Lecture 19

Analysis of hashing

This lecture presents theoretical results, some without proof. The analysis of hashing leads to some interesting and tricky mathematics.

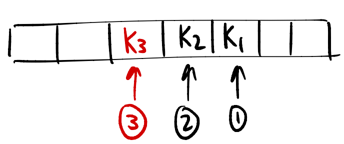
Let's start with a look at a basic result for any table ADT. This is Lemma 3.5 in the textbook.

Lemma 19.1. Suppose that a table is built up from empty by successive insertions, and we then search for a key k uniformly at random. Let $T_{ss}(k)$ (respectively $T_{us}(k)$) be the time to perform successful (respectively unsuccessful) search for k. Then

$T_{us}(k)$ = time of unsuccessful search

- the time taken to retrieve, delete, or update an element with key k is at least $T_{ss}(k)$;
- the time taken to insert an element with key k is at least $\cancel{T_{us}(k)}$; $1 + T_{us}(k)$;
- the average value of $T_{ss}(k)$ equals one plus the average of the times for the unsuccessful searches undertaken while building the table.

Example 19.2. Prove the last statement



Want to put k_3 in

1. look at k_1, make comparison, then probe
2. look at k_2, compare, then probe
3. Probe, see if it's empty,

therefore $= T_{us}(k)$

and then insert $= +1$

$$E\left[T_{ss}(k)\right] = E\left[1 + T_{us}(k)\right]$$

$$= E[1] + E\left[T_{us}(k)\right]$$

$$E\left[T_{ss}(k)\right] = 1 + E\left[T_{us}(k)\right]$$

19.1 Analysis of balls in bins

The probability of no collisions when n balls are thrown into m boxes uniformly at random is $Q(m, n)$ where

first ball into empty bin

second ball into bin, but it already has first ball $\therefore -1$

$$Q(m, n) = \frac{m!}{(m-n)!m^n} = \frac{m}{m}\frac{m-1}{m} \dots \frac{m-n+1}{m}.$$

Note that $Q(m, 0) = 1, Q(m, n) = 0$ for $n > m$. At least one collision when all balls thrown.

For example, $Q(366, 180) \approx 0.4487 \times 10^{-23}$ (negligible chance of no collision when 180 balls thrown into 366 boxes) while $Q(366, 24) \approx 0.4627$ (more likely than not to have a collision when throwing 24 balls in 366 boxes).

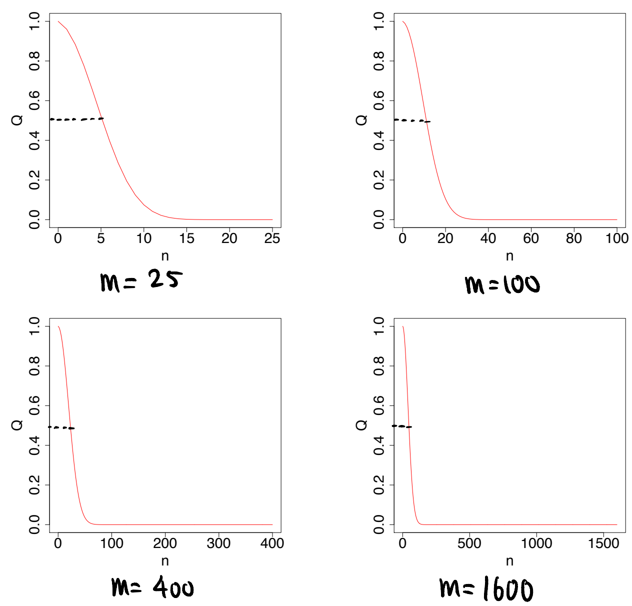


Figure 19.1: Plots of $Q(m, n)$ against n for $m = 25, 100, 400, 1600$

19.2 Analysis of chaining

With chaining, the worst case for searching is $\Theta(n)$, since there may be only one chain with all the keys. The average cost for an unsuccessful search is the average list length, namely λ. $\lambda = \frac{n}{m}$

The average cost for a successful search is then

$$\frac{1}{n}\sum_{k=1}^{n}(1 + \frac{k-1}{m}) = 1 + \frac{n-1}{2m} \approx 1 + \frac{n}{2m} = 1 + \lambda/2.$$

$$\boxed{\frac{1}{n}\sum 1 + \frac{k-1}{m}} = \sum 1 + \sum \frac{k-1}{m}$$
$$= n + \sum \frac{k}{m} - \sum \frac{1}{m}$$
$$= n + \frac{1}{m}\frac{n(n+1)}{2} - \frac{1}{m} n.$$
$$= n + \frac{1}{m}\left(\frac{n(n+1)}{2} - \frac{n^2}{2}\right)$$
$$= n + \frac{1}{m}\left(\frac{n^2 - n}{2}\right)$$
$$= 1 + \frac{1}{m}\left(\frac{n-1}{2}\right)$$

Thus provided the load factor is kept bounded, basic operations all run in constant time on average.

19.3 Analysis of open addressing

We assume ***uniform hashing***: each configuration of n keys in a table of size m is equally likely to occur. In other words, the hash function produces a uniformly random permutation of the keys.

Uniform hashing is a theoretical model and cannot be implemented practically but is a good approximation when double hashing is used.

We show in Section 19.4 that the average cost for insertion (unsuccessful search) under uniform hashing is $\Theta(1/(1-\lambda))$ as $m \to \infty$.

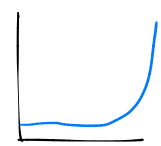
As $\lambda \to 1$, $\frac{1}{1-\lambda} = \frac{1}{0} \therefore \infty$

Linear probing is not well described by the uniform hashing model (because of the clustering). The average insertion cost is $\Theta(1 + 1/(1-\lambda)^2)$ (proof omitted). $\to \infty$ quite fast if $\lambda \to 1$

If the load factor is bounded away from 1, basic operations run in constant time; otherwise performance will be very bad. We should resize as λ grows!

Example 19.3. Sketch the functions $f(\lambda) = 1 + 1/(1-\lambda)^2$ and $g(\lambda) = 1/(1-\lambda)$

λ	$\frac{1}{1-\lambda}$	$1+\frac{1}{(1-\lambda)^2}$
0.5	2	5
0.75	4	17
0.9	10	101
0.99	100	10001



19.4 Unsuccessful search under uniform hashing hypothesis

We'll show that the average number of probes in unsuccessful search is $1/(1-\lambda)$.

- Let X be the number of probes taken for an unsuccessful search. Let p_i be the probability that we need to make exactly i probes (exactly $i-1$ probes hit an occupied cell). And let P_i be the probability we need to make at least i probes. Clearly $p_i = P_i - P_{i+1}$.

- Then

$$E[X] = \sum_{i=1}^{n} i p_i \leq \sum_{i=1}^{\infty} i(P_i - P_{i+1}) = \sum_i P_i$$

where the last equality comes from a telescoping argument.

$\sum i(P_i - P_{i+1}) = 1$(at least 1 − at least 2)
+2(at least 2 − at least 3)
+3(at least 3 − at least 4) ...

only one of these

- For $i \geq 1$, we have

$$P_i = \frac{n}{m}\frac{n-1}{m-1}\cdots\frac{n-i+1}{m-i+1} \leq \lambda^{i-1}.$$

- Thus

$$E[X] \leq \sum_i P_i \leq \sum_i \lambda^{i-1} = \frac{1}{1-\lambda}$$

- It can be shown the bound is tight the average number of probes in an unsuccessful search is $\Theta(1/(1-\lambda))$.
- The time for successful search is $\frac{1}{\lambda}\ln\left(\frac{1}{1-\lambda}\right)$.

Reasonable if $\lambda << 1$

19.5 Statistics for balls in bins: some facts without proof

- When do we expect the first collision? This is the ***birthday problem***. Answer: $E(m) \approx \sqrt{\pi m/2} + 2/3$. So collisions happen even in fairly sparse tables. $m = 365$
- When do we expect all boxes to be nonempty? This is the ***coupon collector*** problem. Answer: After about $m \log m$ balls. It takes a long time to use all lists when chaining. linearithmic
- What proportion of boxes are expected to be empty when n is $\Theta(m)$? Answer: $e^{-\lambda}$. Many of the lists are just wasted space even for pretty full tables. $n = m$ $\quad e^{-m/n} = e^{-1} \approx 0.36$
- When m is $\Theta(n)$, what is the expected maximum number of balls in a box? Answer: About $(\log n)/(\log\log n)$. Some of the lists may be fairly long but not very long. However can reduce to loglogn

19.6 Hashing in practice (Dec 2018)

- Java Collections Framework uses chaining to implement `HashMap`, resizing when $\lambda > 0.75$, and table size a power of 2.
- C++ uses chaining to implement `unordered_map`, resizing when $\lambda > 1$, and prime table size.
- C# uses chaining, resizing when $\lambda > 1$, and prime table size.

- Python uses open addressing, resizing when $\lambda > 0.66$, and table size a power of 2.