Exercise 6

Hardware used for all performance measurements;
# OS: Windows 10; 10.0; amd64
# JVM: Oracle Corporation; 1.8.0_181
# CPU: Intel64 Family 6 Model 60 Stepping 3, GenuineIntel; 8 "cores"
# Date: 2018-10-11T15:09:54+0200
# RAM: 8GB

Exercise 6.1.1

```java
public V get(K k) {
   final int h = getHash(k), stripe = h % lockCount;
   synchronized (locks[stripe]) {
       final int hash = h % buckets.length;
       ItemNode node = ItemNode.search(buckets[hash], k);
       return (node != null) ? (V) node.v : null;
   }
}
```

Exercise 6.1.2

```java
public int size() {
   int sum = 0;
   for (int i = 0; i < lockCount; i++)
       synchronized (locks[i]) {sum += sizes[i]; }
   return sum;
}
```

Locking the stripe makes sure that we can get the most recent value of the stripe size (since we are using the same lock) and that it will not be modified during each read.

Exercise 6.1.3

```java
public V putIfAbsent(K k, V v) {
   final int h = getHash(k), stripe = h % lockCount;
   synchronized (locks[stripe]) {
       final int hash = h % buckets.length;
       final ItemNode<K, V> node = ItemNode.search(buckets[hash], k);
```

```
        if (node != null) return node.v;
        buckets[hash] = new ItemNode<>(k, v, buckets[hash]);
        sizes[stripe]++;
        return null;
    }
}
```

Exercise 6.1.4

```
public V putIfAbsent(K k, V v) {
    final int h = getHash(k), stripe = h % lockCount;
    synchronized (locks[stripe]) {
        final int hash = h % buckets.length;
        final ItemNode<K, V> node = ItemNode.search(buckets[hash], k);
        if (node != null) return node.v;
        buckets[hash] = new ItemNode<>(k, v, buckets[hash]);
        sizes[stripe]++;
        if (sizes[stripe] > (buckets.length/lockCount)) reallocateBuckets();
        return null;
    }
}
```

Exercise 6.1.5

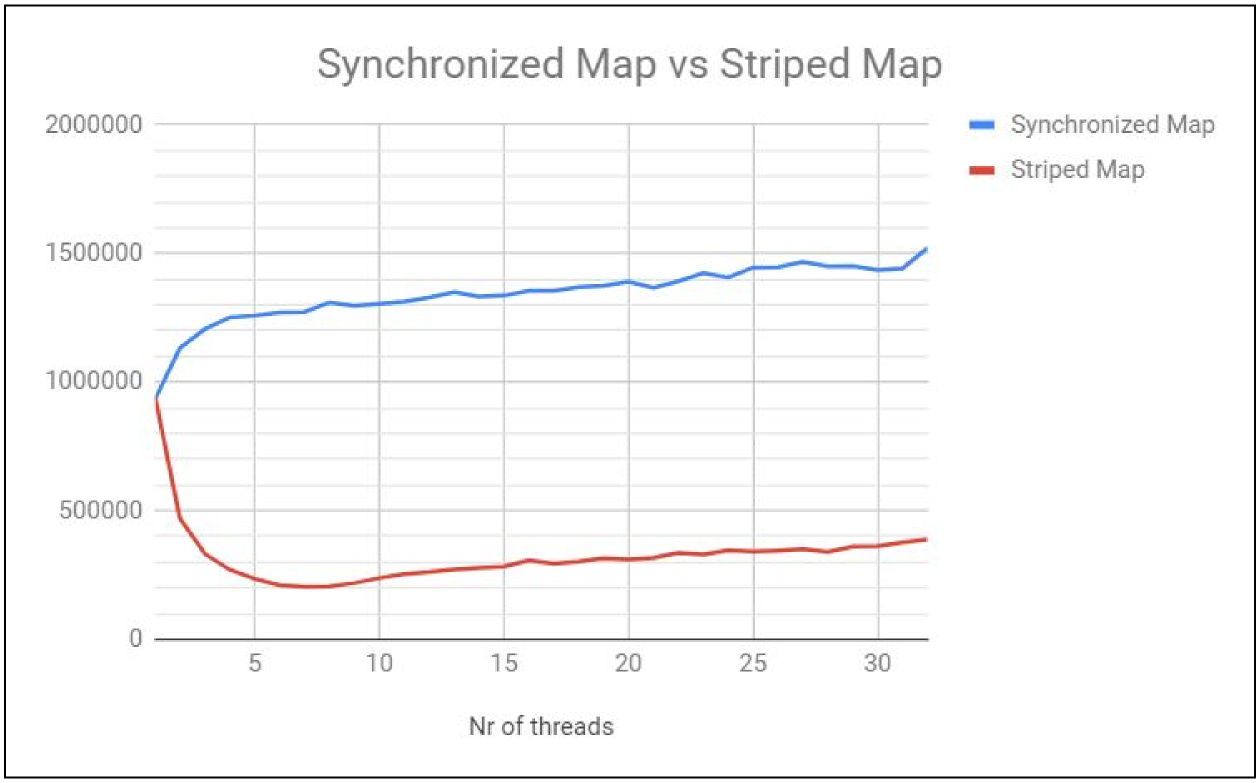
```
public V remove(K k) {
    final int h = getHash(k), stripe = h % lockCount;
    synchronized (locks[stripe]) {
        final int keyIndex = h % buckets.length;
        ItemNode<K, V> node = buckets[keyIndex];
        if (node.k == k)
        {
            V value = node.v;
            buckets[keyIndex] = node.next;
            sizes[stripe]--;
            return value;
        }
        while (node.next != null && node.next.k != k) node = node.next;
        if (node.next == null) return null;
        V value = node.next.v;
        node.next = node.next.next;
        sizes[stripe]--;
        return value;
    }
}
```

Exercise 6.1.6

```
public void forEach(Consumer<K, V> consumer)
{
    for(int stripe = 0; stripe < lockCount; stripe++)
    {
        synchronized (locks[stripe])
        {
            final int buckLength = buckets.length;
            for(int buck = stripe; buck < buckLength; buck+=lockCount)
            {
                ItemNode<K, V> node = buckets[buck];
                while (node != null)
                {
                    consumer.accept(node.k, node.v);
                    node = node.next;
                }
            }
        }
    }
}
```

We implemented version 2 because it would be bad for performance to lock the whole collection whilst doing the foreach. However, we do realize that now, items can be added to other stripes of the collection whilst iterating it meaning some would be included in the foreach and some not.

Exercise 6.1.8



The striped map performed better because we aren't locking the entire map like with the synchronized map, but only the relevant parts (the stripes).

Exercise 6.1.9

What advantages are there to using a small number (say 32 or 16) of stripes instead of simply having a stripe for each entry in the buckets table?

Using stripes means we are locking only part of the map instead of the whole map. If we have 2 stripes and 10 buckets, then then each stripe will be associated with 10/2 = 5 buckets. So when one stripe is locked (5 buckets) the other stripe can be written to which improves performance.

Exercise 6.1.10

Why can using 32 stripes improve performance even if one never runs more than, say, 16 threads?

It makes it less likely that two of the 16 threads are working on the same stripe if the stripes are more than the threads.

Exercise 6.1.11

A comment in the example source code says that it is important for thread-safety of StripedMap and StripedWriteMap that the number of buckets is a multiple of the number of stripes. Give a scenario that demonstrates the lack of thread-safety when the number of buckets is not a multiple of the number of stripes. For instance, use 3 buckets and 2 stripes, and consider two concurrent calls put(k1,v1) and put(k2,v2) where the hash code of k1 is 5 and the hashcode of k2 is 8.

If nr of buckets is not a multiple of number of locks then there will be some buckets that are locked using the wrong stripe. In the provided example, both writes use different stripes but try to write to the same bucket.

Exercise 6.2.1

```java
public V get(K k) {
   final ItemNode<K, V>[] bs = buckets;
   final int h = getHash(k), stripe = h % lockCount, hash = h % bs.length;
   Holder<V> holder = new Holder();
   if (sizes.get(stripe) != 0) ItemNode.search(bs[hash], k, holder);
   return holder.get();
}
```

Exercise 6.2.2

```java
public int size() {
   AtomicInteger totalSize = new AtomicInteger();
   for (int i = 0; i < lockCount; i++)
       totalSize.addAndGet(sizes.get(i));
   return totalSize.get();
}
```

Exercise 6.2.3

```java
public V putIfAbsent(K k, V v) {
   final int h = getHash(k), stripe = h % lockCount;
   ItemNode<K, V>[] bl;
   Holder<V> holder = new Holder<>();
   int afterSize, delta = 0;
   synchronized (locks[stripe]) {
       bl = buckets;
       int index = h % bl.length;
       if (!ItemNode.search(buckets[index], k, holder)) {
           bl[index] = new ItemNode<>(k, v, bl[index]);
           delta++;
       }
       afterSize = sizes.addAndGet(stripe, delta);
   }
   if (afterSize * lockCount > buckets.length) reallocateBuckets(bl);
   return holder.get();
}
```

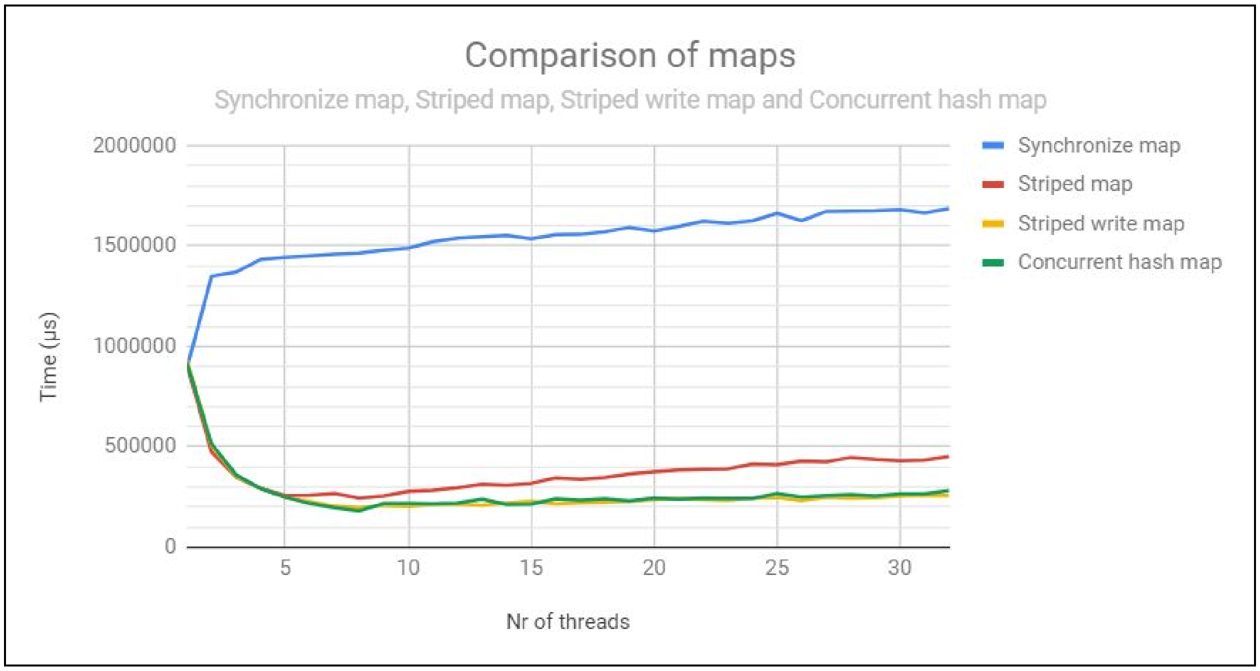
Exercise 6.2.4

```
public V remove(K k) {
    final int h = getHash(k), stripe = h % lockCount;
    ItemNode<K, V>[] bl;
    Holder<V> oldValue = new Holder<>();
    synchronized (locks[stripe]) {
        bl = buckets;
        int index = h % bl.length;
        bl[index] = ItemNode.delete(bl[index], k, oldValue);
    }
    if (oldValue.get() != null) sizes.decrementAndGet(stripe);
    return oldValue.get();
}
```

Exercise 6.2.5

```
public void forEach(Consumer<K, V> consumer) {
    for (int stripe = 0; stripe < lockCount; stripe++) {
        if (sizes.get(stripe) == 0) continue;
        ItemNode<K, V>[] bl = buckets;
        final int buckLength = bl.length;
        for (int buck = stripe; buck < buckLength; buck += lockCount) {
            ItemNode<K, V> node = buckets[buck];
            while (node != null) {
                consumer.accept(node.k,node.v);
                node = node.next;
            }
        }
    }
}
```

Exercise 6.2.6



Exercise 6.3.1

```
current thread hashCode              0.0 us       0.00  134217728
ThreadLocalRandom                    0.0 us       0.00  134217728
AtomicLong                      766182.5 us    3083.42          2
LongAdder                        52420.5 us     933.09          8
LongCounter                     404798.2 us  306382.61          2
NewLongAdder                    373223.4 us   23678.37          2
NewLongAdderPadded               96045.3 us    6649.17          4
```

Exercise 6.3.2

```
current thread hashCode              0.0 us       0.00  134217728
ThreadLocalRandom                    0.0 us       0.00   67108864
AtomicLong                      761387.9 us    8530.06          2
LongAdder                        52085.8 us     709.82          8
LongCounter                     418144.9 us  336197.11          2
NewLongAdder                    354122.3 us   24146.36          2
NewLongAdderPadded               94853.1 us    8083.29          4
NewLongAdderLessPadded          168840.2 us   17046.72          2
```

Very interesting indeed. Removing the object creations made the new adder much slower.
After researching what “false-sharing is” we came up with this answer.

Each core has a cache, and it can happen that a core will invalidate the caches from other cores. This can happen when a value is written to main memory; the cache of the specific core (associated with the value to be written) will be used and if any other cores have a copy of this “cache line” these caches will be invalidated which will force the other cores to read from main memory instead of their caches/cache lines.

Information found from this article;
http://simplygenius.net/Article/FalseSharing