

# CSE 203A: Randomized Algorithms

Spring 2026

**Lecture 8:** Introduction to Hashing

**Date:** 15 Apr 2026

**Instructor:** Daniel M. Kane

**Scribe:** Liam Hardy

## Lecture Overview

### Goal

How can we store a 'dictionary' data structure, i.e. a way to store some subset of elements  $S$  in a universe  $U$ . The properties of such data structure being:

- Lookup of an element and data associated with it
- Add and remove any  $x \in S$

### Results

Good solutions use *hashing*, and we can achieve constant time operations.

### Key terms that appeared

*hashing, pigeonhole principle, hash family,  $k$ -wise independence, collisions, perfect hashing, secondary hashing*

## 1 Introduction

In this lecture, we introduced hashing as a solution to the dictionary data structure problem. This marks the beginning of a new problem in this class.

## 2 Brief Reminders about Previous Topics

We don't really need any content from previous lectures here fortunately!

## 3 Main Lecture Content

### 3.1 Sorting Based Approaches

Solving this dictionary problem can be done by maintaining some sort of sorted data structure, however this only gives us as good as  $\log n$  per operation in terms of input size  $n$ . One benefit here is that **all** we need to be able to do is compare two elements to use this strategy.

### 3.2 Hashing Approaches

#### Idea:

Maintain an array  $A$  of size  $N$ , function  $h : U \rightarrow [m]$  (integers from 1 to  $m$  inclusive).

We store each  $x$  at  $A[h(x)]$ .

**Definition: Hash Function**

A function  $h : U \rightarrow [m]$  that maps keys to bucket indices.

**Problem:**

*Collisions:* if  $x, y \in S$  where  $h(x) = h(y)$ , then what should  $A[h(x)]$  store?

**Definition: Collision**

When two separate inputs  $x \neq y$  result in the same hash value, i.e.  $h(x) = h(y)$

**Definition: Pigeonhole Principle**

Let  $m =$  the size of your array,  $n =$  the number of elements you are storing,  $N =$  the size of your universe.

- If  $m < n$ , you will have collisions
- If  $m < N$ , you will have  $x \neq y$  where  $h(x) = h(y)$

The  $m < N$  aspect of the pigeonhole principle implies that we **can not use a deterministic h** if we want optimal performance in worst case scenarios.

Ideally,  $h$  is uniformly random function  $U \rightarrow [m]$ , but we can't store a truly random function.

**Definition: Hash Family**

A small set of **hash functions** that behave as uniformly random functions

**Calculating  $\mathbb{E}[\# \text{ of collisions}]$** 

Say we have a random  $h : U \rightarrow [m]$  and  $S \subset U$  then

$$\mathbb{E}[\# \text{ of collisions}] = \sum_{x \neq y \in S} \mathbf{1}[h(x) = h(y)]$$

From linearity of expectation, we have

$$\sum_{x \neq y \in S} Pr(h(x) = h(y)) = \frac{\binom{n}{2}}{m} = \boxed{\frac{n^2}{2m} = \mathbb{E}[\# \text{ of collisions}]}$$

Since for each  $x \neq y$ ,  $Pr(h(x) = h(y)) = \frac{1}{m}$

**Definition: k-wise independent family**

A family of hash functions is k-wise independent if for any distinct  $X_1 \dots X_k \in U$ ,  $y_1, \dots, y_k \in [m]$ ,  $Pr_h(h(x_1) = y_1, h(x_2) = y_2, \dots, h(x_k) = y_k) = \frac{1}{m^k}$

i.e. the probability that any  $k$  set of inputs each map to their own specific set of outputs is exactly  $\frac{1}{m^k}$

### 3.3 Dealing with collisions

#### 3.3.1 Basic strategies:

Each element of  $A$  stores not just one entry, but instead a linked list of all entries that map to that value. The runtime of this basic solution however: lookup / insert / delete is  $O(1) + O(\# \text{ of elements of } S \text{ that collide with } x)$ .

#### 3.3.2 Trying to eliminate collisions altogether

Definition: perfect hashing

When the worst case lookup time is  $O(1)$  and total memory is  $O(n)$ . This is easy-ish for static dictionaries, with the easiest way being secondary hashing.

Definition: secondary hashing

For your normal hashmap array, instead of storing the value, store a pointer to a new array  $m_k$  and a hash function  $h_k$ , then  $x \in S$  is actually stored in  $A_k[h_k(x)]$  where  $k = h(x)$ . Key:  $A_k$  should have no collisions.

**Goal:** Use secondary hashing to set up our hash functions in such a way that for static dictionary data structures, we can have guaranteed true  $O(1)$  lookup for elements  $x \in S$ .

**Problem:** How do we actually get  $A_k$  to have no collisions?

Suppose that  $n_k = \#\{x \in S : h(x) = k\}$ .

$\mathbb{E}[\# \text{ of collisions in } A_k] < \frac{n_k^2}{2m_k}$ .

If  $m_k = n_k^2$ , then there is a 50% probability of no collisions. If we try a few different  $h_k$ , we'll eventually get one that will have no collisions. The total space complexity is  $O(n)$  for the top array  $A + \sum_{k=1}^m O(n_k^2)$

*Proof.* WTS: There is some  $h$  s.t. the space complexity is not too large.

Using the identity

$$n_k^2 = \sum_{x \in S} \mathbf{1}[h(x) = k] + \sum_{x \neq y \in S} \mathbf{1}[h(x) = h(y) = k]$$

we have:

$$\begin{aligned} \mathbb{E} \left[ \sum_{k=1}^m n_k^2 \right] &= \sum_{k=1}^m \left( \sum_{x \in S} \Pr(h(x) = k) + \sum_{x \neq y \in S} \Pr(h(x) = h(y) = k) \right) \\ &= \sum_{k=1}^m \frac{n}{m} + \sum_{k=1}^m \frac{n(n-1)}{m^2} \\ &= n + \frac{n(n-1)}{m} = O \left( n + \frac{n^2}{m} \right) \end{aligned}$$

Setting  $m = \Theta(n)$ , total expected space is  $O(n)$ . Since  $\mathbb{E}[\text{space}] = O(n)$ , there must exist some  $h$  achieving  $O(n)$  space.  $\square$

Since constructing this structure and finding such an  $h$  obviously won't be done in constant time, and inserting and deleting may ruin some of the subtle details in a specific secondary hashing structure that works for constant time, this approach falls apart for dynamic dictionary data structures.

## 4 Further Remarks

In the next lecture, we will see strategies for constant time operations in dynamic data structures, not just static ones.

## 5 Summary

In this lecture we proposed the challenge of construction a dictionary like data structure for quick lookup of elements in a subset of a universe. We found that hashing provides an effective means of doing so, and proved the ability to have a static such dictionary with constant time lookup, with dynamic versions coming soon in the next lecture.