COMP 480/580 — Probabilistic Algorithms and Data Structure Sep 5, 2023

Lecture 5: Analysis of Hashing, Chaining, and Probing

Lecturer: Anshumali Shrivastava Scribe By: Z. Lu zfl1, C. Wells crw16, D. Zhu dsz1

1 Hash Table Collisions

1.1 Chaining Analysis

1.1.1 How Chaining Works

Hash m objects into an array of size n. If multiple objects are hashed to the same location, store them in a linked list called a "chain."

Definition 1 *Let* $\alpha = \frac{m}{n}$

We call α the "load factor." Note that α is the expected chain length of an arbitrary slot in our hash table.

1.1.2 Best and Worst Cases

Using these values, we will now analyze searching and insertion using chaining with a 2-universal hash function, $h(x) = (ax + b \mod P) \mod n$.
Note that searching and insertion will take at least $O(1)$ for finding the slot we hash to for an object. We then have to scan down the entire length of the chain in that slot either to insert at the end or to find the thing we're searching for.
This gives us a worst case search/insert of $O(M)$, as that is the longest possible chain if all our objects are hashed to the same slot. In the average case, however, we have $O(1+\alpha)$, as it costs $O(1)$ to execute the hash function and the expected length of the chain is α.

Discussion 1 *In actual applications, it often doesn't matter whether the hash function is independent. This is because the keys themselves, whether they be input urls or querys or other user data, are generated independently by random user actions. Thus, many hash functions that theoretically do not have as good of guarantees on independence and collisions can in practice still give results as good as required for many applications.*
This ***does not*** *apply in the case that we have malicious users - in that case we have to use strong independent hash functions. This only applies for benign users who generate independent keys.*

1.1.3 Finding Probabilities of K-Length Chains

Let us take $m = n$ for simplifying calculations. We want to get some more specific bounds as to the probabilities we have chains of a certain length.

Theorem 1 *The probability we have a chain of length k is* $\binom{n}{k}(\frac{1}{n})^k$*, which is less than* $\frac{1}{k!}$

Proof 1

$$\binom{n}{k}(\frac{1}{n})^k = \frac{n!}{k!(n-k)!n^k} \tag{1}$$

$$\forall k \in \mathbb{N}, n! < (n-k)!n^k \tag{2}$$

$$\frac{n!}{(n-k)!n^k} < 1 \tag{3}$$

$$\binom{n}{k}(\frac{1}{n})^k < \frac{1}{k!} \tag{4}$$

If we let $k = \frac{3 \ln n}{\ln \ln n}$, then $k! > n^2$.
Thus, the probability of any n slots having more than $\ln n$ keys is $< \frac{1}{n}$.

1.2 Power of K Choices

If, for whatever reason, $\ln n$ upper bound is not tight enough for our purposes, we can use this idea:

Discussion 2 *If you run a probabilistic algorithm k times, and choose the "best" outcome from those, your results will be exponentially better.*

1.2.1 How K Choices Work

How this would work with respect to this example is we would take k arrays and have associated with each one a hash function with independently generated a, b. Then, when we get a new object, we hash it with every function and insert it into the one with the smallest chain. This comes with the downside of having to search multiple chains, but the upside is that our chains are much smaller.

1.2.2 Analysis of K Choices

We end up having exponentially shorter chains, in fact, because for a chain to grow, it needs to have every other chain that the object is hashed to be of equal or greater length - and as k increases, the probability of that becomes exponentially smaller. In our specific example, if we used $k = 2$, we already get that the probability of a chain being longer than $\ln \ln n < \frac{1}{n}$, which is exponentially better than $\ln n$.

1.3 Linear Probing

1.3.1 How Probing Works

Say we have a very simple hash function of $h(x) = x \mod 10$, and we are hashing into a table of size 10. Let us say we have already inserted $7, 8$, and 10. When we insert 17 into our table, we will have $h(17) = 7$, and it will collide with the 7 already in the table. We then attempt to insert 17 into the $8th = 7 + 1$ slot of the array, but we also find that that is already taken. We then continue onto the $9th = 7 + 2$ slot of the array, and, having found an open spot, insert our element.

Discussion 3 *In practical applications, we prefer linear probing. This is because we load our table into the cache, and then most of the time, all the probing will examine memory that we*

have already loaded into the cache. In contrast, chaining deals with pointers and dynamically accessing main memory, which is significantly slower from a systems perspective. Further, if we use a sufficiently big table, we can get constant time probing.
One pitfall we have to be aware of is that if we have multiple long probing sequences that become connected to each other, we can have disastrous results with runtimes in linear probing - in contrast, having two long chains hashed right next to each other when using chaining is not any worse than them being anywhere else.

1.3.2 Analysis of Probing

It is not in practice useful for us to analyze hashing objects into an individual slot in our array. We thus consider an entire region in the array for our analysis, i.e. given that we insert an object in a region, what is the probability that, after following the probing, we are still in that region. We also assume $\alpha \leq \frac{1}{3}$ - this is because, as noted in the discussion above, we have disastrous results when m approaches n.
Thus, if we analyze some region, S, the expected number of elements in that region is $\frac{S}{3}$. The number of elements at which we start to see problems with searching and insertion is roughly $\frac{2S}{3}$. Thus, we want to find bounds for the probability that the number of elements in that region is $\frac{2S}{3}$, and we do so using Chebyshev's inequality.

Discussion 4 *We may be tempted to use some other form of uninformed probing, as in if we hash to slot i, we then probe to i^2 and then i^3 or $2i$ and then $4i$ instead of $i+1$ and $i+2$. However, in practice this doesn't have guarantees any better than linear probing, and we lose many of the benefits of cache locality in doing so.*

1.4 Cuckoo Hashing

For chaining and probing, our worst cases are always $O(m)$. Our expected is $O(1)$, but if we cannot stand that, we could use Cuckoo Hashing, which sacrifices insert for worst-case $O(1)$ search.

1.4.1 How Cuckoo Hashing Works

We create two different arrays, with different cheap hash functions, h_1, h_2 for each. We enforce only one property - the only locations that an object could be are $h_1(x)$ and $h_2(x)$. When we insert something, we insert it in the first array at $h_1(x)$, and if something, y, was there, we pop it out and insert it to the second array at $h_2(y)$. If something were there, we then repeat, and hash it to $h_1(z)$, until we find an empty spot. We can sometimes get a cycle, however, and in that case we have to re-seed our functions and re-hash all the objects that are already in our arrays.

1.4.2 Analysis of Cuckoo Hashing

If we have each of the arrays be of size roughly $1.5m$, we get that the probability of cycling per insertion is $\frac{1}{1{,}000{,}000}$. This is fine for when we're inserting around $10,000$ elements, but we may need another fix if we want to insert a greater number. The most cost-effective way of doing this is just by adding more arrays and hash functions - similarly to with the power of K choices, we continue to get exponentially better returns for each array we add, with only minor cost increases to search.

2 Bloom Filters

Bloom filters are a powerful tool for approximate membership testing with limited storage. We will use a problem to illustrate how they can be used.

2.1 Problem

Imagine you have a hash map $h : strings \rightarrow [0 : 999]$. Imagine we have a set of 100 malicious URLs that we want to test against user's querys to see if we should flag the site to them. We **cannot** tolerate any false negatives (i.e. allowing a user unwarned to access a malicious site). Further, we only have 1000 bits that we can use, so we cannot store the malicious strings in a dictionary, as the strings themselves are around $40,000$ bits, assuming each string is on average 50 8-bit characters long.

2.2 Naive Bit-Map Solution

The naive solution to this problem is to simply use our hash map to hash all of our 100 urls into a bitmap of 1000 bits, setting every bit that corresponds to the hash of a malicious string. Assuming the worst case, i.e. no collisions, we will have 100 bits set. Now, when we receive a query from the client, we simply hash their URL, and if it corresponds to a set bit, we pass it along to the server for a more thorough check against our dictionary. If it does not, we allow the request to pass unchecked. This will result in no false negatives, because our hash function definitionally will hash any of our malicious URLs to a set bit. However, it will result in a roughly 10% false positive rate, which is fairly high, especially when those server transactions are expensive. Across N searches, the probability that we get a false positive is $1 - (1 - \frac{1}{R})^N$, where R is the ratio of bits we have access to to malicious strings. If this simply isn't good enough, what can we do to improve it?

2.3 Bloom Filter Solution

Instead of having only 1 hash function, let us use the power of k choices to our advantage, and use k hash functions. Now, for each malicious URL on setup, we set the bits $h_1(x), h_2(x), \ldots h_k(x)$. Then, when a client comes along with a query, q, we do the expensive server access if and only if all of $h_1(q), h_2(q), \ldots h_k(q)$ are set. This, as in the other cases in which we used the power of k choices, provides us with exponentially better results.
The only thing we have to be careful of are if we use too many hash functions for the size of our array, we could set nearly every bit, which would then lead to an increase in the number of false positives.
This solution gives us significantly better probabilities, at only a minor cost per transaction of running k hash functions instead of 1 - which is a minor difference, as we can reasonably use cheap hash functions due to the nature of the requests, as noted in discussion 1.