Exercise 8.1.1

Yes at first the implementation passes the test. The test seems to have good method coverage since it covers all 8 methods. Statement coverage also looks good. We found two issues regarding branch coverage;

- Calls to reallocate buckets from the put() and putIfAbsent() methods are not guaranteed with this test setup. We are calling a version of the reallocate buckets that doesn't do anything.
- Another one is that we never test remove() with a non-existing value.

To force the reallocation of the buckets we came up with the following method;

```
private static void forceReallocateBuckets(OurMap map, int buckets, int
locks) {
   int maxPerBucket = buckets / locks;
   int sizeNeeded = maxPerBucket * buckets + 1;
   for (int i = 0; i <= sizeNeeded; i++) map.put(i, "Test");
   for (int i = sizeNeeded; i >= 0; i--) map.remove(i);
}
```

Also we added "buckets" and "locks" as parameters to the test method, which should always be the same as the map that is passed in. Then we call it within the test method before we add anything to it;

```
private static void testMap(final OurMap<Integer, String> map, int buckets,
int locks) {
   System.out.printf("%n%s%n", map.getClass());
   assert map.size() == 0;
   assert !map.containsKey(117);
   assert !map.containsKey(-2);
   assert map.get(117) == null;
   forceReallocateBuckets(map, buckets, locks); //<- Reallocate buckets
   assert map.put(117, "A") == null;
   assert map.containsKey(117);
   .........
}
```

Then we added this assertion call to remove();

```
assert map.remove(117).equals("C");
assert map.remove(117) == null; //<- New call to remove
```

Exercise 8.1.2

For this we created a new method “testMapConcurrently”, which first creates X number of threads and waits for them all to start (using a Cyclic barrier). Then each thread adds and removes random numbers from the map (10 times for each thread) whilst keeping track of all the work, then after that they all wait for each other to finish.

Then finally we sum all the numbers in the map and assert that it should be the same as the work performed by the threads;

```
private static void testMapConcurrently(final OurMap<Integer, String> map,
final int nrOfThreads) {
   CyclicBarrier barrier = new CyclicBarrier(nrOfThreads + 1);
   AtomicInteger putSum = new AtomicInteger();
   Thread[] threads = new Thread[nrOfThreads];
   for (int i = 0; i < nrOfThreads; i++)
       threads[i] = new Thread(() -> {
           Random random = new Random();
           try {
               barrier.await();
               for (int j = 0; j < 10; j++) {
                   int putKey = random.nextInt(99);
                   int putIfAbsentKey = random.nextInt(99);
                   int removeKey = random.nextInt(99);
                   map.containsKey(random.nextInt(99));
                   if (map.put(putKey, "value") == null)
                       putSum.addAndGet(putKey);
                   if (map.putIfAbsent(putIfAbsentKey, "value") == null)
                       putSum.addAndGet(putIfAbsentKey);
                   if (map.remove(removeKey) != null)
                       putSum.addAndGet(-1 * removeKey);
               }
               barrier.await();
           } catch (InterruptedException e) {
               e.printStackTrace();
           } catch (BrokenBarrierException e) {
               e.printStackTrace();
           }
       });
   for (int i = 0; i < nrOfThreads; i++) {
       threads[i].start();
   }
   try {
```

```
        barrier.await(); //Waits for everyone to start.
        barrier.await(); //Waits for everyone to finish.
        for (int i = 0; i < nrOfThreads; i++) {
            threads[i].join();
        }
        AtomicInteger mapSum = new AtomicInteger();
        map.forEach((k, v) -> mapSum.addAndGet(k));
        assert mapSum.get() == putSum.get();
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (BrokenBarrierException e) {
        e.printStackTrace();
    }
}
```

Exercise 8.1.3

Our test method worked well on both our StripedWriteMap and the WrapConcurrentHashMap. Then for fun we tried to run it on a normal HashMap which thankfully failed, so we concluded that we are doing something right.

Exercise 8.1.4

We added a new atomic integer called threadNumber;

```
threadNumber = new AtomicInteger();
```

Then for each thread we instantiate a final int tNumber;

```
Thread[] threads = new Thread[nrOfThreads];
for (int i = 0; i < nrOfThreads; i++)
    threads[i] = new Thread(() ->
    {
        Random random = new Random();
        try
        {
            final int tNumber = threadNumber.getAndIncrement();
            .........
        }
```

And then we add it as the value in the form (k, “t:k”) like this;

```
if (map.put(putKey,tNumber + ":" + putKey) == null)
   putSum.addAndGet(putKey);
if (map.putIfAbsent(putIfAbsentKey,tNumber + ":" + putIfAbsentKey) == null)
   putSum.addAndGet(putIfAbsentKey);
```

Exercise 8.1.5

To keep track of the counts we used an atomic integer array;

```
AtomicIntegerArray counts = new AtomicIntegerArray(nrOfThreads);
```

Then within each thread we made sure to both increment the appropriate counts each time something was added to the map whilst also making sure to decrement the correct counts when removing values;

```
int putKey = random.nextInt(99);
int putIfAbsentKey = random.nextInt(99);
int removeKey = random.nextInt(99);

String replacedVal = map.put(putKey, tNumber + ":" + putKey);
if (replacedVal == null) {
   putSum.addAndGet(putKey);
   counts.incrementAndGet(tNumber);
} else {
   counts.decrementAndGet(Integer.parseInt(replacedVal.split(":")[0]));
   counts.incrementAndGet(tNumber);
}

if (map.putIfAbsent(putIfAbsentKey,tNumber + ":" + putIfAbsentKey) == null)
{
   putSum.addAndGet(putIfAbsentKey);
   counts.incrementAndGet(tNumber);
}

String removedVal = map.remove(removeKey);
if (removedVal != null) {
   putSum.addAndGet(-1 * removeKey);
   counts.decrementAndGet(Integer.parseInt(removedVal.split(":")[0]));
}
```

Then for the assertion, we instantiated a countsFromMap array and traversed the map, adding each entry to the array hoping we would end up with an identical array to counts;

```
AtomicInteger mapSum = new AtomicInteger();
AtomicIntegerArray countsFromMap = new AtomicIntegerArray(nrOfThreads);
map.forEach((k, v) ->
{
   mapSum.addAndGet(k);
   countsFromMap.incrementAndGet(Integer.parseInt(v.split(":")[0]));
});
```

And finally, we used a for loop to assert that each value within both arrays were the same;

```
for (int i = 0; i < nrOfThreads; i++)
{
   assert countsFromMap.get(i) == counts.get(i);
}
```

Exercise 8.2.1

We tried a couple of things; first we removed synchronization from both of the put methods, which made the test fail on every run because the sums didn't match up. Then we tried it for the lock and then method, which resulted in the same test failure and finally we did it for remove which resulted in the test failing on about 50% of the runs.

Exercise 8.2.2

We tried changing it to synchronize on "this" and on 0, and managed to make the test fail. Basically we ran it within a while loop (slept for 1 second between runs) and just waited until it failed, which was in about 15-20% of the runs.

Exercise 8.2.3

We changed the sizes to an int array, and change how we are using it;

```
if (sizes.get(stripe) == 0) continue; //Old.
if (sizes[stripe] == 0) continue; //New.

sizes.decrementAndGet(stripe);  //Old.
sizes[stripe]--;  //New.

afterSize = sizes.addAndGet(stripe, delta); //Old.
afterSize = sizes[stripe] += delta;  //New.
afterSize = sizes.addAndGet(stripe, newNode == node ? 1 : 0);  //Old.
afterSize = sizes[stripe] = newNode == node ? 1 : 0;  //New.
```

After the changes we ran the test again couple of times and it always failed.

Exercise 8.2.4

We experimented removing reads from the sizes array from couple of places;

- **Foreach**: This did not result in a failure, which makes sense because the check here should only improve the performance.
- **Size**: Here we removed the sizes.get and just added a bogus value;

```
totalSize.addAndGet(7);
```

  This unfortunately always succeeded, but then we realized that made sense since we aren't really checking this property in our tests and no other method of the map uses it.
- **ContainsKey**: We removed the "Sizes.get(stripe) != 0" and still the test succeeded, we think that is because this is a check that would increase performance.
- **Get**: Removing the "Sizes.get(stripe) != 0" made no difference, again we think it is because it is just there to safe time so we don't check a stripe that has a size 0.

Exercise 8.2.5

To test the size() method we added an entry count, counted each entry and asserted that it was the same as the map.size() on it;

```
AtomicInteger mapSum = new AtomicInteger();
AtomicInteger entryCount = new AtomicInteger(); //New variable
AtomicIntegerArray countsFromMap = new AtomicIntegerArray(nrOfThreads);
map.forEach((k, v) ->
{
   mapSum.addAndGet(k);
   countsFromMap.incrementAndGet(Integer.parseInt(v.split(":")[0]));
   entryCount.incrementAndGet(); //Using new variable
});
assert entryCount.get() == map.size();
```

Now when we change the "size()" method the test fails. But still we are only asserting this after all threads have finished so it doesn't tell us much about the visibility of the sizes. Bad visibility would affect the performance so testing for that would be nice.

We tried some other things, we removed volatile from Buckets but still it passed. Also in our methods were we instantiate a new variable to hold on to the buckets we tried to access buckets directly, ended up changing it in all possible places but it always passed.