CSE 373, Winter 2013
Homework Assignment #4 (25 points)
HashMaps vs. Zombies (aka "The Hashing Dead")

Due Friday, February 8, 2013, 11:30 PM

This program focuses on implementation of a map using a hash table array as its internal data structure. Turn in files named `HashMap.java`, `HvZPlayer.java`, and `myzombies.txt` from the Homework section of the course web page.

You will also need to download the support .java files and input .txt files from the Homework section of the course web site; place them in your Java project and add to it. If you are using Eclipse, place .java files in your project's `src/` folder and .txt files in the project root folder (the parent of `src/`). We do not guarantee that the provided tests are exhaustive; perform additional testing of your own before you submit your work.

Part A (Humans vs. Zombies) Description:

Humans vs. Zombies Tag ("HvZT") is a game played on college campuses where two sets of real people chase each other around like lunatics. One set is the "humans", who are trying to escape from players in the other set, the "zombies". If a zombie touches a human, the human is killed and "infected" and becomes a zombie. If a zombie gets more than a certain number of kills, it is a "super zombie." If a zombie doesn't infect any humans for a few days, it dies of starvation. The game ends after some number of days: if all humans have become infected, the zombies win; and if some humans are still alive or all zombies have died of starvation, the humans win.

This assignment has two parts. In the shorter Part A of this assignment, you are provided with some existing code to model a HvZT game. We have already written the main client program, `HvZMain`, along with a class to model the overall game simulation, `HvZSimulation`. The simulation class creates objects of type `HvZPlayer` to represent each player in the game. The state of each `HvZPlayer` object indicates the player's name, whether that player is a human or zombie, its kills, and so on. The simulation runs through a plain text user interface such as the following partial output log:

```
Christine infects Vicki
Christine infects Sara
Doug dies of starvation
Marty dies of starvation
End of day 4
Humans : [Frank(H), Kris(H), Ursula(H)]
Zombies: [Christine(Z:2), Vicki(Z:0), Hank(Z:0), Stuart(Z:3!), Sara(Z:1), Jamie(Z:0)]
Kills  : {Stuart(Z:3!)=Christine+Jamie+Hank, Christine(Z:2)=Vicki+Sara}
The humans are still clinging to survival!
...
```

The code for this HvZT simulation is essentially completely finished already, but it does not work properly. The problem is that the simulation object stores much of its data about the players and game in sets and maps, specifically `HashSet`s and `HashMap`s, and the `HvZPlayer` class is not built properly to be used in such collections. Any class whose objects are to be stored as elements of a `HashSet`, or used as keys of a `HashMap`, must implement a proper equals and `hashCode` method or else the results will be incorrect.

So you must implement the following two methods in the `HvZPlayer` class. You should not need to define any new fields or other methods in the class in order to implement this behavior.

`public boolean` **`equals`**`(Object o)`

In this method you should return `true` if `o` refers to an `HvZPlayer` object with equivalent state to the current object, otherwise `false`. Equivalent state means equal value for all fields: name, human/zombie, and number of kills.

`public int` **`hashCode`**`()`

In this method you should return an integer hash code for the player. The hash code should incorporate all aspects of the player's state (all fields) and should follow the general contract of hash codes: equal for two objects with the same state; consistent across multiple calls if the object's state does not change; and generally well distributed to avoid collisions.

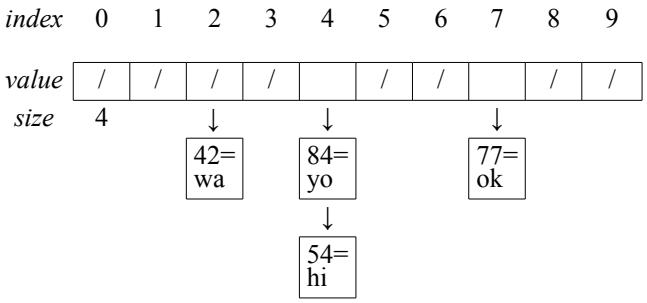
Part B (HashMap) Description:

Part B is the more substantial part of the assignment where you implement a map of your own using a hash table as the internal data structure. You will write a subset of the methods from the real `Map` interface from `java.util`. When you finish Part B, you can go back to the `HvZSimulation` class from Part A and modify it slightly so that it uses your hash map instead of Java's. This provides a good test of your map code.

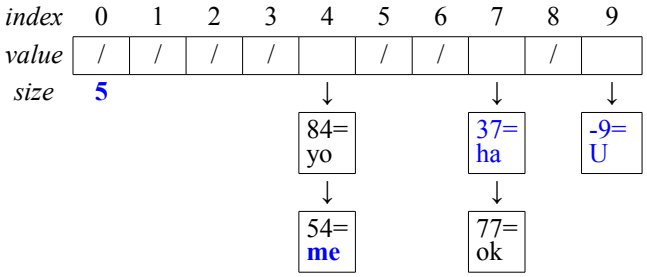
As we have discussed, a **hash table** is an array that uses an algorithm called a hash function to map data into array indexes. Your hash map will provide O(1) expected average runtime for the common operations of adding, removing, and searching for key/value pairs. The hash function you should use is the one shown in class, which asks the key for its `hashCode` value, computes the absolute value of the hash code, and wraps it by the hash table array length.

Hash tables must deal with collisions; in your map you will handle collisions using **separate chaining** as discussed in class. In other words, each index of your hash table array should store a linked list of node objects, where each node stores one key/value pair in the map along with a link to a next node. Write a **`Node` class** similar to the one shown in lecture, where each `Node` object stores a key, a value, and a link to a (possibly `null`) next node. The linked list in each hash bucket index should be initially `null`, but `Node` objects are added to the front of it as the `put` method is called.

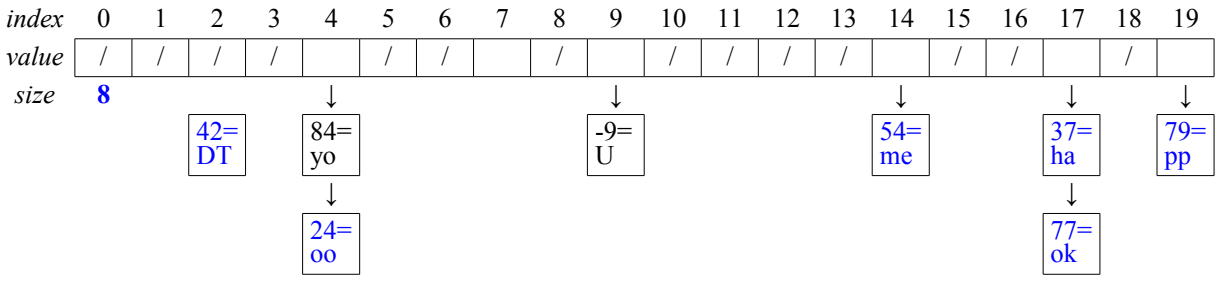
Suppose we have a 10-element hash table representing a map from integers to strings. It is initially empty and then we use the `put` method to add four key/value pairs: 54="hi", 77="ok", 42="wa", and 84="yo". In the picture, `/` means `null`.



If the client calls `remove` on a given key, any existing key/value pair for that key is removed from the hash table. If the client calls `put` with an existing key and a new value, the **old value is replaced**. From the previous hash map state, if the client removes key 42, then puts the pairs 54="ME", -9="U", and 37="ha", the hash table's state is the following. Notice that 54="ME" replaces the previous pair 54="hi" without changing the map's size value. Also notice that 37="ha" is placed at the front of the linked list for hash bucket index 7, rather than at the end, for efficiency.



At the end of any `put` call, if the map's load factor (its ratio of size to capacity) reaches or exceeds 0.75, enlarge it by **rehashing** its data into a new array twice as large. You cannot simply copy over the contents into the same indexes in the new array because the hash code wrapping changes. If we start from the previous hash table and put 24="oo", 42="DT", and 79="pp", the last pair will yield a load factor of 0.8 and will trigger a rehash. Afterward, the hash table will be:



Part B (HashMap) Implementation Details:

Write a class `HashMap<K, V>` that implements our `Map<K, V>` interface. It has the following methods; write them with exactly these headers, so they can be called by the client. You may define additional `private` methods. You will also need to declare any private data fields and private inner classes needed by the class in order to implement this behavior (such as your inner `Node` class). All specified methods must run in O(1) average runtime unless otherwise specified.

`public` **`HashMap`**`()`

In this constructor you will initialize a newly constructed empty map. (Give it a default hash table capacity of 10.)

`public void` **`clear`**`()`

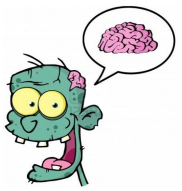
In this method you should remove all key/value pairs from your map. Make sure that your internal hash table stores only `null` after a call to `clear` so that the memory previously occupied by elements can be freed. This method is O(*N*).

`public boolean` **`containsKey`**`(K key)`

In this method you should return `true` if your map contains a key/value pair that includes the given key, else `false`.

`public V` **`get`**`(K key)`

In this method you should return the value associated with the given key, if any. If the map does not contain a key/value pair for this key, you should return `null`.



`public boolean` **`isEmpty`**`()`

In this method you should return `true` if your map does not contain any elements, else `false`.

`public Set<K>` **`keySet`**`()`

In this method you should return a `java.util.Set` object holding all keys from your map. In the real `Map` implementations in `java.util`, the `keySet` is a "view" that links back to the original map; for simplicity you will instead construct and return a separate `HashSet` holding a copy of all keys from your map on each call. It has no link back to the map; if the set is modified by the client, no change to the map occurs, and vice versa. This method is O(*N*).

`public void` **`put`**`(K key, V value)`

In this method you should add the given key/value pairing to your map. If any previous key/value pair exists for that same key, it should be replaced by this new one. After a key/value pair is added, if the load factor of the hash table (the ratio of its size to its capacity) has reached or exceeds 0.75, you should rehash the data into a new array of twice the capacity. Rehashing is O(*N*) but if you resize to a multiple of the old size, the average runtime for this method is O(1).

`public void` **`remove`**`(K key)`

In this method you should remove any existing key/value pair for the given key. After removing a key/value pair, your map should not retain any references to that key or value internally, so that any memory associated with them might be able to be freed by the Java garbage collector.

`public int` **`size`**`()`

In this method you should return the number of key/value pairs currently in your map.

`public String` **`toString`**`()`

In this method you should return a string representation of your map's elements. The string is surrounded by curly braces and contains each key/value pair connected by an equals sign and separated by commas. The order in which the pairs are shown is up to you; typically it reflects the internal ordering of the hash buckets in your array. For example, for the map in the second picture on the previous page, you could return `"{a=b, c=d, e=f}"`. For an empty map you should return `"{}"` and for a one-pair map you should return the one pair in braces such as `"{a=b}"`. This method is O(*N*).

For all methods that accept keys or values as parameters, **check for non-`null` arguments**. If the key and/or value passed is `null`, you should throw a `NullPointerException`. Also, in all methods that look up pairs based on keys in the hash table, equality of keys is determined by calling their **`equals` method**, not using the == operator.

Development Strategy and Hints:

- Write Part A first; it should be relatively straightforward. Compare your output to ours using the Output Comparison Tool on the class web site. (Since your hash codes likely will not be identical to ours, the hash set/map ordering will not match exactly. So check the comparison tool's "ignore character ordering" option.)
- Download `HashSet.java` from lecture as an example. You should borrow heavily from this example class, especially the version using separate chaining. The operations of a hash set are very similar to those of a map. (Don't include our `Set.java` or `HashSet.java` in your project because they would replace Java's.)
- Implement a basic map that supports `put` and `get`, based on the provided `HashSet` shown in class.
  Do not worry about some of the tricky cases like replacing an existing key/value pair or resizing.
  You might want to write debugging code to print out your hash table array and all of its buckets / lists.
  At this point you can also write `isEmpty` and `size`.
- Write a temporary `toString` that just returns the `Arrays.toString` of your hash table.
  This is not the proper behavior, but it will help you debug in the meantime.
- Write `containsKey` and `keySet`. Be mindful of potential redundancy.
- Write the `remove` method. Test several cases: removing from the front of a list, from the middle and end, removing a value that is not found in the table, and so on.
- Make it so that your hash table can resize to a larger array once it becomes full.
- Write the proper version of `toString`, and any other missing behavior. Test thoroughly.

Style Guidelines and Grading:

We will grade your program's behavior and output to give you an **External Correctness** score. You can use the course web site's Output Comparison Tool to check your HvZT simulation and check test case output for your hash map.

We will also grade your code quality (**Internal Correctness**). There are many general Java coding styles that you should follow, such as naming, indentation, avoiding redundancy, etc. For a list of these, please see the **Style Guide** document posted on the class web site. You should follow all of its guidelines as appropriate on this and future assignments.

In past assignments we have used collections heavily to solve problems. But in this assignment (Part B), you are implementing a collection of your own. **For this reason, you may not use any of Java's built-in data structures (such as the ones from `java.util`) to implement your `HashMap`.** For example, declaring a private field in your class of type `ArrayList` or `Map` would be unacceptable. A solution that disobeys this restriction will receive a very substantial deduction. Your map's inner hash table array should be the only data structure of any kind that is declared by your code.

Your map must be implemented using a hash table, including separate chaining for collision resolution, as described in this document. Solutions that do not do so will receive substantial deductions for both external and internal correctness.

You must also match the expected **Big-Oh** demands of each method. Relaxing the Big-Oh might make it easier to implement the functionality; but the whole point is implement the map efficiently. So this will be a grading focus.

Watch out for **redundant code**. If you are performing identical or very similar commands repeatedly, factor out the common code and logic into a helper method, loop, or other facility to remove the redundancy. In particular, consider redundancy between hash table operations like getting a key/value pair and seeing whether a key is contained in the table.

We strongly recommend that you write some **private helper methods** to help you implement the required public functionality of `HashMap`. Think about common operations you would want to perform or values you would want to compute and how methods might be useful in such cases. Our own sample solution uses four private helpers.

Be mindful of your fields. `HashMap` should probably have 2 fields at the most, one for the hash table and one for the size. Do not declare a value as a private field unless it is necessary to retain it as part of the state of your object.

If you have a commonly used literal value, such as a number, declare it at the top of your code as a **class constant**.

In your method comment headers make sure to comment what exceptions (if any) are thrown by each method and under what conditions they are thrown. In addition to including comment headers on each class and each method, also make sure to include inline comments next to any complex or tricky code, briefly explaining the purpose of that code.

This document and its contents are copyright © University of Washington. All rights reserved.