Kotlin Final Test

Imperial College London

Department of Computing

Kotlin Social

Monday 29 April 2024
14:00 to 17:00
THREE HOURS
(including 10 minutes of suggested planning time)

- The maximum total is **100 marks**.
- Credit is awarded throughout for conciseness, clarity, *useful* commenting, and the appropriate use of the various language features.
- **Important:** Marks are deducted from solutions that do not compile in the test environment, which will be the same as the lab machines. Comment out any code that does not compile before you submit.
- After you have finished reading the spec, you can start coding by running the IDEA shortcut on your Desktop, which will open our provided .iml file. **Please do not attempt to open your project in any other way**, and please do not delete any of our files.
- The extracted files should be in your Home folder, under the "`kft`" subdirectory. **Do not move any files**, or the test engine will fail, resulting in a compilation penalty.
- The examples and test cases here are not guaranteed to exercise all aspects of your code. You are therefore advised to define your own tests to complement the ones provided.
- When you are finished, simply **save everything and log out**. **Do not shut down your machine**. We will fetch your files from the local storage.
- If your IDE fails to build your code, you can still compile via the terminal using the provided `./compile-all` perl script. You can then test your code via `./test-all`.

Problem Description

Your task is to write a number of classes and interfaces to model users in a social media application. A user has a set of friends, who are other users, and can receive friend requests which they can accept or reject.

You will first define an interface specifying the service that a user provides.

You will then write a simple implementation of this interface, where sets of friends are represented using lists, and where a user unconditionally accepts a friend request.

You will then refine this simple user implementation so that friend requests are conditionally accepted, and you will write code corresponding to a number of acceptance conditions.

The test then splits into two part that can be attempted in either order:

- **Data structures:** You will design a *linked hashmap* data structure, which is a cross between a hashmap and a linked list, and you will write an optimised user implementation that uses this data structure to represent the user's set of friends.
- **Concurrency:** You will augment the interface for users so that every user has a lock, and you will implement a "matchmaker" class that attempts to create friendships between pairs of users. A matchmaker will rely on the ability to lock users so that multiple matchmakers can make matches between users concurrently, in a thread-safe manner.

Getting Started

The skeleton files are located under the `src` package.

Under `src` there are `main` and `test` sub-directories. The code you write to implement functionality should go under `main`, while any test code should go under `test`.

Under `main` there are `kotlin` and `java` sub-directories, and code for the respective language should be located under the corresponding sub-directory. Within each of the `kotlin` and `java` sub-directories there is a directory called `social`. All functional code you write for this exercise will belong to the `social` package, and thus should reside in one of these `social` directories.

Under `test` there is only a `kotlin` sub-directory (because you are not expected to write tests in Java and all the provided tests are in Kotlin). Under this sub-directory is a `social` sub-directory. All provided tests plus any new tests that you create should be located here, in the `social` package.

You may feel free to add additional methods and classes, beyond those specified in the instructions, as you see fit. For example, this may be in order to follow good object-oriented principles, or for testing. Any new files should be placed in the `social` package for the appropriate programming language.

Testing

There is a test class, `Question`*i*`Tests`, for each question i (with the exception of Question 1). These contain initially commented-out tests to help you gauge your progress. **As you progress through the exercise you should un-comment the test class associated with each question in order to test your work.** In some cases you will be required to add additional tests to these classes as part of your task. Furthermore, you are welcome to add tests to these classes to help you debug your solution.

These tests are not exhaustive and are merely intended to guide you. Your solution should pass all of the given tests. However, passing all of the given tests does not guarantee that your solution is fully correct, and says nothing about your coding style and the appropriateness of your use of Kotlin and Java features. You should thus think carefully about whether your solution is complete, and pay attention to coding style and practices, even if you pass all of the given tests.

What to do

1. **The `User` interface.**

   In this question you will define an interface, `User`, specifying a service via various abstract properties and abstract methods. In this question you are not expected to implement any behaviour for `User`s, which is the subject of later questions.

   In a new file, `User.kt`, create an interface, `User`, specifying the following service:

   Properties A `User` should provide read access to the following properties:

   - `userName`: the name of the user, a string.
   - `yearOfBirth`: the year of the user's birth.
   - `bio`: the user's biography (some text describing the user and their interests), a string.
   - `currentFriends`: the current friends of the user, a list of `User`s.

   Methods A `User` should provide the following public methods:

   - `hasFriend`: takes a `User` and returns a boolean that is true if and only if the receiving user has a friend with the same username as that of the given user.
   - `removeFriend`: takes a `User`, and removes from the receiving user's friends any user with the same username as the given user. Returns a boolean that is true if and only if a friend was removed.
   - `considerFriendRequest`: takes a `User`—a "candidate friend"—as a parameter. Decides whether to add the candidate friend to the friends of the receiving user. Returns a boolean indicating whether or not the candidate friend was added.
   - `removeLongestStandingFriend`: takes no arguments. Returns null if the receiving user has no friends. Otherwise, removes from the receiving user's friends the user who has been in the set of friends the longest, and returns a reference to the removed friend.

   There are no tests for this question because you have not yet written any executable code that can be tested. In the next question, you will write a class implementing the `User` interface.

   [10 marks]

2. **The `SimpleUser` class.**

   In a new file, `SimpleUser.kt`, write a class, `SimpleUser`, that implements the `User` interface. Your implementation should meet the following requirements:

   - The primary constructor of a `SimpleUser` should concretely define the `userName`, `yearOfBirth` and `bio` properties required by the `User` interface.

- The set of friends of a `SimpleUser` should be recorded in an additional property, a mutable list of `User`s. Even though the friends form a set, you will use a list to represent this set to ensure that the order in which friendships were established is known.
- The `currentFriends` property required by the interface should return a *copy* of the user's list of friends, so that subsequent changes to the user's friends will not be reflected in the list returned by a prior call to `currentFriends`.
- If a `SimpleUser` is constructed with a year of birth that is not in the range 1900 to 2100 (both inclusive), an `IllegalArgumentException` should be thrown.
- A `SimpleUser` should accept any friend request, unless the user's list of friends already contains a `User` whose username is the same as that of the candidate friend.
- The remaining methods required by `User` should be implemented in the obvious way. Refer to the accompanying test cases for guidance on what is expected.

Test your solution using (at least) the tests in `Question2Tests`.

[15 marks]

3. **Befriending strategies.**

   A befriending strategy is a function that takes two `User`s as arguments: a *target* user, the user receiving the friend request, and a *candidate* user, the user making the friend request. It returns a boolean that is true if and only if the request should be accepted.

   In a new file, `BefriendingStrategies.kt`, write four top-level functions representing the following befriending strategies:

   - `standardStrategy`: returns true if and only if the target user does not already have a friend whose username is the same as that of the candidate user. This matches the strategy you already implemented in `SimpleUser`.
   - `unfriendlyStrategy`: never returns true (so that all friend requests are rejected).
   - `limitOfFiveStrategy`: returns false if `standardStrategy` does not hold. Otherwise, reduces the number of friends of the target user to less than five by removing as many of the target user's longest-standing friends as necessary and then returns true.
   - `interestedInDogsStrategy`: returns false if `standardStrategy` does not hold. Otherwise, returns true if and only if the list of strings obtained by splitting the candidate user's bio, delimited by spaces, contains the word "dog". Case should be ignored, so that e.g. words such as "DOG" and "dOG" should match the word "dog".

   Test your solution using (at least) the tests in `Question3Tests`.

   [10 marks]

4. **`SimpleUsers` with befriending strategies.**

   Modify your `SimpleUser` class so that a `SimpleUser` can optionally be constructed with an additional property of function type representing a befriending strategy. This property should be named `befriendingStrategy`. If no befriending strategy is provided on construction, a `SimpleUser`'s befriending strategy should default to `standardStrategy`.

   Hint: Consult the tests for this question to see how obtain a reference to a top-level function.

Rewrite the `considerFriendRequest` method in `SimpleUser` so that the given user is added to the receiving user's set of friends if and only if the befriending strategy holds when applied to the receiving user and the given user.

Test your solution using (at least) the tests in `Question4Tests`.

[5 marks]

5. **A linked hashmap data structure.**

   Note: Question 7 relates to concurrency and is almost completely independent from Questions 5 and Question 6. If you feel more confident about Java and concurrency compared with data structures, you may wish to solve Question 7 first.

   In this question you will complete the implementation of a data structure called a linked hashmap. In Question 6 you will write a `User` implementation that uses this data structure.

   A linked hashmap is a cross between a hashmap and a linked list. Like a normal hashmap, it uses buckets to store key-value pairs in an efficient way. However, unlike a normal hashmap, it also records the *order* in which key-value pairs were added to the map by organising the key-value pairs into a doubly-linked list. A doubly-linked list is like the singly-linked lists that were studied in lectures, except that every node in a list has a pointer to a predecessor node as well as a successor node, so that links go in both directions.

   The `OrderedMap` interface. A linked hashmap is an example of an ordered map. Study the provided `OrderedMap` interface. This is the service that your linked hashmap needs to provide, which can be summarised as follows:

   - The `values` property will yield a `List` containing all of the values in the map, ordered according to when they were added to the map.
   - The `size` property will yield the number of key-value pairs currently in the map.
   - The `set` method takes a key and a value, and associates the key with the value. If some value was previously associated with the key, the existing key-value pair should be removed from the map, and the new key-value pair added instead. With respect to the ordering, the new key-value pair should be the *newest* key-value pair, as it has been most recently added to the map. The method should return the value previously associated with the given key, or null if there was no such value.
   - The `containsKey` method takes a key and returns true if and only if the map contains an entry that maps this key to some value.
   - The `remove` method takes a key. If an entry that maps this key to some value exists in the map, this entry should be removed and the value with which the key was associated should be returned. Otherwise, the map should be left unchanged and null should be returned.
   - The `removeLongestStandingEntry` method takes no arguments and should return null if the map is empty. Otherwise, it should remove and return the key-value pair that has been present in the map for the longest.

   You should make one small change to this interface: adapt it so that the notation:

   ```
   m[key] = value
   ```

   can be used to add a key-value pair to a map `m`, as an alternative to:

`m.set(key, value)`.

Completing the `HashMapLinked` class. **Note:** Java provides a class called `LinkedHashMap`. To avoid confusion with this class, the class you will work on is called `HashMapLinked`. You may not use `LinkedHashMap` in your implementation of `HashMapLinked`.

Study the incomplete class, `HashMapLinked`, with which you have been provided.

This class declares a nested class, `Node`, which represents both an entry in a hashmap (via the `key` and `value` properties), and a node in a doubly-linked list (via the `prev` and `next` properties, which are references to the previous and next nodes in a list of nodes).

The class has `head` and `tail` properties: nullable `Node`s that refer to the oldest (`head`) and newest (`tail`) key-value pairs that have been added to the linked hashmap. These properties are both initially null, because the linked hashmap is initially empty.

The `buckets` property represents the hashmap part of the structure: a list of buckets, where each bucket is a list of nodes. This is similar to the kind of hashmap you studied in your lab exercises, except that each bucket element is a `Node`, meaning that as well as representing a key-value pair, it is also part of a doubly-linked list, and thus may have a pointer to a previous node and a next node, which might be located in different buckets from the node and from each other.

Before you start coding, study the diagrams provided in the Appendix at the end of this specification, which illustrate the structure of a linked hashmap, and how this structure changes as entries are added and removed.

You are also provided with:

- An `init` block for setting up the initial hashmap buckets;
- A `getBucket` method that can be used to obtain the bucket associated with a given key;
- A `resize` method that checks whether the hashmap buckets should be resized and performs resizing if so.

You should complete the remaining methods and properties of the class as follows:

- `containsKey`: This involves locating a node in a hash bucket associated with the given key, and returning true if and only if such a node exists.
- `remove`: As in the case of a regular hashmap, this method should determine whether an entry is present for the given key. If so, the associated `Node` should be removed from the hash bucket. Additionally, the `Node` should be unlinked from the doubly-linked list: if the `Node` has a previous (`prev`) node and a next (`node`) node, these should be updated so that the successor of `prev` becomes `next` and the predecessor of `next` becomes `prev`. A challenge here is to think through edge cases, such as when the node being removed is the head or the tail of the doubly-linked list. The size of the linked hashmap should be adjusted appropriately. The `remove` method should *not* perform a `resize` operation: once the number of buckets in a linked hashmap has been increased, it never decreases.
- `set`: Having implemented `remove`, `set` can be implemented relatively easily. First, call `remove` to remove any existing entry associated with the given key, and obtain the associated value if such an entry exists. The result of this call to `remove` (either a value, or null), is what `set` should ultimately return. A new node representing the key-value pair associated with the call to `set` can then be created and placed in a

suitable hash bucket. This node should be added to the end of the doubly-linked list, so that its predecessor is the previous tail of the list, and it becomes the new tail of the list. A challenge here again is to make this work in edge case scenarios, such as when the linked hashmap is empty. The size of the linked hashmap should be adjusted appropriately, and `resize` should be called in case the hash buckets need to be resized.

- `removeLongestStandingEntry`: This can be implemented easily via `remove`.
- `values`: The `get` for this property involves populating a `MutableList` with the values associated with all nodes in the linked hashmap, in the order in which they were added to the map. This involves traversing the list starting at `head`.

Test your solution using (at least) the tests in `Question5Tests`.

[35 marks]

6. **An `OptimisedUser` class.**

   In a new file, `OptimisedUser.kt`, write another implementation of the `User` class.

   This implementation should be similar to `SimpleUser`, but unlike `SimpleUser`, your `OptimisedUser` should represent the user's friends using an `OrderedMap` (instantiated with a `HashMapLinked`) that maps usernames (strings) to `User` instances. Part of your task is to work out which methods of `OrderedMap` are appropriate to meet the needs of an `OptimisedUser`.

   There will be some duplication of code between `SimpleUser` and `OptimisedUser`, and this is acceptable: you do *not* need to take measures to factor out such duplicate code.

   Test your solution using (at least) the tests in `Question6Tests`.

   [10 marks]

7. **Making matches between users in a thread-safe way.**

   Note: the new `Matchmaker` class that you will implement in this question should be written in Java. You may choose instead to write the class in Kotlin, but if you do then you will only receive **half marks** for the question.

   Your task now is to write a "matchmaker": a class that attempts to make friendship matches between users. To allow multiple matchmakers to operate concurrently on a set of users, you will start by equipping users with *locks*, so that a matchmaker can gain exclusive access to the users on which it is operating.

   Add an additional abstract property, `lock`, to the `User` interface, providing read access to a `Lock` from the `java.util.concurrent.locks` package. Adapt your implementations of `User` so that they override this property with a suitable concrete property. Do not make other changes to your `User` implementations. In particular, the operations implemented in `SimpleUser` and `OptimisedUser` (of you have written this class yet) should not acquire the lock.

   In a new file, `Matchmaker.java` (or `Matchmaker.kt` if you have decided to use Kotlin), write a class called `Matchmaker`. This class should be visible to your entire project, and it should not be possible to create subclasses of `Matchmaker`.

   A `Matchmaker` should have a single field of type `BiFunction`, which is from the Java package `java.util.function`. This class represents functions of two arguments. The purpose of the field is to check whether two users are to be considered *compatible* by the

matchmaker—i.e., whether they are likely to make good friends. This is irrespective of whether they would accept one another as friends in practice.

The field should therefore have type `BiFunction<User, User, Boolean>`, representing a function that takes two `User`s and returns true if and only if they are compatible.

The constructor of `Matchmaker` should take a value to populate this field.

`Matchmaker` should have one public method, `tryMatching`, which takes two `User`s as arguments. This method should work as follows:

- Acquire the locks associated with both users.
- Check whether the users are compatible via the `BiFunction` field. To do this, you will need to call the `apply` method of `BiFunction`, which expects arguments for each argument on which the function should be evaluated, and returns the function's result.
- If the users are deemed to be compatible, the `considerFriendRequest` method should be called on the first user, with the second user passed as an argument, and the `considerFriendRequest` method should be called on the second user, with the first user passed as an argument. The results of these method calls should be ignored.

Both of the acquired locks should be released before the `tryMatching` method returns. You should use the Java idiom you were shown in the lectures for ensuring that locks are always released (or the Kotlin idiom for achieving this, if you have chosen to solve this question using Kotlin).

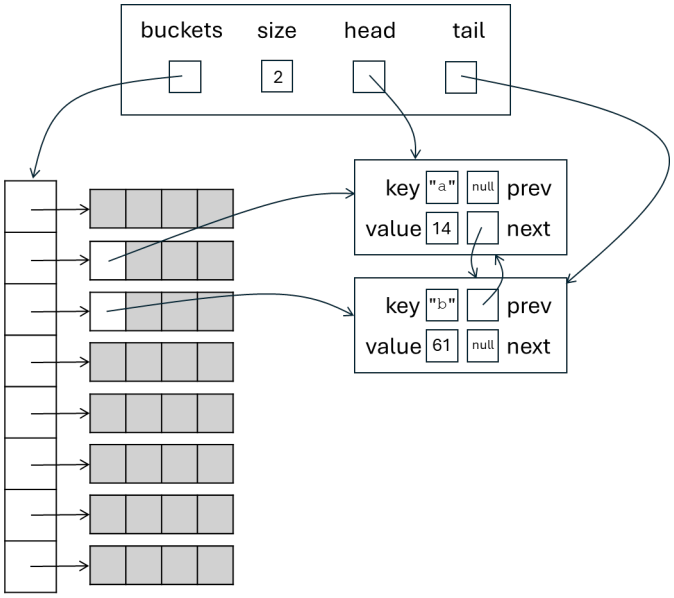
Test your solution using (at least) the tests in `Question7Tests`. These include a concurrency test that runs multiple matchmakers simultaneously on a list of users, repeatedly attempting to match them with one another in random orders.

[**15 marks**] Remember that you will receive **half marks** if you choose to implement the `Matchmaker` class in Kotlin.

Total: 100 marks

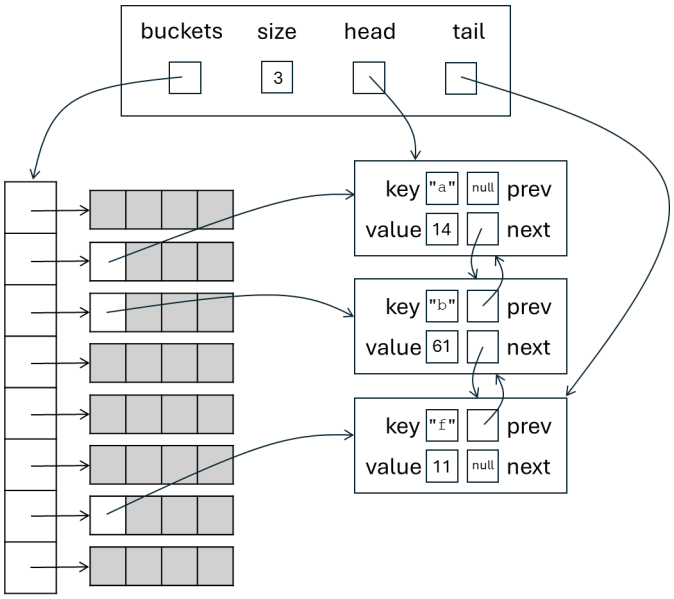
Appendix: linked hashmap diagrams

The following diagram illustrates a linked hashmap that maps strings to integers. There are initially 8 buckets. Buckets are depicted here as array-based lists, with greyed-out squares indicating currently-unused array elements. Let us suppose that the entries `"a"` $\mapsto$ 14 and `"b"` $\mapsto$ 61 have already been added to the linked hashmap:

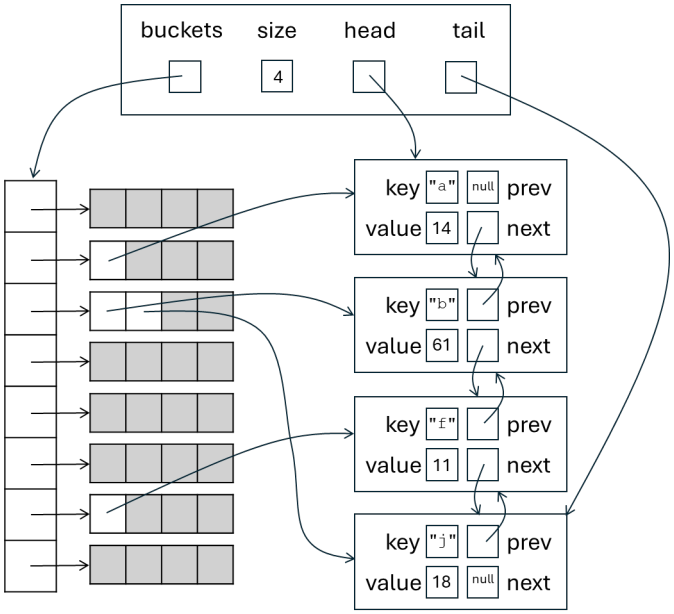


We can see that the entry `"a"` $\mapsto$ 14 was added before the entry `"b"` $\mapsto$ 61, because the node representing `"a"` $\mapsto$ 14 comes before the node representing `"b"` $\mapsto$ 61 in the doubly-linked list.

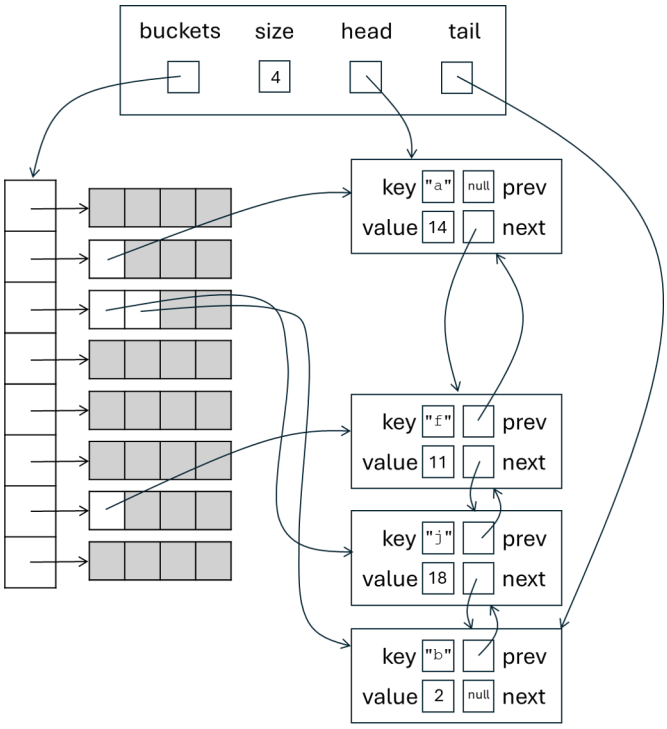
After adding the entry `"f"` $\mapsto$ 11, the linked hashmap has the following structure:



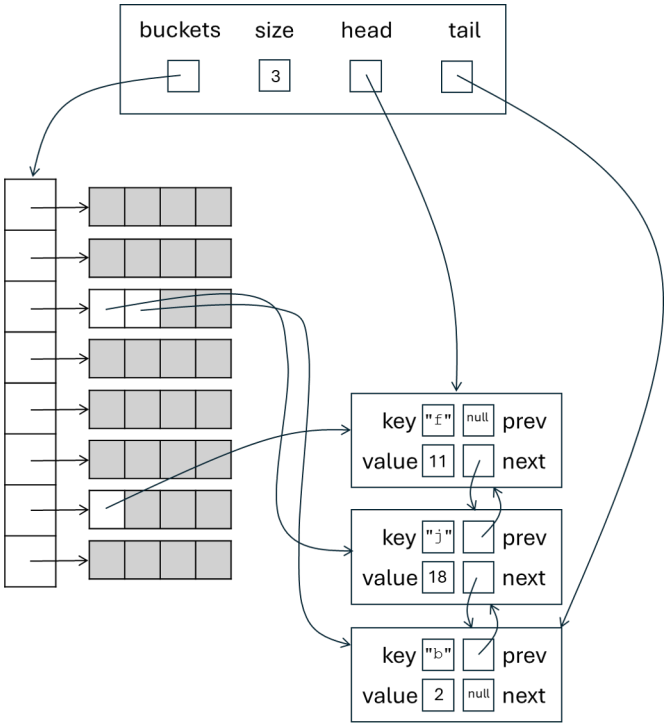
After adding the entry `"j"` $\mapsto$ 18, the linked hashmap has the following structure:



Setting the existing key `"b"` to a new value, 2, leads to the previous entry `"b"` $\mapsto$ 61 being removed, and a new entry `"b"` $\mapsto$ 2 being added. The linked hashmap then has the following structure—notice that the new node for `"b"` $\mapsto$ 2 is at the end of the doubly-linked list:



Finally, removing the entry `"a"` $\mapsto$ 14 leads to the linked hashmap having the following structure:


buckets
size
head
tail
3
key
"f"
null
prev
value
11
next
key
"j"
prev
value
18
next
key
"b"
prev
value
2
null
next