Hashing Worksheet

Q1

Assume we are working with the Java `Integer` class. For each proposed `hashCode` method, explain whether or not the hash code will be valid, and whether or not the hash code will be good.
A note: the `Integer` class extends the `Number` class, a direct subclass of `Object`. The `Number` class `hashCode` method directly calls the `Object` class `hashCode` method.

1. `public int hashCode() {return intValue() * intValue();}`

2. `public int hashCode() {return super.hashCode();}`

3. `public int hashCode() {return intValue();}`

4. `public int hashCode() {return 2*intValue();}`

Q2

For the following two questions, indicate whether the answer is **always**, **sometimes**, or **never**.

1. When you modify a key that has been inserted into a `HashMap` will you be able to retrieve that entry again? Explain.

2. When you modify a value that has been inserted into a `HashMap` will you be able to retrieve that entry again? Explain.

Q3

Consider a hash table that uses external chaining and also keeps track of the number of keys that it contains. It stores each key at most once; adding a key a second time has no effect. It takes the steps necessary to ensure that the number of keys is always less than or equal to twice the number of buckets (i.e., that the load factor is ≤ 2). Assume that its hash function and comparison of keys take constant time. All bounds should be a function of N, the number of elements in the table.

1. Give $\Theta()$ bounds on the worst-case times of adding an element to the table when the load factor is 1 and when it is exactly 2 before the addition.

   Bound for load factor 1:

   Bound for load factor 2:

2. Assume that the hashing function is so good that it always evenly distributes keys among buckets. What now are the bounds on the worst-case time of adding an element?

   Bound for load factor 1:

   Bound for load factor 2:

3. Making no assumption about the goodness of the hashing function, suppose that instead of using linked lists for the buckets, we use some kind of binary search tree that somehow keeps itself bushy. What bound can you place on the worst-case time for `containsKey`?

   Bound:

4. Using the same representation as in part (c), but with a very good hash function, as in part (b), what bound can you place on the worst-case time for testing to see if an item is in the table?

   Bound:

Solutions available on 61B Spring 2017 Dis. 9 and Dis. 9 Exam-Prep:
For #1 and #2: http://datastructur.es/sp17/materials/discussion/discussion9sol.pdf
For #3: http://datastructur.es/sp17/materials/discussion/discussion9epsol.pdf

Quick Overview

Your HashMap resembles an array of linked lists containing key:value entries. Each 'space' in the array is considered a bucket; thanks to our 'linked-list'-like implementation, each bucket can carry many items (this is called external chaining). To PUT a key:value pair or GET a value, there are a few common steps that are followed:

1. Get the HASHCODE of the key
2. Modulo that hash code by the length of the Map Array to get the index
3. Go to that index, and iterate over the linked list to see if the key exists in the list

Note that, in the actual Java implementation of HashMap, the authors defined a nested, inner class called ENTRY. So, HashMap is actually comprised of an array of ENTRY objects, where each ENTRY object contains a key, value, and a pointer to the next ENTRY. (Another way to think of this: each ENTRY object is like an INTLIST node, except it carries information about both key and value).

Runtime Analysis

Runtime analysis on HASHMAP operations depend on the 'goodness' of the HASHCODE method used. In the average case, understand that the *average* number of items per bucket is constant– that's why we can make constant time assumptions. In the worst case, with a bad HASHCODE method, we assume that all items map into the same bucket, so that the number of items per bucket is linear to the number of items total.

If some of these runtimes don't seem obvious to you, I'd recommend walking through the operations with the pseudocode in section 2.

Here is a small table of runtimes, given a HASHMAP with n items

	Worst Case	Average Case
HASHCODE distribution is...	Bad	Good (uniform)
PUT (without resizing)	$O(n)$	$O(1)$
PUT (with resizing)	$O(n)$	$O(n)$

Pseudocode

Here's a pseudocode version of a few common Hash Map operations; this is mainly to give you intuition on how Hash Map operations work.

```
1: function PUT(Key k, Value v)                    ▷ Goal: Add key:value pair k : v to the map
2:     hash := k.HASHCODE()
3:     index := hash % MapArray.length             ▷ Find the index you want in your array
4:     for Entry e in MapArray[index] do           ▷ Iterate over the linked list at that index
5:         if e.key is equal to k then
6:             Replace the value in e with v       ▷ Case 1: The key is already in list; overwrite its value
7:             return                              ▷ After overwriting value, exit function call
8:     MapArray[index].INSERTFRONT(k,v)            ▷ Case 2: Key is not yet in map; insert to the front
9:     NumberItems++
10:    if CurrentLoadFactor≥MaxLoadFactor then     ▷ If load factor is too large, resize
11:        RESIZE()
```

```
1: function RESIZE                                 ▷ Goal: Increase the MapArray's size
2:     NewMapArray := NEW array[MapArray.length*2] ▷ Create a new array of double the size
3:     for each index i in MapArray do
4:         for each Entry e in MapArray[i] do      ▷ Iterate over each item in each bucket
5:             hash := e.key.HASHCODE()
6:             newIndex := hash % NewMapArray.length
7:             NewMapArray[newIndex].INSERTFRONT(e.key, e.value)  ▷ Insert into new location in new map
8:     MapArray := NewMapArray                     ▷ Replace the original map array with the resized array
```

```
1: function GET(Key k)                             ▷ Goal: Get value v associated with k
2:     hash := k.HASHCODE()
3:     index := hash % MapArray.length             ▷ Find the index you want in your array
4:     for Entry e in MapArray[index] do           ▷ Iterate over the linked list at that index
5:         if e.key is equal to k then return e.value   ▷ Case 1: We've found the correct entry
       return null                                 ▷ Case 2: No entry with key k exists
```

Properties of Hash Codes

Validity

1. For the same object, the hash function should always return the same number. Thus, the hash function must be deterministic.

   - **Why?** If our hash function is not deterministic (e.g., it relies on a random number generator), then any call to `get` or `put` will give a different hash code each time. You may not be able to find a key after you put it in the map!

2. If two items are semantically equal (e.g., the `equals` method indicates both are equal), then the hash code for both items should be the same

   - **Why?** First, let's recall the contract between `equals` and `hashCode`:
     (a) If two objects are equal, then they must have the same hash code
     (b) if two objects have the same hash code, they may or may not be equal.
   - Recall also the default implementations of `equals` and `hashCode`.
     (a) If `equals` is not overwritten, the default method is inherited from the `Object` class. In that case, all the `equals` method will do is check to see if the address of two objects are the same.
     (b) If `hashCode` is not overwritten, the default method is inherited from the `Object` class. In that case, all the `hashCode` method will do is convert the address of the object into an integer and return that number.
   - Two issues that may arise:
     Case 1, You overwrite `equals`, but not `hashCode`: Two keys that should be equal instead may map to different buckets (the default `hashCode` method returns the objects' addresses, which are different). So, you may have two equivalent keys in two different places. Violates the first rule in the contract.
     Case 2, You overwrite `hashCode`, but not `equals`: This actually isn't required for a hash code to be valid, but certain problems may arise from this. Example: Two keys that should be equal have been defined to have the same hash code. However, since `equals` is not correctly overwritten, the keys are not seen as equal, and two equivalent keys reside in the same linked list (should instead have been overwritten).

- An implication of writing valid hash codes: the hash function should not rely on mutable values, if possible (sometimes, you can't get around this however, like with Java's `LinkedList`)
  - **Why?** Consider the case where you've put an object into a `HashSet`, and you change that object's fields. If it is the case that the hash function depends on those fields, then you would have changed the hash code as well. You may not be able to find the object anymore
  - **Consequences**: There are three main ways to make sure this doesn't happen:
    (1) Ensure the object itself, or the fields that the hash function rely on are immutable.
    (2) Ensure you don't change an object after you've put it in a hash set.
    (3) Ensure mutable fields that the hash code relies on aren't subject to change. This means that the hash code *could* depend on mutable fields (e.g., a `Person` hash code could make use of the `name` field, even if that field isn't declared as `final`).

Goodness

1. The hash function should be valid (so good hash codes are a subset of valid hash codes)
2. The hash function should evenly distribute keys
   - **Why?** If the hash function does not evenly distribute keys, in the worst case, all items map to the same bucket, and the performance of `get` and `put` runs in time proportional to a linked list, $\theta(n)$
   - A good rule of thumb is that your hash code should depend on everything that uniquely defines an instance of the object.
3. The hash function should be relatively fast
   - We don't really discuss this much in our class, since this requirement is a little vague. The main idea here is to not write a hash code that grows too large relative to the size of your key. Be aware that several hash codes used by Java does run linear to the size of the key (example: `LinkedList` or `Stack`).

1 Concept Checks

- When resizing, why do we rehash each entry? In other words, why can we not just copy over all entries in some index i of the old array to the same index i of the new array?
- A useful fact to know is Java's STRING hash code. For any string s of size n, the hash code is calculated as $s[0] * 31^{n-1} + s[1] * 31^{n-2} + \ldots + s[n-1]$
- HASHMAP is an implementation of the MAP interface, while HASHSET is an implementation of the SET interface. Thus, both data structures have somewhat different guarantees and functionality. However, *internally*, a HASHSET is backed by a HASHMAP! Elements added into a HASHSET are stored as keys in the internal HASHMAP object. The value associated with those keys will be a constant, marking the key as 'PRESENT'.