output (2/2)	j1k21
	1: AuxiliaryCollectionImplTest - Warnings exist. 2: src/datastructures/MemoryImplFineGrainedSync.java:10: warning: [serial] serializable class <code>LockableMap</code> has no definition of serialVersionUID 3: <code>private static class LockableMap<K, V> extends HashMap<K, V> {</code> 4: <code>^</code> 5: src/datastructures/MemoryImplFineGrainedSync.java:11: warning: [serial] non- transient instance field of a serializable class declared with a non-serializable type 6: <code>private final Lock lock;</code> 7: <code>^</code> 8: src/datastructures/MemoryImplFineGrainedSync.java:29: warning: [rawtypes] found raw type: <code>AuxiliaryCollectionImplFineGrainedSync</code> 9: <code>AuxiliaryCollectionImplFineGrainedSync list;</code> 10: <code>^</code> 11: missing type arguments for generic class <code>AuxiliaryCollectionImplFineGrainedSync<K,V></code> 12: where K,V are type-variables: 13: <code>K extends</code> Object declared in class <code>AuxiliaryCollectionImplFineGrainedSync</code> 14: <code>V extends</code> Object declared in class <code>AuxiliaryCollectionImplFineGrainedSync</code> 15: src/datastructures/MemoryImplFineGrainedSync.java:100: warning: [unchecked] unchecked conversion 16: <code>DoublyLinkedListNode<K, V> lruNode = list.getHead().getNext();</code> 17: <code>^</code> 18: required: <code>DoublyLinkedListNode<K,V></code> 19: found: <code>DoublyLinkedListNode</code> 20: where K,V are type-variables: 21: <code>K extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 22: <code>V extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 23: src/datastructures/MemoryImplFineGrainedSync.java:120: warning: [unchecked] unchecked conversion 24: <code>return list.allKeysInOrder();</code> 25: <code>^</code> 26: required: <code>List<K></code> 27: found: <code>List</code> 28: where K is a type-variable: 29: <code>K extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 30: 5 warnings 31: Model Answer's Tests - AuxiliaryCollectionImplTest works! 32: 33: JUnit version 4.12 34: 35: Time: 0.01 36: 37: OK (16 tests) 38: 39: 40: ChaosMonkeyTest - Warnings exist. 41: src/datastructures/MemoryImplFineGrainedSync.java:10: warning: [serial] serializable class <code>LockableMap</code> has no definition of serialVersionUID 42: <code>private static class LockableMap<K, V> extends HashMap<K, V> {</code> 43: <code>^</code> 44: src/datastructures/MemoryImplFineGrainedSync.java:11: warning: [serial] non- transient instance field of a serializable class declared with a non-serializable type 45: <code>private final Lock lock;</code> 46: <code>^</code> 47: src/datastructures/MemoryImplFineGrainedSync.java:29: warning: [rawtypes] found raw type: <code>AuxiliaryCollectionImplFineGrainedSync</code> 48: <code>AuxiliaryCollectionImplFineGrainedSync list;</code> 49: <code>^</code> 50: missing type arguments for generic class <code>AuxiliaryCollectionImplFineGrainedSync<K,V></code> 51: where K,V are type-variables: 52: <code>K extends</code> Object declared in class <code>AuxiliaryCollectionImplFineGrainedSync</code> 53: <code>V extends</code> Object declared in class <code>AuxiliaryCollectionImplFineGrainedSync</code> 54: src/datastructures/MemoryImplFineGrainedSync.java:100: warning: [unchecked] unchecked conversion 55: <code>DoublyLinkedListNode<K, V> lruNode = list.getHead().getNext();</code> 56: <code>^</code> 57: required: <code>DoublyLinkedListNode<K,V></code> 58: found: <code>DoublyLinkedListNode</code> 59: where K,V are type-variables: 60: <code>K extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 61: <code>V extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 62: src/datastructures/MemoryImplFineGrainedSync.java:120: warning: [unchecked] unchecked conversion 63: <code>return list.allKeysInOrder();</code> 64: <code>^</code> 65: required: <code>List<K></code> 66: found: <code>List</code> 67: where K is a type-variable: 68: <code>K extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 69: 5 warnings 70: Model Answer's Tests - ChaosMonkeyTest works! 71: 72: JUnit version 4.12 73: . 74: Time: 15.38 75:			76: OK (1 test) 77: 78: 79: MemoryImplTest - Warnings exist. 80: src/datastructures/MemoryImplFineGrainedSync.java:10: warning: [serial] serializable class <code>LockableMap</code> has no definition of serialVersionUID 81: <code>private static class LockableMap<K, V> extends HashMap<K, V> {</code> 82: <code>^</code> 83: src/datastructures/MemoryImplFineGrainedSync.java:11: warning: [serial] non- transient instance field of a serializable class declared with a non-serializable type 84: <code>private final Lock lock;</code> 85: <code>^</code> 86: src/datastructures/MemoryImplFineGrainedSync.java:29: warning: [rawtypes] found raw type: <code>AuxiliaryCollectionImplFineGrainedSync</code> 87: <code>AuxiliaryCollectionImplFineGrainedSync list;</code> 88: <code>^</code> 89: missing type arguments for generic class <code>AuxiliaryCollectionImplFineGrainedSync<K,V></code> 90: where K,V are type-variables: 91: <code>K extends</code> Object declared in class <code>AuxiliaryCollectionImplFineGrainedSync</code> 92: <code>V extends</code> Object declared in class <code>AuxiliaryCollectionImplFineGrainedSync</code> 93: src/datastructures/MemoryImplFineGrainedSync.java:100: warning: [unchecked] unchecked conversion 94: <code>DoublyLinkedListNode<K, V> lruNode = list.getHead().getNext();</code> 95: <code>^</code> 96: required: <code>DoublyLinkedListNode<K,V></code> 97: found: <code>DoublyLinkedListNode</code> 98: where K,V are type-variables: 99: <code>K extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 100: <code>V extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 101: src/datastructures/MemoryImplFineGrainedSync.java:120: warning: [unchecked] unchecked conversion 102: <code>return list.allKeysInOrder();</code> 103: <code>^</code> 104: required: <code>List<K></code> 105: found: <code>List</code> 106: where K is a type-variable: 107: <code>K extends</code> Object declared in class <code>MemoryImplFineGrainedSync</code> 108: 5 warnings 109: Model Answer's Tests - MemoryImplTest works! 110: 111: JUnit version 4.12 112: 113: Time: 0.008 114: 115: OK (11 tests) 116: 117:	