A Course in Graduate Algorithms **Lecture 9**

Hash Tables

Designed by Prof. Bo Waggoner for the University of Colorado-Boulder *Updated: 2023*

Hash tables are a useful data structure employing randomness. This lecture will introduce hash tables and some interesting phenomena that arise in connection with them, the birthday paradox and the coupon collector problem.

Objectives:

- Work with the *ideal hash function* model and relate it to real world problems.
- Understand tradeoffs involved in hash table space and time.
- Understand how the birthday paradox and coupon collector phenomena work.

1 Overview

Hash tables are data structures commonly used to store *key-value mappings*. For example, if we have a document split into words, we can use a hash table to count how many times each word occurs. Each word is a key, and its value is the current count for that word.

If we wish to store n elements, then a basic hash table will have these performance characteristics:

Operation	**Average-case time**	**Worst-case time**
Add(x, v)	$O(1)$	$O(n)$
Find(x)	$O(1)$	$O(n)$
Remove(x)	$O(1)$	$O(n)$

Here, Add(x, v) puts the key-value pair (x, v) into the table, while Find(x) checks if they key x is in the table and, if so, returns its associated value.

As sets. Hash tables can be used to store *sets* of items, with rapid lookup. In this case, the items act as the keys, and the values can be ommitted (or the value associated with a key is always True). If Find(x) returns that x is present, then the item is in the set; otherwise it is not.

Alternatives. One could also implement these data structures with, for example, a binary search tree (BST) which maintains the keys in sorted order along with their values. BSTs offer $O(\log n)$ worst-case time complexity for Add, Find, and Remove operations.

1.1 General Implementation

Hash tables are implemented with an array, To implement Add(x, v), we use a *hash function* to map x to a location in the array and we store the pair at that location (usually). Formally:

- We have a universe U of possible keys, $|U| = m$.
  We assume keys and values use constant space in word-RAM model.
- Only n of the keys will arrive.[1]

[1] In practice, one usually does not know how many keys will arrive, so hash tables typically dynamically resize themselves as more elements arrive. We will come back to this briefly at the end.

- We use an array of length ℓ.
- We use a *hash function* $h : U \to \{0, \ldots, \ell - 1\}$.
  We assume for our analysis that h can be computed in constant time.

A perfect situation would be if $m = \ell$. Then we could take h to be a bijection between U and $\{0, \ldots, \ell - 1\}$. We would have:

- Add(x, v): set $A[h(x)] = (x, v)$.
- Find(x): return $A[h(x)]$.
- Remove(x): set $A[h(x)]$ =NULL.

Unfortunately, usually our universe U is very large, such as all 64-bit integers ($m = 2^{64}$) or all strings of a certain legnth. So each array location or "bin" will have many universe elements mapping to it. The hope is that, if only n of those actually arrive, then we can minimize these "collisions".

1.2 Chaining

A *collision* occurs when two keys x, x' are inserted into the table at the same location, i.e. $h(x) = h(x')$. The simplest solution for collisions is called chaining. In chaining, each array location (also called "bin" or "bucket") is a pointer to a linked list. To Add(x, v), we append (x, v) to the linked list at $A[h(x)]$. To Find(x) or Remove(x), we first get the linked list at $A[h(x)]$, then search the list for x.

Of course, this can increase the time complexity. If a linked list has length d, then these operations generally take $O(d)$ time. Let

$$\alpha = \frac{n}{\ell},$$

the average load of the bins. If the elements are evenly distributed, then the maximum load would be $d = \lceil \alpha \rceil$. (We'd prefer to use a longer array size ℓ and keep d small, if possible.)

To analyze collisions, we can pretend that h maps each key to a uniformly random location. This would be called a **ideal hash function**. The expected load of any bucket k would be

$$\begin{aligned}
&\sum_{i=1}^{n} \Pr[\text{item } i \text{ lands in bucket } k] \\
&= \sum_{i=1}^{n} \frac{1}{\ell} \\
&= \frac{n}{\ell} \\
&= \alpha.
\end{aligned}$$

But this is only an average. Next we'll analyze how collisions happen.

2 Birthday Paradox

Suppose that the n arriving items are hashed into ℓ buckets with an ideal hash function (i.e. each item placed uniformly at random). This models throwing n balls into ℓ bins, where each ball selects a bin independently and uniformly (i.e. equal probability) at random.

The first question is: **how large must n be until a collision occurs?**

Birthdays. Suppose that each person is born on a uniformly random day of the year, i.e. a $\frac{1}{365}$ chance of January 1, ... (Ignore February 29.) How many people must be gathered in a room before two of them share a birthday? We can think of each person as throwing a ball into one of 365 bins. If two people share a birthday, they've landed in the same bin: a collision.

The balls are numbered $i = 1, \ldots, n$. Let

$$X_{ij} = \begin{cases} 1 & i \text{ and } j \text{ go into the same bin} \\ 0 & \text{otherwise} \end{cases}.$$

Then

$$\begin{aligned} \mathbb{E}[X_{ij}] &= \sum_{k=1}^{\ell} \Pr[i \text{ and } j \text{ both land in bin } k] \\ &= \sum_{k=1}^{\ell} \left(\frac{1}{\ell}\right)\left(\frac{1}{\ell}\right) \\ &= (\ell)\left(\frac{1}{\ell}\right)^2 \\ &= \frac{1}{\ell}. \end{aligned}$$

There is a shorter argument: whichever bin i lands in, j has a $\frac{1}{\ell}$ chance of landing in it as well. Notice that the arguments used independence of balls i and j.

Now, we can calculate the expected number of collisions:

$$\begin{aligned} \mathbb{E}[\text{num. collisions}] &= \mathbb{E}\left[\sum_{\text{pairs } i,j} X_{ij}\right] \\ &= \sum_{\text{pairs } i,j} \mathbb{E}[X_{ij}] && \text{linearity of expectation} \\ &= \sum_{\text{pairs } i,j} \frac{1}{\ell} \\ &= \binom{n}{2}\frac{1}{\ell} \\ &= \Theta\left(\frac{n^2}{\ell}\right). \end{aligned}$$

We conclude that with ℓ bins, we expect a collision after throwing just $n = \Theta(\sqrt{\ell})$ balls. This is often surprising, because the vast majority of bins will be empty (i.e. at least $\ell - \Theta(\sqrt{\ell})$ of them), yet we already expect some collision to occur.

Exercise 1. For the birthday paradox in particular ($\ell = 365$), calculate the fewest people in a room such that at least one collision is expected. Use the exact formula for expected collisions, i.e. $\binom{n}{2}\frac{1}{\ell}$.

Exercise 2 (Optional)**.** Code up a simulation for the birthday paradox and confirm your answer above. About how many people do you need in a room before you expect *two* collisions? Answer with simulations and math.

The case $n = \ell$. Of course, in a hash table for storing n items, we wouldn't want to use any array of size $\ell > n^2$; this would be a quadratic space usage. Another popular calculation comes from when $\ell = n$, i.e. there is one available bin for every item and the average load is $\alpha = 1$. However, due to the randomness in the bin assignments, some bins may be empty while others contain multiple balls.

For any given bin, its probability of being empty is the chance that all n balls do *not* land in this bin. The chance for each ball is $\frac{\ell-1}{\ell} = 1 - \frac{1}{\ell}$, so the chance of being empty is

$$\left(1 - \frac{1}{\ell}\right)^n \approx e^{-n/\ell}.$$

When $n = \ell$, this gives a $\frac{1}{e} \approx 0.367\ldots$ probability of being empty. By linearity of expectation, we expect $\frac{\ell}{e}$ empty bins.

Meanwhile, a well-known analysis gives that, when $n = \ell$, the max-loaded bin contains approximately $\frac{\ln(n)}{\ln(\ln(n))}$ balls in expectation. Therefore, we can say that an Add, Find, or Remove operation should take at most this long with the basic chaining technique.

In fact, for any given x, we can show that the expected time for Add, Find, or Remove is $O(1)$. To see this, note that expected time is $O(1)$ plus the expected number of other items hashed to x's bin. For any given bin, the expected number of the n items that hash there is $\alpha = n/\ell$. So for $\alpha = 1$, the expected time for these operations is $O(1)$.

3 Coupon Collector

Now we ask a different question: How many items, n, must be hashed until *all* of the ℓ bins are nonempty? Here, the related story is the *coupon collector* problem. Suppose each box of cereal we purchase comes with a uniformly random coupon, with a total of ℓ possible coupons. How many boxes n must we purchase before we've collected at least one of each type of coupon?

To analyze this process, let Y_t = the first throw on which t bins are nonempty. Set $Y_0 = 0$. Note that $Y_1 = 1$. If, for example, the first five balls land in the same bin and the sixth lands in a different bin, then Y_2 would equal 6, because the sixth throw is when a second bin became nonempty. And so on up to Y_ℓ, the throw on which the last empty bin receives a ball. For $1 \le t \le \ell$, let

$$Z_t = Y_t - Y_{t-1},$$

the number of throws after Y_{t-1} until a ball lands in an empty bin.

Now, we can analyze the geometric random variable Z_t as follows. Each throw, there are $\ell - (t-1)$ empty bins. So the probability of hitting one is

$$p_t := \frac{\ell - t + 1}{\ell}.$$

Now Z_t is the number of independent trials with this probability until the first success. It turns out that

$$\mathbb{E}[Z_t] = \frac{1}{p_t} = \frac{\ell}{\ell - t + 1}.$$

This allows us to calculate the expected time until we've collected all the coupons. We have (remem-

ber $Y_0 = 0$)

$$\begin{aligned}
\mathbb{E}[Y_\ell] &= \mathbb{E}[Y_0 + (Y_1 - Y_0) + (Y_2 - Y_1) + \cdots + (Y_\ell - Y_{\ell-1})] \\
&= \mathbb{E}\left[\sum_{t=1}^{\ell} Z_t\right] \\
&= \sum_{t=1}^{\ell} \mathbb{E}[Z_t] \\
&= \sum_{t=1}^{\ell} \frac{\ell}{\ell - t + 1} \\
&= \ell \sum_{t=1}^{\ell} \frac{1}{\ell - t + 1} \\
&= \ell \left(\frac{1}{\ell} + \frac{1}{\ell - 1} + \cdots + 1\right) \\
&= \Theta(\ell \ln(\ell)).
\end{aligned}$$

Here we used that the harmonic sum satisfies $\sum_{j=1}^{\ell} \frac{1}{j} = \Theta(\ln(\ell))$.

We conclude that we expect on the order of $\ell \ln(\ell)$ coupons before we obtain one of each kind. In other words, we expect to throw about $\ell \ln(\ell)$ balls into ℓ bins before every bin is nonempty.

4 On Implementation of Hash Tables

Our analyses above assume an **ideal hash function**. But can we implement it? If not, can we approximate it?

Of course, the following idea fails: *Each time we call $h(x)$, pick uniformly at random from $\{1, \ldots, \ell\}$.* The problem is that the hash function has to give the same answer for $h(x)$ every time. That way we can find x again after we insert it.

Here's an idea that would work, but is not practical in terms of time and space:

- Before using the data structure, pick h uniformly at random from the space of all functions $U \to \{0, \ldots, \ell - 1\}$.
  - That is, for each $x \in U$, pick a choice for $h(x)$ uniformly from $\{0, \ldots, \ell - 1\}$.
  - Now each time $h(x)$ is called, we return this choice.
- Use this h for this hash table.

This would implement an ideal hash function exactly. Unfortunately, this is not practical because we would have to store all of the information that determines h in an extremely large array.

In practice, what we do instead is pick a simple function or formula that seems to approximate an ideal hash function. One should be careful if the built-in hash table for a programming language relies on a known formula: it becomes very desirable for attackers. This can be mitigated by having a formula with a parameter that is chosen randomly each time the program starts.

4.1 Growing the hash table

We generally don't know n, the number of items, in advance. The ideal approach would be to dynamically grow ℓ as more items are added. A basic approach is:

- Set some desired maximum load factor α, e.g. $\alpha = 0.7$ or $\alpha = 0.9$.
- Pick some initial ℓ, e.g. 1 or 100.
- When average load exceeds α (i.e. $n \geq \alpha\ell$):
  - Create a new array of size 2ℓ.
  - Insert all the items into the new array.
  - Delete the old array.

Resizing a table of size j takes $O(j)$ work, but it only happens rarely. When adding n total elements, the total amount of work done in resizing is on the order of $1 + 2 + 4 + 8 + 16 + \cdots + n \leq 2n = O(n)$. So the *amortized* time complexity is an average of $O(1)$ work per time step, although some steps will take longer. More advanced approaches can incrementally transfer elements into the new array so that no one step takes too much time.