6.856 — Randomized Algorithms

Spring Term, 2021

March 10, 2021 — Problem Set 4, Due 3/17

Problem 1. Improving the two-choice bound.

(a) In class we proved that the two-choices approach improves the maximum load to $O(\log\log n)$. A generalization is that choosing the least loaded of d choices reduces the maximum load to $O(\log_d \log n)$. Explain what changes to the proof are needed to derive this result. Give only the diffs; do not bother writing a complete proof.

(b) **Optional:** Suppose that instead of making two choices at random, you divide the bins into a left and right half and break all ties by putting items in the left bin. Show that the maximum load improves by a constant factor, from $O(\log\log n/\log 2)$ to $O(\log\log n/2\log\phi)$ where $\phi = (1+\sqrt{5})/2$ is the golden ratio. Again, only the diffs are required. **Hint:** for the number of height i bins on each side, use different recurrences β_i for the left side and γ_i for the right side. Show that $\beta_{i+1} \le c_1\beta_i\gamma_i/n$ while $\gamma_{i+1} \le c_2\beta_{i+1}\gamma_i/n$.

(c) **Optional:** Generalize (b) to d bins, showing a load of $O((\log\log n)/d)$.

Problem 2. A set of variables is *k-way independent* if every subset of at most k of them is mutually independent.

(a) Suppose n balls are placed uniformly at random in n bins. Let $C_{ijk} = 1$ if balls i, j, k are all placed in the same bin. Compute $E[\sum C_{ijk}]$.

(b) Suppose a bin contains m items. Give a lower bound on $\sum C_{ijk}$.

(c) Conclude that the maximum bin size is probably $O(n^{1/3})$.

(d) Argue this result holds so long as ball locations are 3-way independent.

(e) Consider the following hash function from $Z_p \to Z_p$. Choose three random values $a, b, c \in Z_p$ and define $h_{abc}(x) = ax^2 + bx + c$. Prove that for any three $x_1 \ne x_2 \ne x_3 \in Z_p$, the hash values $h_{abc}(x_i)$ are mutually independent. In other words, h_{abc} generates three-way-independent values. **Hint:** you may use without proof the determinant of a Vandemonde Matrix.

This result generalizes in the obvious way to k-way independent hash functions. In particular, setting $k = O(\log n)$ is sufficient to yield a maximum load of $O((\log n)/\log\log n)$.

Problem 3. Cuckoo hashing is nice, but does cost a factor of two in space (our analysis required $2m < n$ to work). Develop a related approach that uses less space while still guaranteeing worst-case constant-time lookups. Use the following ideas:

- Probing more than twice in a table increases the chances of finding an empty cell.
- If after some probes you fail to find an empty cell, move the failed item into an "overflow" table that uses cukoo hashing

Use this to achieve constant-time lookups using only $(1+\epsilon)m$ space for any constant ϵ. Determine the best tradeoffs you can between number of probes required and amount of space used. You may use the uniform hashing assumption.

Problem 4. Another problem with the distributed caching system we discussed is that it is hard to keep track of which machines are up or down. Different clients may learn about different caches' states at different times. And if different clients have different opinions about which caches are up, they will have different opinions about which cache to contact to retrieve a given item. Suppose that each of the n clients knows about at least half of the caches that are up at a any given time. Prove that with high probability, $O(\log n)$ caches will be asked to deal with any given data item, regardless of the number m of caches. You may use the uniform hashing assumption.

Problem 5—This problem should be done without collaboration. In class we argued that a consistent hash function (without replication) could be evaluated in $O(\log m)$ time by putting the bucket IDs in a binary search tree. Argue that this can be improved to $O(1)$ time in expectation, and $O(\log\log m)$ with high probability, using only $O(m)$ space for the data structure. **Hint:** Use the fact that the bucket positions are random, and consider breaking the ring into m equal sized intervals that could be represented as a size-m array.

Problem 6—Optional. In class we showed how to construct *perfect hash functions* that can be evaluated in constant time while producing no collisions at all. Perfect hashing is nice, but does have the drawback of taking quadratic space. Consider the following alternative approach to producing a perfect hash table that uses less space. Consider the following alternative approach to producing a perfect hash function with a small description. Define *bi-bucket hashing*, or *bashing*, as follows. Given n items, allocate *two* arrays of size $n^{1.5}$. When inserting an item, map it to one bucket in *each* array, and place it in the emptier of the two buckets.

(a) Suppose a random function (i.e., all function values are uniformly random and mutually independent) is used to map each item to buckets. Give a good upper bound on the expected number of collisions (i.e., the number of pairs of items that are placed in the same bucket).

Hint: What is the probability that the k^{th} inserted item collides with some previously inserted item?

(b) Argue that bashing can be implemented efficiently, with the same expected outcome as in (a), using the ideas from 2-universal hashing.

(c) Conclude an algorithm with linear expected time (ignoring array initialization) for identifying a perfect bash function for a set of n items. How large is the description (the space required to encode the function) of the resulting bash function?

(d) **(OPTIONAL)** Generalize the above approach to use less space by exploiting tri-bucket hashing (trashing), quad-bucket hashing (quashing), and so on.