Lab 8: HashMap

Introduction

In this lab, you'll work on **MyHashMap**, an implementation of the `Map61B` interface, which represents a hash map. This will be very similar to Lab 7, except this time we're building a Hash Map rather than a Tree Map.

After you've completed your implementation, you'll compare the performance of your implementation to a list-based Map implementation `ULLMap` as well as the built-in Java `HashMap` class (which also uses a hash table). We'll also compare the performance of `MyHashMap` when it uses different data structures to be the buckets.

MyHashMap

Overview

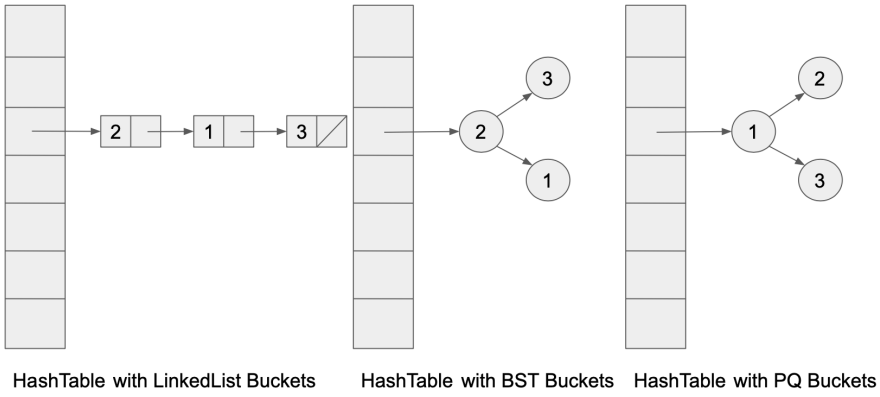
We've created a class **MyHashMap** in `MyHashMap.java`, with very minimal starter code. Your goal is to implement all of the methods in the **Map61B** interface from which **MyHashMap** inherits, *except for* `remove`. For `remove`, you should throw an `UnsupportedOperationException`. Note that you should implement `keySet` and `iterator` this time, where `iterator` returns an `Iterator` that iterates over the stored keys. Both of these functions may return the keys in any order. For these

methods, we recommend you simply create a `HashSet` instance variable that holds all your keys.

Note that your code will not compile until you implement all the methods of **Map61B**. You can implement methods one at a time by writing the method signatures of all the required methods, but throwing `UnsupportedOperationExceptions` for the implementations, until you get around to actually writing them.

Skeleton Code

You might recall from lecture that when we build a hash table, we can choose a number of different data structures to be the buckets. The classic approach is to choose a `LinkedList`. But we can also choose `ArrayList`s, `TreeSet`s, or even other crazier data structures like `PriorityQueue`s or even other `HashSet`s!



During this lab, we will try out hash tables with different data structures for each of the buckets, and see empirically if there is an asymptotic difference between using different data structures as hash table buckets.

You can maybe imagine that if we implemented `MyHashMap` without any care, it would take a lot of effort with Find + Replace to be able to change out the bucket type with a different bucket type. For example, if we wanted to change all our `ArrayList` buckets to `LinkedList` buckets, we would have to Find +

Replace for all occurrences of `ArrayList` and replace that with `LinkedList`. This is not ideal.

The purpose of the starter code is to have an eaiser way to try out different bucket types with `MyHashMap`. It accomplishes this through polymorphism and inheritance, which we learned about earlier this semester. It also makes use of **factory methods**, which are used to create objects. We will use factory methods to create the buckets. The inheritance structure of the starter files is as follows:

```
Map61B.java
└── MyHashMap.java
    ├── MyHashMapALBuckets.java
    ├── MyHashMapHSBuckets.java
    ├── MyHashMapLLBuckets.java
    ├── MyHashMapPQBuckets.java
    └── MyHashMapTSBuckets.java
```

`MyHashMap` implements the `Map61B` interface through use of a hash table. In the starter code, we give the instance variable `private Collection<Node>[] buckets`, which is the underlying data structure of the hash table. Let's unpack what this code means:

- `buckets` is a `private` variable in the `MyHashMap` class
- It is an array (or table) of `Collection<Node>` objects, where each `Collection` of `Node`s represents a single bucket in the hash table
- `Node` is a private helper class we give that stores a single key-value mapping. The starter code for this class should be straightforward to understand, and should not require any modification
- `java.util.Collection` is an interface which most data structures inherit from, and it represents a group of objects. The `Collection` interface supprots methods like `add` to the group, `remove` from the group, and `iterate` over a

group. Many data structures in `java.util` implement `Collection`, including `ArrayList`, `LinkedList`, `TreeSet`, `HashSet`, `PriorityQueue`, and many others. Note that because these data structures implement `Collection`, we can assign them to a variable of static type `Collection` with polymorphism

- Therefore, our array of `Collection<Node>` objects can be instantated by many different types of data structures, e.g. `LinkedList<Node>` or `ArrayList<Node>`
- When creating a new `Collection<Node>[]` to store in our `buckets` variable, be aware that in Java, you cannot create an array of parameterized type. `Collection<Node>` is a parameterized type, because we parameterize the `Collection` class with the `Node` class. Therefore, the expression `new Collection<Node>[size]` is illegal, for any given `size`. To get around this, you should instead create a `new Collection[size]`, where `size` is the desired size. The elements of a `Collection[]` can be a collection of any type, like a `Collection<Integer>` or a `Collection<Node>`. For our purposes, we will only add elements of type `Collection<Node>` to our `Collection[]`.

Each of the `MyHashMap*Buckets` classes instantiates `buckets` with a different type of data structure. For example, `MyHashMapLLBuckets` instantiates `buckets` with a `new LinkedList<Node>()`. The mechanism by which this happens is a factory method `protected Collection<Node> createBucket()`, which simply returns a data structure that implements `Collection`. For `MyHashMap.java`, you can choose any data structure you'd like. For example, if you choose `LinkedList`, the body of `createBucket` would simply be:

```
protected Collection<Node> createBucket() {
        return new LinkedList<>();
```

```
}
```

Instead of creating new bucket data structures with the `new` operator, you must use the `createBucket` method instead. This might seem useless at first, but it allows the `MyHashMap*Buckets.java` classes to override the `createBucket` method and provide different data structures as each of the buckets. Then, we end up with multiple different classes (`MyHashMapTSBuckets.java` `MyHashMapPQBuckets.java`, etc.) that all use your implementation in `MyHashMap` but they provide different types for the buckets (`TreeSet`, `PQ`, etc.). We can even have a hash table that has buckets that are other hash tables (`MyHashMapHSBuckets.java`)! We can then directly compare each of the `MyHashMap*Buckets.java` classes directly in a speed test similar to the one you saw in Lab 7.

We provide additional factory methods `createTable` to create the backing array of the hash table and `createNode` to create new `Node` objects as well. It's okay if you use `new` operators to create the backing array and `Node` objects instead of the factory method, but we added them for uniformity.

Implementation Requirements

You should implement the following constructors:

```
    public MyHashMap();
    public MyHashMap(int initialSize);
    public MyHashMap(int initialSize, double loadFact
```

Some additional requirements for `MyHashMap` are below:

- Your hash map should initially have a number of buckets equal to `initialSize`. You should increase the size of your `MyHashMap` when the load factor exceeds the set `loadFactor`. Recall that the **load factor** can be computed

as `loadFactor = N/M`, where `N` is the number of elements in the map and `M` is the number of buckets. The load factor represents the amount of elements per bucket, on average. If `initialSize` and `loadFactor` aren't given, you should set defaults `initialSize = 16` and `loadFactor = 0.75` (as Java's built-in HashMap does)

- You should handle collisions with separate chaining. You may not import any libraries other than `ArrayList`, `LinkedList`, `Collection`, `HashSet`, `iterator` and `Set`. This means that for `MyHashMap.java`, you should use either an `ArrayList`, `LinkedList` or `HashSet` for the bucket types in `MyHashMap.java`. For more detail on how you should implement separate chaining, see the **Skeleton Code** setion above.
- Because we use a `Collection<Node>[]` as our `buckets`, when implementing `MyHashMap`, you are restricted to using methods that are supported by the `Collection` interface. The only methods you will need are `add`, `remove`, and `iterator`. When you are searching for a `Node` in a `Collection`, simply iterate over the `Collection`, and find the `Node` whose `key` is `.equal()` to the key you are searching for
- When resizing, make sure to multiplicatively resize, not additively resize. You are **not** required to resize down
- Your `MyHashMap` operations should all be constant amortized time, assuming that the `hashCode` of any objects inserted spread things out nicely (recall: every `Object` in Java has its own `hashCode()` method). **Note**: `hashCode()` can return a *negative value*! Please take this into account when writing your code. Take a look at the lecture slides linked below for a hint on how to cleanly handle this case.

- If the same key is inserted more than once, the value should be updated each time. You can assume `null` keys will never be inserted.

Testing

You can test your implementation using `TestMyHashMap.java`. If you choose to implement the additional `remove` method, we provide tests in `TestHashMapExtra.java`. If you've correctly implemented generic `Collection` buckets, you should also be passing the tests in `TestMyHashMapBuckets.java`. The `TestHashMapBuckets.java` file simply calls methods in `TestMyHashMap.java` for each of the different map subclasses that implement a different bucket data structure.

Before moving on from this section, be sure you're passing the tests in `TestMyHashMap.java` and `TestMyHashMapBuckets.java`.

Resources

You may find the following resources useful:

- HashMap code from pages 136 and 137 of Data Structures Into Java, from our course references page
- Chapter 3.4 of our optional Algorithms textbook
- HashTable code from our optional textbook
- `ULLMap.java` (provided), a working unordered linked list based **Map61B** implementation
- Lecture slides on HashMaps, inheritance, and subtype polymorphism

HashMap Speed Test

There are two interactive speed tests provided in `InsertRandomSpeedTest.java` and `InsertInOrderSpeedTest.java`. Do not attempt to run these tests before you've completed **MyHashMap**. Once you're ready, you can run the tests in IntelliJ.

The `InsertRandomSpeedTest` class performs tests on element-insertion speed of your **MyHashMap**, **ULLMap** (provided), and Java's built-in **HashMap**. It works by asking the user for an input size `N`, then generates `N` Strings of length `10` and inserts them into the maps as `<String, Integer>` pairs.

Try it out and see how your data structure scales with `N` compared to the naive and industrial-strength implementations. Record your results in the provided file named `lab8/speedTestResults.txt`. There is no standard format required for your results, and there is no required number of data points.

Now try running `InsertInOrderSpeedTest`, which behaves similarly to `InsertRandomSpeedTest`, except this time the Strings in `<String, Integer>` key-value pairs are inserted in lexicographically-increasing order. Note that unlike Lab 7, your code should be in the rough ballpark of Java's built in solution – say within a factor of `10` or so. What this tells us is that state-of-the-art `HashMaps` are relatively easy to implement compared to state-of-the-art `TreeMaps`. When would it be better to use a `BSTMap`/`TreeMap` instead of a `HashMap`? Discuss this with your labmates, and add your answer to `speedTestResults.txt`.

Change Bucket Types: Speed Test

If you've correctly implemented generic `Collection` buckets, most of the work is done! We can directly compare the different data structures `MyHashMap*Buckets.java`. We provide `speed/BucketsSpeedTest.java`, which is an interactive test that queries the user for an integer `L` for the length of string to use on subsequent operations. Then, in a loop, it queries the user for an integer `N`, and runs a speed test on each of the five data structures:

- `MyHashMapALBuckets`, which uses `ArrayList` buckets
- `MyHashMapLLBuckets`, which uses `LinkedList` buckets
- `MyHashMapTSBuckets`, which uses `TreeSet` buckets
- `MyHashMapPQBuckets`, which uses `PriorityQueue` buckets
- `MyHashMapHSBuckets`, which uses `HashSet` buckets

Try it out and compare how the different implementations scale with `N`. Discuss your results with your labmates, and record your responses in `speedTestResults.txt`.

You might notice that our implementation of `MyHashMapTSBuckets` and `MyHashMapHSBuckets` searches for a `Node` by iterating over the entire data structure. But from what we know, trees and hash tables support more efficient lookups than that. Would our hash table speed up if we were able to use a logarithmic search over the `TreeSet` or a constant-time search over the `HashSet`? You do not need to implement anything new here, just discuss with your labmates, and record your ideas in `speedTestResults.txt`.

Optional Exercises

This will not be graded, but you can still receive feedback on the autograder.

Implement the methods `remove(K key)` and `remove(K key, V value)`, in your **MyHashMap** class. For an extra challenge, implement `keySet()` and `iterator` without using a second instance variable to store the set of keys.

For `remove`, you should return null if the argument key does not exist in the **MyHashMap**. Otherwise, delete the key-value pair (key, value) and return value.

Lab Debrief and Submission

At the end of lab, your TA will go over the reference solution. This will be helpful if you haven't finished the lab, since we don't want you to be stuck working on lab too much outside of lab. (This is also an incentive for you to go to lab!)

Make sure to submit your completed `MyHashMap.java` and `speedTestResults.txt`, and submit through git and Gradescope as usual.