| | | |
|---|---|---|
| ETEC2101 | Lab5: **Hash Sets and HashMaps** | Points: **100**[1] |
| Assigned: **10/17/2016** | Due: **10/28/2016 at 3:00pm**[2] | |

**Tasks:**

1. Implement a HashMap data structure
   a. (**10 points**) Overall structure.
   b. Should be generic in *two* ways (the key (of type K) and value (of type V))
   c. For all these methods, you will lose a lot of points if you scan the entire array! HashMaps are supposed to be O(1) for all operations (**-40 points**)
   d. You can internally either use circular / linear probing or buckets (your choice).
   e. Should internally use an array-ish[3] structure to hold to hash table. You'll lose a lot of points if you don't have this (**-40 points**)
   f. The constructor should take three values:
      i. int **initial_capacity**: Make sure the internal "array" has this many slots (initially holding nulls)
      ii. int **capacity_increase**: When we exceed the load factor, re-size the array to include this many more values. It's very important when you re-size that you re-hash all the elements when moving them over to the new array. Don't use System.arraycopy!
      iii. **load_factor** (float in the range (0.0 to 1.0, not including end-points)).
   g. (**8 points**) void **set**(K key, V value): place the value in the hashed-to position in the array. If that key already exists, replace the value, but don't increase the size. If that value isn't in the array.
   h. (**8 points**) V **get**(K key): Return null if that key isn't in the hash-map, or return the value if it is.
   i. (**2 points**) int **getSize**(): Return the number of *used* spots in the hash-map. Ideally, you'll calculate this value as you add things (so you don't have to scan the entire array to calculate it).
   j. (**2 points**) **HashMapIterator**<K, V> iterator(IteratorType t): Should return an iterator that will either walk through all the values or the keys in the hash-map (see below for details). You should create the enum IteratorType in the HashMap class.
   k. (**8 points**) void **remove**(K): Removes the given key-value pair from the hash-map.
   l. (**10 points**) String **toString**(): Output all the values in this form:
      `{key:value, key:value, … }`
2. (**20 points**) Implement an Iterator for the HashMap:
   a. The next and hasNext methods should just "consider" non-null entries in the HashMap.
   b. Next should either return a value of type K or of type V (depending on the IteratorType value that is passed to the constructor).
3. Create a HashSet class
   a. (**10 points**) Overall structure.
   b. Should extend from HashMap (you'll probably need two types here if you did the bonus from 1.d. The only difference is we store the keys and index by keys!
   c. (**7 points**) void add(K item): call the set method from the base class.
   d. (**8 points**) String toString(): Similar to HashMap's toString, but only shows the keys (no colon).
   e. (**15 points**) Create methods *(each will operate on "this" set and another, which is passed to you)* to do the following (see overview of https://en.wikipedia.org/wiki/Venn_diagram):
      i. Create the union of two sets
      ii. Create the intersection of two sets
      iii. Create the relative difference (sometimes called relative complement) of two sets.
      iv. Symmetric difference
4. (**10 points**) Good documentation (using Javadoc syntax)
5. (**12 points**) Devise your own test program that *thoroughly* tests all your code (if you don't finish all items, I'll "pro-rate" this value.

[1] There are 130 points possible. If you more than 100 points, the rest is bonus.

[2] -10 if submitted by noon 10/29/2016, -20 if submitted by noon 10/30/2016, -30 if submitted by noon 10/31/2016 (and no more)

[3] I think a "normal" array of Objects or a java.util.Vector (which work very similarly to ArrayLists) object would work nicely here. Both allow you to control (and see) the capacity. ArrayList and more "advanced" list-like objects don't give you as much control.