Analysis of Algorithms I: Basic Hashing

Xi Chen

Columbia University

Hashing is a great practical but sometimes mysterious technique. The goal here is to build a data structure to support the dictionary operations: Insert, Delete, and Search.

Let $U = \{0, 1, \ldots, D-1\}$ denote a universe set. We call the integers $k \in U$ keys. In most situations, $D = |U|$ is very large. The goal is to design a data structure (a hash table T here) to maintain a subset $S \subset U$ of keys. Starting with $S = \emptyset$, we need to handle any sequence of n operations each of the form:

$$\text{Insert}\,(T, k) : S \leftarrow S \cup \{k\}$$

$$\text{Delete}\,(T, k) : S \leftarrow S - \{k\}$$

$$\text{Search}\,(T, k) : \text{return 1 if } k \in S \text{ and 0 otherwise}$$

for some key $k \in U$.

An application of a dictionary data structure: Counting the number of distinct keys in a sequence A of n keys from U.

1. Set $t = 0$ and initialize T, a dictionary data structure
2. For i from 1 to n
3. If Search$(T, A[i]) = 0$
4. $t = t + 1$ and Insert$(T, A[i])$

Given any input sequence A of n keys, the execution of this algorithm makes a sequence of $O(n)$ calls to the dictionary operations. An observation that will be useful in the next class when we discuss Universal Hashing is that, given any input A, this sequence of $O(n)$ calls, including both the type of each call and the key used in each call, is fixed and is independent of our implementation of the dictionary data structure (and in particular, independent of the hash function).

Ideally, it would be great to have a dictionary data structure that, for any sequence of n operations, takes $O(n)$ time in total. Is this achievable? (We will see later that search trees have worst-case running time $O(\lg n)$ for each operation and $O(n \lg n)$ in total, though they support more than just the dictionary operations.)

The simplest implementation of a dictionary is the direct-address table discussed in Section 11.1. However, it uses a table / array of size $|U|$ and is impractical if the size of the universe set is huge. For example, consider the problem of counting distinct numbers in a sequence of length n, where the range is $\{0, 1, \ldots, 2^n\}$.

Now what is a hash table? A hash table $T[0, 1, \ldots, m-1]$ is simply an array of length m. For now consider m to be of order similar to n and is much smaller than $|U|$. We will see how to choose m later (depending on what we want to get from T).

A hash function is then a map h from U to $\{0, 1, \ldots, m-1\}$. It maps a key $k \in U$ to a slot $h(k)$ of the hash table T. Ideally, a hash table supports the three operations as follows:

1. Search (T, k): If k is stored in slot $h(k)$, return 1; otherwise 0
2. Insert (T, k): Store k in slot $h(k)$ of T
3. Remove (T, k): If k is stored in slot $h(k)$, remove it

Of course there is a serious problem with this implementation. What if h maps multiple keys to the same slot? If we insert two keys and they are mapped to the same slot, we cannot just replace the first key stored in that slot by the second one. We call this situation a collision with respect to a hash function h: two keys k and $k' \in U$ that satisfy $h(k) = h(k')$.

The hope is that we pick a "good" hash function that spreads the keys evenly over the slots so that when, e.g., m is a little larger than n, no collision happens during any sequence of n operations. This means whenever we insert a key into the table, it is mapped by h to a slot not used yet. But is this possible?

A typical hash function is the division method:

$$h(k) = k \mod m \in \{0, \ldots, m-1\}, \qquad \text{for any } k \in \{0, \ldots, D-1\}$$

where h maps k to the reminder of k divided by m. If we use this function and insert two keys k and $k' \in U$ with the same reminder, collision happens. What if we use a "better" hash function?

Impossible, as long as $|U| > m$ (recall that usually $|U| \gg m$). For any function $h : U \to \{0, 1, \ldots, m-1\}$, there must exist two keys $k \neq k'$ in U such that $h(k) = h(k')$. The so-called Pigeonhole Principle: No matter how we put D balls in m bins, as long as $D > m$, there must be a bin ends up with more than one ball.

Now we see that collision is unavoidable, how to deal with it? Chaining. For each slot j of T, we create a linked list $L[j]$ to store all the keys mapped to this slot so that we do not lose any keys. Using this strategy, we change the three operations accordingly:

1. Search (T, k): Search for k in the linked list $L[h(k)]$
2. Insert (T, k): Search (T, k) first. If it returns 0, we insert k at the head of the linked list $L[h(k)]$
3. Remove (T, k): Remove k if it is in the list $L[h(k)]$

The worst-case running time of any operation on $k \in U$ clearly is linear in the number of collisions between k and the keys currently in the hash table: Let S denote the current set of keys in the hash table. Then the worst case running time of Search(T, k), Insert (T, k) or Remove(T, k) is linear in

$$\text{length of } L[h(k)] = \text{number of } k' \in S \text{ with } h(k') = h(k)$$

It is not surprising that the number of collisions is crucial to the performance of a hash table.

However, we show below that whenever $|U| \geq nm$, no matter which hash function h is used, one can easily construct a sequence of $O(n)$ operations such that the hash table with h takes time $\Omega(n^2)$. To see this, we use the Pigeonhole Principle again and for any h, there are n keys $k_1, \ldots, k_n$ from U that collide with each other:

$$h(k_1) = h(k_2) = \cdots = h(k_n)$$

Then a nightmare for the hash table with h is the following sequence: First insert $k_1, \ldots, k_{n-1}$ into the hash table; then search k_n for n times. Check that the hash table with h takes $\Omega(n^2)$ time to handle this sequence of $2n - 1$ operations.

The situation is somewhat similar with Quicksort. We have two great techniques with very good reputation. However, it is very easy to beat them with simple worst-case examples. One way to justify their success is to use average-case analysis, instead of worst-case analysis.

In Quicksort, it is shown that if the input sequence is a random permutation, then the expected running time of Quicksort is $O(n \lg n)$. Also in Section 11.2, it is shown that if the keys in a sequence of n operations are equally likely to hash into any of the m slots (the assumption of simple uniform hashing), then by setting $m = n$ the expected running time of a hash table is $O(n)$.

Instead, we will make no assumption on the keys to hash but use randomization. Roughly speaking, we will "randomly" construct a hash function every time we are asked to build a hash table and to handle a sequence of dictionary operations. A clear advantage is that an adversary cannot predict the hash function we actually use and it becomes almost impossible to slow down a hash table with collisions. This is kind of similar to the idea behind Randomized Quicksort. Instead of making an assumption on the input, we use randomization to show that a hash table of size $m = n$ can handle any sequence of n dictionary operations with running time $O(n)$ in expectation, when a hash function is "randomly" picked.