**COMP232 – Data Structures & Problem Solving**

**Homework #8**

**Hashing**

1. Explain the importance of having a hash function that approximately satisfies the simple uniform hashing assumption.

2. Consider the following sequence of Map operations to be applied to a hash map with capacity 10 and the simple hash function h(k)=k:

add(57, “A”)

add(107, “X”)

add(27, “Q”)

add(37, “B”)

add(88, “M”)

add(89, “R”)

add(100, “Z”)

add(66, “N”)

a. Show the contents of the hash map after this sequence of operations if it were using open addressing with linear probing. Assume that the hash table does not resize.

b. What elements are in a primary cluster?

c. Show the contents of the hash map after this sequence of operations if it were using closed addressing with chaining. Assume that the hash table does not resize.

d. What is the load factor of the hash tables in parts a and c?

3. In class we saw that for a hash map using closed addressing with chaining the average case for the get operation had an asymptotic bound of O(1).

a. What is the worst-case asymptotic bound for get in this implementation of hashing? Explain.

b. Explain how could you obtain a worst case bound of O(lg n) for the get operation with closed addressing?

c. Explain why the added overhead of your approach in part b be worth it, or not?

4. Discuss the major tradeoff that exists between the use of open and closed addressing.

5. As we saw in class the hash map operations with open hashing run in O(n/m) time. If we can assume Simple Uniform Hashing and ensure that n<=m then this reduces to O(1) expected time per operation. The current implementation in CS232OpenHashMap does not guarantee that n<=m because the hash table is never resized. It thus does not guarantee O(1) average time per operation. Add resizing to the put operation such that the hash table is doubled in size whenever the load factor equals or exceeds the MAX\_LOAD\_FACTOR. The No5Tests class contains tests that you can use to check your implementation of this functionality. All but 2 of the tests pass using the provided code because they do not rely in the resize functionality. These tests are included, along with those that test the resizing, to ensure that the resize operation does not break the existing functionality.

6. Complete the remove method in the CS232OpenHashMap so that it executes in O(1) expected time. In practice, the remove method will decrease the size of the hash table to reduce wasted space if the load factor becomes too small. It is not necessary to implement this reduction in the size of the hash table for this problem. The No6Tests class contains tests that you can use to check your implementation of this functionality. All of the tests from No5Tests are also run to ensure that the remove operation does not break the existing functionality.

7. Implement the following methods in the CS232ClosedHashMap using linear probing:

a. put and get. You do not have to implement the resizing of the hash table in the put method. The No8aTests class contains tests that you can use to check your implementation of this functionality.

b. remove with appropriate modifications to put and get to deal with deleted elements. The No8bTests class contains tests that you can use to check your implementation of this functionality.

8. Consider each of the following possible hash functions where m is the capacity of the underlying hash table:

a. h(k) = k/m

b. h(k) = 1

c. h(k) = (k + rnd.nextInt(m))

rnd.nextInt(m) gives a uniformly distributed random integer in the range [0…m).

For each of these hash functions, discuss why it would not be a good hash function to use.

1. h(k) = k/m

b. h(k) = 1

c. h(k) = (k + rnd.nextInt(m))

rnd.nextInt(m) gives a uniformly distributed random integer in the range [0…m).