Lab 2: Ad-hoc Key-Value Stores

Due date: January 19, beginning of lab period.

Lab Assignment

Assignment Preparation

This is conceptually an individual lab. However, for this lab, I actually *encourage* you to continue talking to your Lab 1 partner (or another person in the class) to discuss design and implementation of the required software. I will allow collaboration and code sharing one **one specific aspect** of this lab. The instructions are provided at the end of this document.

Key-Value Stores

Key-Value store is a popular type of a distributed system. A Key-Value store is a system that stores a collection of objects, each of which has a unique explicit, known to the user, key, and provides the API for retrieval of individual objects.

In addition to this functionality, many Key-Value stores provide other functions, some of which require careful approach to storage of data and its manipulation.

Key-Value stores can be thought of as the most primitive database management systems. They store data in the format:

```
<key>: <value>
```

where `<key>` is a unique value of a data type specified ahead of time (for example, and integer, or a string), and `<value>` is an arbitrary object, often thought of simply as a byte array.

Key-Value stores support the following functionality:

- **Create Collection.** This operation creates a new empty collection of Key-Value pairs.
- **Clear Collection.** This operation removes all Key-Value pairs stored in the collection, but keeps the collection itself.
- **Drop Collection.** This operation deletes the collection and all Key-Value pairs stored in it.
- **Put Key-Value Pair.** This operation, given a Key-Value pair and a collection, inserts the Key-Value pair into the collection.
- **Get Value By Key.** This operation, given a collection and a key, returns the value stored under the key in the collection (or `null` if no such key exists.
- **Check if a Key is in Collection.** This operation, given a collection and a key, simply returns whether the collection contains a record under the given key.
- **Update a Key-Value pair in Collection.** Given a key-value pair and a collection, find the key in the collection and update the value with the new value provided.
- **Delete a Key-Value pair from Collection.** Given a collection and a key value, find the key value in the collection and remove the key-value pair from it.

Key-Value stores can also support some additional functionality, some of which is discussed in your assignment.

The Task

For this assignment, you will test how far a *naïve* implementation of a Key-Value store using Java's `Map<K,V>` interface will take you.

There are two key artifacts to this assignment, your code, and a report that details the results of your investigation/experimentation. These are described below.

The Software

You will implement a simple in-memory Key-Value store for reading and storing the data contained in the JSON documents produced by your `BeFuddled` and `ThghtShre` JSON generators. The specifics of the storage are discussed below.

The internal implementation of the Key-Value store shall rely on one of Java's implementations of the `Map<K,V>` interface. The most "pure" implementation of this interface is the `Hashmap` class, and I highly recommend that you use it.

Your program shall implement the following classes:

1. `KeyValueStore`. This is essentially a container class for a list of of Key-Value collections. The API for the `KeyValueStore` itself largely revolves around collection creation and management.
2. `KVCollection`. This class is essentially a wrapper around a single `HashMap` that is to be used for storing a single collection of Key-Value pairs.
3. Experiment classes. You will create a number of experiments that use your `KeyValueStore` implementation to create Key-Value stores and test their performance.

In addition to these classes you may need to implement/re-implement/reuse the `BeFuddled` and `ThghtShre` parsers for reading in the JSON objects produced by your `Lab 1-1` JSON generators.

General Notes

Your Key-Value store implementation shall handle the following type of Key-Value pairs:

1. **Keys.** All collections of your Key-Value store will always use integer keys. To acoommodate how `Map<K,V>` interfaces work, you will have to use the Java `Integer` object type for the keys.
2. **Values.** All collections will always store JSON objects. Use the `org.json` `JSONObject` class instances for values.

Class Collection Overview

The `KVCollection` class represents a single Key-Value collection managed by the Key-Value store. This class is essentially a wrapper around one of Java's implementation of the `Map<K,V>` interface (e.g.,`HashMap`).

You can design the instance variables for an instance of `KVCollection` to suit your needs. There is no restriction on what you can or cannot use there.

The `KVCollection` class shall provide access to the following methods and constructors.

Constructors. A simple implementation of `KVCollection` shall implement just one constructor.

- `KVCollection()` creates an empty Key Value collection. The collection shall use integers (Java's `Integer` objects) as keys, and `org.json`'s (or other JSON library's) `JSONObject` objects as values.

You may add other constructors to your implementation if you find them convenient.

Methods. Your `KVCollection` class shall implement the following methods.

1. `boolean clear()`: removes all key-value pairs stored in the collection. Returns `true` is successful, `false` otherwise.

2. `boolean containsKey(int key)`: given a key, returns `true` if the key is found in the collection, returns `false` otherwise.

3. `JSONObject get(int key)`: given a key, retrieve an object stored in the collection under this key. Returns the object, or `null` if key does not exist.

4. `int put(int key, JSONObject value)`: inserts the `<key, value>` pair into the collection. Return value is set as follows:

Return value	Meaning
1	key-value pair succesfully inserted
-1	attempt to insert duplicate key. key-value pair not inserted.
0	key-value pair could not be inserted for any reason other than duplicate key.

5. `int remove(int key)`: given a key, removes the key-value pair with that key from the collection. The return value is set as follows:

Return value	Meaning
1	key is in collection. key-value pair succesfully removed.
-1	key is not in collection.
0	key is in collection but removal operation failed.

6. `int replace(int key, JSONObject value)`: attempts to replace the contents of the object stored under the given key value with the new object. Return value is set as follows:

Return value	Meaning
1	key-value pair succesfully replaced
-1	key is not in the collection. Replace is not performed.
0	key-value pair is in the collection, but replace failed.

7. `int size()`: returns the number of key-value pairs currently stored in the collection.

8. `int upsert(int key, JSONObject value)`: if object with a given key exist - replaces the value with the new object. Otherwise, inserts a new key-value pair. Return value is set as follows:

Return value	Meaning
1	key-value pair succesfully inserted
2	key-value pair successfully replaced
0	operation failed

In addition to these methods, implementing essentially the Key-Value Store API, your program shall implement one more method for the `KVCollection` class.

- `JSONArray find(String jsonFieldName, Object jsonFieldValue)`: this method shall perform a search *among all key-value pairs* stored in the `Collection` instance for JSON objects (stored in `value` components of the key-value pairs), which satisfy the following search condition:

  > The JSON object must contain a field/attribute with the name `jsonFieldName` whose content is equal to (deep equality) the content of `jsonFieldValue`.

  The method constructs a `JSONArray` object that consists of the JSON objects built out of the selected JSON objects as follows. Let us say that a key-value pair `<Key, Value>` where `Key` is an `Integer` and `Value` is a JSON object satisfies the search condition. Then, the method shall construct the following JSON object:

  ```
  {
   "key": Key,
   "value": Value
  }
  ```

  and will append it to the output `JSONArray`.

For example, consider the following method call: `find("name", "Bob")`. Consider the following JSON object

```
 {
  "name":"Bob",
  "grade":"A"
}
```

stored in the collection under the key 123. Then, the `find` method will add the following JSON object to the `JSONArray`:

```
{
 "key": 123,
 "value": { "name":"Bob",
            "grade":"A"
          }
}
```

At the bare minimum, your implementation must correctly search for strings and numbers (integers, floating points). If you want to split the `find` method into multiple, based on the type of the second argument, you can do it.

KeyValueStore Class Overview

`KeyValueStore` class is the class implementing the overall in-memory key-value store. Essentially, a Key-Value store is a collection of `KVCollection` objects, where each `KVCollection` object is given a unique name. `KeyValueStore` provides access to collection manipulation functionality that allows the users to add collections to the store, and select collection objects from the store.

Your program is expected to maintain a single instance of a `KeyValueStore` object - all collections will be stored inside it.

You are not restricted in the design of the instance variables for the `KeyValueStore` instances.

The `KeyValueStore` class will implement the following constructors and methods.

Constructors. You shall implement three constructors. One creates an empty `KeyValueStore` capable of storing arbitrary many collections. The second creates an empty `KeyValueStore` that will store no more than a given number of collections. The third method shall create a `KeyValueStore` prepopulated with a specific set of empty collections.

- `KeyValueStore()`: this constructor creates an empty Key-Value Store, which, in the future can store an arbitrary number of collections. (an empty Key-Value Store means a Key-Value Store with no collections in it).
- `KeyValueStore(int limit)`: this constructor creates an empty Key-Value Store, which in the future can store no more than `limit` different collections.
- `KeyValueStore(Set<String> collections)`: the constructor creates a Key-Value Store prepopulated with the set of empty `KVCollection` collections, whose names are provided in the `collections` parameter. The Key-Value store created this way will admit no new collections.

You may add any other constructors if you find them convenient.

Methods. The `KeyValueStore` class will implement the following methods.

1. `clear()`: remove all collections from the Key-Value Store. Leaves the Key-Value Store empty.
2. `int addCollection(String name)`: create an empty collection and add it to the Key-Value store under a given name. The return value indicates the status of the operation as follows:

Return value	Meaning
1	collection succesfully inserted
-1	duplicate name. collection not created.
-2	limit on the number of collections is exceeded
0	collection not inserted for any other reason

3. `KVCollection getCollection(String name):` return a `KVCollection` object representing the collection with the given name. Returns `null` if collection with the given name does not exist.

4. `Set<String> list():` return the list of collection names. You can replace `Set<>` with any other Java type that implements `List<>` interface (e.g., `Collection`) if it is more convenient for you.

5. `boolean isEmpty():` return `true` if the Key-Value store contains no collections, `false` otherwise.

6. `int size():` return the number of collections in the Key-Value store.

7. `int getNumObjects():` return the number of objects in **all** collections in the Key-Value Store.

8. `int getLimit():` return the maximum number of collections allowed by the Key-Value Store. Return -1 is there is no limit set.

Storing JSON Objects in the Key-Value Store

In this assignment, you will be storing the JSON objects created by your `Lab 1-1` generators in the Key-Value store that you implement.

The driver program for storing your objects will be tasked with generating the keys under which each JSON objects is stored. Please use the following key generation strategies:

- BeFuddled **data.** BeFuddled JSON objects lack an immediate unique key. Therefore, for each object, your driver program shall generate a unique integer key. The simplest thing to do is start with 1, and autoincrement the key, but you can use other strategies if you want.
- ThghtShre **data.** ThghtShre messages have unique ids in them. You driver shall extract the unique id of each message and use it as the key. Note that you will store the JSON object itself intact as the `value` under the key (i.e., do not remove the messageid tag from the JSON object).

Experiments to Conduct

Using the implementation of the `KeyValueStore` that you build, you need to conduct a series of experiments to determine the answers to the following questions.

Note. In many of your experiments you will be tasking the resources of the system on which your code is running. In the report you prepare to document your findings, you will be running your code on one of the CSL servers (`unix11`, `unix12`, etc.) with default Java settings (specifically, with the default Java setting for the amount of heap memory).

1. How many `BeFuddled` JSON objects does it take to fill the memory available to you?
2. How many `ThghtShre` JSON objects does it take to fill the memory available to you?
3. How much time does it take to create a **Key-Value Store** of a specific size?
4. How fast can you retrieve a single JSON object by its key from a single collection, based on the size of the collection?
5. How fast can you find what JSON objects in a single collection possess a certain property (based on the size of the collection)? (that is, how fast is your `find()` implementation). Experiment with objects of both types.
6. How much does the efficiency of retrieval of an object by key depend on how many collections you have to search for it? (see below)

To study these questions, implement multiple driver programs that parse the JSON files produced by your beFuddled and ThghtShre generators, store the data in your `KeyValueStore` instances and measure what you need. Each question will probably require a separate driver, and sometimes - multiple drivers.

The experimental design for the first five questions is left up to you. For question 6, I suggest the following approach.

- Experiment with a different number of collections. Vary the number of collections from 1 to some upper bound you are comfortable with (10, 20, ...).
- In each round of the experiment develop a specific "sharding" strategy for your data. I would test multiple sharding strategies as specified below. A "sharding" strategy is the means by which you decide into which collection the specific JSON object is going.
- Insert the objects into the key-value store using your sharding strategy.
- Perform multiple searches of an object- generate a valid key, search for it, repeat however many time you feel is needed. Time each search, compute the average and the standard deviation of your timings.

- Different sharding strategies can affect how you implement the search. At the bare minimum, I would try two different strategies: one in which given a key you can determine ahead of time, which collection (shard) to search for it in, and one, where you would not know in which collection to look for a specific key.
- The sharding strategy of the first type is, for example the one where you separate all keys into ranges, and store a specific range of keys in each collection. To test this approach, you must make sure to generate enough keys so that you attempted retrieval from each collection (as collection sizes can vary).
- The sharding strategy of the second type is, for example, a round-robin strategy, where you insert key-value pairs into each collection in turn, or a random strategy, where you randomly choose which collection to insert the key-value pair into. In both cases, we expect roughly balanced in size collections, but your search may potentially need to access all collections.

Report

Part of your assignment is the written report documenting what you have learned from your experiments.

Your report shall be a PDF document formatted as an academic paper (you do not need to use any specific publication format though). It shall contain the following information.

- Title, your name, email address, course number.
- Implementation overview. A short section that discusses how you chose to implement your `KeyValueStore` and `KVCollection` classes. Specify any functionality you added to the required API, as well as any design decisions that were not obvious/trivial.
- Experiments conducted. For each question, provide the following information:
  - Brief description of how you elected to answer the question.
  - Brief description of what you have measured.
  - Brief description of the implementation of your experimental driver/drivers.
  - Instructions for graders on how to run the driver/drivers.
  - Results you have obtained.
  - Analysis and discussion of the results, conclusions. These can be brief.
- Reflection. Provide a short description of what you have learned for yourself while implementing this lab.

Collaboration and Sharing

You must abide by the following rules concerning collaboration and sharing of code on this assignment.

- Each student must implement `KVCollection` and `KeyValueStore` **individually**. You *may discuss* the design with your Lab 1 partner (or another student in the class), but there shall be **no code sharing** for this part of the assignment.
- You are **allowed to discuss, jointly implement, use, and submit the results of running** the experimental drivers. That is, feel free to work with a partner (or a small group of partners) on design and development of the experiments. You can share the code for those experiments between all members who actively participated in its design and development.
- You must run all your experiments (when testing) either on the output generated by your Lab 1-1 code, or on any sample data posted by the instructor.
- You must write an **individual** report.
- You must **explicitly acknowledge** all students with whom you discussed the implementation of any stage of this assignment in your report and in the comments to your code. Any shared code must contain the names of all students who shared it. If you modified some common code (e.g., to adapt a jointly developed experiment to the quirks in your API), mention this in the code comments as well.
- Your collaborative efforts should be restricted to groups of 2-3 people. **Do NOT** gang up larger groups on the experiments.
- Contact me if you want to discuss these rules any further.

Submission

Submit all your code in a single archive (zip or tar.gz). Submit your report as a separate file (outside of the archive).

Use `handin` to submit as follows:

Section 01:

```
$ handin dekhtyar lab2-01 <FILES>
```

Section 03:

```
$ handin dekhtyar lab2-03 <FILES>
```

Good Luck!