Warm-up

Problem 1. Consider a hash table of size $N > 1$, and the hash function such that $h(k) = k \bmod 2$ for every k. We insert a dataset S of size $n < N$. After that, what is the typical running times of GET for chaining and open addressing (as a function of n)?

Solution 1. $\Theta(n)$ for both. Since the hash function maps all elements to only two values, 0 and 1, there are at least $n/2$ elements hashed to the same value – for instance, 0. In the case of chaining, that means that at least $n/2$ elements of S are now part of a linked list, and so doing a GET on any of those boils down to doing a linear search into a linked list of size $\Theta(n)$: $\Theta(n)$. In the open addressing case, similarly we have a sequence of at least $n - 1$ contiguous positions corresponding to those elements: indeed, everything is mapped to either 0 or 1, so after the first 0 everything collides at 1 and every single following index until an empty slot is found (and if there is no element mapped to 0, then all n insertions collide at 1), and so we end up doing a search in an unsorted array of size $\Theta(n)$.
Upshot: the choice of hash function matters! Don't come up with your own, it may be a bad idea...

Problem 2. Suppose you are given a hash function h mapping 10-digit integers to integers in $\{1, 2, \ldots, 10000\}$. Show that there is some dataset S of size $1,000,000$ such that all keys of S are hashed to the *same* value.

Solution 2. There are 10^{10} different 10-digit integers. By the Pigeonhole Principle, there exists some $i \in \{1, 2, \ldots, 10000\}$ such that $|h^{-1}(i)| \geq \frac{10^{10}}{10000} = 10^6$ (where $h^{-1}(i) \subseteq S$ is the set of keys k such that $h(k) = i$) . Let S be this set $h^{-1}(i)$.
Upshot: for every hash function h, there exists some dataset $S = S(h)$ (depends on the hash function!) for which h is arbitrarily bad. So all we can ask for is that hash functions be good for *most* (i.e., "typical" for our applications) datasets, not *all*.

Problem 3. Work out the details of implementing cycle detection in cuckoo hashing based on the number of iterations of the eviction sequence.

Solution 3. There are $2N$ entries in total, so if the cuckoo eviction process runs more than $4N$ times, we are guaranteed to have a cycle: there can be $2N$ shifts to move all entries to their alternative position and then another $2N$ to move everything back to their initial positions. By adding a counter that is incremented every time we evict an entry, this is easily checked in $O(1)$ time after every eviction.

Problem 4. Work out the details of implementing cycle detection in cuckoo hashing based on keeping a flag for each entry.

Solution 4. We augment the hash table entries with a boolean attribute called flag. Assume that at the start of the put routine all entries are unflagged (i.e., all flags are

set to false). Suppose we are trying to put a new element x into the hash table and that the element could store in hash entries i or j. To test if there exists an eviction path starting at, say, i, we design a recursive auxiliary routine TEST-CHAIN(x, i).

This auxiliary routine first checks if i is empty; if that is the case, then the test is successful. If the entry is not empty, it checks if the entry is flagged; if that's the case the test is unsuccessful. Finally, if the entry is not empty and not flagged, let y be the item currently stored at i and k be the alternative entry for y. We flag entry i, and call TEST-CHAIN(y, k). Regardless of the outcome (successful or unsuccessful) before returning we unflag the entry previously flagged so that all entries are unflagged at the start of the next put call.

If both TEST-CHAIN(x, i) and TEST-CHAIN(x, j) are unsuccessful, then it is not possible to put x into the has table. The running time is proportional to the length of the eviction chain tests.

Note that the unsuccessful eviction chain test could be much longer than the successful one. We could modify the algorithm to interleave the search for a chain so that we do work proportional to the length of the shortest successful chain (if one exists) but this would complicate the algorithm a great deal and would require us to keep two flags. Not really worth the effort.

Problem solving

Problem 5. Design a sorted hash table data structure that performs the usual operations of a hash table with the additional requirement that when we iterate over the items, we do so in the order in which they were inserted into the hash table. Iterating over the items should take $O(n)$ time where n is the number of items stored in the hash table. Your data structure should only add $O(1)$ time to the standard put, get, and delete operations.

Solution 5. In addition to the hash table, we keep a doubly linked list of the entries and augment each entry in the hash table to also have a pointer to its position in the list.

When we need to put a new item, after inserting it into the hash table in the usual way, we add it to the end of the list and set the pointer of the entry in the hash table accordingly in $O(1)$ time.

When we remove an item, after removing it from the hash table in the usual way, we use the pointer to the doubly linked list to remove it from there as well in $O(1)$ time.

When we update an existing item, after updating it in the usual way, we move its position in the list to the end in $O(1)$ time or we leave it in place depending on how we want to interpret this case; i.e., we care about the order of when the key of the item arrived or when the value of the item arrived.

Whenever we need to iterate over the entries, we use the linked list. Since iterating through all elements of a doubly linked list takes $O(n)$ time, this satisfies our requirements.

Problem 6. Given an array with n integers, design an algorithm for finding a value that is most frequent in the array. Your algorithm should run in $O(n)$ expected time.

Solution 6. Keep a hash table with $2n$ entries using linear probing where the keys are the integers in the input array and the value is the frequency of the key.

We scan the integers in the array, updating their frequency in our hash table. That is, when processing k, we first try to get the entry for k. If there is no entry, we put $(k, 1)$; otherwise, if there is already an entry (k, f) we updated with $(k, f + 1)$. At the end we do a scan of the table to find the integer with maximum frequency.

Given that the load factor of the hash table is $\leq 1/2$, the hash table operations take $O(1)$ expected time. Finally, scanning the whole hash table to find the maximum frequency integer take $O(n)$ time.

It is worth noting that we can avoid the scan of the hash table if we also keep track of the maximum frequency item we have seen so far. Every time we update the frequency of a number, we check if its frequency is larger than the maximum frequency so far, and if so, we update it.

Problem 7. A multimap is a data structure that allows for multiple values to be associated with the same key. The method GET(k) should return all the values associated with key k. Describe an implementation where this method runs in $O(1 + s)$ expected time, where s is the number of values associated with k.

Solution 7. *(Sketch)* Each entry, instead of having a single value, has a linked list of values associated with the key. When putting a new item (k, v) we add v to the list in the entry associated with k. When getting a key k, we find it in $O(1)$ expected time and we go over the list in the entry associated with k (which takes $O(s)$ time).

Problem 8. Suppose that you have a group of n people and you would like to know if there are two people that share a birthday. Design an $O(1)$ time algorithm that given the information about the n people's birthdays, finds a pair that shares a birthday, or reports that no such pair exists.

Solution 8. Map the birthday to a number in $[1, 365]$ and use a hash function on integers to build a hash table of size $2 * 365$ using linear probing. We iterate over the elements in the list, for each person p we treat their birthday as a key k and try to get the entry with k. If there is no such key we add (k, p) to the hash table; otherwise, if we find the item (k, p') we have our pair p, p' of people sharing a birthday and we can stop.

If we scan the whole set of n people without finding a match, we report that none share a birthday.

For the time complexity, we note that if $n > 365$ we are guaranteed to find a matching pair of people in the first 366 entries of the array so the algorithm terminates after $O(1)$ iterations and each iteration takes $O(1)$ time since the hash table has $O(1)$ size.

Problem 9. In computational linguistics, texts (such as a book or an article) are modelled as a sequence of words. A k-gram is a sequence of k consecutive words.

A common task in language modelling requires that we compute the frequency of all k-grams that appear in the text.

Given a text with n words, design an $O(n)$ expected time algorithm that computes the frequency of all k-grams that appear at least once in the text.

Solution 9. We use an approach similar Problem 6, but instead we use the k-grams in the text as our keys. To design a hash function for these keys we can use polynomial evaluation (since the order of the words of the k-gram matters).

Note that there can be at most $n - k + 1$ distinct k-grams in the text, so keeping a hash table of size $2n$ using linear probing yields the desired running time.