Don't Put All Your Eggs In One Basket; Put Them In A Hashtable

Ziheng Chen

September 21, 2023

Abstract

It's well known that eggs in a basket are fragile. What is the alternative choice then? In a hashtable! In this talk, we'll introduce one of the most fundamental data structures in computer science. We'll begin with the typical applications and then move on to practical construction and performance analysis. As a particular example of a random algorithm, the hashtable has a few exciting properties and generalizations.

1 Introduction: Where's my egg?

Say we wish to run an egg business, essentially selling eggs at a grocery store. Let's further assume that we are a crazy lover of automation, so we're designing a fully automatic egg stashing-retriving system where there are countless robots running behind the scenes to transport the eggs for the customers. When the customers arrive, the robot goes to the storage and checks their name, trying to locate the corresponding package with their name on it, or returning without finding any. We may add new packages as the customers are quite enthusiastic of tasting our eggs with good quality. Now, how should we design the system that is most efficient and consistent?

Let us assume there are n potential customers with different names (or at least there is a way to tell them apart, e.g. check their IDs). The most naive setup is to put down the names in a fixed order; the robot then checks if the label matches the order, from left to right, one after another.

			robot here		
customer	Joe	Lucy	Peter	Tom	...
basket					

Table 1: The sequential check takes $\mathcal{O}(n)$ time on average.

This plan is, of course, easy to implement, but can be costly if n becomes large since $\mathcal{O}(n)$ time is needed for *each* query performed. The insertion is probably cheap, since the robot just dumps the new package to the rightmost end.

What if we email the customers their order numbers so we can probably sort all the packages in an increasing order, from left to right.

			robot here		
order number	156	502	2133	5412	...
basket					

Table 2: The bisection check takes $\mathcal{O}(\log n)$ time on average.

This plan looks slightly nicer, since you might have heard of the bisection algorithm, which essentially pins down the basket in $\mathcal{O}(\log n)$ time. This may look promosing, but there are two down sides:

1. it takes $\mathcal{O}(n \log n)$ time to build the sorted basket sequence in the first place; the overhead cost is not negligible, and
2. it takes $\mathcal{O}(n)$ time to insert a new package since we need to shift the packages followed to the right,

not to mention that bisection only work for numerical values since customers may find it hard to remember long order numbers.

What if I tell you that there is a data structure that achieves (roughly speaking) $\mathcal{O}(1)$ time for query, insertion, and deletion? To understand why we prefer to put eggs in a hashtable instead of a line of baskets, we need to study the construction and do some calculation.

2 Hashtable

Hashtable is a data structure that is widely used in most programming languages. This structure serves as a container that memories key-value pairs,

meaning that the indexing is done by showing a key and the value is retrived (there is no way for the customer to tell the robot "hey, I guess the second package from the left is mine"). The keys and values can be non-numerical, making the problem a bit tougher to solve.

The hash function plays a central role in operating a hastable. To put it simply, the hash function h makes it possible to use a (usually short) integer $1 \leq h(k) \leq m$ to describe the key object $k \in U$. We will postpone the construction of such functions to the end of this section, but take it for granted that the hash function looks "pretty random" in terms of the output $h(k)$.

To build a hashtable, we simply allocate a storage of m slots and put the value v in the slot with number $h(k)$, for all given key-value pairs (k, v).

			robot here		
hashes	1	2	3	4	...
basket	v_2		v_1	v_3	

Table 3: The robot goes to the third slot since the hash function is defined by $h(k_1) = 3, h(k_2) = 1, h(k_3) = 4$.

What if two hashes happen the same number (which we call collision)? One way to solve this issue is to nest a linked list that place the key-value pairs in sequential order as follows:

			robot here		
hashes	1	2	3	4	...
basket	v_2		(k_1, v_1)	v_3	
			(k_4, v_4)		

Table 4: A collision scenario $h(k_1) = h(k_4) = 3, h(k_2) = 1, h(k_3) = 4$.

which is also known as hashtable with separate chaining. There are also other ways to address collision but that is a story for another time.

3 Collision analysis

Let's formalize the collision problem in the math language.

Definition. $b_1, b_2, \ldots, b_n$ are n i.i.d. variables that take value uniformly in $[m] := \{1, 2, \ldots, m\}$. Let

$$X_i := \sum_{j=1}^{n} \mathbf{1}_{b_j = i}$$

be the count of bin i.

Question. *What can we know about X_i? For example, $\mathbb{E}X_i$, $\max X_i$, ...*

Claim. $\mathbb{E}X_i = 1$ by linearity.

Proposition 1. $\max_i X_i = \mathcal{O}\left(\frac{\log n}{\log\log n}\right)$ *with high probability* $1 - n^{-c}$.

Proof. Notice that $\mathbb{P}\left(\max_i X_i \geq l\right) \leq n\,\mathbb{P}\left(X_1 \geq l\right)$ and for each bin

$$\mathbb{P}\left(X_1 \geq l\right) \leq \sum_{S\in[n],|S|=l} \mathbb{P}\left(S \text{ falls in bin } 1\right) = \left(\begin{array}{c} n \\ l \end{array}\right) \frac{1}{n^l}.$$

For the combinatorial term, we use

$$\left(\begin{array}{c} n \\ l \end{array}\right) \leq \left(\frac{en}{l}\right)^l$$

to conclude that

$$\mathbb{P}\left(\max_i X_i \geq l\right) \leq \frac{ne^l}{l^l}.$$

To match the right hand side above with n^{1-c} $(c > 1)$, we take logarithm

$$l\,\log\frac{e}{l} = -c\,\log n$$

where LHS is decreasing w.r.t. l, so we can assume

$$l = \frac{A\,\log n}{\log\log n}$$

and a proper A can be solve via

$$A\left(1 + \frac{\log\frac{A}{e} - \log\log\log n}{\log\log n}\right) = c.$$

□

Prop. 1 is nice since it gives a worst case bound (under high probability), but it does not improve too much from the bisection algorithm which takes $\mathcal{O}\left(\log n\right)$ time. In fact, the complexity can be improved to use multiple hash functions to further average out the randomness.

Algorithm. *For each successive egg, pick two random bins and the egg goes to the bin with fewer ones.*

Solution. We will not give a rigorous proof as we'll probably be running out of time, but we aim to give an intuitive explanation. The general idea is that, inserting a new egg is not going to increase the maximum height, as the egg has to be unlucky enough to land on two highest piles. Let's introduce $v_i(t)$ as the number of bins that contains no less than

$$v_i(t) = \#\{\text{bins of height} \geq i \text{ after inserting } t \text{ eggs}\}.$$

Then, if we manage to show $v_i(t) \leq \beta_i n$ regardless of t via induction, then

$$\mathbb{P}[\text{egg placed at height} \geq i+1] \leq \beta_i^2,$$

leading to

$$\mathbb{E}[v_{i+1}(t)] = \mathbb{E}[\#\text{bins of height} \geq i+1] \leq \mathbb{E}[\#\text{eggs of height} \geq i+1] \leq n\beta_i^2.$$

We need an external tool called Chernoff inequality, which essentially tell us that it's very unlikely to get twice the expectation, leading to

$$v_{i+1}(t) \leq n\beta_i^2 \Rightarrow \beta_{i+1} = \beta_i^2.$$

Since $\log \beta_{i+1} = 2\log \beta_i$ doubles every time that starts from $\mathcal{O}(1)$ and ends to $\mathcal{O}(\log n)$, we only need $\mathcal{O}(\log \log n)$ steps.

4 Hash functions

You may have heard of cryptographic hash functions, but here we only need very simple hash functions since we do not seek for the safety inversion properties. We name a few ones if anyone is interested:

- Carter-Wegman family: pick prime $p > m$ and define

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

- Bit-multiplication: for $U = 2^n$ and $M = 2^m$, we define

$$h_a(x) = (ax \mod U) \gg (u-m).$$

- Matrix-finite-field: pick $A \in \mathbb{F}_2^{m \times n}, b \in \mathbb{F}_2^m$, then for $x \in \mathbb{F}_2^n$,

$$h_{A,b}(x) = Ax + b.$$