Problem Set 7

Data Structures and Algorithms, Fall 2021

Due: November 11, in class.

Problem 1

Suppose that we have a hash table with n slots, with collisions resolved by chaining, and suppose that n keys are inserted into the table. Each key is equally likely to be hashed to each slot. Let M be the maximum number of keys in any slot after all the keys have been inserted. Your mission in this problem is to prove an $O((\lg n)/\lg\lg n)$ upper bound on $\mathbb{E}[M]$, the expected value of M.

(a) Fix an arbitrary slot, let Q_k be the probability that exactly k keys hash to this slot. Prove that Q_k is less than e^k/k^k. *(Hint: you may need to use Stirling's approximation.)*

(b) Let P_k be the probability that $M = k$, that is, the probability that the slot containing the most keys contains k keys. Prove that $P_k \leq nQ_k$. *(Hint: union bound.)*

(c) Prove that there exists a constant $c > 1$ such that $P_k < 1/n^2$ when $k \geq (c \lg n)/\lg\lg n$.

(d) Prove that $\mathbb{E}[M] = O((\lg n)/\lg\lg n)$. *(Hint: recall how we bound the cost for searching operations when discussing skiplist.)*

Problem 2

(a) Consider a version of the division method in which $h(k) = k \mod m$, where $m = 2^p - 1$, k is a character string interpreted in radix 2^p, and $p > 1$ is an integer. (For example, if we use the 7-bit ASCII encoding, then $p = 7$ and string AB has key value $65 \times 128 + 66$.) Show that if we can derive string x from string y by permuting its characters, then x and y hash to the same value.

(b) Consider an open-address hash table. Suppose that we use double hashing to resolve collisions—that is, we use the hash function $h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$. Show that if m and $h_2(k)$ have greatest common divisor $d \geq 1$ for some key k, then an unsuccessful search for key k examines $1/d$ fraction of the hash table before returning to slot $h_1(k)$. Thus, when $d = 1$, meaning m and $h_2(k)$ are relatively prime, the search may examine the entire hash table.

Problem 3

Many theoretical analysis of hashing assumes *ideal random hash functions*. Ideal randomness means that the hash function is chosen *uniformly* at random from the set of *all* functions from U to $\{0, 1, \ldots, m-1\}$. Intuitively, this means for each new item x, we roll a new m-sided die to determine the hash value $h(x)$.

Suppose your boss wants you to find a *perfect* hash function for mapping a known set of n items into a table of size m. A hash function is *perfect* if there are no collisions: each of the n items maps to a different slot in the hash table. Notice a perfect hash function is only possible if $m \geq n$. After cursing your algorithms instructor for not teaching you about (this kind of) perfect hashing, you decide to try something simple: repeatedly pick ideal random hash functions (with replacement) until you find one that happens to be perfect.

(a) Suppose you pick an ideal random hash function h. What is the *exact* expected number of collisions, as a function of n (the number of items) and m (the size of the table)?

(b) What is the *exact* probability that a random hash function is perfect?

(c) What is the *exact* expected number of different random hash functions you have to test before you find a perfect hash function?

(d) What is the *exact* probability that none of the first N random hash functions you try is perfect?

(e) How many ideal random hash functions do you have to test to find a perfect hash function *with high probability* (that is, with probability at least $1 - 1/n$)?

Problem 4

Suppose we perform a sequence of n operations on a data structure in which the i-th operation costs i if i is an exact power of 2, and 1 otherwise. Prove that the amortized cost per operation is $O(1)$.

Problem 5

Suppose we can insert or delete an element into a hash table in $O(1)$ time. In order to ensure that our hash table is always big enough, without wasting a lot of memory, we will use the following global rebuilding rules: (1) after an insertion, if the table is more than $3/4$ full, we allocate a new table twice as big as our current table (this takes $O(1)$ time), insert everything into the new table, and then free the old table (this takes $O(1)$ time); (2) after a deletion, if the table is less than $1/4$ full, we allocate a new table half as big as our current table (this takes $O(1)$ time), insert everything into the new table, and then free the old table (this takes $O(1)$ time). Now, prove that for any sequence of insertions and deletions, the amortized time per operation is still $O(1)$.