# Algorithm
# Design and Analysis

**Hashing: Universal and Perfect Hashing**

# Roadmap for today

- Review the ***dictionary problem*** and motivate hashing
- See ***universal hashing*** and how to prove that a family is universal
- See an algorithm for ***static perfect hashing***

# Formal model of computation

- ***Model (word-RAM)***:
  - We have unlimited constant-time addressable memory ("registers")
  - Each register can store a $w$-bit integer (a "word")
  - Reading/writing, arithmetic, logic, bitwise operations on a constant number of words takes constant time
  - With input size $n$, we need $w \geq \log n$.

# Dictionaries & Hashing

# The dictionary problem

The dictionary data type stores ***items*** that have associated unique ***keys***

**unique key** →

| ***STUDENT*** | |
|---|---|
| id: | *integer* |
| name: | *string* |
| grade: | *character* |

***Dictionary Interface***

***insert***(*item*): Insert the given item (associated with its key)

***lookup***(*key*): Return the item with the given key if it exists

***delete***(*key*): Delete the item with the given key if it exists

***Python equivalent***

*d[key] = item*

*item = d[key] (throws **KeyError** if not present)*

*d.pop(key) (throws **KeyError** if not present)*

# Formal setup for hashing/hash tables

- The keys come from $U = [0 \ldots u-1]$ (the ***universe*** of keys)
- We want to store items in a table $A$ of size $m$. Assume $u \gg m$, so we can not just store key $x$ at $A[x]$
- ***Key idea (Hash function):*** Define a function

$$h : U \to \{0,1,\ldots,m-1\}$$

- Try to store item with key $x$ at $A[h(x)]$

# Handling collisions

***Approach #1 (Open addressing)***: When a collision occurs, cleverly find a **different location in the table** for the new item

- Very hard to analyze, bad performance if not implemented well
- Amazing performance if done well! **All state-of-the-art hashtables do this**

***Approach #2 (Chaining)***: Instead of storing a single item in each slot, store a **list of items**. Add all items that hash to that slot to the list

- Simple to analyze and implement
- Decent performance in practice, used by the C++ standard library
- Much easier to parallelize

# Prehashing non-integer keys

*Idea (prehashing):* For non-integer keys, we want to convert them into some representative integer.

*Example (strings):* Strings can be interpreted as integers by interpreting each character as a digit, in base alphabet size

**B A C Z**

**1 0 2 25**

$$= \mathbf{1 \cdot 26^3 + 0 \cdot 26^2 + 2 \cdot 26 + 25}$$

$$= \mathbf{17653}$$

# Choosing a hash function $h$

- **Main goal:** We want it to be unlikely that $h(x) = h(y)$ for $x \neq y$
- We want $m = O(n)$, where $n$ is the number of keys in the table
  - We could just pick $m = u$ then there are no collisions!!
  - But this is an unacceptable amount of memory if $u \gg n$
- We also want $h(x)$ to be fast to compute. Ideally $O(1)$ time
- How long does a hashtable operation take using chaining?

# Can we just pick… the best hash function?

- For any hash function you choose, I can find a set of $n$ items that hash to the same location…
- There's no such thing as a hash function that works for every input.
- ***Big idea (randomization):*** We need to employ randomization to build a hash function that doesn't have a horrible worst-case behaviour
- **Specifically, we want to choose a random hash function from some big set of possible hash functions**

# Random hash families

- ***Definition (totally random hash)***: A set $\mathcal{H}$ of hash functions is ***totally random*** if for all $x \in U$, $t \in \{0, \dots m-1\}$, independent of all $y \in U$

$$\Pr_{h \in \mathcal{H}}[h(x) = t] = \frac{1}{m}$$

- Essentially equivalent to "*Simple uniform hashing*" (if you know it)

- Totally random hashing has all nice properties, but its not possible to do practically…

# Less random, but still random

- **Goal**: We need a hash function that is still "pretty random", but not totally random, since that's too expensive

***Definition (Universal Hashing)***: A set $\mathcal{H}$ of hash functions $h : U \to \{0, \dots, m-1\}$ is called ***universal*** if for all $x \neq y$

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$$

Can compute probability by counting:

$$\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{|h(x) = h(y)|_{h \in \mathcal{H}}}{|\mathcal{H}|}$$

# Examples: Universal or not?

$$|\boldsymbol{U}| = \mathbf{2}, \qquad \boldsymbol{M} = \mathbf{2}$$

| | $a$ | $b$ |
|---|---|---|
| $h_1$ | 0 | 0 |
| $h_2$ | 0 | 1 |

| | $a$ | $b$ |
|---|---|---|
| $h_1$ | 0 | 0 |
| $h_2$ | 1 | 1 |

| | $a$ | $b$ |
|---|---|---|
| $h_1$ | 0 | 1 |
| $h_2$ | 1 | 0 |

| | $a$ | $b$ |
|---|---|---|
| $h_1$ | 0 | 1 |
| $h_2$ | 1 | 0 |
| $h_3$ | 0 | 1 |

# More examples

$$|U| = 3, \qquad M = 2$$

| | $a$ | $b$ | c |
|---|---|---|---|
| $h_1$ | 0 | 0 | 1 |
| $h_2$ | 1 | 1 | 0 |
| $h_3$ | 1 | 0 | 1 |

$$|U| = 3, \qquad M = 3$$

| | $a$ | $b$ | c |
|---|---|---|---|
| $h_1$ | 0 | 0 | 0 |
| $h_2$ | 0 | 1 | 2 |
| $h_3$ | 1 | 2 | 0 |
| $h_4$ | 2 | 0 | 1 |

# Analysis of Universal Hashing

***Theorem***: If $\mathcal{H}$ is a universal family, then for any set $S \subseteq U$ with $|S| = n$, for any $x \in S$, if $h$ is chosen at random from $\mathcal{H}$, then the **expected** number of collisions between $x$ and other elements is at most $n/m$.

# Corollary

***Definition (Load Factor):*** The quantity $n/m$ is called the ***load factor***

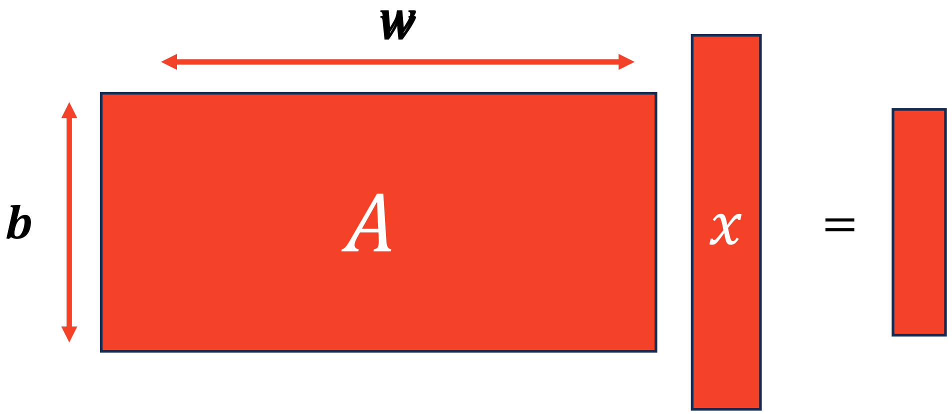
***Corollary:*** Using separate chaining, given a universal family $\mathcal{H}$, the expected cost of each operation is $O(1 + n/m)$

***Corollary***: Using separate chaining, given a universal family $\mathcal{H}$, if the load factor is always at most $1$, for any sequence of $L$ insert, lookup, delete operations, the expected cost of the $L$ operations over a random $h \in \mathcal{H}$ is $O(L)$.

*Assumes $h$ can be computed in $O(1)$ time*

# Okay… how do we construct one?

- Universal families sound great. How do we make one?
- ***Construction (Random binary matrix)***: Assume $|U| = 2^w, m = 2^b$
  - Let $A$ be a random $w \times b$ matrix of zeros and ones
  - Interpret $x \in U$ as a $w$ length vector of its bits
  - Let $h(x) = Ax \bmod 2$, again interpreting $h(x)$ as a $b$ length vector of bits



# Analysis of random binary matrix

***Theorem***: Its universal, i.e., for $x \neq y$, $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] = \frac{1}{m}$

# Wait, that's not constant time!

- How efficient is computing $h(x)$?

- Thankfully, there exists universal families whose hash functions can be computed in constant time (but they are harder to analyze).

***Example (The multiplication method):*** Suppose $|U| = 2^w$ and choose a power of two table size $m = 2^r$ and a **random odd integer $\boldsymbol{a}$**

$$h(x) = [(ax) \bmod 2^w] \gg (w - r)$$

# Even more randomness!

- Can we make a hash family that is "more random" than universal, but still less than totally random?  Yes!

- ***Definition (pairwise independent)*****:** A hash family $\mathcal{H}$ is called ***pairwise independent*** if for every pair $x_1 \neq x_2$ of distinct keys and every pair of values $v_1, v_2 \in \{0, \dots, m-1\}$ (not necessarily distinct),

$$\Pr_{h \in \mathcal{H}}[h(x_1) = v_1 \text{ and } h(x_2) = v_2] = \frac{1}{m^2}$$

*Intuitively, for every pair of distinct keys $(x_1, x_2)$, all pairs of values $(v_1, v_2)$ are equally likely to occur (there are $m^2$ possible pairs of values).*

# Example

- Is this hash function pairwise independent? Is it totally random?

| | $a$ | $b$ | c |
|---|---|---|---|
| $h_1$ | 0 | 0 | 0 |
| $h_2$ | 0 | 1 | 1 |
| $h_3$ | 1 | 0 | 1 |
| $h_4$ | 1 | 1 | 0 |

| | $a$ | $b$ |
|---|---|---|
| $h_1$ | 0 | 0 |
| $h_2$ | 0 | 1 |
| $h_3$ | 1 | 0 |
| $h_4$ | 1 | 1 |

| c |
|---|
| 0 |
| 1 |
| 1 |
| 0 |

$= h(a) \oplus h(b)$

| | $a$ | c |
|---|---|---|
| $h_1$ | 0 | 0 |
| $h_2$ | 0 | 1 |
| $h_3$ | 1 | 1 |
| $h_4$ | 1 | 0 |

| $b$ |
|---|
| 0 |
| 1 |
| 0 |
| 1 |

$= h(a) \oplus h(c)$

| | $b$ | c |
|---|---|---|
| $h_1$ | 0 | 0 |
| $h_2$ | 1 | 1 |
| $h_3$ | 0 | 1 |
| $h_4$ | 1 | 0 |

| $a$ |
|---|
| 0 |
| 0 |
| 1 |
| 1 |

$= h(b) \oplus h(c)$

# Even more randomness!

- ***Definition (k-wise independent)***: A hash family $\mathcal{H}$ is called $k$-wise independent if for every set of $k$ distinct keys $x_1, \dots, x_k$ and $k$ values $v_1, \dots, v_k$ (not necessarily distinct) we have

$$\Pr_{h \in \mathcal{H}}[h(x_1) = v_1 \text{ and } \dots \text{ and } h(x_k) = v_k] = \frac{1}{m^k}$$

- The $k = 1$ case is usually called *uniform* (since "1-wise independent" sounds funny)

- The $k = 2$ case is pairwise independence from the previous slides

# Static perfect hashing (not required)

# Static perfect hashing

***Problem***: Suppose we ***know the $n$ keys in advance*** want deterministic constant query time in the worst case? Is this possible?

***Idea***: Reduce collision probability by making the table really really big!

***Theorem***: Given a universal family $\mathcal{H}$, taking $m = n^2$ gives us

$$\Pr_{h \in \mathcal{H}}[\text{no collisions}] \geq \frac{1}{2}$$

# Some analysis

***Theorem***: Given a universal family $\mathcal{H}$, taking $m = n^2$ gives us

$$\Pr_{h \in \mathcal{H}}[\text{no collisions}] \geq \frac{1}{2}$$

# That's a bit too much

- Okay, no collisions is nice, but $n^2$ space is way too much.
- Can we achieve the same with only $O(n)$ space?
- ***Idea***: The number of collisions per element is usually small anyway. Squaring those numbers might not be too big

# FKS Hashing

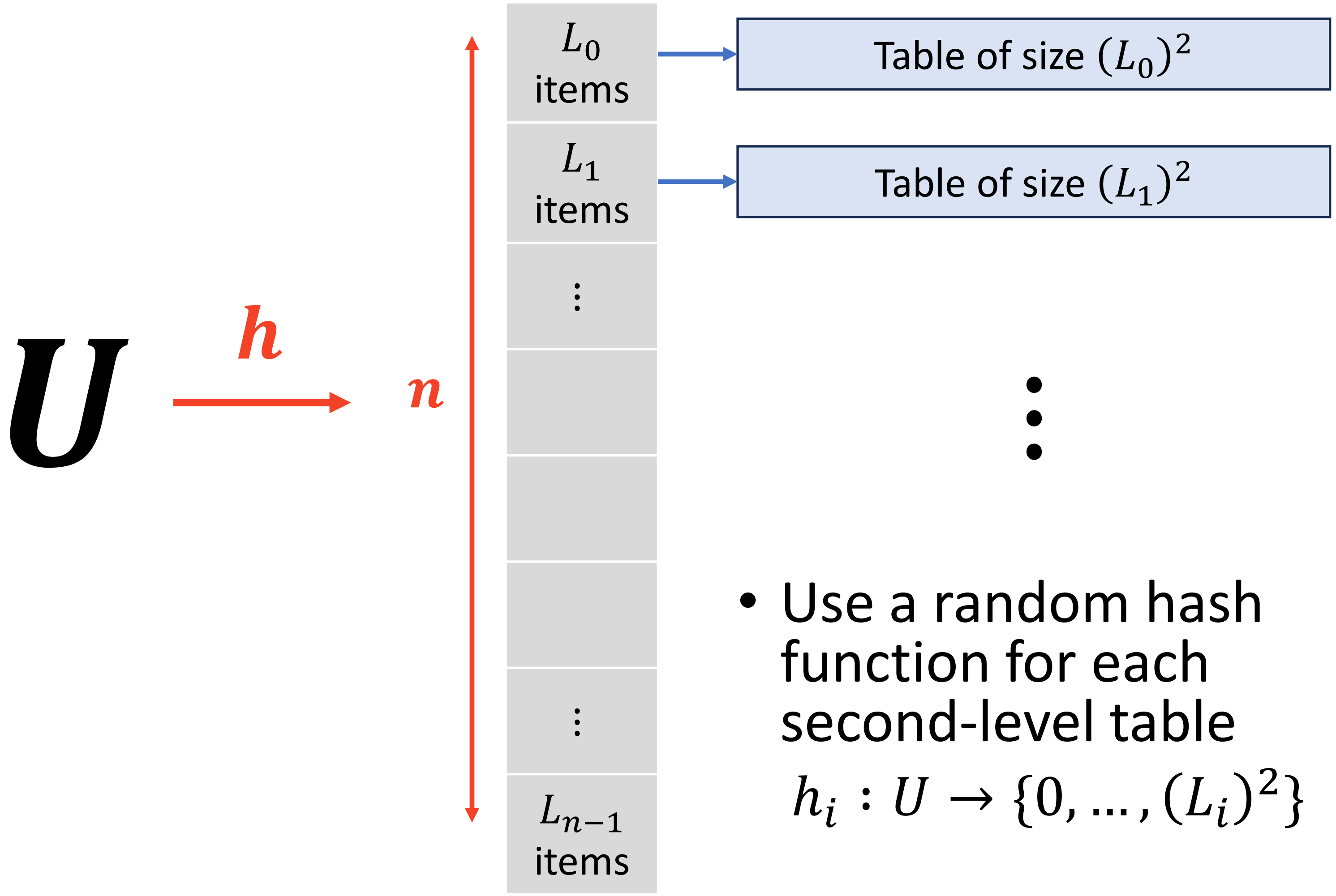
- Choose a hash function $h \in \mathcal{H}$ (universal)

  $h: U \to \{0, \ldots, n-1\}$

- Let $L_i$ be the number of keys $x$ such that

  $$h(x) = i$$

- Store the $L_i$ items at position $i$ in a second-level table of size $(L_i)^2$




- Use a random hash function for each second-level table

  $h_i : U \to \{0, \ldots, (L_i)^2\}$

# Analysis of second-level tables

- We know that for each second-level table, we have a $\geq 1/2$ probability that there are no collisions
- There are $n$ such tables, so there are bound to be ***some*** with collisions
- ***Solution***: If there are collisions in a second-level table, just pick another random hash from the family until there isn't.

# Analysis of top level

***Theorem***: If $h$ is chosen from a universal family $\mathcal{H}$, then

$$\Pr_{h \in \mathcal{H}}\left[\sum L_i^2 > 4n\right] \leq \frac{1}{2}$$

# Analysis continued…

***Lemma***: Define $C_{xy} = 1$ if $h(x) = h(y)$, else $C_{xy} = 0$

$$\sum (L_i)^2 = \sum \sum C_{xy}$$

# Analysis continued continued…

***Lemma***: If $h$ is chosen from a universal family $\mathcal{H}$, then

$$\mathbb{E}\left[\sum (L_i)^2\right] < 2N$$

# Completing the analysis

***Theorem***: If $h$ is chosen from a universal family $\mathcal{H}$, then

$$\Pr_{h \in \mathcal{H}}\left[\sum L_i^2 > 4n\right] \leq \frac{1}{2}$$

# Summary of today

- ***Universal hashing*** gives us "enough" randomness to get nice results
  - Operations on a hash table with separate chaining run in $O(1 + n/m)$ time.
  - Static FKS hashing gives deterministic lookup in constant worst-case time.
- ***Proving that a hash family is universal / $k$-wise independent*** can be quite tricky, but is very important
- For "more randomness", we can employ pairwise independent, or $k$-wise independent hashing.