Kotlin Maps – Part 2

COMP40009 – Computing Practical 1

26th February – 1st March 2024

Aims

- To gain experience writing Java code by implementing a fundamental data structure.
- To gain more experience with concurrency by making existing Kotlin data structures thread-safe.

Introduction

During the first part of this two-week "Maps" exercise, you created two implementations of a `CustomMutableMap` interface:

- `ListBasedMap`, which represented a map as a linked list of key-value pairs;
- `HashMapBackedByLists`, which represented a map as an array of buckets, where each bucket was a `ListBasedMap`, and where hashing was used to associate a key with one of the buckets.

Your `HashMapBackedByLists` should have been based on a `GenericHashMap` abstract class that takes care of the details of the hashmap data structure, but is parameterised by the kinds of buckets that it works with.

There are two orthogonal strands to this week's exercise.

Strand 1: implementing a tree-based map using Java. In this strand, you will create a class, `TreeBasedMap`, that implements the `CustomMutableMap` interface using a binary search tree representation. This class can serve as a map in its own right. Alternatively, a hashmap can use `TreeBasedMap` as its bucket type; you will create another subclass of `GenericHashMap`, called `HashMapBackedByTrees`, that follows this design.

Your task for strand 1 will be to implement `TreeBasedMap` and `HashMapBackedByTrees` (and any supporting classes) using Java. Because Kotlin and Java have been designed to interoperate in a pretty seamless manner, you will find that the Java code you write can integrate directly into the infrastructure that you have already created for writing map implementations. In particular, you can write a Java class that implements a Kotlin interface, and a Java class that extends a Kotlin class.

Strand 2: making list-based maps and hashmaps thread-safe. In this orthogonal strand, you will use locks to design *thread-safe* versions of `ListBasedMap` and `GenericHashMap`. The aim is to make these implementations work correctly in an environment where multiple threads are accessing the map concurrently. You will start by implementing basic thread-safe maps that

use a "big global lock" (one lock protecting the entire map structure). These implementations are pleasingly simple to get right, but are not very efficient. You will then implement a thread-safe hashmap that uses a finer-grained locking strategy.

The work in strand 2 should be implemented using Kotlin.

The strands are independent. You can work on them in either order.

Getting Started

As per the previous exercises, use `git` to clone the repository with the skeleton files for this exercise, via

```
git clone git@gitlab.doc.ic.ac.uk:lab2324_spring/kotlinmapspart2_username.git
```

As this is part 2 of a two-part exercise, you should start by adding your code from part 1 to the repository. Add all of the source files that you created, and any new test files that you created. You are provided with a new and slightly modified version of `CustomMutableMapTest`, so do not import this class or its subclasses. (If you added new tests to your version of this class during part 1 then you should import those tests individually to the new version of the class.)

Strand 1: tree-based maps in Java

You should attempt this part of the exercise in Java, and you are encouraged to write Java code directly. However, if you are having trouble getting to grips with Java you might prefer to initially get a solution working written in Kotlin and then port this solution to Java. It is expected that you will need to use more features of Java to implement this part of the exercise compared with the example set of features that were covered in the lectures. You should therefore do additional online research into the Java language as needed when implementing your solution. This is good practice for future software development work that you are likely to do during your studies and career, where you will need to work with a language that you are not yet all that familiar with. Over time you will find that, as a skilled programmer in general, it becomes easier and easier to pick up new programming languages.

The Java code that you write for this strand of the exercise should be placed under `src/main/java/maps`. Any additional tests that you write can be implemented in Kotlin and placed under `src/test/kotlin/maps` (because Kotlin code is able to call Java code directly). However, if you would like to write tests for your Java code using Java you are welcome to do so and should then place the tests under `src/test/java/maps`.

This part of the exercise requires you to build on last week's code by implementing a key-value map using a binary tree. Each node of the tree should contain a key and a value and the tree should be ordered based on the key.

The map should contain at most one item with a given key (there may be duplicate values, but all keys will be unique).

Tree nodes

Your first job is to define a class for implementing the *nodes* in a binary tree of key-value pairs (with key and value types `K` and `V` respectively). Hence, write a class `TreeMapNode<K, V>` that contains a key and a value and two subtrees, each of type `TreeMapNode<K, V>`. Java requires

constructors to be defined explicitly, so define a constructor `TreeMapNode(K key, V value)` that builds a tree map node with the specified key and value, and two empty subtrees.

Unlike Kotlin, Java does not have the notion of nullable types, so the children of a `TreeMapNode<K, V>` should simply have type `TreeMapNode<K, V>`, which means that they can either be null or can refer to `TreeMapNode<K, V>` objects. However, JetBrains provide a set of annotations for Java that allow nullability information to be provided in Java code. Feel free to investigate and use these annotations if you are interested.[1]

The following getter/setter methods should be members of the class; their definitions should be obvious:

- `void setLeft(TreeMapNode<K, V> left)`
- `void setRight(TreeMapNode<K, V> right)`
- `TreeMapNode<K, V> getLeft()`
- `TreeMapNode<K, V> getRight()`
- `K getKey()`
- `V getValue()`
- `void setKey(K key)`
- `void setValue(V value)`

Note that these methods, and the constructor, will be "package-private", as defined, because this is the default visibility level in Java. The "package-private" visibility level means that a declaration is visible anywhere within its package, but not outside its package. Kotlin does not have an equivalent visibility level.

The `TreeMapNode<K, V>` class is so straightforward that you may decide not to write tests for it directly, but rather to rely on it being tested via tests for the tree-based map class that you will build based on it. However, feel free to write test cases for this class if you wish.

Tree-based map

Now define a class, `TreeBasedMap<K, V>`, for representing an *ordered* binary tree of key-value pairs, with the ordering based on the keys.

An ordered binary tree is often referred to as a binary *search* tree. In general, a binary tree is a tree of nodes, where each node stores a value and has (possibly empty) left and right subtrees. A binary search tree is a binary tree for which the following property holds for every node n in the tree: the values stored in all nodes in n's left subtree are less than or equal to the value stored in the node; the values stored in all nodes in n's right subtree are greater than or equal to the value stored in the node.

In the context of a tree-based map, the values are key-value pairs, the ordering is based on the key, and duplicate keys are not allowed. Thus, an *in-order* traversal of the tree, where the nodes of a left subtree are visited first, followed by the root node, and then the nodes of the right subtree, will visit the nodes in ascending key order.

[1] `https://www.jetbrains.com/help/idea/annotating-source-code.html#jetbrains-annotations`

Because binary search trees are ordered, you need to capture the ordering when you define the tree. In this exercise you should do this by equipping a `TreeBasedMap` with a comparator field, provided in the constructor of the map.

As a reminder, the `Comparator<T>` interface has a method `int compare(T o1, T o2)` that defines the ordering on the element type `T`. The ordering here should be based on the key within a given node.

In addition to the comparator, a field of the class should also represent the *root* node of the tree (initially `null` to indicate that the tree is empty).

The class definition should look like this:

```
public class TreeBasedMap<K, V> implements CustomMutableMap<K, V> {...}
```

and the constructor like this:

```
public TreeBasedMap(Comparator<K> keyComparator) {...}
```

Tree-based maps

You will need to provide definitions of the following methods in the `CustomMutableMap` interface:

- `public Iterable<Entry<K, V>> getEntries()` – return the entries in the tree (see below).
- `public Iterable<K> getKeys()` – return the keys associated with the items in the tree.
- `public Iterable<V> getValues()` – return the values associated with the items in the tree.
- `public V get(K key)` – return the value associated with the given key, or null if there is no entry with that key.
- `public V put(K key, V value)` – add the key-value pair to the map and return the previous value associated with the key, or null if there was none.
- `public V remove(K key)` – remove the entry with the given key and return the previous value associated with the key, if there was one. If there was no entry with that key, return null.

The remaining member functions of `CustomMutableMap` also need to be implemented, even if you have provided them as default methods of `CustomMutableMap`. This is because at present, without using special compiler options, a Java class cannot rely on default methods defined in a Kotlin interface.

Traversing a binary search tree

The `get`, `put` and `remove` methods will need to navigate through the tree to locate entries or insertion points based on a given key. At each node you move left if the key is smaller than the key within the node and right otherwise. This is where the tree's `Comparator` should be used.

For `get` you can simply iterate through the tree until you find the matching key, if there is one. If you reach null, the key wasn't present in the tree, so you return null.

For `put` you need to modify the existing tree by adding a new leaf node (a node with two empty subtrees). This means changing either the left or right subtree of one of the nodes already in

the tree. The tricky bit is to ensure that you return the right value (`V`), i.e. null if the value wasn't present in the map and the previous value associated with the given key if it was.

You can implement this in any way you wish, but may find that the easiest approach to iterate through the tree using `current` and `parent` node references, checking whether the subtree you are about to move into is null and then using `setLeft/setRight` to effect the change. You will also need an additional check at the beginning to determine whether the tree root is null.

For `remove` you first need to locate the entry to be removed (if there is one) and the suggestion is that you use a similar approach to that for `insert`, i.e. using current and parent references to iterate through the tree. If you reach null the key wasn't present, so you return null. If the `compare` method returns 0 (a key match) then you need to remove the entry at the current node, so it's a good idea to store the associated value, as you'll need to return it later.

Because it's very easy to miss an edge case you might find it useful at this point to define a helper method `deleteNode`, say, which takes the current node and its parent and which has the effect of deleting the current node from the tree by connecting the parent to one the current node's subtrees.

It's important to establish the conditions under which `deleteNode` will be called: the current node will be non-null but at least one of its subtrees will be null (it will be clear why when you read on). If the parent is non-null then there are four possible cases: the current node could be either left or right of its parent and either of its left or right subtrees will be null. In each case it should be clear how to connect the parent to the subtree 'opposite' the one that's null using the parent's `setLeft/setRight` methods. Note that if both subtrees happen to be null, that's OK, as the parent will link to a null subtree, as required. It's not all over yet(!): if the parent is null, then the current node will be at the top of the tree, so you need to change the tree's root to one of the current node's subtrees accordingly; as above, there are two cases, as at least one of those subtrees will be null.

If you've defined `deleteNode` as described, it should be fairly easy to complete `remove`: if either the left or right subtree of the current node is null, call `deleteNode` to remove the current node (if they're both null then `deleteNode` will still do the right thing). Otherwise the entry in the current node will need to be replaced with either the largest entry in the left subtree, or the smallest entry in the right subtree; that entry will then need to be deleted. You could choose one of the two options at random, or base it on the sizes of the two subtrees, or you could implement a more elaborate structure that ensure that the tree is always balanced. To keep it simple, the suggestion is that you always remove the smallest entry from the right subtree for now; this may result in slightly "lopsided" trees with generally fewer entries to the right of each node than the left, but that's fine for the purposes of the exercise. You can explore other alternatives as an extension (see below).

Thus, having remembered the current node (the one whose entry needs to be overwritten), use the parent and current references, or use a helper method, to walk the 'left spine' of the right subtree until the current node's left subtree is null. At this point the current node will contain the smallest entry in the right subtree, so you can use `setKey` and `setValue` to overwrite the node containing the key that you previously remembered. The final job is to delete the current node (`deleteNode`) and you're done. To wrap up, return the value previously associated with the key being removed.

Defining the iterators

To implement the `getEntries()` method, define an `EntriesIterator` inner class that implements Java's `Iterator` class. To do this you'll need to define the methods `hasNext()` and `next()`. The class definition should look like this:

```
private class EntriesIterator implements Iterator<Entry<K, V>> {...}
```

Note that despite being an *inner* class, we do not use the `inner` keyword when defining `EntriesIterator`. This is not a keyword in Java, and a class declared inside another class is an inner class in Java by default.[2]

To support an in-order traversal of the tree you should use a *stack* to keep track of which node to choose next, having visited a node – you can use the `LinkedList` class from `java.util` to save defining your own. This implements the `Dequeue` (double-ended queue) interface, which contains the familiar stack manipulation methods `isEmpty`, `push` and `pop`.[3] Because it is good practice to program against an interface, not an implementation of that interface, you should use `Deque` as the *apparent type* of your stack, and only use `LinkedList` when constructor of the stack, to specify its *actual type*.

To use the stack, you "keep left" as you walk the tree, pushing nodes onto the stack as you go, until you hit null. The item returned by the iterator's `next()` method is then an `Entry` whose key and value are obtained from the node `pop`ped from the top of the stack. Having popped the stack, the next node to visit (if there is one) is the one at the top of the right subtree of the popped node. Note that if this is null, you'll end up popping the stack again and the process repeats.

To implement the `getKeys()` and `getValues()` iterators, it is recommended that you define similar inner classes that implement `Iterator`. But rather than duplicating logic for iteration, each of these classes can be constructed with a reference to an `EntriesIterator`. It can then provide `next()` and `hasNext()` by delegating to the `EntriesIterator` object, projecting either the key or value of the entry as appropriate in the case of `next()`.

Testing your tree-based maps solution

Use the `TreeBasedMapTest` class to test your solution. This is a subclass of `CustomMutableMapTest`, which provides a general-purpose test suite for map implementations. Running all tests in the `TreeBasedMapTest` class will cause these general purpose tests to be executed against your tree-based map implementation.

Writing and testing a hashmap backed by tree-based maps

In part 1 of this exercise you implemented a class called `HashMapBackedByLists` that used list-based maps to represent the buckets of a hashmap.

Follow a similar approach and implement a class called `HashMapBackedByLists` that uses tree-based maps to represent the buckets of a hashmap. This should be a very simple class and you can implement it either in Kotlin or in Java, as you prefer.

[2] To achieve in Java what is called a *nested* class in Kotlin (which is what you get if you nest one class inside another without using the `inner` keyword), you need to declare the class as `static`.

[3] Note that a stack results in the tree being traversed in depth-first order; if you ever needed to traverse it in breadth-first order, you would use a queue.

Use the `HashMapBackedByTreesTest` class to test that this class works as expected.

Strand 2: thread-safe maps in Kotlin

To give you more experience working with lock-based concurrency, this strand involves writing versions of list-based maps and hashmaps that are made *thread-safe* through the use of locks.

A collection is said to be *thread-safe* if the collection can be used by multiple threads simultaneously in a manner that does not lead to the state of the collection being corrupted, and that leads to threads observing sensible results when interacting with the collection. For example, consider a linked list class. Without any synchronisation, it might be possible for two threads to attempt to remove an element from the head of the list simultaneously, and instead of this leading to two elements being removed, and each thread obtaining one of the elements, for just one element to be removed and each thread obtaining this element (as an exercise, think about how data races could lead to this occurring). As another example, suppose a hashmap has not been protected using synchronisation, two threads attempt to add an element to the hashmap concurrently, and both threads determine that the hashmap should be resized. Depending on how resizing is implemented this race condition could have negative consequences. It might lead to the hashmap being doubly-resized (if one resize completes before the other starts). Or it might lead to one of the added elements being lost, if one of the threads adds its element and gets on with its resize before the other thread has managed to add its element.

The general point is that for collections to be used in a multi-threaded environment they must be thread-safe, and one way to make collections thread-safe is through the use of locks.

Thread-safe maps using a "big global lock"

Your first class is to write an abstract class, `LockedMap<K, V>` that implements `CustomMutableMap<K, V>`. This class should be able to provide a thread-safe version of *any* `CustomMutableMap<K, V>` implementation via *delegation*. Specifically, a `LockedMap<K, V>` should have two properties: a lock, and a reference to a *target map* of type `CustomMutableMap<K, V>`. The methods of `CustomMutableMap` should be implemented in `LockedMap<K, V>` by delegating to the target—i.e., invoking the corresponding method of the target map. However, the delegation calls should all be protected by the lock, so that no two calls to the target map can proceed concurrently.

Care is needed when providing access to the entries, keys and values of a map via their associated properties. These properties have type `Iterable`, and it is recommended that you implement the properties in `LockedMap` by creating a collection that contains a copy of all the entries, keys or values of the target collection and then returning this copy. This is needed if the target map would provide a "lazy" `Iterable` object to give access to this data. Protecting the logic that implements the properties using a lock would only ensure that the statements required to return the iterator are lock-protected. Subsequent access to the data returned by calls to the `next()` method of the iterator that the `Iterable` returns would not, which could lead to data races on the target map.

This simple approach to synchronisation is called the "big global lock" approach. It is simple and easy to get right. However, it has the drawback that it does not admit much parallelism: it makes it impossible for two pieces of work to be performed on a collection in parallel. That is fine if the collection is not being used in scenario where performance matters. For example, an application might use a map to occasionally log information. In this case one would need to make the map thread-safe if multiple threads can use it for logging (to avoid the map becoming

corrupted due to data races), but because concurrent calls to the map are not expected to occur very often, contention for the lock that guards the map will not be high: in the majority of cases a thread who needs to use the map will obtain its lock straightaway. However, a "big global lock"-based approach may not perform well enough if a data structure is frequently accessed by multiple threads concurrently, as the threads will often end up being blocked waiting for the lock to become available.

You can now make two concrete subclasses of `LockedMap<K, V>`:

- `LockedListBasedMap<K, V>`, which instantiates its `LockedMap<K, V>` parent class using a `ListBasedMap<K, V>` instance, providing a thread-safe version of `ListBasedMap<K, V>`
- `LockedHashMapBackedByLists<K, V>`, which instantiates its `LockedMap<K, V>` parent class using a `HashMapBackedByLists<K, V>` instance, providing a thread-safe version of `HashMapBackedByLists<K, V>`

Test these implementations using the `LockedListBasedMapTest` and `LockedHashMapBackedByListsTest` classes. Similar to the test classes in Strand 1, and in part 1 of the exercise, these test classes subclass an abstract class that provides general tests for concurrent maps.

Because concurrency tests need to run multiple times, to guard against tests passing by chance (due to bug-inducing nondeterministic behaviour not occurring, even though it can occur), these tests give output indicating how many times each has executed. You will find that the `LockedListBasedMapTest` tests take a while to run, partly because mainly because all list-based map operations have linear time complexity.

The provided tests for thread-safe maps do not exercise all the methods of the `CustomMutableMap` interface. Use code coverage analysis to find out which methods are not exercised, and write additional tests that cover these methods. Try to think up tests that might fail if you do not use locks to synchronise concurrent accesses, and see whether your new tests can indeed fail if you temporarily remove locking from your `LockedMap` class.

A "striped" thread-safe hashmap

Your final task for this strand is to implement a so-called "striped" hashmap. This will involve writing a subclass of your `GenericHashMap` class and adding synchronisation-related code in the subclass (making minimal changes to the superclass as required).

Recall that a hashmap initially has some number of buckets, and that the number of buckets is increased each time the hashmap is resized. It is common to double the number of buckets, and for this strand that is how your `GenericHashMap` class should work—if you used a different strategy for resizing in part 1 of the exercise, then you should change your implementation so that doubling is used.

A striped hashmap is a hashmap that achieves thread-safety in the following way:

- The hashmap stores a list of locks, one for each bucket originally present in the hashmap. So, if the hashmap originally has 32 buckets, it should be equipped with a list of 32 locks. In what follows, call the original number of hashset buckets N.
- Initially, the lock at position i in the list protects bucket number i: any operation that will access bucket i should do so by first locking lock i, and on completion should release lock i.

- When a thread determines that the hashmap needs to be resized, it should acquire all N bucket locks in order. This is because resizing involves accessing all the buckets of the hashmap, and no bucket should be tampered with until exclusive access to all buckets has been ensured. Once a thread has required all of the bucket locks, it should check that the hashmap *still* needs to be resized: it could be that in the time it took for the thread to obtain all of the locks, another thread managed to acquire the locks, resize the hashmap and release the locks. Once the thread has finished resizing (or realised that it does not need to resize) it should release all of the locks.
- Resizing should double the number of buckets but should *not* double the number of locks. That is, even though the number of buckets will grow from N to $2\dot{N}$ to $4\dot{N}$, etc., the number of bucket locks will always be N. (A hashmap in which the locks are also resized is called a *refinable* hashmap, and is significantly more challenging to implement – see the extensions.)
- After each resize, every lock should protect twice the number of buckets that it previously protected: after the first resize, the lock at position i should protect bucket i and bucket $i+N$. After the second resize, this lock should protect bucket i, $i+N$, $i+2N$ and $i+3N$. In general, after some number of resizes, the lock at position i in the list of locks should protect buckets $i\dot{k}\dot{N}$, for all $1 \leq k \leq R+1$, where R is the number of resizes that have taken place.

The advantage of a striped hashset over a hashset protected by a "big global lock" is that a striped hashset allows for certain accesses to the hashset to be concurrent: accesses to buckets that are protected by different locks.

To put this into practice, write an abstract subclass `StripedGenericHashMap<K, V>` of `GenericHashMap<K, V>`. The intention is that this subclass should take care of details of the "striped" synchronisation method, but still leave abstract the details of how buckets are represented. These will be provided in further subclasses as explained below.

Equip `StripedGenericHashMap` with a property representing a list of locks, initialised to the initial number of buckets used by your `GenericHashMap` class. Ensure that the initial number of buckets is something reasonable, like 32, so that there is a reasonable chance that data items will be stored in buckets protected by different locks to allow for parallelism when concurrent accesses to buckets occur.

Override the various methods of `GenericHashMap` that access hashset buckets so that in the `StripedGenericHashMap` subclass each method acquires the appropriate bucket lock, calls the superclass method, and then releases the bucket lock. Use Kotlin's `withLock` feature to avoid the potential for a thread to fail to release a lock due to an exception being thrown or the thread returning before explicitly performing an `unlock` call.

Find a way to ensure that resizing is overridden in `StripedGenericHashMap`, so that locks are used to properly synchronise resizing as described above. Because resizing involves acquiring multiple locks you cannot use `withLock` to take care of lock management. Instead, make use of the `try...finally` pattern that was shown in the Java programming lectures as an alternative.

You should also override the properties that provide access to the entries, keys and values of a map, and ensure that (a) these properties acquire all of the bucket locks before attempting to access the map's buckets (this is important because they will need to access all buckets), and (b) the `Iterable` instances that these properties return will not have access to the buckets of the map (because such accesses would not be lock-protected). This is similar to the issue

discussed above when implementing these properties using the "big global lock" approach to synchronisation.

As an overall reference for your `StripedGenericMap` implementation, you may wish to refer to Java code for how to implement a striped hashset (which is similar to a striped hashmap) in Section 13.2.2 of "The Art of Multiprocessor Programming", which is available online.[4]

Once you have finished coding `StripedGenericHashMap`, it is time to create some concrete subclasses that provide thread-safe hashmaps that use the "striped" mechanism for synchronisation, but use concrete map types to represent buckets. In this regard, create the following subclasses of `StripedGenericHashMap`:

- `StripedHashMapBackedByLists`, which should use `ListBasedMap` as its bucket type.
- `StripedHashMapBackedByTrees`, which should use `TreeBasedMap` as its bucket type (come back to this if you have decided to attempt strand 2 before strand 1).

Test your solutions using the `StripedHashMapBackedByListsTest` and `StripedHashMapBackedByTreesTest` test classes.

Performance comparisons

The point of a striped hashmap is that it should provide better performance compared with a hashmap that uses a "big global lock" when accessed frequently by many threads concurrently. Write some example code to confirm that this is the case. Use the example code for measuring time in Kotlin that were demonstrated for the "parallel dot product" example in the concurrency part of the course.

Extensions

Porting more code to Java

To gain more experience writing Java, you could try rewriting more (or even all) of the code you have written for parts 1 and 2 of this exercise to Java. You may even want to try porting the tests.

Porting everything to Kotlin

To gain more experience writing Kotlin, you could try rewriting the tree-based map code that you implemented in Java in Kotlin. The IntelliJ IDE provides a plug-in for automatically converting Kotlin to Java. One idea would be to manually port the tree-based map code to Kotlin, and then compare the results that you get via manual porting to the results you get by using the IntelliJ plug-in for conversion. (To make this work you should give different names to the Kotlin classes that you manually port from Java, or put them in a different package, to avoid name collisions.)

Providing support for balanced binary trees

Modify the `BinarySearchTree` class so that the trees are *balanced*. There are various way that a tree can be considered "balanced": for each node, it could mean that the depths of the two

[4] `https://library-search.imperial.ac.uk/permalink/44IMP_INST/f7tnsv/alma991000618299901591`

subtrees differ by at most one, or that the number of elements in those subtrees differs by at most one.

You could investigate data structures such as *red-black* trees, or *AVL* trees. Alternatively, you could implement a more straightforward algorithm that uses *rotations* to move nodes around in order maintain the balance property when you insert/remove items. A left rotation moves the root of the right subtree upwards and pushes the original root downwards; a right rotation does the opposite. A rotation preserves the order in the tree but changes the depths of the two subtrees: one decreases in depth by 1 and the other increases in depth by 1. See for example, `https://en.wikipedia.org/wiki/Tree_rotation`.

Implementing a refinable hashmap

As noted above, implementing a *refinable* hashmap, where the number of bucket locks is resized in step with the number of buckets, is challenging. Section 13.2.3 of "The Art of Multiprocessor Programming" (*link in a footnote above*) shows how this can be achieved in Java for a hashset, which is similar to a hashmap. Study this section of the book and try to implement a refinable hashmap using Kotlin. Compare the performance of your refinable hashmap compared with your striped hashmap using high contention workloads to confirm that the refinable approach yields better performance.

Note: your repository containing the skeleton for this lab can be found at *https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinmapspart2_username.* ***As always, you should use LabTS to test and submit your code.***

Assessment

```
F - E: Very little to no attempt made.
       Submissions that fail to compile cannot score above an E.

D - C: Implementations of most functions attempted;
       solutions may not be correct, or may not have a good style.

B:     Implementations of all functions attempted, and solutions
       are mostly correct. Code style is generally good.

A:     There are no obvious deficiencies in the solution or
       the student's coding style. In addition, there is
       evidence of productive testing.

A*:    As for an A -- plus the student has done additional work
       beyond the basic spec, e.g. by considering (and clearly
       commenting) interesting variations or extensions to the
       given functions; e.g. based on their own research.
```