

CSE 203A: Randomized Algorithms

Spring 2026

Lecture 8: Introduction to Hashing

Date: 4/15/26

Instructor: Daniel M. Kane

Scribe: Kai Wang

1 Introduction

In this lecture, we introduced the dictionary data structure and hashing. We discussed different methods to reduce collisions and time complexity.

2 Prerequisite Knowledge

There isn't much prerequisite knowledge on previous topics aside from basic probability tools:

Linearity of Expectation

Markov Inequality

3 Main Lecture Content

3.1 Introduction to Dictionaries

Definition 3.1 (Static Dictionary). Let the universe of all words be U , with $|U| = N$ being inconceivably large. A static dictionary is a data structure on some subset $S \subset U$ of words, with $|S| = n < N$, that answers two types of queries:

- Determine if a word $x \in S$.
- Retrieve data associated with element $x \in S$.

Definition 3.2 (Dynamic Dictionary). A dynamic dictionary is a static dictionary that supports two additional queries:

- Insert $x \in U$ into S .
- Delete $x \in U$ from S .

The dictionary data structure is very useful for storing a set of elements that is much smaller than the set of all possible words. For example, in a Scrabble game, a static dictionary is useful to determine whether a word is valid, since the set of all possible strings of letters is extremely large. In this example, the extra data associated with each x could correspond to the point value of each valid word.

A dynamic dictionary is more suitable for tasks where insertion and deletion are common, such as database management.

Remark 3.3 (Sorting-Based Approaches). A dictionary can be implemented in $O(\log(n))$ time per query by sorting the elements in S . For example, a balanced binary tree supports lookup, insertion, and deletion in $O(\log(n))$ time. This requires elements in U to have a valid total ordering.

Hashing approaches aim to answer each query in $O(1)$ time complexity.

3.2 Introduction to Hashing

The central idea behind hashing is to map elements in U to elements in $\{1, 2, \dots, m\}$. Then, we can store elements in S in a size m array A . We can use a hash function $h(x) : U \rightarrow \{1, 2, \dots, m\} = [m]$ to store element x in $A[h(x)]$.

The problem with this approach are collisions, which happen when $h(x) = h(y)$ for $x \neq y$. Collisions are inevitable if $n > m$ due to the Pigeonhole Principle. Additionally, if $N > m$, then there are some $x, y \in U$ such that $h(x) = h(y)$. This makes deterministic hash functions hard to use when implementing dictionaries, as there will always be some sets S which have collisions.

Ideally, we want the hash function to be random, uniformly distributed from $U \rightarrow [m]$. However, storing and computing this random function is difficult.

Definition 3.4 (Hash Family). A hash family h is a set of efficiently computable hash functions $h_i(x)$.

By sampling from a hash family, we can simulate a truly uniform random function.

If we use a uniform random function, the expected number of collisions can be calculated as follows:

$$E\left[\sum_{x,y \in S, x \neq y} h(x) = h(y)\right] \tag{1}$$

$$= \sum_{x,y \in S, x \neq y} Pr(h(x) = h(y)) \tag{2}$$

Linearity of Expectation

$$= \sum_{x,y \in S, x \neq y} 1/m \tag{3}$$

Uniformly Random

$$= \binom{n}{2} / m \tag{4}$$

$$\approx n^2 / 2m \tag{5}$$

We want the hash family approach to be "random enough" for this analysis to still hold. Note that the only assumption made in the calculation is that $Pr(h(x) = h(y)) = 1/m$ when $x \neq y$.

Definition 3.5 (k -wise independence). A hash family H is k -wise independent if for any distinct $x_1, x_2, \dots, x_k \in U$ and any distinct $y_1, y_2, \dots, y_k \in [m]$, $Pr_h(h(x_1) = y_1, h(x_2) = y_2, \dots, h(x_k) = y_k) = 1/m^k$, where h is sampled from H .

Remark 3.6 (Random Hash Functions and Hash Families). We can think of a hash family as a probability distribution over different hash functions. Practically speaking, sampling from this distribution should be fast.

If a hash family is 2-wise independent, then the hash family behaves as well as a uniformly random function in terms of expected number of collisions.

Example 3.7 (Random Polynomial over a Finite Field). For some finite field \mathbb{F} , the hash family of random polynomials of degree less than k is k -wise independent. This comes from the fact that there is a unique degree k polynomial for a given set of $k + 1$ or more points. If we want to map to $[m]$, then we can use a random polynomial $\mathbb{F} \rightarrow \mathbb{F}$ and a deterministic function $\mathbb{F} \rightarrow [m]$.

Remark 3.8 (Birthday Paradox). It is difficult to do better than the previously established $n^2/2m$ collisions. This connects to the birthday paradox, where the probability of two people in a room sharing the same birthday increases quadratically as more people are added.

3.3 Dealing with Collisions

A basic approach to dealing with collisions is to allow each index to store multiple elements. For example, $A[k]$ could store a linked list containing all x such that $h(x) = k$. This approach works, but has implications on the time complexity. To look up x , we need $O(1)$ to compute $h(x)$ and $O(\# \text{ of elements in } S \text{ that collide with } x)$ to locate x . Deletion and insertion queries are the same, as we need to search the linked list to remove x or to ensure that no duplicates are added.

By linearity of expectation, using a 2-wise independent hash family means that the number of elements that collide with x is $n * 1/m = n/m$. If $m = O(n)$, then this gives an average case complexity of $O(1)$ per query. However, the worst case complexity could be very bad. This is especially relevant in adversarial attacks, where an attacker may run a single slow query multiple times to cause massive slowdowns.

If we want perfect hashing, the worst case time complexity must also be $O(1)$. Additionally, the space complexity of the dictionary should be $O(n)$. This is much easier to accomplish for a static dictionary.

3.4 Secondary Hashing

The idea behind secondary hashing is to eliminate collisions by using two hash functions. In each entry $A[k]$, store a pointer to another hash table A_k of size m_k using a separate hash function $h_k(x)$. Each A_k should have zero collisions. Using this approach, element x is located at $A_{h(x)}[h_{h(x)}(x)]$. We must now determine the values of m_k .

Let n_k be the expected number of elements x with $h(x) = k$. Then, the expected number of collisions in A_k is $n_k^2/2m_k^2$ from section 3.2. If $m_k = n_k^2$, then by the Markov inequality, the probability that a random hash function has no collisions is greater than 50%. Therefore, if we set $m_k = n_k^2$ and try a few different h_k , we will eventually find one that results in no collisions.

The space complexity of this approach can be split into two parts. The original hash table takes $O(m)$ space, and each secondary hash table takes $O(n_k^2)$ space. Calculating the expectation of the second term gives:

$$E\left[\sum_{k=1}^m n_k^2\right] \quad (6)$$

$$= \sum_{k=1}^m E[n_k^2] \quad \text{Linearity of Expectation} \quad (7)$$

$$= \sum_{k=1}^m E\left[\sum_{x,y \in S} h(x) = h(y) = k\right] \quad n_k^2 = \# \text{ of pairs} \quad (8)$$

$$= \sum_{k=1}^m (E\left[\sum_{x \in S} h(x) = k\right] + E\left[\sum_{x,y \in S, x \neq y} h(x) = h(y) = k\right]) \quad \text{Linearity of Expectation} \quad (9)$$

$$\approx m * (1/n * n + n^2/m^2) \quad \text{2-wise independence} \quad (10)$$

$$= m + n^2/m \quad (11)$$

If $m = O(n)$, then this expected value is $O(n)$. Using the Markov inequality, we are able to try some different choices of h and select one that works with very high probability. Once we have the hash function choices, we can easily build the entire data structure in linear time.

This approach does not work for dynamic hashing, as we are not able to "try" different hash functions, and the collision analysis breaks down. Next lecture will be on Cuckoo Hashing, which aims to solve this problem.

4 Summary

- Static and dynamic dictionaries
- Hash functions
- Randomized hash functions, hash families
- k -wise independent hash families
- Hash collisions
- Secondary hashing

References

- [1] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.