COMP480, Fall 2024
`https://www.cs.rice.edu/~as143/COMP480_580_Fall24/`.
Instructors: Anshumali Shrivastava, Rice University.

Lecture 1, 2024-09-12. Scribed by David Su (js202), Yimo Wang (yw104), Mary Nam (mn80).

1 Materials from the last class

1.1 k-universal hashing family

We hash m objects into an array of size n. Upon collision, we list the objects together, forming a chain. The $E[chainlength] \leq 1 + \frac{m-1}{n}$. Following convention, we can define $\alpha = \frac{m}{n}$.
Then, the worst-case insertion and search time for hashing with chaining is $O(m)$, and occurs in the case where all m elements collide. In the average-case, the expected insertion and search time for hashing with chaining is $O(1+\alpha)$; $O(1)$ time to compute the hash, and $O(\alpha)$ time to navigate the length of the chain.
This is generally an acceptable runtime. However, is it possible to achieve better runtime, such as $O(ln(n))$? To achieve this runtime, we need to consider the probability that there exist a chain of $size \geq log(n)$.

Theorem: For the special case where $m = n$, with probability at least 1-1/n, the longest list is O(ln n / ln ln n).
Proof: Let $X_{i,k}$ be the indicator that key i hashes to slot k, with $\Pr(X_{i,k} = 1) = \frac{1}{n}$. The probability that a particular slot k receives more than κ keys, where $m = n$, can be determined by assuming a high level of independence. If we choose $\kappa = \frac{3 \ln n}{\ln \ln n}$, then it follows that $\kappa! > n^2$ and $\frac{1}{\kappa!} < \frac{1}{n^2}$. Consequently, the probability that any of the n slots receives more than κ keys is less than $\frac{1}{n}$.

1.2 Power of k choices

We want to further reduce the length of the chain. How can we do better?

Intuition: Use k hash functions $h_1(x), h_2(x), ..., h_k(x)$, and insert the element into the location with the smallest chain. The resulting chains are exponentially shorter (and thus, exponentially better!). Because in order for a chain i to grow, we must satisfy the condition that all chains j, $j \neq i$, has already grown to equal or greater length.

For example, let $k = 2$. Examining this, we see that P(chain length $\geq \ln(\ln(n))) < \frac{1}{n}$. Recall that with $k = 1$, if $m = n$, P(chain length $\geq \ln(n)) < \frac{1}{n}$. With 2 chains, we are already at an exponential improvement from 1 chain.

2 Linear probing

Consider a hash function $h(x) = x \mod A$, where A is the size of the array. When we perform linear probing, we follow a **probe sequence** until we find an open spot in the array. Define a simple probe sequence as follows:

$$\begin{aligned}
\text{0th probe: } & h(k) \mod \text{A} \\
\text{1st probe: } & (h(k)+1) \mod \text{A} \\
\text{2nd probe: } & (h(k)+2) \mod \text{A} \\
& \ldots \\
\text{ith probe: } & (h(k)+i) \mod \text{A}
\end{aligned}$$

Let's now perform a simple example of insertion. Using the hash function above on an array of size 10, we attempt to insert 38, 19, 8 in that order.

1. Insert 38: $38 \mod 10 = 10$, so we insert 38 at index 8.
   $[\ ,\ ,\ ,\ ,\ ,\ ,\ ,\ , 38,\ ]$
2. Insert 19: $19 \mod 10 = 9$, so we insert 19 at index 9.
   $[\ ,\ ,\ ,\ ,\ ,\ ,\ ,\ , 38, 19]$
3. Insert 8: $8 \mod 10 = 8$, but index 8 is already occupied, so we check $(8+1) \mod 10 = 9$. Index 9 is also already occupied, so we check $(8+2) \mod 10 = 0$. Index 0 is empty, so we insert 8 at index 0.
   $[8,\ ,\ ,\ ,\ ,\ ,\ ,\ , 38, 19]$

In practice, linear probing is one of the fastest hashing strategies. What makes it good?
Memory: Only requires an array and a hash function to be stored.
Locality: Due to the nature of how probing is done, in the unfortunate event of collisions, we only need to search in adjacent locations, making it easy to traverse.
Combined: Combining the low memory overhead and excellent locality, we get a great cache performance from linear probing.

2.1 Expected cost of linear probing

For simplicity, let's assume a load factor of $\alpha = \frac{m}{n} = \frac{1}{3}$.
Let's denote a "region" of size m to be a consecutive set of m locations in the hash table. Then, an element q hashes to region R if $h(q) \in$. Note that due to probing, q may not ultimately be placed in R. Given this load factor, a region of size 2^S would be expected to have at most $\frac{1}{3}2^S$ elements in it. It would be very unlucky if a region had twice as many elements in it as expected. A region of size 2^s is **overloaded** if at least $\frac{2}{3}2^s$ elements hash to it.

We want to show that the probability of this unlucky event is very low.

Theorem: The probability that the query element q ends up between 2^s and 2^{s+1} steps from its home location is upper-bounded by $c\cdot$ P(the region of size 2^s centered on $h(q)$ is overloaded) for some fixed constant c independent of S.
The proof for this theorem is outside the scope of this class. For interested individuals, see https://arxiv.org/abs/1509.04549.

Overall, we can write the expectation as :

$$\begin{aligned}\mathrm{E}(\text{lookup time}) &\leq O(1)\sum_{1}^{\log(n)} 2^{s+1}\cdot \mathrm{P}(q \text{ is between } 2^s \text{ and } 2^{s+1} \text{ slots away from } h(q)) \\ &= O(1)\sum_{1}^{\log(n)} 2^{s}\cdot \mathrm{P}(\text{the region of size } 2^s \text{ centered on } h(q) \text{ is overloaded})\end{aligned}$$

For query q, let B_s be the number of keys that hash into the block of size 2^s centered on $h(q)$. $\mathrm{P}(B_s \geq \frac{2}{3}\cdot 2^s) =$? That is, what is the probability that B_s is overloaded? Assuming h is at least 2-independent, $\mathrm{E}(B_s) = \frac{1}{3}\cdot 2^s$.

$$\mathrm{P}(B_s \geq \frac{2}{3}\cdot 2^s) = \mathrm{P}(B_s \geq 2\cdot \mathrm{E}(B_s)).$$

Thus,

$$\mathrm{E}(\text{lookup time}) \leq \sum_{1}^{\ln(n)} 2^s \cdot \mathrm{P}(B_s \geq 2\cdot \mathrm{E}(B_s))$$

Variance: Assuming 3-independence and using Chebyshev inequality, we can see that E(lookup time) $\leq O(\log(n))$.

3 Mark and recapture

Goal: understand how randomized estimation process works
Problem setting: Count Turtles in a Pond.
Option: take a random sample of the pond.Mark the captured turtles with a tag. Then release them. When you capture them again, if every turtle that comes up already has a tag, you can be reasonably certain that all turtles have been tagged.

Let's say I capture K_1 of n total turtles, mark all K_1 of them, and put them back in the pond. Now, after 10 days (why 10 days? This allows use to assume a uniformity – that the tagged turtles sufficiently mix with the untagged ones, guaranteeing the next sample will be randomized.) I capture another K_2 turtles, and find that M of them are marked. So $\frac{M}{K_2} \approx \frac{K_1}{n}$. That is $\frac{M}{K_2}$ should represent the fraction of marked turtles in the pond. Then, $n \approx \frac{K_1 K_2}{M}$.
In this problem, we are making an assumption: the 10 days of mixing creates a uniform distribution of the tagged and untagged turtles.

3.1 In terms of Indicator variables

Create a random variable for each turtle:

For the n turtles, we have $X_1, ...X_n$.

$X_i = 1$ if turtle i is marked, else 0.

After the first capture, we have $\sum_{i=1}^{n} X_i = K_1$.

After the recapture of K_2 turtles, we have $M = \sum_{i=1}^{K_2} X_i$ are marked.

$P(X_1 = 1) = \frac{K_1}{n}$ = probability of any given turtle being marked.

$E[X_1] = E[X_2] = ... = E[X_i] = \frac{K_1}{n}$.

Note that for all $i, j \in 1, ..., n$, X_i and X_j are correlated. That is, if you know that X_i is marked, your belief of whether X_j is marked changes.

$$E(M) = \sum_{i=1}^{K_2} \frac{K_1}{n}$$

$$E(M) = \frac{K_1 K_2}{n}$$

Let us now revisit the expression we previously derived: $n \approx \frac{K_1 K_2}{M}$.

The more accurate correct expression is $n = \frac{K_1 K_2}{E(M)}$.