CSC317 Fall 2022
Instructor: Odelia Schwartz
Assignment 5

Myles Greene-Beaupre

Please complete your own homework and do not copy solutions from the web.

Due ***Thursday October 6 2022*** by midnight, on Blackboard. Note: Some of the questions are based on the Cormen textbook. Points listed below are for relative weighting of the questions in this assignment. Each assignment will in the end be weighted equally.

1. **[5 points] Consider a hash table with collisions resolved by chaining. Assume simple uniform hashing.**

(a) Consider the case that the number of keys (corresponding to elements) n stored in the hash table is 50 times the number of slots m. What is the average time for an unsuccessful search?

The load factor is given by n/m where n is the number of elements and m is the number of slots. Since we have 50 times more elements than slots, we can say our load is 50m/m = 50. Therefore, on average we have 50 elements per linked list, and the average runtime for an unsuccessful search is given by θ(load + 1) = θ(50 + 1) where the 1 represents the cost to run the hash function. However, since n and m are of the same order (n is not equal to something like m^3, for example), we can ignore constants and say our average runtime is simply θ(1) or constant time.

(b) Consider the case that there are m slots in the hash table, and that the number of keys stored in the hash table is m^2. What is the average time for an unsuccessful search?

In this case, when we calculate our load factor we get n/m = m^2/m = m. So our linked lists have, on average, length m. Therefore, our average runtime for an unsuccessful search is θ(load + 1) = θ(m + 1). Ignoring constants, this gives us a runtime of θ(m).

2. **[5 points] Explain why the average search time in a hash table with collisions resolved by chaining is not always $O(1)$.**

   We only get an average search time of O(1) when our load factor is a constant. This happens when n and m are of the same order, for example if we have a number of elements equal to four times our number of slots, we can see that our load is 4m/m, and the m cancels out entirely and we just get a load of 4, which is a constant. With something like a number of elements equal to m^4, we have m^4/m = m^3, which is an exponential load factor (not constant), so our average search time could not possibly be a constant O(1). The load factor

allows us to find our average search time because the load factor gives us the average length of our linked lists, which we have to search through in linear time in order to find the element we are looking for.

3. **[5 points] Consider an open-address hash table with uniform hashing.**
(a) **What is the expected number of probes (at most) in an unsuccessful search when the load factor is 6/7?**

The expected number of probes in an unsuccessful search is equal to 1 / (1 – load) = 1 / (1 – 6/7) = 1 / (1/7) = 7. So we would expect at most 7 probes during an unsuccessful search in these conditions.

(b) **What is the expected number of probes (at most) in an unsuccessful search when the number of keys in the array are 5 and the number of slots 7?**

We can calculate our load with n/m = 5/7. Then, our expected number of probes is equal to 1 / (1 – load) = 1 / (1 – 5/7) = 1 / (2/7) = 7/2 = 3.5. So we would expect at most 3.5 probes during our unsuccessful search, which doesn't really make sense because there is no such thing as half of a probe, but it gives us a ballpark idea of how many probes we can expect on average.

4. [5 points] Consider a hash table with collisions resolved by open addressing, with n keys and m slots. What is the worst case time to search for a key? Explain your answer.

In an open addressing hash table, our worst case would be when n = m (hash table is completely full) and we have really bad clustering such that when you go to search for an item, you do not find it at its calculated hash value index but instead you have to go through each element in the hash table to find it. For example, assume you calculate a hash value that corresponds to the first index of the array, but the element you are looking for is not there. If the element you are looking for resides at the last index of the array, or if it is not in the array at all (unsuccessful search), then you will end up going through all n elements and checking each one to see if it is the one you are searching for. Thus, this worst case scenario would result in a runtime of O(n).