CS124 Lecture 15 Notes

Scribe: Ashley Zhuang

March 28, 2022

1 Hashing with Chaining

Recall that hash functions are functions of the form $H : U \to \{0, 1, \ldots, m-1\}$. In CS124, we will take hash functions as a blackbox with the following properties: it is a deterministic function, but looks random (independent and uniform). In other words, $x \neq y \implies \Pr[H(x) = H(y)] = 1/m$. Check out CS223 for more on hash functions.

Last time, we discussed sets as an application of hashing. For this, we stored an m-bit array. To add x to the set, we set $H(x)$ to 1 in the array. To check if y is in the set, we check if $H(y)$ is 1 in the array. With this method, we will have false positives; e.g. if $m/n = 8$, the false positive probability is approximately $e^{-1/8} \approx 12\%$.

To get 0% error, we would have to store the x's in our set, rather than just storing 1 at it's hash value, since we need to check that we actually have x when we have an input y that hashes to $H(x)$. This will cost us extra memory.

More formally, we can store an array of m linked lists. Then, each time we add an element x to our set, we add it to the linked list stored at index $H(x)$. This gives us expected lookup time $O(1 + \frac{n}{m})$ (we analyzed this in detail in section last week).

	Lookup time	Space	False Positive Prob.
Hash with collisions	$O(1)$	$O(m)$	$1 - e^{-n/m}$
Bloom filter	$O(m/n)$	$O(m)$ bits	$(.6185)^{m/n}$
Hash with chaining	$O(1 + m/n)$ exp.	$O(m)$ words + items	0

Table 1: Comparison between different implementations of set membership data structures.

2 Table Doubling

However, we don't always know n ahead of time, but we want to keep our expected time $O(1 + \frac{n}{m})$ small even as n increase (i.e. as we add more items to our set). We'll achieve this with table doubling: whenever $n \geq m/2$, we destroy the table and rebuild it, now with size $2m$.

We claim that with table doubling, we have $O(1)$ amortized time/insert.

2.1 Amortized Analysis: Direct Proof

Consider doing a series of inserts, starting with a table of size 1:

m	n	Rebuilding work
1	0	
2	1	1
4	2	2
4	3	0
8	4	4

We see from the pattern above that when $n = k$, we have $m = 2^{\lfloor \log k \rfloor + 1} \leq 2k$. Thus, the total work for k inserts is

$$2^{\lfloor \log k \rfloor} + \frac{1}{2}2^{\lfloor \log k \rfloor} + \ldots + 1 = 2^{\lfloor \log k \rfloor} \cdot 2 \leq 4k = O(k).$$

2.2 Amortized Analysis: Money Proof

The intuition for this proof is that we can store $O(1)$ dollars for every insert, and then use the money that we've stored up to pay for the expensive $O(n)$ cost of table rebuilding.

For every insert, store \$2. For every rebuild, gather all the money. If we are rebuilding at size n, then our last rebuild was at size $n/2$. This means we've inserted $n/2$ times since the last rebuild, so we have $n/2 \cdot 2 = n$ dollars stored up, which is exactly how much we need to rebuild.

2.3 Amortized Analysis: Potential Energy Proof

Define our potential function for a table of size m with n elements to be

$$P(n, m) = 2(n - \frac{m}{2}).$$

We define work to be

$$\text{Work} = \text{time} + \triangle P.$$

When we do an insert that doesn't double our table, the time is $O(1)$ and the change in potential is $O(1)$ (since only n increases by 1). Thus, the work to insert is $O(1)$. When we do an insert that does double our table, the time to rebuild is n and the change in potential will be $-n+2$, so the total work to rebuild is also $O(1)$. Thus, the total work for any insert operation is $O(1)$ amortized.

If we modify our method so that we can also remove items (specifically, when we remove an item, if $n \leq 4m$ we will destroy the table and rebuild at size $2n$), the same potential energy function should still work for this analysis.

3 Fingerprinting

Suppose we're trying to find a pattern string P of length k (where k is very big) as a substring of a document D. The algorithm we will use to achieve this proceeds as follows:

1. Hash the pattern P.
2. Hash each length k substring of D.
3. Double check matches.

For example, if we have $D = 6386179357342$ and $P = 17935$, we would have $k = 5$ and $n = 14$ here. We would hash 63861, then 38617, then 86179, etc. in step 2 of the algorithm (it's like a sliding window). Then, we can check if any of these hashes match the hash of P, and then if we get a match, we should double-check that the substring actually matches P to avoid false positives.

If hashing is $O(1)$, this algorithm is $O(n)$. But how do we hash in time $O(1)$ if we are hashing k-length substrings? Surely it should take at least as long as the size of the input to calculate the hash value of some input. Then, if hashing is $O(k)$, this algorithm is $O(nk)$ (this is expensive). Thus, to do better, we'll choose our own hash function here. We'll treat the strings P and D as sequences of digits for simplicity.

Specifically, our hash function H will be $H(x) = x \mod p$, where p is some prime smaller than 10^k. Going back to our example where $P = 17935$, if $p = 251$, then $H(P) = P \mod p = 114$. Thus, when we are hashing the substrings of D, we'll get $63861 \mod 251 = 107$, then $38617 \mod 251 = 214$, and so on as our hash values. Each of these mods still takes $O(k)$ time if we are brute-forcing. However, notice that we can be smart here: 63861 and 38617 overlap in 4 digits. Similarly, 38617 and 86179 also overlap in 4 digits; specifically, $86179 = 10 \cdot 38617 + 9 - 3 \cdot 10^5$. Thus, once we've done the work to calculate the first hash, we can calculate the next one in $O(1)$ time. For example, if we already have our hash $38617 \mod 251 = 214$, then $86179 \mod 251 = 10 \cdot 214 + 9 - 3 \cdot 149 = 86$.

Note that $17935 \mod p = 114$, but also $57342 \mod 114 = 114$ too. Thus, we'll get a false positive here unless we double check matches (which would mean we need to actually store P). Suppose we choose a random prime for our p. Let's find the probability of a false positive/double-check. If we have a false positive x, this means that $x \equiv P \pmod{p} \implies x - P \equiv 0 \pmod{p}$. Thus, we want to know how many primes could possibly divide $x - P \leq 10^k$. If t primes p_i (which are all at least 2) divide $x - P$, then $|x - P| \geq p_1 p_2 \ldots p_t \geq 2^t$. Thus, $2^t \leq 10^k \implies t \leq k \log 10$. Another useful fact: the number of primes less than or equal to some number x (which we'll denotes as $\pi(x)$) satisfies

$$x/\log x \leq \pi(x) \leq 2(x/\log x)$$

for $x > 2$. Hence, the false positive probability is bounded by $\frac{k \log 10}{\pi(x)}$. Exercise: How big should we choose p to make so that the false positive probability is less than 0.01?

One final loose end we have is how to choose a random prime. We'll discuss this in a later lecture.