Kotlin Maps – Part 1

COMP40009 – Computing Practical 1

19th – 23rd February 2024

Aims

- To implement some fundamental data structures in Kotlin.
- To write programs that build and make use of mutable iterators over collections.
- To gain further experience working with classes, interfaces and inheritance.

Introduction

The objective of this exercise is to given you experience of implementing some of the core data structures that are ubiquitous in computer science. This is a two-week exercise and the ultimate objective is to design and evaluate your own implementations of *maps*, including both sequential and concurrent versions of list-based maps, hashmaps and tree maps, using a combination of Kotlin and Java code.

This week's part of the exercise (covered by this spec) focuses on the implementation of sequential list-based maps and hashmaps.

As you already know from working with Kotlin maps, a map is used to store key-value pairs.

A list-based map is a simple map implementation that represents key value pairs as a list.

A hashmap is a more efficient data structure that provides efficient mechanisms for inserting new key-value pairs and for looking up values based upon their key. Hashmaps are widely used because they can offer constant-time insertion, deletion and look-up functions in the average case, depending on the choice of underlying data structure (collection) and hashing function used.

Although there are extensive pre-defined libraries to support hash maps and the data structures upon which they depend, this exercise requires you to develop your own implementations from scratch.

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/kotlinmaps_username.git
```

Background

The `Iterable` and `MutableIterable` interfaces

This exercise makes use of Kotlin's `Iterable` and `MutableIterable` interfaces, which you should briefly study by looking at the Kotlin documentation regarding these interfaces.

The `Iterable` interface is used to describe any collection that can be iterated over. It is implemented directly or indirectly by all Kotlin collection classes: e.g. all classes that implement `List` and `Set` are `Iterable`s. The `Iterator` interface requires implementing classes to provide the `iterator()` operator, which provides access to elements of the underlying collection.

The `MutableIterable` interface extends `Iterable`, and has the effect of strengthening the requirement of the `iterator()` operator so that it must return a `MutableIterator`, rather than any old `Iterator`. The `MutableIterator` interface extends the `Iterator` interface with a method `remove()`, the effect of which is to remove the last element returned by the iterator's `next()` method from the underlying collection.

Hashmaps

A hashmap incorporates two fundamental data structures. The first is an array of *buckets* (see below), indexed by the *hash code* of a given key. The job of the hashing function is to map an object of some specified type to an integer array index: if the array has N elements then the hash function must generate an integer in the range $0, ..., N-1$.

Because two different keys may hash to the same array index it is necessary to store all items that hash to the same index – so called "collisions" – in a secondary data structure, commonly referred to as a *bucket*, containing a *map* of key-value pairs whose keys hash to the same index Insertions, deletions and look-ups all happen within the buckets. For example, when looking up a given key we first use the hash of the key to determine the correct bucket then search the bucket to find the the value associated with the exact key being looked up.

In the best case, if there are N key-value items to be stored and N buckets in the hashmap array then a "perfect" hashing function will allocate exactly one item to each bucket. In the worst case the hashing function will compute the same index for every item, in which case the hashmap degenerates to that of the structure of an individual bucket. Perfect hashing functions rarely exist, however, even when the items to be stored are known in advance. The hope is that, in practice, a reasonably well-behaved hash function will spread the key-value pairs sufficiently evenly among the buckets so that each contains a small number of items whose size is independent of N – hence the $O(1)$ average time complexity.

In order to improve performance the size of the hashmap array is set to be larger than the number of key-value pairs by using a so-called *load factor*. This is typically a number between 0.5 and 1.0 so, for example, a load factor of 0.75 will guarantee that a hashmap with 75 entries will have at least 100 buckets. If the number of entries exceeds the product of the number of buckets and the load factor, then the array is *re-sized*, e.g. by doubling its size, and re-inserting its entries into the re-sized hashmap.

In this first week you will develop a sequential (non-concurrent) hashmap where the buckets contain custom-built lists of key-value pairs.

Custom lists

Your first task is to build a custom implementation of linked lists. You should build this from scratch, i.e. do *not* delegate the task to Kotlin's existing `List<T>`, or similar!

The code you write for this implementation of linked lists will be similar to the code you have written in exercises during the course for a singly-linked list class, so you will be able to reuse that knowledge and experience (though the requirements and suggested design here are slightly different).

Write a class `CustomLinkedList<T>` that implements Kotlin's `MutableIterable<T>` interface. The `MutableIterable` interface extends the `Iterable` interface and has a function `iterator()` that returns a `MutableIterator`. The difference between a `MutableIterator` and an `Iterator` is that the former has a function `remove()` that removes the last element that was returned by the `next()` function from the underlying collection. The ability to remove arbitrary elements will be useful later on when you use lists to build maps (sets of key-value pairs).

Lists should be represented by chaining together individual list nodes. To support this:

- Define an interface `Node<T>` containing a single property (or function) that will return the next item in the list.
- Your lists should always contain a *root* node which links to zero or more other nodes containing the list items; the list is empty if the link in the root node is null. Thus, define two classes – instances of `Node<T>` – representing the root node and value nodes.

You can now implement the class `CustomLinkedList` as follows:

- Define the `root` node of the list – a `RootNode` with a null reference.
- Define the obvious functions for manipulating lists, such as `isEmpty`, `head` (or maybe call it `peek`) (inspect the head without removal), `add` and `remove`, with appropriate types. Note that `add` and `remove` should operate on the head end of the list. As above, you may choose to implement functions like `head` and `isEmpty` as `get` and `set` functions on `head` and `isEmpty` properties. Note that `head` and `remove` may return null, so their return types should be nullable.
- Finally, override the `iterator` function, which should return a `MutableIterator`. An attempt to call `next()` when there is no next element should throw a `NoSuchElementException` exception. Similarly, an attempt to remove the next item when there is none should throw a `UnsupportedOperationException` exception.

Testing your custom list code

To allow you flexibility in the custom list methods that you define, no tests for this part of the exercise are provided. You will use your custom lists below when implementing maps, and the tests for maps will provide a degree of indirect testing of your implementation of custom lists.

However, it is strongly recommended that you write a suite of unit tests for your custom list classes and check that they pass before you proceed to implement map data structures that use your lists. You risk getting into a tangle if you do not test your list implementations sufficiently thoroughly at this stage—you may find yourself debugging issues in your map data structures that stem from a combination of bugs in your list implementations *and* bugs in your map implementations that use these lists. Bugs are much easier to deal with one at at time!

List-based maps

Your next task is to write a map implementation that is backed by a list. This is a precursor to implementing a more efficient hashmap data structure: your list-based maps can be used as maps in their own right, and they will also form part of your hashmap implementation.

The `CustomMutableMap` interface

Have a look at the provided interface, `CustomMutableMap`. This interface describes the service that a map should provide. The list-based map class that you will write should implement this interface.

You should not change the service that this interface provides—i.e. you should not add or remove methods or properties, or change the signatures of methods or types of properties. However, you are free to provide default implementations for some of the methods and properties if you see fit.

Note that the interface provides `get` and `set` as operator functions. This allows maps to be used with array-like indexing, using the '`[]`' operator. However, it is more traditional to use the name `put` for the operation of assigning a value to a key in a maps, hence `put` is also specified as a method of the interface.

The `ListBasedMap` class

Write a class `ListBasedMap` that implements `CustomMutableMap`. The `ListBasedMap` class should use your `CustomLinkedList` class to build a key-value map.

Hints: There are various ways to define `remove()` but you may find that the easiest solution is to add a separate (overloaded) `remove` function to `CustomLinkedList`, or even as an extension function on `MutableIterable`s, that removes an item from an arbitrary position in the list based on a given predicate (type `(T) -> Boolean`). To define this you can use the `hasNext()`, `next()` and `remove` functions from the `MutableIterable`'s iterator. Note that if you add an entry to a map that already contains the same key then the corresponding value should be replaced by the new one; the simplest way to implement this is to `remove` any existing values associated with the key and then add the new one using `put`.

Testing your code

The provided test class `CustomMutableMapTest` provides a suite of tests for implementations of `CustomMutableMap`. Briefly study this class. Notice that it is an *abstract* class. Also notice that it does not refer to any particular `CustomMutableMap` implementation. Instead, it has an abstract function, `emptyMap()` that subclasses must implement. This function should provide an empty `CustomMutableMap` of the relevant implementation of `CustomMutableMap` that a subclass is designed to test. The tests in `CustomMutableMapTest` all start with an empty map, and then use the methods of the `CustomMutableMap` interface to manipulate this map, by adding and removing entries.

Now look at the `ListBasedMapTest` subclass of `CustomMutableMapTest`. Observe that all this does is override the `emptyMap()` method. At present, it uses Kotlin's `TODO` feature to have the overridden method throw a `NotImplementedError`. To instead facilitate testing of your `ListBasedMap` class, change this method so that it returns a `ListBasedMap` instance.

You should now be able to test your `ListBasedMap` class by running all tests in `ListBasedMapTest`. This will cause all of the tests in the `CustomMutableMapTest` superclass to be executed, and they will all operate on `ListBasedMap` instances thanks to the overridden `emptyMap()` method in `ListBasedMapTest` returning a `ListBasedMap`.

Debug your solution based on these tests, and assess how much code coverage the tests achieve on the classes that you have implemented so far. Write additional tests to improve the percentage of statements that are covered.

Aside: If you want, you can *narrow* the return type of `emptyMap()` in `ListBasedMapTest` to document that it returns a more specific type than `CustomMutableMap`—you can change it to `ListBasedMap`. When overriding a method, it is always acceptable to narrow the return type of the method to a subtype of its original return type.

Implementing hashmaps

Your task is now to build a hashmap whose buckets are unordered lists of key value pairs. However, you will design this hashmap so that it is easy to change the bucket data structure, e.g. to use an array (one of the extensions for this week's exercise), or an (ordered) tree (trees will feature in the second part of the exercise).

A generic hashmap abstract class

Because we would like to vary the type of bucket used in a hashmap it is convenient to define a *generic* hashmap that provides all the common functionality required without explicit reference to the concrete bucket type.

Hence, write an abstract class, `GenericHashMap` that implements `CustomMutableMap`. A generic hashmap should maintain an array of buckets, where every bucket is itself a `CustomMutableMap`. To make this work, we need to find some way to build a particular bucket type when constructing the hashmap array. You can implement this any way you wish, but a neat trick is to provide the `GenericHashMap` constructor with a function that, when invoked, will build a bucket of the required type. Functions that build data structures are sometime referred to as *factory functions*, so a type alias something like this might be a good way to get started:

```
typealias BucketFactory<K, V> = () -> CustomMutableMap<K, V>

abstract class GenericHashMap<K, V>(private val bucketFactory: BucketFactory<K, V>) :
  CustomMutableMap<K, V> {
  ...
}
```

Notice the types: a `GenericHashMap` is a `CustomMutableMap`, as are the buckets from which it is made. Because a `GenericHashMap` is a `CustomMutableMap`, you will need to provide definitions for all of the (non-default) interface properties and methods.

First, use Kotlin's `TODO` feature to implement stubs for these methods that throw a `NotImplementedError`.

A `HashMapBackedByLists` concrete class

Your generic hashmap abstract class is not yet fully-featured—many of the implementations of interface methods simply throw exceptions. Still, you can go ahead and write a concrete

subclass of `GenericHashMap` that represents buckets using your `ListBasedMap` class. You can then use this concrete hashmap class to test your solution as you implement the features of `GenericHashMap`.

Write a subclass of `GenericHashMap` called `HashMapBackedByLists`. This class should "complete" the `GenericHashMap` implementation by providing a `ListBasedMap` object whenever a new bucket is required. If you do this correctly, the `HashMapBackedByLists` class will be extremely short!

Confirming that the tests compile, but fail

Now that you have a `HashMapBackedByLists` class, you can adapt the `HashMapBackedByListsTest` test class so that its `emptyMap()` method returns a `HashMapBackedByLists()` object.

If you now run all tests in the `HashMapBackedByListsTest` class you should find that compilation succeeds, but that all tests fail due to the `TODO`s in `GenericHashMap`. You are now ready to tick off these `TODO`s!

Implementing the functionality of `GenericHashMap`

You are now ready to provide implementations in `GenericHashMap` of the various `CustomMutableMap` interface methods, such as `get`, `put` and `remove`. You will also need to provide access to the `entries` of the hashmap which involves combining the elements in each bucket (a `flatMap` perhaps?).

Indexing the hashmap

Given a search `key` the bucket index is found by using the `hashCode()` of `key`; every object in Kotlin inherits this function by virtue of it being defined in the `Any` class. A has code is a 32-bit signed integer, so in order to convert it to an array index you need to take the value modulo the number of buckets[1]. The hashmap's configuration parameters (size, load factor) can be defined as private constants, or passed as parameters to the class's constructor.

Resizing the map

If, when you add (`put`) a new item into the hashmap, the number of elements exceeds the product of the number of buckets and the load factor, you should resize the array by constructing a new array that is twice the size of the original and add each element of the old array to the new one.

Testing as you go

Once all the above is in place, all the tests of `HashMapBackedByListsTest` should now pass. (Recall that these tests are actually defined in the `CustomMutableMapTest` superclass.)

However, it is good practice to test your solution as you proceed, rather than implementing all the methods of `GenericHashMap` and only running tests once you are done. Each time you implement a method of `GenericHashMap`, add at least one new test to `CustomMutableMapTest` that should pass as a result of the method you have implemented now being in place. (By adding the test to `CustomMutableMapTest`, this test will also provide additional testing of your `ListBasedMap` class, and any other map implementations that you write in the future.) Make

[1] If the number of buckets is a power of two then you can replace the modulus operation with and `and` operation, which turns out to quite a lot faster, viz. `key.hashCode() and (numBuckets - 1)`.

sure the new test passes, and note whether any of the provided tests also pass as a result of your new method. Make sure that you are always making forward progress: once a test is passing, changes you make in the future should not cause it to fail.

Extensions

Array-based hashmaps

Implement a hashmap using an array of entries, rather than our custom list, building on the experience you have gained during the course exercises on building resizing array lists. You'll need to re-size the array if the number of elements in the bucket exceeds the size of the array. As per the lectures, one solution is to replace the original array with one twice the size and re-insert the elements into the new array. Be mindful of how the resizing operation affects the amortized time complexity of insertion.

Name this class `HashMapBackedByArrays`, and make it a subclass of `GenericHashMap`, similar to what you did for `HashMapBackedByLists`. Use the `HashMapBackedByArraysTest` test class to test your solution.

Cuckoo hashmaps

Implement a simple form of *cuckoo hashing*, which uses two hashing functions and a single hashmap array. As before, each hash function maps a key to a position in the hash array. The idea is to displace any entry that clashes with one you're trying to insert and try to put it somewhere else, akin to a cuckoo chick displacing its host's eggs from the nest (except the displaced eggs gets to live on!).

To insert a key-value pair, you apply the following process,

- First, find the initial position using the first hash function
- If the initial position is empty, store the key-value pair there.
- If the initial position is occupied, you apply the second hash function to find an alternate position.
- If the alternate position is empty, store the key-value pair there.
- If the alternate position is occupied, replace the existing key-value pair with the new one and recursively try to insert the displaced pair elsewhere in the array.
- If displacement fails to find an empty position after a specified number of iterations (a parameter of the scheme) then you stop trying to find an empty position add the item to a collection of collisions, as in the simple scheme above.

For look-up and deletion, you apply both hash functions to find the possible positions of the searched-for item in the hash array. If the key is found you respectively either return the associated value or remove the key-value pair.

There are more complicated schemes involving multiple arrays and hashing functions, but the above will suffice for the exercise.

Tests for this kind of hashmap are not provided, but as long as the hashmap implements `CustomMutableMap` (directly or indirectly), you should be able to utilise the tests in `CustomMutableMapTest` by writing a suitable subclass of that test class.

Note: your repository containing the skeleton for this lab can be found at *https://gitlab.doc.ic.ac.uk/lab2324_spring/kotlinmaps_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.