HW 3: Hash Table

I have written a program to analyze word counts in texts. It relies on a `HashTable` object, that you must write.

You may choose to write your hash table using either separate chaining, or linear probing. Some of the methods you write that return statistics about the table will behave slightly differently, depending on which one you choose (see below). You will receive a small amount of extra credit if you choose to implement linear probing.

The program loads up a text, and counts the number of unique words it contains. It accepts queries from the user regarding counts of different words. Here is a typical run of the program (with separate chaining) run on Lewis Carroll's "Alice's Adventures in Wonderland":

```
$ java WordFreqs alice.txt
The text contains 2629 unique words.
Please enter a word to query, "!help" for help, or "!exit" to exit.
> !stats
Hash table statistics:
    Size (n): 2629
    Table size (m): 1535
    Load factor (α): 1.713
    Empty units: 277
    Longest run: 8
> alice
"alice" appears 386× in the text.
> -alice
386 entries of "alice" have been deleted.
> alice
"alice" is not in the text.
> +alice
"alice" now appears 1×.
> alice
"alice" appears 1× in the text.
> !exit
Goodbye!
```

Several public-domain texts have been provided for you on the class web page.

Here is the API of the `HashTable` object, with its methods' expected time complexities:

- `HashTable(int initialSize)` (constructor) ($O(m_0)$). Creates the hash table, with the initial internal array size as given. It is linear with respect to the initial size m_0 (which is probably very small).
- `HashTable()` (constructor) ($O(1)$). Creates the hash table, using the default initial size of `11`.

- `void put(K key, V value)` (amortized O(1)) Inserts a new key-value pair.
- `V get(K key)` (amortized O(1)) Returns the value corresponding to the given key, or `null` if the key is not present.
- `V delete(K key)` (amortized O(1)) Removes a key-value pair, returning the deleted value. Returns `null` if the key wasn't present.
- `boolean containsKey(K key)` (amortized O(1)) Searches for the key, and returns true if it is present.
- `boolean containsValue(V value)` (O(n)) Searches for the value, and returns true if it is present.
- `boolean isEmpty()` (O(1)) Returns true if the table is empty.
- `int size()` (O(1)) Returns n, the number of key-value pairs in the table.
- `K reverseLookup(V value)` (O(n) or O(m)) Finds a key that maps to the given value, or returns `null` if there is none.
- `int getTableSize()` (O(1)) Returns m, the size of the primary internal array.
- `double getLoadFactor()` (O(1)) Returns α, which is n/m.
- `int countEmptySlots()` (O(m)) For separate chaining, counts the number of chains with no entries. For linear probing, counts the number of entries with no valid key-value pair.
- `int findLongestRunLength()` (O(m)) For separate chaining, returns the length of the longest chain. For linear probing, returns the length of the largest cluster.

You will probably find it easiest to make private `rehash()` and `hashAndMod()` methods. But which private methods you make is ultimately your choice. And as before, this list uses Java-based syntax. You are responsible for adjusting the methods as needed if you are using a different language.

Your program must rehash its table periodically, in order to maintain constant times. The load factor α must be kept within a certain critical range.

- For separate-chaining tables, $0.5 < \alpha < 2.0$.
- For linear-probing tables, $0.125 < \alpha < 0.75$.
- When α gets too large on an add, rehash from size m to a new size $2m + 1$. (This doesn't consistently produce prime numbers, but it works pretty well.)
- When α gets too small on a deletion, rehash from size m to a new size $\lfloor m/2 \rfloor$. However, you should only rehash down when $m \geq 10$. Small tables may thus have values of α below the usually allowed range.

And of course, style still matters! Make your code readable and follow the proper standards.