CSE332 Week 4 Section Worksheet

1. For each of the following types of hash-tables, insert the following numbers into an initially empty table of size 10 (use h(k) = k%tableSize as the primary hash function):

23, 27, 94, 33, 14, 3

1. Separate Chaining (with each bucket being an unsorted linked list). Here items are inserted at the beginning of the list, though the end would work just as well.
2. Double Hashing, with h2(k) = 7- (k%7) as the secondary hash function

 When done, give the load factor for the resulting table.

The load factor is the same for both: .6 (6 elements divided by a table size of 10).

1. In 1.b., what’s wrong with h2(k) = k%7 as a secondary hash function?

When k is a multiple of 7 it evaluates to 0 - this is very problematic for a secondary hash function. That means the 2nd, 3rd, 4th, etc. indices checked would all be the same as the initial index.

1. Imagine we have a hash table with a poorly chosen primary hash function (or we just get very unlucky with our insertions) and all the keys are mapped to the same index by our primary hash function. How would this affect a table using separate chaining? How would this affect a table using double hashing (assume that the secondary hash function distributes the keys fairly evenly)?

With separate chaining we’d have all elements in a single unsorted linked list – operations would take linear time, in the worst case.

If the table used double hashing, we’d have a collision at first, but the location after that would depend on the secondary hash function. If the secondary hash function were pretty good, we would get a fairly good distribution of indices after that initial index.

1. a. We would choose the hash table under most circumstances, as it is likely to be faster most of the time.

b. While the hash table would be a better choice for the most part, it doesn’t give us a nice O(logn) bound for lookups/inserts like the AVL tree. For the hash table, we may get linear time performance due to bad-luck, or while rehashing. So, in time-critical applications, where operations \*must\* run in logn or less, an AVL tree would make sense.

As someone pointed out in section, we could build a hash table with an AVL tree in each cell instead of a linked list – this would give us a better guarantee on time (except for rehashing). Although, if our primary hash function is that bad for our keys, we should probably choose a new hash function anway.