CSCI241 Winter 2019: Assignment 3
Due March 6th at 9:59pm

Your submissions, for this and all future homework assignments, must be yours. You may chat with other students on a high-level about topics and concepts, and brainstorm using a white board, but you cannot share, disseminate, co-author, or even view, another student's code. See also the academic honesty guidelines on the syllabus. If any of this is unclear, please ask for further clarification.
If you rely on any external resources (e.g., the internet, other textbooks, etc.), you MUST cite that resource(s) in a comment portion at the top of your `Heap.java` file. Under no circumstances may you cut-and-paste entire blocks of code—if you do, you will receive an F in the course and be reported to the Dean of Students.

1 Overview

A3 consists of three phases.

1. Implement a standard min-heap in Heap.java.
2. Implement a hash table in HashTable.java.
3. Augment your heap in Heap.java to implement efficient `contains` and `changePriority` methods.

   See the lecture slides from Lecture 16 for some additional context.

2 Getting Started

The Github Classroom invitation link for this assignment is in Assignment 3 on Canvas. Begin by accepting the invitation and cloning a local working copy of your repository as you have for past assignments.

The Heap class depends on the AList class you wrote for Lab 5. Copy your AList.java from your lab 5 repository into your A3 repo's src/main/java/heap/ directory.

3 Unit Tests and Gradle

You are provided with unit tests for each phase. All tests live in A3Test.java, and they are numbered with three-digit numbers; the hundreds place indicates which phase the test pertains to. **Test often**, and make sure you pass any tests associated with a TODO item before moving on to the next one. You may find it helpful to run only the tests for the phase you're currently working on using the –tests flag with a wildcard; for example to run only the phase 2 tests, you could run

```
gradle test --tests "test2*"
```

4 Phase 1

In Phase 1, you'll complete the implementation of a min-heap given in the skeleton repository's Heap.java file. For details on how these operations work, see the slides from Lectures 13 and 14.

The tasks listed below are marked in Heap.java as `TODO 1.0`, `TODO 1.1`, and so on. **Phase 3 involves further modifications to Heap.java. For now, ignore anything in the code marked as Phase 3 Only, or** `TODO 3.x`.

1.0 Read and understand the class invariant in the comment at the top of Heap.java. Ignore the Phase 3 parts for now. This specifies the properties the Heap must satisfy. All public methods are responsible for making sure that the class invariants are true before the method returns.

1.1 Implement the `add` method, using the `bubbleUp` helper according to its specification.

1.2 Implement the `swap` helper method, which you'll use in both bubbling routines.

1.3 Implement the `bubbleUp` helper method. Feel free to use private helper methods to keep this code clean.

1.4 Implement `peek`. Recall that peek returns the minimum element without modifying the heap.

1.5 Implement `poll` using `bubbleDown`, which you'll implement next.

1.6 Implement `bubbleDown`. You are highly encouraged to use one or more helper methods to keep this code clean. In fact, we've included a suggested private method `smallerChild`, along with its specification, that we used in our solution.

5 Phase 2

In Phase 2 you'll implement a hash table with chaining for collision resolution. Although we could base it on our AList class, we need access to the internals of the growth process to handle growing and rehashing as needed. Similarly, for the underlying storage we could use an array of LinkedLists, but dealing with Java's LinkedList machinery ends up being a bit of a headache—more so than simply writing a little linked list code to do the chaining by hand. For these reasons, you'll complete a standalone hash table implementation in HashTable.java without using any tools from `java.util`. The following major design decisions have been made for you:

- The hash table encapsulates its key-value pairs in an inner class called `Pair`.
- The hash table uses chaining for collision resolution.

- The `Pair` class doubles as a linked list node, so it has fields to store its `key`, `value` and a reference to the `next` Pair in the chain.
- The underlying array doubles in size when the load factor exceeds 0.8.

Here's some sample code using the hash table and its output using my solution. The `dump` method shows the internal layout of the table: each bucket in the `buckets` array stores a reference to Pair objects, each of which stores a reference to the next Pair in the chain, or `null`.

```
Code:
    HashTable<Integer,Integer> hm = new HashTable<Integer,Integer>(4);
    hm.put(0,0);
    hm.put(4,1);
    hm.put(19,1);
    hm.put(19,4);
    hm.dump()

Output:
    Table size: 3 capacity: 4
    0: -->(4, 1)-->(0, 0)--|
    1: --|
    2: --|
    3: -->(19, 4)--|
```

Your job in Phase 2 is to implement four methods: (`get`, `put`, `containsKey`, and `remove`). Details of how you implement them are left up to you - be sure to read the specification of each method carefully to be sure you're implementing the specified behavior correctly, completely, and according to the given efficiency bounds.

Your tasks:

2.0 To get familiar with the underlying storage mechanism, read the existing code in HashTable.java: note the . In particular, read the javadoc comments above the class, the comments describing each field and its purpose, and the Pair inner class. Then take a look at the provided `dump` method, which may be useful for debugging.

2.1 Implement `get(K key)`.

2.2 Implement `put` without worrying about load factor and array growth. Make sure that you replace the value if the key is already in the map, and insert a new key-value pair otherwise. For now, don't worry about load factor or growing the array. After implementing `get` and `put` without growing the array, you should pass test210PutGet, test211PutGet, test212Put, test213Put, test230PutGet, and test231Put.

2.3 Implement `containsKey` Your code should pass test240containsKey and test241containsKey.

2.4 Implement `remove`. Your code should pass test250Remove and test251Remove.

2.5 Finally, modify `put` to check the load factor and grow and rehash the array if the load factor exceeds 0.8 after the insertion. Use the `createBucketArray` helper method to create the new array. We've also included a method stub for a `growIfNeeded` private helper method with a suggested specification; you are welcome to implement and use this, but not required to. At this point your code should pass all tests.

6 Phase 3

In Phase 3, we turn back to the `Heap` class. Now that we have a working HashTable implementation, we can overlay the hash table on the heap in order to make the `contains` and `changePriority` operations efficient. This is a common strategy when building data structures in the real world: often a textbook data structure does not provide efficient runtimes for all the operations you need, so you can combine multiple textbook data structures to get more efficient operations at the cost of some extra bookkeeping and storage.

In the Phase 1 heap, finding a given value in the tree requires searching the whole tree, so the runtime is O(n). In Phase 3, we'll use a HashTable to map from **values** to **heap indices**, which allows us to find any value in the heap in expected O(1) time, as we discussed in Lecture 16. Keeping the HashTable up to date requires small changes throughout the Heap class to make sure that the HashTable stays consistent with the state of the heap - whenever the heap is modified, we need to update the HashTable to match.

One constraint imposed by the use of a HashTable is that each **value** can only map to a single **index**. This means that if we insert two entries with equal **value** into the table, we can't differentiate between them and store both indices HashTable. To deal with this, we will simply add the requirement that all **values** stored in the Heap must be distinct. Note that two different **values** may still have equal **priorities**.

Your tasks are as follows:

3.0 Introduce the `map` field into Heap.java by doing the following:

   a. Uncomment the `map` field in Heap.java.

   b. Uncomment the line of the constructor that initializes the `map` field.

   c. Add a `throws` clause to the `add` method's header to indicate that the method will throw an `IllegalArgumentException` if the user calls `add` with a value that's already contained in the heap.

3.1 Update the `add` method to keep the `map` consistent with the state of the heap. Also be sure to throw an exception if a duplicate value is inserted—the `map` makes it possible to check this efficiently. For now, don't worry about the map during `bubbleUp`—the next TODO will handle this. At this point, your code should pass test300Add.

3.2 3.2 swap - update the swap method to keep the HashTable consistent with the heap. If you used `swap` to implement both bubbling routines, `bubbleUp` and `bubbleDown` You should now pass test310Swap and test315Add_BubbleUp.

3.3 Update `poll`. Your code should now pass test330Poll_BubbleDown_NoDups and test340testDuplicatePriorities.

3.4 Implement `contains`. Once again, the map makes this easy and efficient. You should pass test350contains.

3.5 Implement `changePriority` by finding the value in question, updating its priority, and fixing the Heap property by bubbling it up or down. You should now pass test360ChangePriority.

At this point, your code should be correct! Check the following things before you submit:

1. Each method adheres to the asymptotic runtime given in its specification, if any.

2. Your code follows the style guidelines set out in the rubric and the syllabus.
3. Your submission compiles and passes all tests on the command line in Linux without modification.
4. All code is committed and pushed to your A3 GitHub repository.

7 Extra Credit: Test Suite Gaps

This assignment has no optional enhancements - the base assignment will be worth all 50 points. There is an opportunity for extra credit if you discover a gap in my test suite. Specifically, if you discover that your implementation passes **all** the provided test cases (for AList or for any phase of A3) but has a bug, you can earn up to 3 points of extra credit for providing a correct test case that catches the bug.

If you find such a bug:

1. With the bug present in your solution code, Create a "testing" branch in your repository (`git checkout -b testing`).
2. Modify the appropriate testing class (AListTest.java or A3Test.java) to include your additional test case. When this test file is run, your test should be the only one that fails.
3. Commit your test file changes and push your testing branch to github.
4. Email Prof. Wehrwein (scott.wehrwein@wwu.edu) with a description of the bug and the test you've written.

Do not wait until submission time—submit these whenever you encounter them. You can switch back to your master branch, fix the bug in your solution, and continue working on the project.

8 How and What to Submit

Submit the assignment by pushing your final changes to GitHub (`git push origin master` or just `git push`) before the deadline.

Rubric

You can earn points for the correctness and efficiency of your program, and points can be deducted for errors in commenting, style, clarity, and following assignment instructions. Correctness will be judged based on the unit tests provided to you.

Git Repository	
Code is pushed to github and hours spent appear as a lone integer in hours.txt	1 point
Code : Correctness	
Each unit test is worth 1 point. A full report including any unit test failures will be committed to your repository in the `grading` branch.	32
Code : Efficiency	
`Heap.add` is average-case O($\log n$) and worst-case O(n), unless rehashing is necessary in which case the runtime is expected O($C+n$) and worst-case O($C+n^2$), where C is the smaller array's capacity.	2
`Heap.peek` is O(1)	1
`Heap.poll` is average-case O($\log n$) and worst-case O(n)	2
`HashTable.get` is average-case O(1), worst-case O(n)	2
`HashTable.put` is average-case O(1), worst-case O(n)	2
`HashTable.containsKey` is average-case O(1), worst-case O(n)	2
`HashTable.remove` is average-case O(1), worst-case O(n)	2
`Heap.contains` is average-case O(1) and worst-case O(n)	2
`Heap.changePriority` is average-case O($\log n$) and worst-case O(n)	2
Clarity deductions (up to 2 points each)	
Include author, date and purpose n a comment comment at the top of each file you write any code in	
Methods you introduce should be accompanied by a precise specification	
Non-obvious code sections should be explained in comments	
Indentation should be consistent	
Method should be written as concisely and clearly as possible	
Methods should not be too long - use private helper methods	
Code should not be cryptic and terse	
Variable and function names should be informative	
Total	50 points