Section B

Q1	5 /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 45 / 50

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.

```java
package datastructures;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

@SuppressWarnings({"rawtypes", "unchecked"})
public class AuxiliaryCollectionImplFineGrainedSync<K, V> implements AuxiliaryCollectionI<K, V> {

  private LockableDoublyLinkedNode<K, V> head;
  private LockableDoublyLinkedNode<K, V> tail;
  private AtomicInteger size;

  public AuxiliaryCollectionImplFineGrainedSync() {
    head = new LockableDoublyLinkedNode<>(null, null);
    tail = new LockableDoublyLinkedNode<>(null, null);
    head.setNext(tail);
    tail.setPrevious(head);
    size = new AtomicInteger(0);
  }

  /**
   * Removes the given node from the Collection. It assumes that the input is either null or a node
   * which exists in the Collection.
   *
   * @param toRemove The DoublyLinkedNode to remove.
   * @return Returns false if the input is null or true if the removal is successful.
   */
  @Override
  public boolean remove(DoublyLinkedNode toRemove) {
    ((LockableDoublyLinkedNode) toRemove).lock();
    ((LockableDoublyLinkedNode) toRemove.getPrevious()).lock();
    ((LockableDoublyLinkedNode) toRemove.getNext()).lock();

    if (toRemove == null) {
      ((LockableDoublyLinkedNode) toRemove).unlock();
      ((LockableDoublyLinkedNode) toRemove.getPrevious()).unlock();
      ((LockableDoublyLinkedNode) toRemove.getNext()).unlock();
      return false;
    }

    toRemove.getPrevious().setNext(toRemove.getNext());
    toRemove.getNext().setPrevious(toRemove.getPrevious());
    size.decrementAndGet();

    ((LockableDoublyLinkedNode) toRemove).unlock();
    ((LockableDoublyLinkedNode) toRemove.getPrevious()).unlock();
    ((LockableDoublyLinkedNode) toRemove.getNext()).unlock();

    return true;
  }

  /**
   * Appends the given node to the end of the Collection.
   *
   * @param toAdd The DoublyLinkedNode to be appended.
   * @return Returns false if the node cannot be added or true otherwise.
   */
  @Override
  public boolean append(DoublyLinkedNode toAdd) {
    ((LockableDoublyLinkedNode) toAdd).lock();
    tail.lock();
    ((LockableDoublyLinkedNode) tail.getPrevious()).lock();

    if (toAdd == null) {
      ((LockableDoublyLinkedNode) toAdd).unlock();
      tail.unlock();
      ((LockableDoublyLinkedNode) tail.getPrevious()).unlock();
      return false;
    }

    toAdd.setPrevious(tail.getPrevious());
    toAdd.setNext(tail);

    tail.getPrevious().setNext(toAdd);
    tail.setPrevious(toAdd);

    size.incrementAndGet();

    ((LockableDoublyLinkedNode) toAdd).unlock();
    tail.unlock();
```



```java
    ((LockableDoublyLinkedNode) tail.getPrevious()).unlock();

    return true;
  }

  /**
   * Searches for a node in the Collection.
   *
   * @param key The key of the DoublyLinkedNode to look for.
   * @return Returns a DoublyLinkedNode which has a key equal the input key or null if such a node
   *         was not found.
   */
  @Override
  public DoublyLinkedNode<K, V> find(K key) {
    DoublyLinkedNode<K, V> curr = head;

    while (curr != tail) {
      if (curr.getKey() == key) {
        return curr;
      }
      curr = curr.getNext();
    }

    return null;
  }

  /**
   * The size of the Collection.
   *
   * @return Returns an int representing the number of items currently in the Collection.
   */
  public int size() {
    return size.get();
  }

  /**
   * Returns the head of the Collection.
   *
   * @return A DoublyLinkedNode which is currently the head of the Collection, or null if the
   *         Collection is empty.
   */
  public DoublyLinkedNode<K, V> getHead() {
    return this.head;
  }

  /**
   * Traverses the Collection and stores the keys of all items currently in the Collection.
   *
   * @return Returns a List of keys of all the items currently in the Memory.
   */
  @Override
  public List<K> allKeysInOrder() {
    var collector = new ArrayList<K>();
    var curr = head.getNext();

    while (curr != tail) {
      collector.add(curr.getKey());
      curr = curr.getNext();
    }

    return collector;
  }
}
```

```java
package datastructures;

import java.util.ArrayList;
import java.util.List;

/*
  NOTE: I have chosen to use dummy nodes for head and tail
  for my implementation of AuxiliaryCollectionImpl to simplify
  bookkeeping and avoid edge cases, such as when the list is
  empty or only has one element. As a result of this design
  decision, the following provided methods have been modified
  to achieve the expected test results:
  - The constructor
  - find
  - allKeysInOrder

  The commented code is from my initial attempt of implementing
  the class without using dummy nodes.
 */

@SuppressWarnings({"rawtypes", "unchecked"})
public class AuxiliaryCollectionImpl<K, V> implements AuxiliaryCollectionI<K, V> {

  private DoublyLinkedNode<K, V> head;
  private DoublyLinkedNode<K, V> tail;
  private int size;

  public AuxiliaryCollectionImpl() {
    head = new DoublyLinkedNode<>(null, null);
    tail = new DoublyLinkedNode<>(null, null);
    head.setNext(tail);
    tail.setPrevious(head);
    size = 0;
  }

  /**
   * Removes the given node from the Collection. It assumes that the input is either null or a node
   * which exists in the Collection.
   *
   * @param toRemove The DoublyLinkedNode to remove.
   * @return Returns false if the input is null or true if the removal is successful.
   */
  @Override
  public boolean remove(DoublyLinkedNode toRemove) {
    if (toRemove == null) {
      return false;
    }
    toRemove.getPrevious().setNext(toRemove.getNext());
    toRemove.getNext().setPrevious(toRemove.getPrevious());
    size--;
    //    if (toRemove.getPrevious() != null) {
    //      toRemove.getPrevious().setNext(toRemove.getNext());
    //    }
    //    if (toRemove.getNext() != null) {
    //      toRemove.getNext().setPrevious(toRemove.getPrevious());
    //    }
    //    size--;
    //    head = toRemove.getNext();
    //    if (size <= 0) {
    //      tail = null;
    //    }
    return true;
  }

  /**
   * Appends the given node to the end of the Collection.
   *
   * @param toAdd The DoublyLinkedNode to be appended.
   * @return Returns false if the node cannot be added or true otherwise.
   */
  @Override
  public boolean append(DoublyLinkedNode toAdd) {
    if (toAdd == null) {
      return false;
    }
    toAdd.setPrevious(tail.getPrevious());
    toAdd.setNext(tail);
    tail.getPrevious().setNext(toAdd);
    tail.setPrevious(toAdd);
    size++;
    //    if (head == null && tail == null) {
```

you need to handle the head and tail ok

8/10

5/5


```java
    //      // There are no nodes in the list, i.e. size = 0
    //      head = toAdd;
    //    } else {
    //      // There is at least one node in the list, i.e. size >= 1
    //      toAdd.setPrevious(tail);
    //      tail.setNext(toAdd);
    //    }
    //    tail = toAdd;
    return true;
  }

  /**
   * Searches for a node in the Collection.
   *
   * @param key The key of the DoublyLinkedNode to look for.
   * @return Returns a DoublyLinkedNode which has a key equal the input key or null if such a node
   *         was not found.
   */
  @Override
  public DoublyLinkedNode<K, V> find(K key) {
    DoublyLinkedNode<K, V> curr = head;

    while (curr != tail) {
      if (curr.getKey() == key) {
        return curr;
      }
      curr = curr.getNext();
    }

    return null;
  }

  /**
   * The size of the Collection.
   *
   * @return Returns an int representing the number of items currently in the Collection.
   */
  public int size() {
    return size;
  }

  /**
   * Returns the head of the Collection.
   *
   * @return A DoublyLinkedNode which is currently the head of the Collection, or null if the
   *         Collection is empty.
   */
  public DoublyLinkedNode<K, V> getHead() {
    return this.head;
  }

  /**
   * Traverses the Collection and stores the keys of all items currently in the Collection.
   *
   * @return Returns a List of keys of all the items currently in the Memory.
   */
  @Override
  public List<K> allKeysInOrder() {
    var collector = new ArrayList<K>();
    var curr = head.getNext();

    while (curr != tail) {
      collector.add(curr.getKey());
      curr = curr.getNext();
    }

    return collector;
  }
}
```

```java
package datastructures;

import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class LockableDoublyLinkedNode<K, V> extends DoublyLinkedNode<K, V> {
  private final Lock lock;

  public LockableDoublyLinkedNode(K key, V value) {
    super(key, value);
    lock = new ReentrantLock();
  }

  public void lock() {
    lock.lock();
  }

  public void unlock() {
    lock.unlock();
  }
}
```



```java
package datastructures;

import java.util.List;
import java.util.Optional;

public class MemoryImplCoarseGrainedSync<K, V> extends MemoryImpl<K, V> {
  public MemoryImplCoarseGrainedSync(int capacity) {
    super(capacity);
  }

  @Override
  public synchronized Optional<V> read(K key) {
    return super.read(key);
  }

  @Override
  public synchronized boolean write(K key, V value) {
    return super.write(key, value);
  }

  @Override
  public synchronized int size() {
    return super.size();
  }

  @Override
  public synchronized List<K> allMemKeysInOrder() {
    return super.allMemKeysInOrder();
  }

  @Override
  public synchronized int getCapacity() {
    return super.getCapacity();
  }
}

/*
  Fine-grained implementation:

  For the implementation of the MemoryI interface, the Map data structure
  should be replaced with a thread-safe/atomic map data structure to avoid
  race conditions when accessing the map.

  For the implementation of the AuxiliaryCollectionI interface,
  the following changes to the class design are proposed:

  1. Instead of using an int for storing the size, an AtomicInteger
  should be used instead.

  2. A lock should be added to every DoublyLinkedNode, and corresponding
  public methods, e.g. lock() and unlock(), should be implemented to
  allow the AuxiliaryCollectionI implementation to lock and unlock the nodes.
  Alternatively, add a lockable subclass of DoublyLinkedNode as an inner class
  of the AuxiliaryCollectionI implementation.

  The following changes to the method implementations are proposed:

  1. append: acquire locks on tail and tail.getPrevious() before reassigning
  pointers. Release the locks before returning from the method.

  2. remove: acquire three locks in total: toRemove, toRemove.getPrevious(),
  and toRemove.getNext(). Release the locks before returning from the method.

  3. read: acquire a lock on the map object before checking if the key
  is in the map. If the key is in the map, obtain the value of the node with
  the key and store it in a variable, then release the lock. Otherwise,
  release the lock and return from the method immediately. Since the implementation
  of AuxiliaryCollectionI is thread-safe, the call to update (private method in my
  implementation of MemoryImpl) should be thread-safe as well.

  4. write: acquire a lock on the map object before accesing it. Release the lock
  right after the last access.

  This implementation should entirely respect the LRU cache policy, as only one thread
  is allowed to access the tail end of the AuxiliaryCollectionI at any given moment.
 */
```

10/10

2/3

what order?

what order?

```java
package datastructures;

import java.util.HashMap;
import java.util.List;
import java.util.Optional;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class MemoryImplFineGrainedSync<K, V> implements MemoryI<K, V> {
  private static class LockableMap<K, V> extends HashMap<K, V> {
    private final Lock lock;

    public LockableMap(int capacity) {
      super();
      lock = new ReentrantLock();
    }

    public void lock() {
      lock.lock();
    }

    public void unlock() {
      lock.unlock();
    }
  }

  private final int capacity;
  LockableMap<K, DoublyLinkedNode<K, V>> map;
  AuxiliaryCollectionImplFineGrainedSync list;

  public MemoryImplFineGrainedSync(int capacity) {
    map = new LockableMap<>(capacity);
    list = new AuxiliaryCollectionImplFineGrainedSync<>();

    this.capacity = capacity;
  }

  /**
   * The size of the LRU memory.
   *
   * @return Returns an int equal to the number of items currently in the LRU memory.
   */
  public int size() {
    return map.size();
  }

  /**
   * The maximum capacity this Memory can hold.
   *
   * @return Returns an int equal to the maximum number of items that can be held in this
   *         Memory.
   */
  @Override
  public int getCapacity() {
    return this.capacity;
  }

  /**
   * Reads an item with the given key from the LRU memory.
   *
   * @param key The key of the item to read.
   * @return Returns an empty Optional for invalid keys or the value of the item upon a successful
   *         read.
   */
  @Override
  public Optional<V> read(K key) {
    map.lock();
    if (!map.containsKey(key)) {
      map.unlock();
      return Optional.empty();
    }
    Optional<V> value = Optional.of(map.get(key).getValue());
    map.unlock();
    update(key);
    return value;
  }

  /**
   * Adds a new item with key and value to the LRU memory. Allows overwriting existing keys.
   *
   * @param key   The key of the item
```



```java
   * @param value The item's value
   * @return Returns true when completed.
   */
  @Override
  public boolean write(K key, V value) {
    if (key == null) {
      return false;
    }

    map.lock();

    if (map.containsKey(key)) {
      map.get(key).setValue(value);
      update(key);
      map.unlock();
      return true;
    }
    if (size() >= capacity) {
      DoublyLinkedNode<K, V> lruNode = list.getHead().getNext();
      list.remove(lruNode);
      map.remove(lruNode.getKey());
    }
    DoublyLinkedNode<K, V> node = new DoublyLinkedNode<>(key, value);
    map.put(key, node);

    map.unlock();

    list.append(node);
    return true;
  }

  /**
   * Finds the keys of all items currently in Memory.
   *
   * @return Returns a List of keys of all the items currently in Memory.
   */
  @Override
  public List<K> allMemKeysInOrder() {
    return list.allKeysInOrder();
  }

  /**
   * Updates the node with the specified key such that it becomes
   * the most recently used.
   *
   * @param key The key of the node to be updated.
   */
  private void update(K key) {
    if (!map.containsKey(key)) {
      return;
    }
    DoublyLinkedNode<K, V> node = map.get(key);
    list.remove(node);
    list.append(node);
  }
}
```

```java
package datastructures;

import java.util.HashMap;
import java.util.List;
import java.util.Optional;

/*
  NOTE: I have chosen to use sentinel nodes for head and tail
  for my implementation of AuxiliaryCollectionImpl.
 */
@SuppressWarnings({"rawtypes", "unchecked"})
public class MemoryImpl<K, V> implements MemoryI<K, V> {

  private final int capacity;
  HashMap<K, DoublyLinkedNode<K, V>> map;
  AuxiliaryCollectionImpl list;

  public MemoryImpl(int capacity) {
    map = new HashMap<>(capacity);
    list = new AuxiliaryCollectionImpl<>();

    this.capacity = capacity;
  }

  /**
   * The size of the LRU memory.
   *
   * @return Returns an int equal to the number of items currently in the LRU memory.
   */
  public int size() {
    return map.size();
  }

  /**
   * The maximum capacity this Memory can hold.
   *
   * @return Returns an int equal to the maximum number of items that can be held in this
   *         Memory.
   */
  @Override
  public int getCapacity() {
    return this.capacity;
  }

  /**
   * Reads an item with the given key from the LRU memory.
   *
   * @param key The key of the item to read.
   * @return Returns an empty Optional for invalid keys or the value of the item upon a successful
   *         read.
   */
  @Override
  public Optional<V> read(K key) {
    if (!map.containsKey(key)) {
      return Optional.empty();
    }
    update(key);
    return Optional.of(map.get(key).getValue());
  }

  /**
   * Adds a new item with key and value to the LRU memory. Allows overwriting existing keys.
   *
   * @param key   The key of the item
   * @param value The item's value
   * @return Returns true when completed.
   */
  @Override
  public boolean write(K key, V value) {
    if (key == null) {
      return false;
    }
    if (map.containsKey(key)) {
      map.get(key).setValue(value);
      update(key);
      return true;
    }
    if (size() >= capacity) {
      DoublyLinkedNode<K, V> lruNode = list.getHead().getNext();
      list.remove(lruNode);
      map.remove(lruNode.getKey());
```



```java
    }
    DoublyLinkedNode<K, V> node = new DoublyLinkedNode<>(key, value);
    map.put(key, node);
    list.append(node);
    return true;
  }

  /**
   * Finds the keys of all items currently in Memory.
   *
   * @return Returns a List of keys of all the items currently in Memory.
   */
  @Override
  public List<K> allMemKeysInOrder() {
    return list.allKeysInOrder();
  }

  /**
   * Updates the node with the specified key such that it becomes
   * the most recently used.
   *
   * @param key The key of the node to be updated.
   */
  private void update(K key) {
    if (!map.containsKey(key)) {
      return;
    }
    DoublyLinkedNode<K, V> node = map.get(key);
    list.remove(node);
    list.append(node);
  }
}
```

```
Warnings exist.
src/datastructures/MemoryImplFineGrainedSync.java:10: warning: [serial] serializable class LockableMap has no definition of serialVersionUID
  private static class LockableMap<K, V> extends HashMap<K, V> {
                 ^
src/datastructures/MemoryImplFineGrainedSync.java:11: warning: [serial] non-transient instance field of a serializable class declared with a non-serializable type
    private final Lock lock;
                  ^
src/datastructures/MemoryImplFineGrainedSync.java:29: warning: [rawtypes] found raw type: AuxiliaryCollectionImplFineGrainedSync
  AuxiliaryCollectionImplFineGrainedSync list;
  ^
  missing type arguments for generic class AuxiliaryCollectionImplFineGrainedSync<K,V>
  where K,V are type-variables:
    K extends Object declared in class AuxiliaryCollectionImplFineGrainedSync
    V extends Object declared in class AuxiliaryCollectionImplFineGrainedSync
src/datastructures/MemoryImplFineGrainedSync.java:100: warning: [unchecked] unchecked conversion
      DoublyLinkedNode<K, V> lruNode = list.getHead().getNext();
                                                      ^
  required: DoublyLinkedNode<K,V>
  found:    DoublyLinkedNode
  where K,V are type-variables:
    K extends Object declared in class MemoryImplFineGrainedSync
    V extends Object declared in class MemoryImplFineGrainedSync
src/datastructures/MemoryImplFineGrainedSync.java:120: warning: [unchecked] unchecked conversion
    return list.allKeysInOrder();
                               ^
  required: List<K>
  found:    List
  where K is a type-variable:
    K extends Object declared in class MemoryImplFineGrainedSync
5 warnings
Model Answer's Tests - AuxiliaryCollectionImplTest works!

JUnit version 4.12
................
Time: 0.01

OK (16 tests)


ChaosMonkeyTest - Warnings exist.
src/datastructures/MemoryImplFineGrainedSync.java:10: warning: [serial] serializable class LockableMap has no definition of serialVersionUID
  private static class LockableMap<K, V> extends HashMap<K, V> {
                 ^
src/datastructures/MemoryImplFineGrainedSync.java:11: warning: [serial] non-transient instance field of a serializable class declared with a non-serializable type
    private final Lock lock;
                  ^
src/datastructures/MemoryImplFineGrainedSync.java:29: warning: [rawtypes] found raw type: AuxiliaryCollectionImplFineGrainedSync
  AuxiliaryCollectionImplFineGrainedSync list;
  ^
  missing type arguments for generic class AuxiliaryCollectionImplFineGrainedSync<K,V>
  where K,V are type-variables:
    K extends Object declared in class AuxiliaryCollectionImplFineGrainedSync
    V extends Object declared in class AuxiliaryCollectionImplFineGrainedSync
src/datastructures/MemoryImplFineGrainedSync.java:100: warning: [unchecked] unchecked conversion
      DoublyLinkedNode<K, V> lruNode = list.getHead().getNext();
                                                      ^
  required: DoublyLinkedNode<K,V>
  found:    DoublyLinkedNode
  where K,V are type-variables:
    K extends Object declared in class MemoryImplFineGrainedSync
    V extends Object declared in class MemoryImplFineGrainedSync
src/datastructures/MemoryImplFineGrainedSync.java:120: warning: [unchecked] unchecked conversion
    return list.allKeysInOrder();
                               ^
  required: List<K>
  found:    List
  where K is a type-variable:
    K extends Object declared in class MemoryImplFineGrainedSync
5 warnings
Model Answer's Tests - ChaosMonkeyTest works!

JUnit version 4.12
.
Time: 15.38

```



```
OK (1 test)


MemoryImplTest - Warnings exist.
src/datastructures/MemoryImplFineGrainedSync.java:10: warning: [serial] serializable class LockableMap has no definition of serialVersionUID
  private static class LockableMap<K, V> extends HashMap<K, V> {
                 ^
src/datastructures/MemoryImplFineGrainedSync.java:11: warning: [serial] non-transient instance field of a serializable class declared with a non-serializable type
    private final Lock lock;
                  ^
src/datastructures/MemoryImplFineGrainedSync.java:29: warning: [rawtypes] found raw type: AuxiliaryCollectionImplFineGrainedSync
  AuxiliaryCollectionImplFineGrainedSync list;
  ^
  missing type arguments for generic class AuxiliaryCollectionImplFineGrainedSync<K,V>
  where K,V are type-variables:
    K extends Object declared in class AuxiliaryCollectionImplFineGrainedSync
    V extends Object declared in class AuxiliaryCollectionImplFineGrainedSync
src/datastructures/MemoryImplFineGrainedSync.java:100: warning: [unchecked] unchecked conversion
      DoublyLinkedNode<K, V> lruNode = list.getHead().getNext();
                                                      ^
  required: DoublyLinkedNode<K,V>
  found:    DoublyLinkedNode
  where K,V are type-variables:
    K extends Object declared in class MemoryImplFineGrainedSync
    V extends Object declared in class MemoryImplFineGrainedSync
src/datastructures/MemoryImplFineGrainedSync.java:120: warning: [unchecked] unchecked conversion
    return list.allKeysInOrder();
                               ^
  required: List<K>
  found:    List
  where K is a type-variable:
    K extends Object declared in class MemoryImplFineGrainedSync
5 warnings
Model Answer's Tests - MemoryImplTest works!

JUnit version 4.12
...........
Time: 0.008

OK (11 tests)


```