19.1 The dictionary problem

Consider the following data structural problem, usually called the *dictionary problem*. We have a set of items. Each item is a (key, value) pair. Keys are in the range $\{1,\ldots,U\}$, and values are arbitrary. A data structure supporting the following operations is called a *dynamic dictionary*.

- **insert(k,v):** insert a new item into the database with key k and value v. If an item with key k already exists in the database, update its value to v.
- **delete(x):** delete item x from the database (we assume x is a pointer to the item)
- **query(k):** return the the value associated with key k, or `null` if key k is not in the database

The fact that we must not only support **query** but also **insert** and **delete** makes the above variant of the dictionary problem *dynamic*. (Dynamic variants of data structure problems are ones that support the data set being updated.)

19.1.1 Dynamic dictionary via hashing with chaining

One randomized solution to the dynamic dictionary problem is *hashing with chaining*. A hash family $\mathcal{H}$ is a set of functions $h : \{1,\ldots,U\} \to \{1,\ldots,m\}$. The basic idea of this solution is to maintain an array of size m where an item with key k is stored in the $h(k)$th position of the array. Unfortunately this does not work as is due to *collisions*: two distinct keys k,k' in our database may have $h(k) = h(k')$, so how do we resolve this collision? In hashing with chaining, we initialize some array A of size m, where $A[i]$ stores the pointer to the head of a doubly linked list. We pick some h uniformly at random from $\mathcal{H}$. Then $A[i]$ will be a doubly linked list containing (key,value) pairs of *all* items whose key k satisfies $h(k) = i$. That is, colliding items are stored in the same linked list. To insert a (k,v) pair, we simply insert this new item to the head of the list at $A[h(k)]$. To query k, we traverse the list at $A[h(k)]$ until we find an item with key k (or discover that none exists). Deletion splices the item out of the doubly linked list when it finds it.

Of course, the above scheme will not always be efficient. For example, consider $\mathcal{H}$ only containing a single hash function h satisfying $h(i) = 1$ for all i. Then our entire data structure is a single linked list. However, if $\mathcal{H}$ is *nice* in a certain way and m is sufficiently large, we will be able to prove good guarantees.

Definition 19.1 *We say a family $\mathcal{H}$ of hash function mapping $\{1,\ldots,U\}$ into $\{1,\ldots,m\}$ is* universal *if for all $1 \le x < y \le U$,*

$$\mathbb{P}_{h\in\mathcal{H}}(h(x) = h(y)) \le \frac{1}{m},$$

where h is chosen uniformly at random from $\mathcal{H}$.

Example 19.2 The set $\mathcal{H}$ of *all* functions mapping $\{1,\ldots,U\}$ into $\{1,\ldots,m\}$ is universal and has $|\mathcal{H}| = m^U$. That is, an element $h \in \mathcal{H}$ can be described using $\log|\mathcal{H}| = U\log m$ bits of space.

Example 19.3 Another option is to pick some prime $p \ge U$ and define $h_a(x) = (ax \mod p) \mod m$. Then we let $\mathcal{H} = \{h_a : 0 < a < p\}$. Then $|\mathcal{H}| = p-1$, and we can choose p for example to be at most polynomial in U, so a random $h \in \mathcal{H}$ can be represented using $\log|\mathcal{H}| = O(\log U)$ bits. The analysis of this scheme requires some abstract algebra beyond the scope of this course, but this family turns out to be "almost universal", in the sense that for any $x \ne y$, $\mathbb{P}_{h\in\mathcal{H}}(h(x) = h(y)) \le C/m$ for some constant C that tends to 1 as p grows large (and for most applications, this weaker property suffices).

Theorem 19.4 *Consider a hash table with chaining on a database with n items using a universal hash family $\mathcal{H}$ with $m \ge n$. Then executing a query takes expected time $O(1+T)$, where T is the cost of evaluating a hash function $h \in \mathcal{H}$.*

Proof: Let the query be on some key k. A query performs one hash evaluation, taking time T, followed by traversing the list at $A[h(k)]$. For $i = 1,\ldots,n$ let X_i be a random variable which is 1 if the ith key k_i in the database has $h(k_i) = h(k)$, and X_i is 0 otherwise. The the running time of a query is proportional to

$$T + \sum_{i=1}^{n} X_i.$$

Thus by linearity of expectation, the expected running time of a query is proportional to

$$T + \sum_{i=1}^{n} \mathop{\mathbb{E}}_{h} X_i = T + \sum_{i=1}^{n} \mathbb{P}_{h\in\mathcal{H}}(h(k_i) = h(k)) \le T + \sum_{i=1}^{n} \frac{1}{m}$$

Noting that $m \ge n$, the above is $T+1$. ∎

19.1.2 Static dictionary via perfect hashing

In this section we study the static dictionary problem and describe a solution based on two-level perfect hashing. Sometimes it is also called "FKS perfect hashing" after its inventors: Fredman, Komlós, and Szemerédi. Recall

in the static dictionary problem, there are no item insertions or deletions, but rather the set of items S is fixed in the beginning. Letting $K \subset U$ denote the keys that appear amongst (key, value) pairs in S, we say a hash function $h : [K] \to [m]$ is a *perfect* hash function for S if it is injective on K, that is $\forall k, k' \in K, k \neq k' \Rightarrow h(k) \neq h(k')$. If we could efficiently find a perfect hash function h for S, with a hash function h that is quickly computable, it would imply a very simple solution to the static dictionary problem: namely initialize an array A of length m, and for any $(k,v) \in S$, store v in $A[h(k)]$.

In this section we describe a solution to the static dictionary problem which yields $O(1)$ worst case query time, $m = O(n)$ space, and $O(n)$ expected preprocessing time to create the data structure.

To begin, recall the birthday paradox where, assuming random birthdays, you shouldn't be surprised that two people have the same birthday when you have at least $\approx \sqrt{365}$ people in one room.

19.1.2.1 Quadratic space

If we were willing to make a table whose size is quadratic in size n. Then we can easily construct a perfect hash value. Let $\mathcal{H}$ be a universal hash family and $m = n^2$.

Claim: If $\mathcal{H}$ is universal and $m = n^2$, then $P_h(\text{no collisions}) \geq 1/2$ when using hash function $h \in \mathcal{H}$.

Proof: In order for a collision to occur, elements x, y must equal each other. Of the $\binom{n}{2}$ pairs, the chance they collide is $\leq 1/m$ by definition of universal hash family. Then the probability a collision occurs $P_h(\text{collision occurs}) \leq \binom{n}{2}/m < 1/2$.

This is the opposite of the birthday paradox since we are looking for the probability that no pair of people has the same birthday. The complement then shows that the probability of no collisions must $\geq 1/2$. Our method then involves trying a random h from $\mathcal{H}$. If we have any collisions, we pick another h. On average, we would only need to do this twice.

19.1.2.2 Linear space

Let's say we want to get a better space complexity. The general idea is that we are going to perform a 2-level hashing scheme: first we pick a random function $h_i \in \mathcal{H}$ from the universal hash family and hash all elements. Let B_i represent the number of items that hash to bucket i. We then wish to keep picking random $h_i \in \mathcal{H}$ until we find h_i such that

$$\sum_{i=1}^{m} B_i^2 \leq 4n$$

Note that we do not know how long it will take to find h_i that fulfills the above condition. Once we do find our satisfactory h_i, we can use the birthday paradox, which states that if we have n possible days in the year, once we have $\sqrt{n}$, we can expect to find two people with the same birthday. Likewise, if we have significantly fewer than $T = \sqrt{n}$, if we taking values between 1 and T^2 and we have much less than T people, we can be pretty sure that there isn't a collision.

For bucket i, we can then hash all B_i of the values in it to $1, \ldots, B_i^2$ using $h_i : [U] \to [10B_i^2]$, then by the birthday paradox, there are probably no collisions. To summarize, the entire method is then

(1) First we take our elements that take on values $v \in [n]$ and hash them using first-level hash function $h : [U] \to [m]$ such that

$$\sum_{i=1}^{m} B_i^2 \leq 4n$$

where B_i is the number of items in the bucket i. This creates first-level table A.

(2) Next, we use m second-level hash functions $h_1, \ldots, h_m$ and m second-level tables $A_1, \ldots, A_m$. Note that we pick h_i such that there are no collisions in A_i.

Thus, we can ensure no collisions in the 2-level hash table structure. Our space complexity is $O(m + \sum_{i=1}^{m} 10B_i^2)$ since we need m first-level buckets and $\sum_{i=1}^{m} 10B_i^2$ second-level buckets. Note that since we chose h such that the sum of $\sum_{i=1}^{m} B_i^2 \leq 4m$, our space complexity is $\Theta(n)$.

Let's back up though. We initially said to keep picking h such that $\sum_{i=1}^{m} B_i^2 \leq 4n$. How long will that take?

Claim: $P_h(\sum_{i=1} B_i^2 > 4n) \leq 1/2)$

Proof: Let $Q_{ji} = 1$ if item j hashes to i and 0 otherwise. We can rewrite B_i as

$$\begin{aligned} B_i &= \sum_{j=1}^{n} Q_{ji} \\ B_i^2 &= \left(\sum_{j=1}^{n} Q_{ji}\right)^2 = \sum_{j=1}^{n} Q_{ji}^2 + \sum_{j\neq k} Q_{ji}Q_{jk} \\ &= \sum_{j} Q_{ji} + \sum_{j\neq k} Q_{ji}Q_{ki} \qquad (19.1) \\ \sum_{i=1}^{m} B_i^2 &= \sum_{i=1}^{m}\sum_{j=1}^{n} Q_{ji} + \sum_{i=1}^{m}\sum_{j\neq k} Q_{ji}Q_{ki} \\ &= n + \sum_{i=1}^{m}\sum_{j\neq k} Q_{ji}Q_{ki} \qquad (19.2) \\ \mathbb{E}\left(\sum_{i=1}^{m} B_i^2\right) &= n + \mathbb{E}\left(\sum_{i=1}^{m}\sum_{j\neq k} Q_{ji}Q_{ki}\right) \\ &= n + \mathbb{E}\left(\sum_{i=1}^{m}\sum_{j\neq k} Q_{ji}Q_{ki}\right) \\ &= n + n - 1 = 2n - 1 \qquad (19.3) \end{aligned}$$

Above, we show a few significant and potentially non-intuitive steps. In (19.1), we can drop the square of $\sum_{j=1}^{n} Q_{ji}$ since Q_{ji} is either 0 or 1. In (19.2), we can reorder the summations: we know that $\sum_{i=1}^{m} Q_{ji} = 1$ since item j will hash to exactly one of the m values. We then find $\sum_{j=1}^{n} 1 = n$.

For (19.3), we know that $Q_{ji}Q_{ki}$ will have a product of 1 if and only if $h(j) = h(k)$, meaning j and k hash to the same bucket. Our resulting term is then

$$\mathbb{E}\sum_{j\neq k}[1 \text{ if } h(j) = h(k)]$$

We can then linearize the expectation inside the summation, yielding the probability that the two item j,k collide, which is exactly universal hashing. The probability of collision $\leq 1/m$, and there are $n(n-1)$ possibilities since there are n choices for j and $n-1$ choices for k, giving us

$$\sum_{j\neq k} P(\text{collide}) = \frac{n(n-1)}{m}$$

Since $m = n$, we get that $\sum_{j\neq k} P(\text{collide}) = n - 1$.

From Markov's inequality, we get

$$\begin{aligned} P(X > \lambda\mathbb{E}X) &< \frac{1}{\lambda} \\ P\left(\sum_{i=1}^{m} B_i^2 > 4n\right) &< \frac{1}{2} \end{aligned}$$

This clears up the mystery of why we chose 4 as a coefficient in $\sum_{i=1}^{m} B_i^2 \leq 4n$: because it's twice the expectation.

Now we know when picking h randomly from $\mathcal{H}$, we have to pick an expected two times to fulfill our condition. Picking h_i is a similar story.

Claim: When picking $h_i : [U] \to [10B_i^2]$, $P(\exists \text{ collision}) < 1/2$

Proof: We define again $X_{jk} = 1$ if j,k collide under h_i and 0 otherwise. The expected number of collisions is then

$$\begin{aligned}
\mathbb{E}(\#\text{ of collisions}) &= \mathbb{E}\sum_{j=1}^{B_i}\sum_{k=1}^{j-1} X_{jk} \\
&= \sum_{j=1}^{B_i}\sum_{k=1}^{j-1} \mathbb{E}X_{jk} \\
&= \sum_{j=1}^{B_i}\sum_{k=1}^{j-1} P(j,k \text{ collide under } h_i) \\
&\leq \sum_{k<j} \frac{1}{10B_i^2} \\
&< \frac{B_i^2}{2} \cdot \frac{1}{10B_i^2} = \frac{1}{20}
\end{aligned}$$

By Markov's inequality again, $P(\#\text{ of collisions} \geq 1) < 1/20$

We have now shown that we can construct a 2-level hash table using hash functions h and $h_1, \ldots, h_m$ to make a static dictionary since h and $h_1, \ldots, h_m$ can be chosen in constant time each and linear time overall.