Lecture 10

Hashing and Amortization

Supplemental reading in CLRS: Chapter 11; Chapter 17 intro; Section 17.1

10.1 Arrays and Hashing

Arrays are very useful. The items in an array are statically addressed, so that inserting, deleting, and looking up an element each take $O(1)$ time. Thus, arrays are a terrific way to encode functions

$$\{1,\dots,n\} \to T,$$

where T is some range of values and n is known ahead of time. For example, taking $T = \{0,1\}$, we find that an array A of n bits is a great way to store a subset of $\{1,\dots,n\}$: we set $A[i] = 1$ if and only if i is in the set (see Figure 10.1). Or, interpreting the bits as binary digits, we can use an n-bit array to store an integer between 0 and $2^n - 1$. In this way, we will often identify the set $\{0,1\}^n$ with the set $\{0,\dots,2^n - 1\}$.

What if we wanted to encode subsets of an arbitrary domain U, rather than just $\{1,\dots,n\}$? Or to put things differently, what if we wanted a *keyed* (or *associative*) *array*, where the keys could be arbitrary strings? While the workings of such data structures (such as dictionaries in Python) are abstracted away in many programming languages, there is usually an array-based solution working behind the scenes. Implementing associative arrays amounts to finding a way to turn a key into an array index. Thus, we are looking for a suitable function $U \to \{1,\dots,n\}$, called a **hash function**. Equipped with this function, we can perform key lookup:

$$U \xrightarrow{\text{hash function}} \{1,\dots,n\} \xrightarrow{\text{array lookup}} T$$

(see Figure 10.2). This particular implementation of associative arrays is called a **hash table**.

There is a problem, however. Typically, the domain U is much larger than $\{1,\dots,n\}$. For any hash function $h : U \to \{1,\dots,n\}$, there is some i such that at least $\frac{|U|}{n}$ elements are mapped to i. The set

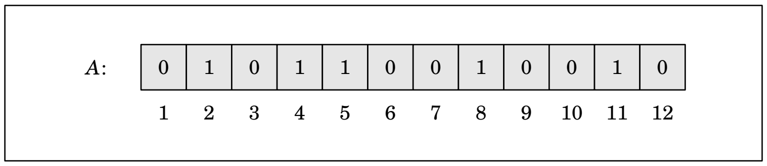

Figure 10.1. This 12-bit array encodes the set $\{2,4,5,8,11\} \subseteq \{1,\dots 12\}$.

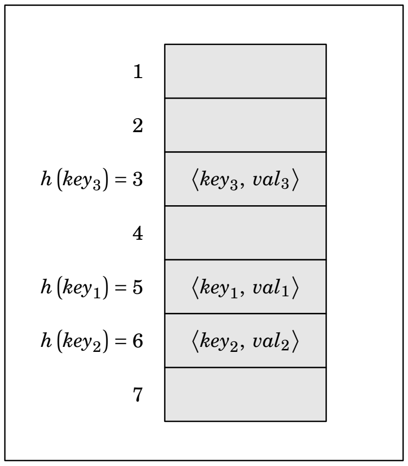
1
2
$h(key_3) = 3$
$\langle key_3, val_3 \rangle$
4
$h(key_1) = 5$
$\langle key_1, val_1 \rangle$
$h(key_2) = 6$
$\langle key_2, val_2 \rangle$
7

Figure 10.2. An associative array with keys in U and values in T can be implemented as a $(U \times T)$-valued array equipped with a hash function $h : U \to \{1, \ldots, n\}$.

$h^{-1}(i)$ of all elements mapped to i is called the *load* on i, and when this load contains more than one of the keys we are trying to store in our hash table we say there is a **collision** at i. Collisions are problem for us—if two keys map to the same index, then what should we store at that index? We have to store both values somehow. For now let's say we do this in the simplest way possible: storing at each index i in the array a linked list (or more abstractly, some sort of bucket-like object) consisting of all values whose keys are mapped to i. Thus, lookup takes $O\left(h^{-1}(i)\right)$ time, which may be poor if there are collisions at i. Rather than thinking about efficient ways to handle collisions,[1] let's try to reason about the probability of having collisions if we choose our hash functions well.

10.2 Hash Families

Without any prior information about which elements of U will occur as keys, the best we can do is to choose our hash function h at random from a suitable hash family. A **hash family** on U is a set $\mathscr{H}$ of functions $U \to \{1, \ldots, n\}$. Technically speaking, $\mathscr{H}$ should come equipped with a probability distribution, but usually we just take the uniform distribution on $\mathscr{H}$, so that each hash function is equally likely to be chosen.

If we want to avoid collisions, it is reasonable to hope that, for any fixed $x_1, x_2 \in U$ ($x_1 \neq x_2$), the values $h(x_1)$ and $h(x_2)$ are completely uncorrelated as h ranges through the sample space $\mathscr{H}$. This leads to the following definition:

Definition. A hash family $\mathscr{H}$ on U is said to be **universal** if, for any $x_1, x_2 \in U$ ($x_1 \neq x_2$), we have

$$\Pr_{h \in \mathscr{H}}\left[h(x_1) = h(x_2)\right] \leq \frac{1}{n}.$$

[1] If you are expecting lots of collisions, a more efficient way to handle things is to create a two-layered hash table, where each element of A is itself a hash table with its own, different hash function. In order to have collisions in a two-layer hash table, the same pair of keys must collide under two different hash functions. If the hash functions are chosen well (e.g., if the hash functions are chosen randomly), then this is extremely unlikely. Of course, if you want to be even more sure that collisions won't occur, you can make a three-layer hash table, and so on. There is a trade-off, though: introducing unnecessary layers of hashing comes with a time and space overhead which, while it may not show up in the big-O analysis, makes a difference in practical applications.

Similarly, $\mathscr{H}$ is said to be **ϵ-universal** if for any $x_1 \neq x_2$ we have

$$\Pr_{h \in \mathscr{H}}\left[h(x_1) = h(x_2)\right] \leq \epsilon.$$

The consequences of the above hypotheses with regard to collisions are as follows:

Proposition 10.1. *Let $\mathscr{H}$ be a universal hash family on U. Fix some subset $S \subseteq U$ and some element $x \in U$. Pick $h \in \mathscr{H}$ at random. The expected number of elements of S that map to $h(x)$ is at most $1 + \frac{|S|}{n}$. In symbols,*

$$\mathop{\mathbb{E}}_{h \in \mathscr{H}}\left[\left|h^{-1}\left(h(x)\right)\right|\right] \leq 1 + \frac{|S|}{n}.$$

If $\mathscr{H}$ is ϵ-universal rather than universal, then the same holds when $1 + \frac{|S|}{n}$ is replaced by $1 + \epsilon|S|$.

Proof. For a proposition φ with random parameters, let I_φ be the indicator random variable which equals 1 if φ is true and equals 0 otherwise. The fact that $\mathscr{H}$ is universal means that for each $x' \in U \setminus \{x\}$ we have

$$\mathop{\mathbb{E}}_{h \in \mathscr{H}}\left[I_{h(x)=h(x')}\right] \leq \frac{1}{n}.$$

Thus by the linearity of expectation, we have

$$\begin{aligned}
\mathbb{E}\left[\left|h^{-1}\left(h(x)\right) \cap S\right|\right] &= I_{x \in S} + \mathop{\mathbb{E}}_{h \in \mathscr{H}}\left[\sum_{\substack{x' \in S \\ x' \neq x}} I_{h(x)=h(x')}\right] \\
&= I_{x \in S} + \sum_{\substack{x' \in S \\ x' \neq x}} \mathop{\mathbb{E}}_{h \in \mathscr{H}}\left[I_{h(x)=h(x')}\right] \\
&\leq 1 + |S| \cdot \tfrac{1}{n}.
\end{aligned}$$

The reasoning is almost identical when $\mathscr{H}$ is ϵ-universal rather than universal. □

Corollary 10.2. *For a hash table in which the hash function is chosen from a universal family, insertion, deletion, and lookup have expected running time $O\left(1 + \frac{|S|}{n}\right)$, where $S \subseteq U$ is the set of keys which actually occur. If instead the hash family is ϵ-universal, then the operations have expected running time $O\left(1 + \epsilon|S|\right)$.*

Corollary 10.3. *Consider a hash table of size n with keys in U, whose hash function is chosen from a universal hash family. Let $S \subseteq U$ be the set of keys which actually occur. If $|S| = O(n)$, then insertion, deletion, and lookup have expected running time $O(1)$.*

Let $\mathscr{H}$ be a universal hash family on U. If $|S| = O(n)$, then the expected load on each index is $O(1)$. Does this mean that a typical hash table has $O(1)$ load at each index? Surprisingly, the answer is no, even when the hash function is chosen well. We'll see this below when we look at examples of universal hash families.

Examples 10.4.

1. The set of *all* functions $h : U \to \{1, \ldots, n\}$ is certainly universal. In fact, we could not hope to get any more balanced than this:

- For any $x \in U$, the random variable $h(x)$ (where h is chosen at random) is uniformly distributed on the set $\{1, \ldots, n\}$.
- For any pair $x_1 \neq x_2$, the random variables $h(x_1), h(x_2)$ are independent. In fact, for any finite subset $\{x_1, \ldots, x_k\} \subseteq U$, the tuple $\left(h(x_1), \ldots, h(x_k)\right)$ is uniformly distributed on $\{1, \ldots, n\}^k$.

The load on each index i is a binomial random variable with parameters $\left(|S|, \frac{1}{n}\right)$.

Fact. *When p is small and N is large enough that Np is moderately sized, the binomial distribution with parameters (N, p) is approximated by the* **Poisson distribution** *with parameter Np. That is, if X is a binomial random variable with parameters (N, p), then*

$$\Pr\left[X = k\right] \approx \frac{(Np)^k}{k!} e^{-Np} \quad (k \geq 0).$$

In our case, $N = |S|$ and $p = \frac{1}{n}$. Thus, if L_i is the load on index i, then

$$\Pr\left[L_i = k\right] \approx \frac{\left(\frac{|S|}{n}\right)^k}{k!} e^{-|S|/n}.$$

For example, if $|S| = n$, then

$$\begin{aligned} \Pr\left[L_i = 0\right] &\approx e^{-1} \approx 0.3679, \\ \Pr\left[L_i = 1\right] &\approx e^{-1} \approx 0.3679, \\ \Pr\left[L_i = 2\right] &\approx \tfrac{1}{2} e^{-1} \approx 0.1839, \\ &\vdots \end{aligned}$$

Further calculation shows that, when $|S| = n$, we have

$$\mathbb{E}\left[\max_{1 \leq i \leq n} L_i\right] = \Theta\left(\frac{\lg n}{\lg \lg n}\right).$$

Moreover, with high probability, $\max L_i$ does not exceed $O\left(\frac{\lg n}{\lg \lg n}\right)$. Thus, a typical hash table with $|S| = n$ and h chosen uniformly from the set of all functions looks like Figure 10.3: about 37% of the buckets empty, about 37% of the buckets having one element, and about 26% of the buckets having more than one element, incuding some buckets with $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$ elements.

2. In Problem Set 4 we considered the hash family

$$\mathscr{H} = \left\{h_p : p \leq k \text{ and } p \text{ is prime}\right\},$$

where $h_p : \{0, \ldots, 2^m - 1\} \to \{0, \ldots, k - 1\}$ is the function

$$h_p(x) = x \bmod p.$$

In Problem 4(a) you proved that, for each $x \neq y$, we have

$$\Pr_p\left[h_p(x) = h_p(y)\right] \leq \frac{m \ln k}{k}.$$

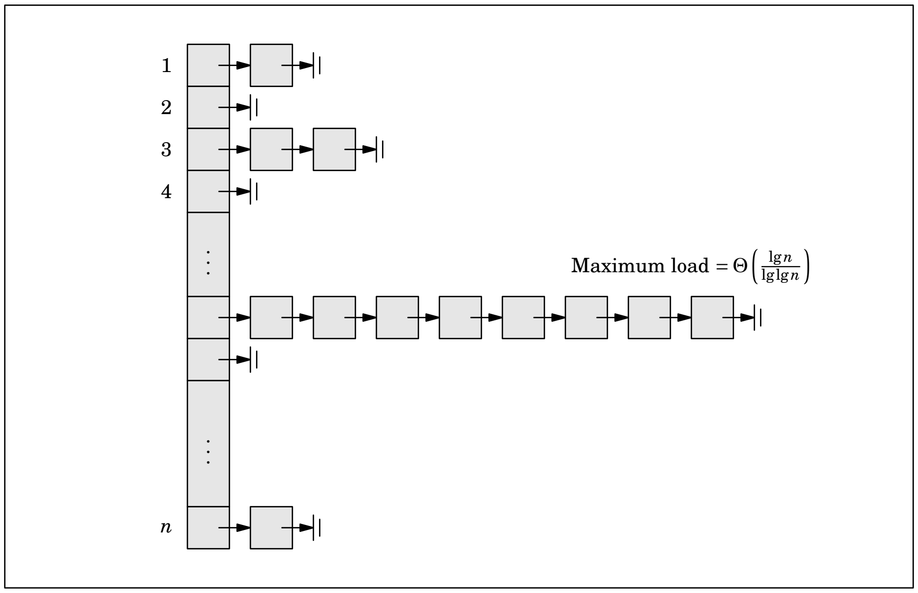

Figure 10.3. A typical hash table with $|S| = n$ and h chosen uniformly from the family of all functions $U \to \{1,\ldots,n\}$.

3. In Problem Set 5, we fixed a prime p and considered the hash family

$$\mathscr{H} = \left\{h_{\vec{a}} : \vec{a} \in \mathbb{Z}_p^m\right\},$$

where $h_{\vec{a}} : \mathbb{Z}_p^m \to \mathbb{Z}_p$ is the dot product

$$h_{\vec{a}}(\vec{x}) = \vec{x} \cdot \vec{a} = \sum x_i a_i \pmod p.$$

4. In Problem Set 6, we fixed a prime p and positive integers m and k and considered the hash family

$$\mathscr{H} = \left\{h_A : A \in \mathbb{Z}_p^{k\times m}\right\},$$

where $h_A : \mathbb{Z}_p^m \to \mathbb{Z}_p^k$ is the function

$$h_A(\vec{x}) = A\vec{x}.$$

5. If $\mathscr{H}_1$ is an ϵ_1-universal hash family of functions $\{0,1\}^m \to \{0,1\}^k$ and $\mathscr{H}_2$ is an ϵ_2-universal hash family of functions $\{0,1\}^k \to \{0,1\}^\ell$, then[2]

$$\mathscr{H} = \mathscr{H}_2 \circ \mathscr{H}_1 = \{h_2 \circ h_1 : h_1 \in \mathscr{H}_1, h_2 \in \mathscr{H}_2\}$$

is an $(\epsilon_1 + \epsilon_2)$-universal hash family of functions $\{0,1\}^m \to \{0,1\}^\ell$. To see this, note that for any $x \neq x'$, the union bound gives

$$\Pr_{\substack{h_1 \in \mathscr{H}_1 \\ h_2 \in \mathscr{H}_2}} \left[h_2 \circ h_1(x) = h_2 \circ h_1(x')\right]$$

[2] To fully specify $\mathscr{H}$, we have to give not just a set but also a probability distribution. The hash families $\mathscr{H}_1$ and $\mathscr{H}_2$ come with probability distributions, so there is an induced distribution on $\mathscr{H}_1 \times \mathscr{H}_2$. We then equip $\mathscr{H}$ with the distribution induced by the map $\mathscr{H}_1 \times \mathscr{H}_2 \to \mathscr{H}$, $(h_1, h_2) \mapsto h_2 \circ h_1$. You could consider this a mathematical technicality if you wish: if $\mathscr{H}_1$ and $\mathscr{H}_2$ are given uniform distributions (as they typically are), then the distribution on $\mathscr{H}_1 \times \mathscr{H}_2$ is also uniform. The distribution on $\mathscr{H}$ need not be uniform, however: an element of $\mathscr{H}$ is more likely to be chosen if it can be expressed in multiple ways as the composition of an element of $\mathscr{H}_2$ with an element of $\mathscr{H}_1$.

$$= \Pr\left[h_1(x) = h_1(x') \text{ or } \left(h_1(x) \neq h_1(x') \text{ and } h_2 \circ h_1(x) = h_2 \circ h_1(x')\right)\right]$$

$$\leq \Pr\left[h_1(x) = h_1(x')\right] + \Pr\left[h_1(x) \neq h_1(x') \text{ and } h_2 \circ h_1(x) = h_2 \circ h_1(x')\right]$$

$$\leq \epsilon_1 + \epsilon_2.$$

In choosing the parameters to build a hash table, there is a tradeoff. Making n larger decreases the likelihood of collisions, and thus decreases the expected running time of operations on the table, but also requires the allocation of more memory, much of which is not even used to store data. In situations where avoiding collisions is worth the memory cost (or in applications other than hash tables, when the corresponding tradeoff is worth it), we can make n much larger than S.

Proposition 10.5. *Let $\mathscr{H}$ be a universal hash family $U \to \{1,\ldots,n\}$. Let $S \subseteq U$ be the the set of keys that occur. Then the expected number of collisions is at most $\binom{|S|}{2} \cdot \frac{1}{n}$. In symbols,*

$$\mathop{\mathbb{E}}_{h \in \mathscr{H}}\left[\sum_{x \neq x' \in U} I_{h(x)=h(x')}\right] \leq \binom{|S|}{2} \cdot \frac{1}{n}.$$

Proof. There are $\binom{|S|}{2}$ pairs of distinct elements in S, and each pair has probability at most $\frac{1}{n}$ of causing a collision. The result follows from linearity of expectation. ☐

Corollary 10.6. *If $n \geq 100|S|^2$, then the expected number of collisions is less than* 1/200*, and the probability that a collision exists is less than* 1/200.

Proof. Apply the Markov bound. ☐

Thus, if n is sufficiently large compared to S, a typical hash table consists mostly of empty buckets, and with high probability, there is at most one element in each bucket.

As we mentioned above, choosing a large n for a hash table is expensive in terms of space. While the competing goals of fast table operations and low storage cost are a fact of life if nothing is known about S in advance, we will see in recitation that, if S is known in advance, it is feasible to construct a **perfect** hash table, i.e., a hash table in which there are no collisions. Of course, the smallest value of n for which this is possible is $n = |S|$. As we will see in recitation, there are reasonably efficient algorithms to construct a perfect hash table with $n = O(|S|)$.

10.3 Amortization

What if the size of S is not known in advance? In order to allocate the array for a hash table, we must choose the size n at creation time, and may not change it later. If $|S|$ turns out to be significantly greater than n, then there will always be lots of collisions, no matter which hash function we choose.

Luckily, there is a simple and elegant solution to this problem: **table doubling**. The idea is to start with some particular table size $n = O(1)$. If the table gets filled, simply create a new table of size $2n$ and migrate all the old elements to it. While this migration operation is costly, it happens infrequently enough that, on the whole, the strategy of table doubling is efficient.

Let's take a closer look. To simplify matters, let's assume that only insertions and lookups occur, with no deletions. What is the worst-case cost of a single operation on the hash table?

- Lookup: $O(1)$, as usual.
- Insertion: $O(n)$, if we have to double the table.

Thus, the worst-case total running time of k operations ($k = |S|$) on the hash table is

$$O(1 + \cdots + k) = O\left(k^2\right).$$

The crucial observation is that this bound is *not* tight. Table doubling only happens after the second, fourth, eighth, etc., insertions. Thus, the total cost of k insertions is

$$k \cdot O(1) + O\left(\sum_{j=0}^{\lg k} 2^j\right) = O(k) + O(2k) = O(k).$$

Thus, in any sequence of insertion and lookup operations on a dynamically doubled hash table, the average, or **amortized**, cost per operation is $O(1)$. This sort of analysis, in which we consider the total cost of a sequence of operations rather than the cost of a single step, is called **amortized analysis**. In the next lecture we will introduce methods of analyzing amortized running time.