CS2110: Hashing Problem Set

Instructors: Gries and Wehrwein *Made by: Your 2110 TAs*

1 Hash Functions

1. The best hash functions have the most amount of clustering. **True/False**
2. For a class with an equals function, function hashCode has to satisfy a certain property. What is that property?

3. Below is part of class HashMe. Circle the best hash function for it from the list below. Also, underline any valid hash functions (they could be terrible, but as long as they work). Assume that `timeOfDayInSeconds()` returns an int. Justify your rationale

   (a) `return 0;`

   (b) `return id;`

   (c) `return x;`

   (d) `return timeOfDayInSeconds();`

```
public class HashMe() {
    private final int id; // Won't change after construction, used in equals().
    private int x; // Might change after construction, not used in equals().

    @Override public int hashCode() {
        // What goes here?
    }

    // ... other methods ...
}
```

2 Time Complexity

You can look up answers to these questions in the Java API specs.

1. What is the time complexity of `HashSet.get()`?

2. What is the time complexity of `HashSet.add()`?
3. What is the time complexity of `HashMap.contains()`?
4. What is the time complexity of `HashMap.add()`?

3 Collision Resolution

Assume that the following array backs a HashSet. The top line represents the index and the bottom represents the value in the set. Each of the questions in this section are **independent of each other**. Assume **for this section** that you don't have to worry about resizing.

0	1	2	3	4	5
MA			NY		VA

1. What is the load factor of this array?
2. Assuming that we are using chaining and that CA hashes to index 3, which bucket does CA end up in, and what is the size of the list in that bucket?
3. Assuming that we are using linear probing and that CA hashes to index 3, which bucket does CA end up in, and how many state objects are in that bucket?
4. Assuming that we are using linear probing, CA hashes to index 3 and CA has already been inserted. How many buckets would linear probing need to probe if we were to insert AK, which also hashes to index 3?
5. Assuming that we are using quadratic probing, CA hashes to index 3 and CA has already been inserted. How many buckets would quadratic probing need to probe if we were to insert AK, which also hashes to index 3?

4 Rehashing Practice

The following is the initial configuration of an array backing a HashSet. On the right is a mapping from states to their hash values. Follow the directions below and draw the backing array at the end of each step. Assume that the largest load factor allowed in the HashSet is $\frac{1}{2}$ and that it uses linear probing. If you have to increase the length of the array, double its length.

0	1	2	3	4	5
MA					VA

State	Hash
MA	6
NY	4
VA	5
MO	14
AZ	2

1. Add NY to the HashSet

2. Add MO to the HashSet

3. Add AZ to the HashSet

4. Add AZ to the HashSet

5 Karma

1. Do a Google/Bing/Yahoo search for md5sum. Realize that hashing is used in many different aspects of computing. Read up a little. Understand that that's actually what you see in CMS when you upload a document.

2. Implement md5sum(). You get nothing for completing this other than the undying respect of your TAs, and you will be a legend. (We're joking, please don't try this.)

6 Function BugTree.equals

The children of a BugTree node are implemented as a Set, and Set has a function containsAll: set1.containsAll(set2) if every element of Set set1 is contained in Set set2. Therefore, we should be able to write function BugTree.equals as on the next page:

```
/** Return true iff this is equal to ob. */
public boolean equals(Object ob) {
  if (!(ob instanceof BugTree)) return false;
  if (ob == this) return true;
  BugTree bt= (BugTree) ob;
  if (root != bt.root) return false;
  if (children.size() != bt.children.size())  return false;

  // Return true iff children contains all the trees in bt.children.
  return children.containsAll(bt.children);
}
```

Please study the whole method. Look at its structure. Look at how simply it handles one case at a time, returning what is known based on the if-conditions. Was the first part of your function equals written so simply?

Now, ask yourself: What does the function call children.containsAll(bt.children) use to tell whether some tree in children equals some function in bt.children? It uses the equals function that appears above! This is a form of mutual recursion.

Now, this does not work yet because the children are implemented as HashSets and the major principle needed for hashing to work is not satisfied: If tree1.equals(tree2), then tree1.hashCode() must equal tree2.hashCode().

Please do the following to see this. First, copy this method into your A4 –you can comment out your own function equals. Second, copy the code at the end of this document into your class BugTreeTest. That code was used to test our own implementation of equals. Note how we have two methods to construct a tree and then procedure testEquals, which is quite simple. Now execute your testing class; you will see that it doesn't work.

Now put the following extremely simple function into BugTree. It hashes everything to the first location. Run your testing class again. Trees bt6 and bt7 will be equal, but bt8 and bt7 will not be (because they are not).

```
@Override public int hashCode() { return 0;}
```

Here are some questions to answer:

1. Suppose HashSet is implemented using Chaining and we use the hash code given above. Draw a diagram showing what the hash table looks like after inserting three trees t1, t2, and t3. How does this work? Name another Collections class that works in the same manner:

2. Suppose HashSet is implemented using linear probing and we use the hash code given above. Draw a diagram showing what the hash table looks like after inserting three trees t1, t2, and t3.How does this work? Name another Collections class that works in the same manner:

3. Change the hash function to return the hash function of the Human at the root of the tree. This will mean that two equal trees will hash to the same index. Test it to make sure your function equals stil works. Now, there are LOTS of tree that have the same root. So for the trees we might generate in A4, this hash function may not be so good. But note that the children of any such tree are all different humans, so that for the children sets, it's much better.

```
/** Return this tree: A [G, C [E [F]], B [D]] */
    public BugTree make_ABCDEFG() {
        BugTree bgr=  new BugTree(humans[0]);
        bgr.add(humans[0], humans[1]);
        bgr.add(humans[0], humans[2]);
        bgr.add(humans[1], humans[3]);
        bgr.add(humans[2], humans[4]);
        bgr.add(humans[4], humans[5]);
        bgr.add(humans[0], humans[6]);
        return bgr;
    }

    /** Return this tree:
     * 0      A
     *       / \
     * 1    B   C
     *     /   / \
     * 2  D   E   F
     *       /     \
     * 3    G       H     */
     public BugTree make_ABCDEFGH() {
        BugTree bgr=  new BugTree(humans[0]);
        bgr.add(humans[0], humans[1]);
        bgr.add(humans[0], humans[2]);
        bgr.add(humans[1], humans[3]);
        bgr.add(humans[2], humans[4]);
        bgr.add(humans[2], humans[5]);
        bgr.add(humans[4], humans[6]);
        bgr.add(humans[5], humans[7]);
        return bgr;
     }

     @Test
     public void testEquals() {
         // make_ABCDEFGH(): A [B [D], C [F [H], E [G]]]
         BugTree bt6= make_ABCDEFGH();
         BugTree bt7= make_ABCDEFGH();
         assertTrue(bt6.equals(bt7));
         // make_ABCDEFG(): A [C [E [F]], G, B [D]]
         BugTree bt8= make_ABCDEFG();
         assertFalse(bt8.equals(bt7));
     }
```