COMP 480/580 — Probabilistic Algorithms and Data Structure Sept 9, 2025

Lecture 4: Diving Deeper into Chaining and Linear Probing

Lecturer: Anshumali Shrivastava

Scribe By: Andrew Chu (ahc12), John Tian (jt81), Arnan Bawa (ab223)

1 Introduction & Review of Previous Topics

1.1 Hashing

A perfect hash function is a function h that takes an object O and guarantees that if $O_1 \neq O_2$, then $h(O_1) \neq h(O_2)$. However, creating a perfect hash function efficiently is a hard open problem. Therefore, we must consider a the tradeoffs of relaxing the "perfect" condition and using k-universal hashing. This lecture explores this tradeoff and its implications on hash table performance.

1.2 Tail Bounds

Tail bounds, such as Markov, Chebyshev, and Chernoff bounds, allow us to analyze processes and set stronger limits on their behavior. Analysis that uses expectation (mean) only gets us so far; we should use tail bounds to examine the behavior of hash table collisions.

2 k-Universal Hash Functions

A hash function from a family H is k-universal if for any set $x_1, x_2, \ldots, x_k$, for h sampled from H, $h(x_1), h(x_2), \ldots, h(x_k)$ are independent random variables: $Pr(h(x_1) = h(x_2) = \cdots = h(x_k)) \leq \frac{1}{n^{k-1}}$, where n is the size of the set x_i is sampled from.

In previous lectures, we examined the 2-universal hash family. More generally, a k-independent family means our hash function is formed by a polynomial expression of degree k An example of such is the following:

$$h(x) = (a_k x^k + a_{k-1}x^{k-1} + \ldots + a_1 x + a_0) \bmod P \bmod R$$

There is a tradeoff between hash function performance and the chances of a collision. On one end of the spectrum, 2-universal hash functions are easy to compute but have weak guarantees regarding collisions. On the other end, perfect hash functions provide the highest guarantees regarding collisions but are difficult to create.

As we will soon see, using these easily computable hash functions can result in performance (time complexities) similar to that of using perfect hash functions with small probabilities of failure.

3 Hash Tables with Chaining

Previously, we examined how hash tables are constructed using k-Universal hashing and separate chaining to handle collisions. With this technique, keys that hash to the same value are appended to a linked list whose head is the entry in the hash table corresponding to the key's hash value:

3.1 Example

Suppose our key space is integers, and we have a hash table of size 10 indexed from 0 to 9 with a hash function $h(K) = K \bmod 10$. We attempt to insert keys in the following order: 7, 41, 18, 17

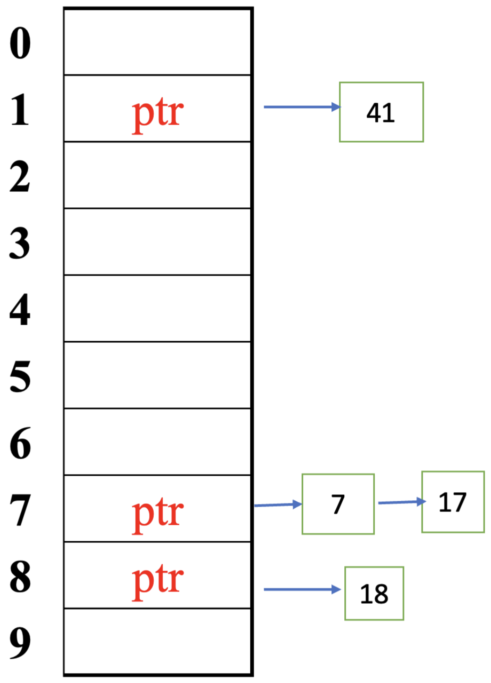

Figure 1: Chaining [2]

As seen in the image, keys that map to the same slot as a previous key, such as 17 which has the same hash value as 7, are chained to that same slot, forming a chain of 7 and 17.

3.2 Analysis of chaining

Since the hash function we use is k-universal, we benefit from the independence. If m elements are inserted into a hash table with n slots, the expected length of the chain is bounded by $1 + \frac{m-1}{n}$. The runtime of hash table operations then depends on the load factor, $\alpha = \frac{m}{n}$. Addition and insertion then have complexity $O(1 + \alpha)$. The worst case complexity, however, is still $O(m)$, when all elements hash to the same value.

It is interesting to note that the average case for this technique is $O(1)$. However, we must note that expected values do not capture the full picture. We must examine the probability of undesirable cases using tail bounds.

3.3 What is a good running time?

We aim for $O(\log(n))$ running time for our hash table. This leads naturally to the question: with chaining, what is the probability that a chain of length greater than or equal to $\log(n)$ will exist in our table?

3.4 Analyzing the probability of short chains

Theorem. *For the special case where $m = n$, the longest chain length is $O(\frac{\ln n}{\ln \ln n})$ with probability at least $1 - \frac{1}{n}$.*

Proof. [1] Let $m = n$ and $X_{i,k}$ be the indicator variable that key i hashes to slot k. With k-Universal hashing, we have $Pr(X_{i,k} = 1) = \frac{1}{n}$. Assuming lots of independence, the probability that a particular slot receives κ keys is

$$\binom{n}{\kappa} \frac{1}{n^\kappa} \left(1 - \frac{1}{n}\right)^{n-\kappa} < \binom{n}{\kappa} \frac{1}{n^\kappa} < \frac{n^\kappa}{\kappa!} \frac{1}{n^\kappa} = \frac{1}{\kappa!}.$$

We can then bound the probability that a particular slot receives κ or more keys is at most $\frac{n}{\kappa!}$, since the probability of having more than κ keys is less than $\frac{1}{\kappa!}$. By the union bound, the probability that any slot has chain length at least κ is at most the sum of the individual probabilities, $n \cdot \frac{n}{\kappa!} = \frac{n^2}{\kappa!}$. We examine this expression to find the value of κ allows us to have a probability of long chains to be at most $\frac{n^2}{\kappa!} = \frac{1}{n}$. We must have $\kappa = n^3$. To find such a value of κ, we use Stirling's approximation:

$$\ln \kappa! = \ln n^3 \geq \kappa \ln \kappa - \kappa$$

$$3 \ln n \geq \kappa(\ln \kappa - 1)$$

$$\frac{3 \ln n}{\ln \kappa - 1} \geq \kappa$$

Suppose $\kappa = \frac{3 \ln n}{\ln \ln n}$. Then, we have

$$\frac{3 \ln n}{\ln \frac{3 \ln n}{\ln \ln n} - 1} \geq \frac{3 \ln n}{\ln \ln n}$$

$$\frac{\ln \ln n}{\ln \frac{3 \ln n}{\ln \ln n} - 1} \geq 1.$$

For large enough n ($n \geq 3$), this holds. Thus, $\kappa = \frac{3 \ln n}{\ln \ln n}$, and so $O(\frac{\ln n}{\ln \ln n})$ is a bound for the longest chain length with probability $1 - \frac{1}{n}$.

□

3.5 Extracting better performance with chaining

We can improve the performance of hash tables that use chaining by using multiple hash functions, and inserting keys at the chain with the smallest length. In the case where $m = n$, as in the previous subsection, we find that with probability at least $1 - \frac{1}{n}$, the length of the longest list is $O(\log \log n)$.

4 Probing & Open Addressing

Chaining is actually not widely implemented due to its reliance on linked lists, which yield subpar performance at scale due to poor spatial locality and cache affinity.

Instead, we introduce the idea of **linear probing**. Suppose we have a hash function h. If the slot at $h(K)$ is filled for some key K, we will attempt to insert K at slot $h(K) + 1$. If that slot is filled, we will attempt insertion at slot $h(K) + 2$, and so on. Notice that insertion is guaranteed to succeed if there is at least one open spot in the hash table. More concisely, a general formula for linear probing for a hash table with m entries can be expressed as

$$h(k,i) = (h'(k) + i) \bmod m$$

where $h(k,i)$ is the new hash value for key k at the i-th probe, $i \in \{0,1,2,\ldots\}$, and h' is the hash function.

When we search for an element in such a hash table, we follow the same probing sequence that resulted from the element's insertion. The search stops when we find the element or reach an empty slot. The empty case implies that the desired element is not in the hash table.

However, we must carefully consider situations in which we delete elements from a hash table; if we were to naively delete an element and leave its slot empty, we would break the probe chain and any element that was inserted after the deleted element would be unreachable. We can solve this issue by introducing a marker. Instead of immediately deleting an item from the table, we would place a marker in its slot. During a search, we would treat this marker as an occupied slot and continue probing. During insertion, we would treat the marker as an empty slot.

4.1 Example

Suppose our key space is integers, and we have a hash table of size 10 indexed from 0 to 9 with a hash function $h(K) = K \bmod 10$. We attempt to insert keys in the following order: 38, 19, 8, 109, 10 to get the result shown in the image.

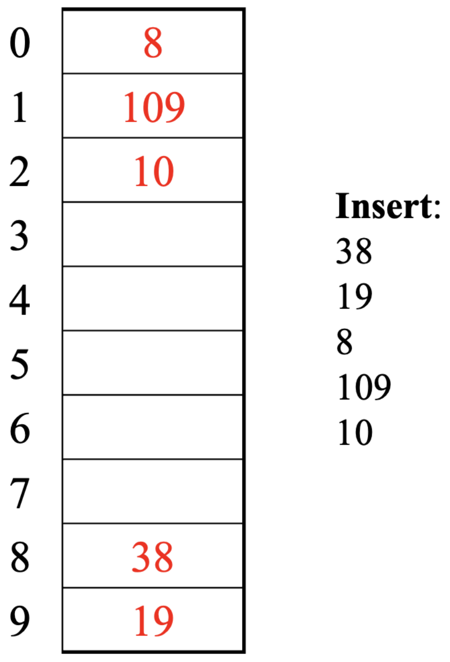

Figure 2: Linear Probing [2]

4.2 Linear Probing in Practice

Linear probing is one of the fastest general-purpose hashing strategies available in practice. The primary reasons for this performance are the following:

- Low memory overhead: Linear probing works with only an array of size n and a hash function.
- Excellent locality: When collisions occur, we only need to search within the contiguous array, as opposed to traversing a linked list in hashing with chaining.
- Great cache performance: The two above factors make caching for the hash table easy.

4.3 Analyzing the Expected Cost of Linear Probing

Suppose the load factor is $\alpha = \frac{m}{n} = \frac{1}{3}$. Since it is difficult to reason directly about the load of any particular slot in linear probing, we instead examine regions of the array.

A region of size m is defined as a consecutive set of m locations in the hash table. An element q is said to hash to a region R if $h(q) \in R$, even though q may not ultimately be placed inside R due to collisions. On expectation, a region of size 2^s should contain at most $\frac{1}{3} \cdot 2^s$ elements that hash to it.

References

[1] Jeff Erickson. CS 472 - Algorithms Lecture 7. Lecture Notes, Feb 2010.

[2] Anshumali Shrivastava. COMP 480/580 Probabilistic Algorithms and Data Structures Lecture 5. PowerPoint presentation, Sep 2024.