CSE 521 Winter 2006
Notes on Hashing

Instructor: Venkatesan Guruswami

1 The Dictionary Data Structure

The dictionary data structure is ubiquitous in computer science. A dictionary holds a set of items from an ordered universe and supports the operations of inserting an item, deleting an item, and searching for an item (membership queries). This is the *dynamic dictionary* problem. If the set of items is fixed and never changes, we only need to efficiently support membership queries (of the form "$x \in S$?"), and this is called the *static dictionary* problem.

The fundamental "time-space" trade-off in any data structures problem is the space needed to store the data structure, and the time required to perform the operations (such as updates and answering queries). Let us consider this trade-off for the dictionary data structure.

Let us assume the universe U is the set $[N] = \{0, 1, 2, \ldots, N-1\}$, and we need to store a subset $S \subseteq U$ of $n \ll N$ items. A trivial solution is to store a bit array of size N that holds the characteristic vector of the set. This supports $O(1)$ time operations but has very poor storage. On the other hand, at the other extreme, we can store the set as a linked list, which has optimal $O(n)$ storage, but requires $\Theta(n)$ time for operations. A binary search tree improves upon both of these with $O(n)$ storage and $O(\log n)$ time operations. For the static problem, a sorted array supports search in $O(\log n)$ time. The logarithmic time is in fact optimal in the comparison model (if only allowed operations are comparisons between integers). The objective of hashing is to use other operations on integers and simultaneously achieve optimal $O(n)$ storage as well as $O(1)$ time for the operations.

2 Hashing to search in $O(1)$ expected time

Let us think about the static problem. To store a set S of n integers, we will use an array/table T of size m where $m = O(n)$, say $m = 2n$ (any $m \geq (1+\varepsilon)n$ for $\varepsilon > 0$ will work). For an integer M, let $[M]$ denote the set $\{0, 1, \ldots, M-1\}$. We will use a *hash function* $h : [N] \to [m]$ and store element $x \in S$ in the entry $T[h(x)]$. Since more than one element can map to the same table entry, $T[i]$ will in general point to a linked list of all items $x' \in S$ such that $h(x') = i$. Since $m \ll N$, for any function h, there will be sets S for which all items map to the same location, and this solution is no better than the naive linked list one. Since we want to make a claim for worst-case inputs S, it is clear a deterministic choice of the hash function will not work for us. (Even if the hash function is cryptographic for which it is hard to find collisions, it will still not give a solution that works for all input sets S.)

We will pick the hash function at random from a family $\mathcal{H}$ of functions. The key property we would like from $\mathcal{H}$ is for collisions (that is, events where $x \neq y \in S$ map to the same location $h(x) = h(y)$ for a random choice $h \in \mathcal{H}$) to be rare. If $\mathcal{H}$ is the space of all functions from $[N] \to [m]$, then the probability that $h(x) = h(y)$ for a random $h \in \mathcal{H}$ is exactly $1/m$, for each fixed pair $x \neq y$. It follows that for $u \in U$ and $S \subseteq U$ of size n, the expected time to look for u is at most $O(1 + n/m) = O(1)$. But we also need to store the hash function h that we used in the data structure so we can later compute where an item may be stored, and this itself will take $N \log m > N$ space! We therefore look for much smaller families of functions which also guarantee the above property concerning collisions. This motivates the following definition.

Definition 1 (2-universal hash families) *A set $\mathcal{H}$ of functions from $[N]$ to $[m]$ is said to be 2-universal family if for every $x, y \in [N]$, $x \neq y$,*

$$\mathop{\mathbf{Pr}}_{h \leftarrow \mathcal{H}}[h(x) = h(y)] \leq 1/m \ .$$

Lemma 1 *Let $\mathcal{H}$ be a 2-universal hash family with domain $[N]$ and range $[m]$. Let S be an arbitrary subset of $[N]$ of size at most n and let $u \in U$ be arbitrary. Then, for a random choice of $h \in \mathcal{H}$, the expected number of $x \in S$ for which $h(x) = h(u)$ is at most n/m.*

Proof: Define an indicator r.v I_x for the event $h(x) = h(u)$. The desired random variable Z (that equals the number of $x \in S$ for which $h(x) = h(u)$) equals $\sum_{x \in S} I_x$. By definition of 2-universality, $\mathbf{E}[I_x] \leq 1/m$ for each $x \in S$. By linearity of expectation $\mathbf{E}[Z] = \sum_{x \in S} \mathbf{E}[I_x] \leq |S|/m \leq n/m$. $\Box$

Thus if we had a 2-universal family $\mathcal{H}$ of size polynomial in N so that we can store a member of the family in $O(1)$ RAM locations, and we compute $h(x)$ for any member $h \in \mathcal{H}$ and $x \in U$ in $O(1)$ time (in the RAM model), then in view of the above lemma we can search for an item in $O(1)$ *expected time* and using $O(n)$ storage. We can also support insert/delete operations in expected $O(1)$ time, except that once the size of the set S becomes comparable to m, we will need to rehash. This leads to an amortized expected insertion time of $O(1)$.

It therefore remains to give such a 2-universal hash family, and we will have a solution to the dictionary problem with expected $O(1)$ time operations. We turn to this next.

2.1 A simple 2-universal hash family

Lemma 2 *Let p be a prime in the range $[N, 2N)$. For $a \in \{1, \ldots, p-1\}$ and $b \in [p]$, define the function $h_{a,b} : [N] \to [m]$ as*

$$h_{a,b}(x) = ((ax + b) \mod p) \mod m \ .$$

The family $\mathcal{H}_{p,m} = \{h_{a,b} \mid a \in \{1, 2, \ldots, p-1\}, b \in \{0, 1, \ldots, p-1\}\}$ is a 2-universal hash family. Note that $\mathcal{H}_{p,m}$ is a family of size $O(N^2)$.

Proof: Fix $x, y \in [N]$ with $x \neq y$. Let us determine the size of the set $C_{x,y} = \{(a, b) \mid h_{a,b}(x) = h_{a,b}(y)\}$ of hash functions that collide on x, y. We wish to show that $|C_{x,y}| \leq p(p-1)/m$, so that the probability of a collision which is equal to $|C_{x,y}|/|\mathcal{H}_{p,m}| = |C_{x,y}|/(p(p-1))$ is at most $1/m$.

The bound on $|C_{x,y}|$ is immediate from the following two easy claims that hold for each fixed $x \neq y$:

- The number of ordered pairs $(r, s) \in [p] \times [p]$ with $r \neq s$ satisfying $r \mod m = s \mod m$ is at most $p \cdot (\lceil p/m \rceil - 1)$. Since p is a prime, $\lceil p/m \rceil \leq (p+m-1)/m$, so the bound $p(\lceil p/m \rceil - 1)$ is at most $p(p-1)/m$.

- For each pair $(r, s) \in [p] \times [p]$ with $r \neq s$, there is a *unique* pair (a, b) with $a \in \{1, \ldots, p-1\}$ and $b \in [p]$ such that $(ax + b) \mod p = r$ and $(ay + b) \mod p = s$. $\Box$

3 Perfect Hashing: Searching in $O(1)$ worst-case time

The above allowed us to answer membership queries in $O(1)$ *expected* time. Now we turn to a method called perfect hashing that gives $O(1)$ search time in the *worst-case.* We will only focus on the static problem in this section – the much harder dynamic case will be dealt with later via an approach different from perfect hashing. Throughout this section $S \subseteq [N]$ is the fixed set that has to be stored, and $|S| = n$.

3.1 Quadratic space solution

We will begin with a very simple quadratic space solution. Let us pick the table size $m = n^2$. Using an argument identical to Lemma 1, we can show that the expected number X of pairs $x \neq y \in S$ for which $h(x) = h(y)$ taken over a random choice of $h \in \mathcal{H}_{p,m}$ is at most $\binom{n}{2}/m < 1/2$. By Markov inequality, we have

$$\mathbf{Pr}[X \geq 1] \leq \frac{\mathbf{E}[X]}{1} < \frac{1}{2} \ .$$

Hence, with probability at least $1/2$, a random choice of $h \in \mathcal{H}_{p,m}$ creates NO collisions, and thus perfectly hashes S into $[m]$. We will hence find such a perfect hash function for S within two random trials in expectation, and when we do, we just store S based on this function, leading to $O(1)$ worst-case search time.

3.2 Perfect hashing in $O(n)$ space

To reduce the space to $O(n)$, we use a 2-level hashing scheme. The idea is to first hash using a function h in $\mathcal{H}_{p,m}$ to a table T of size m with $m = O(n)$, say $m = 2n$. If n_j elements of S fall into location $T[j]$ for $j = 0, 1, \ldots, m$, then these are further hashed using a perfect hash function h_j from $\mathcal{H}_{p,m_j}$ to a second level table of size m_j where $m_j = n_j^2$ (which is found using the method of Section 3.1). The rationale is that we do not expect too many collisions after the first hashing, i.e., n_j should be much smaller than n, and so we incur the quadratic cost for a much smaller subset of S. The below calculation makes this precise.

Since all collisions are resolved by the second hash, and each of the hash functions h and $\{h_j \mid j \in [m]\}$ can be computed in $O(1)$ time this indeed enables searching in $O(1)$ time. Each h and h_j can be stored in $O(1)$ space, the first hash table has size $m = O(n)$, so the expected space usage equals

$$O(n) + \sum_{j=0}^{m-1} \mathbf{E}[n_j^2] = O(n) + \sum_{j=0}^{m-1} \mathbf{E}\Big[n_j + 2\binom{n_j}{2}\Big] = O(n) + 2\,\mathbf{E}[Y] \ . \tag{1}$$

where Y is the random variable measuring the number of collisions of h. If for $x \neq y \in S$, $I_{x,y}$ is indicator random variable for the event $h(x) = h(y)$, then $\mathbf{E}[I_{x,y}] \leq 1/m$ (by 2-universality of $\mathcal{H}_{p,m}$) and $Y = \sum_{x,y} I_{x,y}$. Therefore, $\mathbf{E}[Y] \leq \binom{n}{2}/m \leq n/2$. Using this in (1), we conclude that the expected storage is $O(n)$. By Markov inequality, for a random choice of h, we have $Y = O(n)$ with probability at least $1/2$, so we can simply try (an expected $O(1)$ times) till we find a hash function h with $O(n)$ collisions and hence the resulting 2-level table has size $O(n)$.

3.3 Dynamic Perfect Hashing

It is possible to use perfect hashing to solve the dynamic dictionary problem with $O(1)$ insert/delete time with high probability (and $O(1)$ worst-case search time). This scheme is quite complicated.

Instead, we will present in class a different approach called “Cuckoo Hashing” introduced by Pagh and Rodler in 2001 (yes, it is quite recent!). It is cleaner and simpler than previous dynamic dictionaries and it does not use perfect hashing. It uses two hash tables, and each element is stored in its hash location in one of them. Cuckoo hashing achieves $O(1)$ expected time for updates and $O(1)$ worst-case time for membership queries.