Exercise 7.1

Question 1

We have implemented `V get(K k)` similarly to SynchronizedMap where we only lock on the relevant stripe.

Question 2

We iterate through each stripe to get its count. We lock the specific stripe when adding its count to the total count. We do this to ensure visibility.

Question 3

This is implemented in the same way as `put` just without replacing any previous values.

Question 4

This is implemented as in `SynchronizedMap`, but where we only lock the relevant stripe.

Question 5

We have implemented the `forEach` by first making a local reference to the array stored in `buckets` to avoid iterating over the same `ItemNode` several times if `reallocateBuckets` is called simultaneously. Then we iterate through the stripes and lock each of them and iterate its buckets. This will acquire each lock only once.

Question 6

By using the tests we discovered an error in our `remove` method where instead of picking the first `ItemNode` in the chain and iterating, we just picked the `ItemNode` we wanted to remove and thereby we were not able to change the references to exclude it.

Question 7

Our performance measurements are documented in the 7_1_7.txt and 7_1_7.png in the 'benchmarks' folder. The graph is also available at: http://goo.gl/ogw5Un.

This results is partially as expected. The `SynchronizedMap` is fastest when using just a single thread. When using more than one thread, it is twice as slow no matter the amount of threads available. This was surprising to us, and we do not have a reasonable idea of why this happens. Our best guess is that using the synchronized methods introduces an extra overhead, when using more than one thread, increasing the execution time.

The two maps are equally fast when using a single thread, but after that `StripedMap` is remarkable

faster. The execution time decreases for the first 6-8 threads (on an 8 core machine), but after that the performance slows down as more threads are fighting to acquire the same locks and more threads are scheduled on the same core. This is to be expected.

Question 8

If we had a lock for each bucket, we would need to create new locks, as the number of buckets increased. The current resizing ensures that a key is always associated with the same stripe, and thereby the same lock. As the `size` array contains the count for each stripe/lock, we would have to recalculate the count for each bucket when resizing. This would take an unreasonably long time.

Question 9

We want to lower the probability of a lock being unavailable. We need at least as many locks as threads, so each thread ideally could hold a lock simultaneously. Having more locks increases the chance of a lock being available when a thread tries to take it.

Question 10

Having the number of buckets a multiple of the number of stripes ensures that each bucket always will be within the same stripe. ???? ???? ???? ???? ???? //Not true-> If not, there is a risk that an interviening call to `reallocateBuckets` could allocate the needed entry to a different stripe which could make the thread lose the lock for the specific entry leading to the risk of losing updates to the entry.

Exercise 7.2

Question 1

We implemented the size method, as described in the assignment text, by iterating through the `sizes` array and summing up its sizes. This is done without the use of locks as the array is an AtomicIntegerArray.

Question 2

If nothing was added in the call to `putIfAbsent` we don't need to write to the stripe `sizes` array in order to ensure visibility.

Question 3

We have implemented the `remove` method just as described in the assignment text.

Question 4

In our implementation of `forEach` we do stripe-wise locking. To ensure proper visibility we read the stripe size before iterating the buckets in the stripe. This ensures that we get the latest updates of the specific stripe. If updates are made after the size is read, we won't be able to see them. This is not a problem since the `forEach` only will guarantee a rolling snapshot.

Question 5

The result of our performance measurement is placed in the file '7_2_5.txt' in the 'benchmark' folder.

The results of our measurements are as expected. `SynchonizedMap` is by far the slowest with `StripedMap` being a good improvement. The non-locking reads in the `StripedWriteMap` gives us an extra performance boost, which is similar to, but not as fast as, the `WrapConcHashMap`.

Question 6

Exercise 8.3

Question 1

We chose to implement method 1 described in the assignment. We create N DownloadWorker instances each downloading a singe webpage. All the DownloadWorkers are executed in parallel. Their result is appended to the textarea through the done method invoked on the event dispatch thread when the task is completed.

Question 2

For cancellation we check before starting the download whether or not the SwingWorker subclass has already been cancelled. If the cancel button is pressed, an ActionListener on the cancel button calls the cancel method on each worker which throws an InterruptedException in the get method.

Question 3

We have made a shared AtomicInteger as a counter keeping track on the running tasks. This is decremented every time a DownloadWorkers `done` method is called and afterwards the progress bar is updated using the `setProgress` method. We do not need the atomicity of the counter, since all the calls on it are made by the event thread. It is just to allow all the workers to work of the same counter.