Hashing

Data Structures and Algorithms

Nanjing University, Fall 2021
郑朝栋

Efficient implementation of **OSet**

	`Search(S,k)`	`Insert(S,x)`	`Remove(S,x)`
`BinarySearchTree`	$O(h)$ worst-case	$O(h)$ worst-case	$O(h)$ worst-case
`Treap`	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation
`RB-Tree`	$O(\log n)$ worst-case	$O(\log n)$ worst-case	$O(\log n)$ worst-case
`SkipList`	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation

Can we be faster?

(if we only care about **Search**/**Insert**/**Remove**)

Search/Insert/Remove in $O(1)$ time

- Assume keys are *distinct* integers from universe $U = \{0, 1, \cdots, m-1\}$
- Easy, just allocate an array of size $m = |U|$.
- **Search/Insert/Remove** can be done in $O(1)$ time.

Direct-address Tables

Any potential issue?

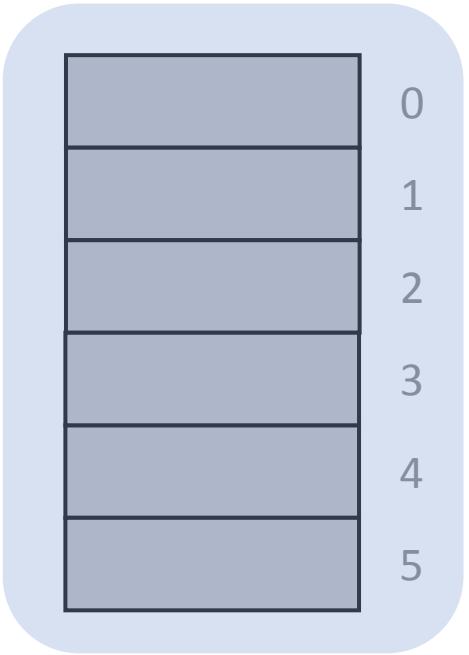
Insert(2)

Insert(3)

Search(2)

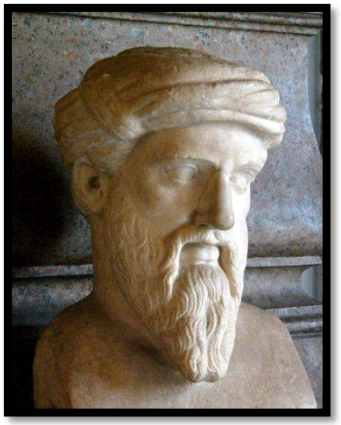
Remove(3)

Search(3)



Direct-address Tables

- Assume keys are *distinct* integers from universe $U = \{0, 1, \cdots, m-1\}$
- **Direct-address table:** allocate an array of size $m = |U|$.
- **Search/Insert/Remove** can be done in $O(1)$ time.
- Potential issues:
- **Q:** What if keys are distinct, but not integers (e.g., strings).
- **A:** "Everything is a number." Said Πυθαγόρας. That is, Pythagoras. This is especially true for modern computers…

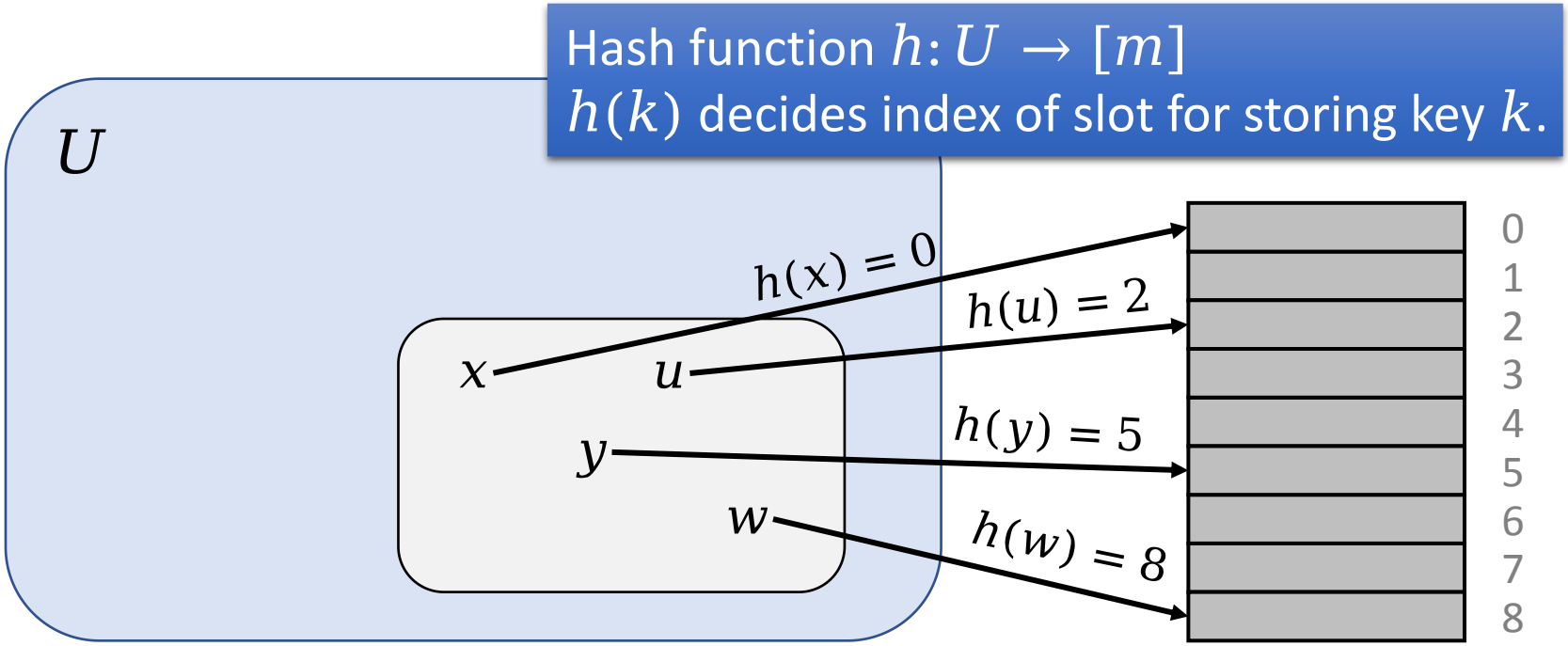


Direct-address Tables

- Assume keys are *distinct* integers from universe $U = \{0, 1, \cdots, m-1\}$
- **Direct-address table:** allocate an array of size $m = |U|$.
- **Search/Insert/Remove** can be done in $O(1)$ time.
- The real problem: the universe can be large, very large!
  (E.g., U is the set of 64-bit integers.)
- The space complexity is unacceptable!

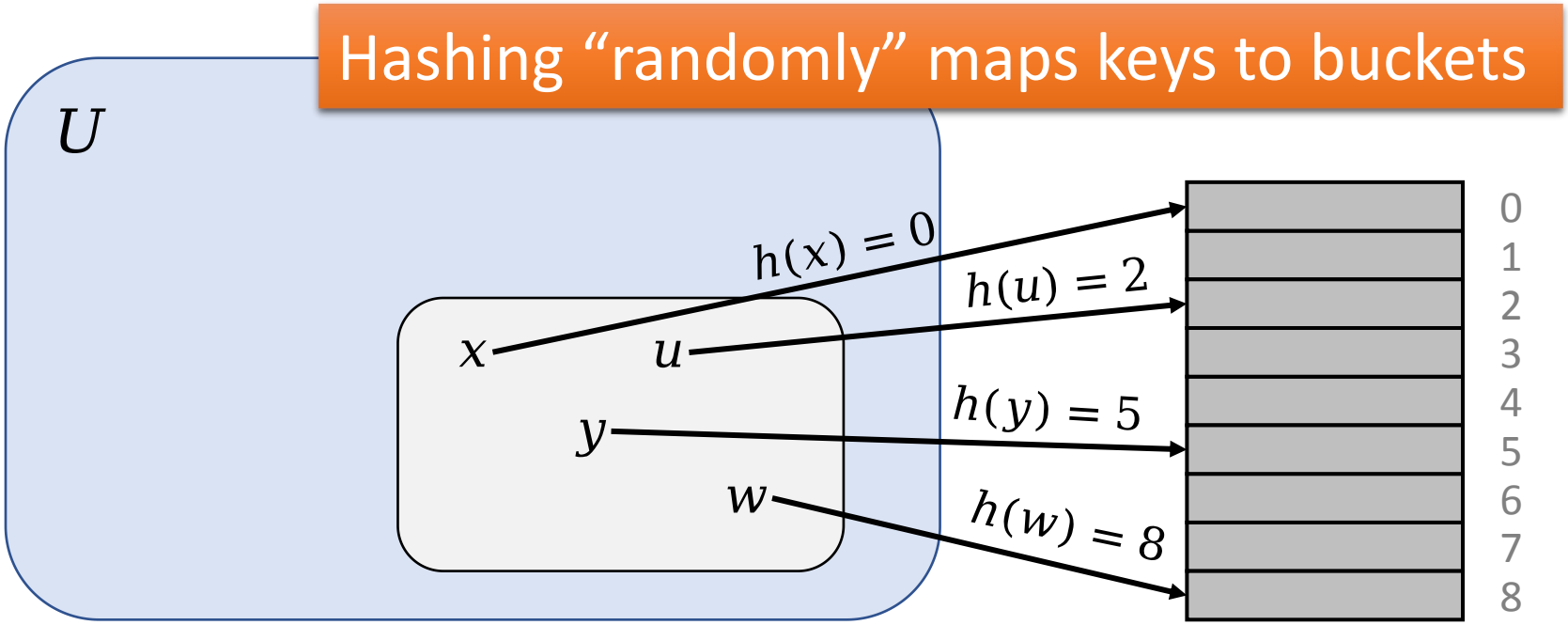
Hashing

- Huge universe U of possible keys.
- Much smaller number n of actual keys.
- Only want to spend $m \approx n$ (i.e., $m \ll |U|$) space.
  (Meanwhile support very fast **Search**/**Insert**/**Remove**.)



Hashing

- Keys from huge universe U, but much smaller number n of actual keys.
- Spend $m \approx n$ (i.e., $m \ll |U|$) space for very fast **Search/Insert/Remove**.
- Design **hash function** $h: U \rightarrow [m]$
- Use $h(k)$ as the index of slot for storing element with key k.

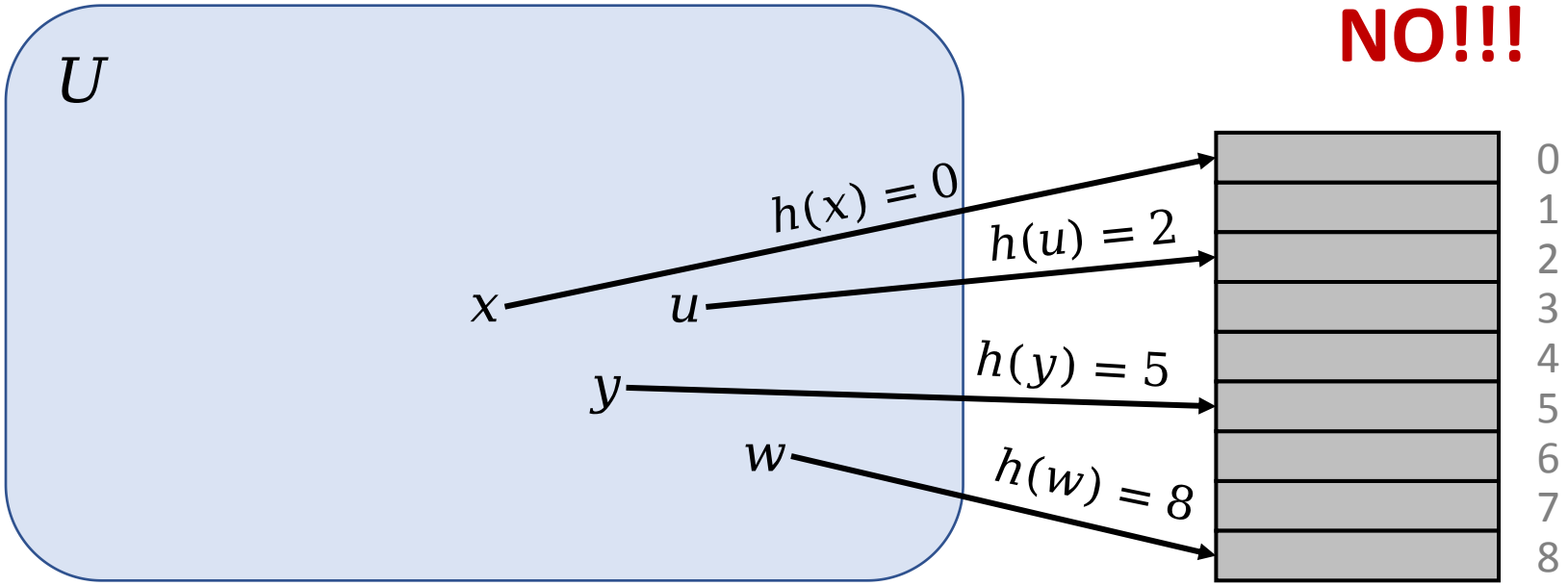


Hashing

- Design **hash function** $h: U \to [m]$ $m \ll |U|$
- Use $h(k)$ as the index of slot for storing element with key k.
- Assume computing h is always fast. (E.g., in $O(1)$ time.)
- ~~Assume h maps distinct keys to distinct indices.~~
- **Search/Insert/Remove** can be done in $O(1)$ time!

But is this possible?

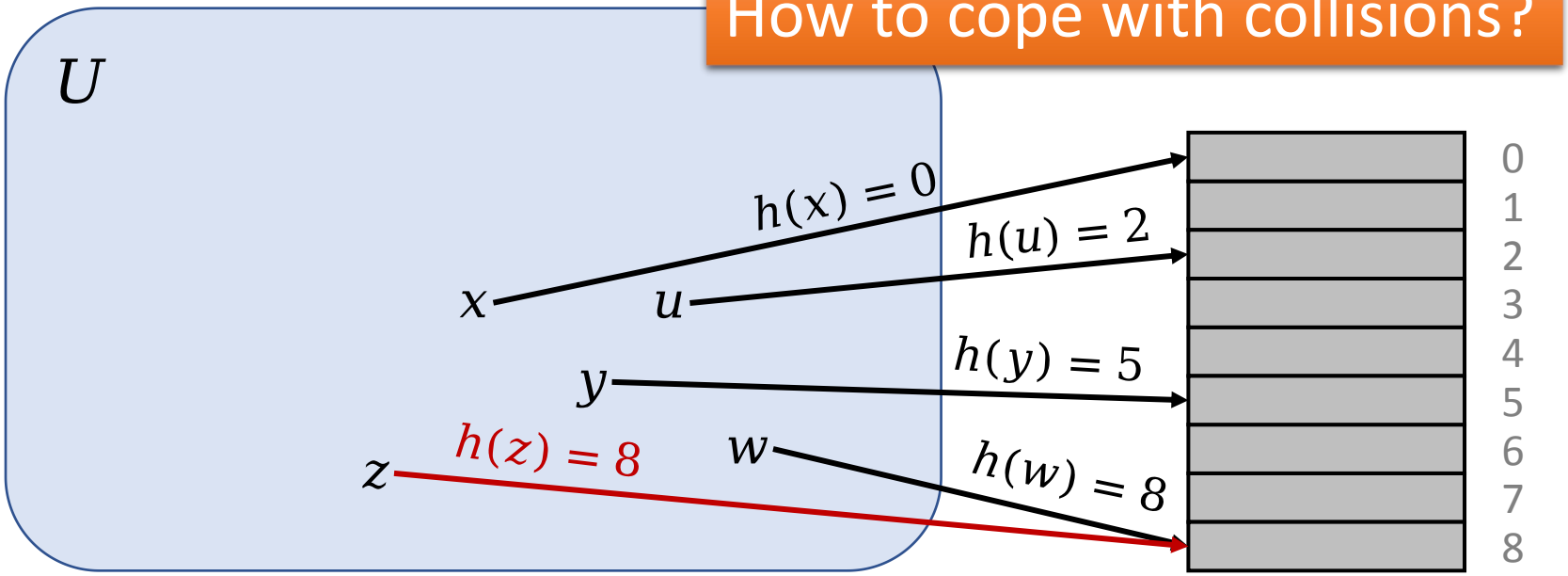
NO!!!



Collisions in hashing

- Hash function $h: U \rightarrow [m]$
- Two *distinct* keys k_1 and k_2 **collide** if: $h(k_1) = h(k_2)$
- Collisions are unavoidable!
  - Proof: $m \ll |U|$ and pigeonhole principle.

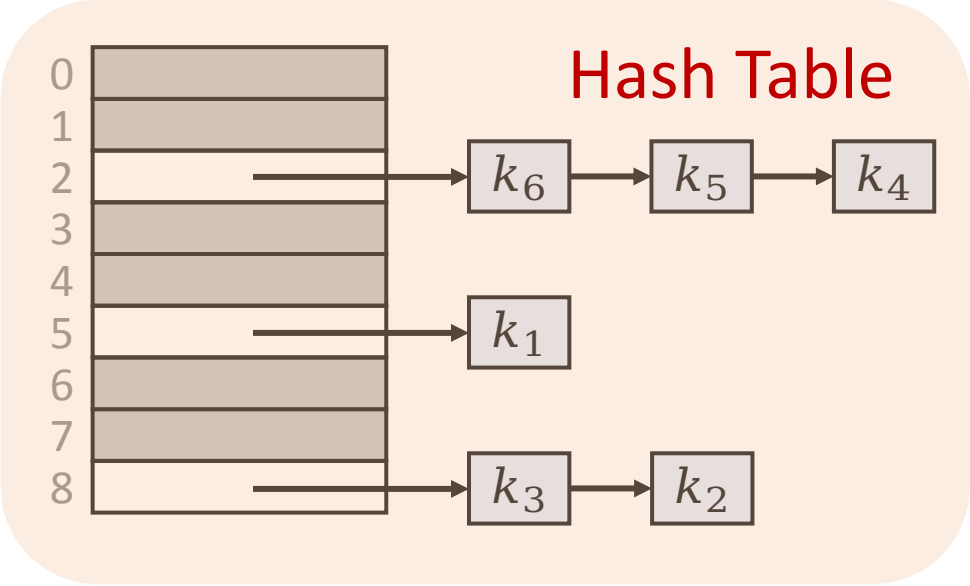
How to cope with collisions?



Coping with collisions in hashing

Chaining

- Each bucket i stores a pointer to a linked list L_i.
- All keys that are hashed to index i go to L_i.



$h(k_4) = h(k_5) = h(k_6) = 2$

Space cost:

$\Theta(m)$ for pointers;

$\Theta(n)$ for actual elements.

Hashing with Chaining

- **Search**(k) where k is a key. Time depends on length of the linked list!
  - Compute $h(k)$
  - Go through the corresponding list to search item with key k.
- **Insert**(x) where x is a pointer to an item. $O(1)$
  - Compute $h(x.key)$
  - Insert x to the *head* of the corresponding list.
- **Remove**(x) where x is a pointer to an item. $O(1)$
  - Simply remove x from the linked list.

- Time complexities? (Assume computing h takes $O(1)$ time.)
- **Search** can cost $\Theta(n)$ in worst-case. (All keys hash to same value.)

Let's be optimistic (for now):

The "Simple Uniform Hashing" Assumption

- Every key is *equally likely* to map to every bucket.
- Keys are mapped *independently*.

- Recall hash function h is fixed and deterministic:
  Making assumptions regarding input keys' distribution!

- Why this helps?
- Each key goes to a randomly chosen bucket, if there are enough number of buckets (w.r.t. *actual* number of keys to be stored), each bucket will not have too many keys.

Assuming “Simple Uniform Hashing”

Performance of hashing with chaining

- Consider a hash table containing m buckets, storing n keys.
- Define **load factor** $\alpha = n/m$.
  (This is the expected number of keys in each bucket.)
- Intuitively, **Search** will on average cost $O(1 + \alpha)$:
  - $O(1)$ for computing hash value;
  - $O(\alpha)$ for traversing linked list.

- Expected cost is $\Theta(1 + \alpha)$,
  for both successful and unsuccessful search!

Assuming "Simple Uniform Hashing"

m buckets, n keys;
load factor $\alpha = n/m$.

Performance of hashing with chaining

- Expected cost of unsuccessful search is $\Theta(1+\alpha)$.
  - Cost: compute hash value + traverse entire linked list in a bucket.
  - The key being searched is equally likely to map to every bucket.
  - $\Theta(1) + \Theta(\alpha) = \Theta(1+\alpha)$
- Expected cost of successful search is $\Theta(1+\alpha)$, too!
  - Cost: compute hash value + traverse linked list in a bucket till key found.
  - Let C_i be the cost for finding the i^{th} inserted element x_i.
    We want to compute $(1/n) \cdot \sum_{i=1}^{n} \mathbb{E}[C_i]$
  - Let X_{ij} be i.r.v. taking value 1 iff $h(x_i.key) = h(x_j.key)$
  - $\frac{1}{n} \cdot \sum_{i=1}^{n} \mathbb{E}[C_i] = \frac{1}{n} \cdot \sum_{i=1}^{n} \mathbb{E}\left[\left(1 + \sum_{j=i+1}^{n} X_{ij}\right)\right]$
    $= \frac{1}{n} \cdot \sum_{i=1}^{n} \left(1 + \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]\right) = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(1 + \sum_{j=i+1}^{n} \frac{1}{m}\right)$
    $= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1+\alpha)$

Assuming "Simple Uniform Hashing"

Performance of hashing with chaining

- Consider a hash table containing m buckets, storing n keys.
- Define **load factor** $\alpha = n/m$.
- Expected cost is $\Theta(1 + \alpha)$ for the **Search** operation.
- If $m = \Theta(n)$, hash table costs $\Theta(n)$ space, but **Search/Insert/Remove** all take $O(1)$ time, on average.
- But what is the expected *maximum* cost for **Search**? (Search for a key that maps to the heaviest bucket.)
- That is: expected length of the longest linked list?
- Alternatively: throw n balls into m bins uniformly at random, max number of balls in a bin, in expectation?
- If $m = \Theta(n)$, the answer is $\Theta\left(\frac{\lg n}{\lg \lg n}\right)$.

Reality fights back

- "Simple Uniform Hashing" does not hold!
  - Keys are not that random (they usually have patterns).
  - Patterns in keys can induce patterns in hash functions, unless you are very, very careful.
  - Bottom line: once h is fixed and known, you can find a set of "bad" keys that hash to same value.

How to design hash functions?

Stop! Please don’t try!

Unless you really need to…

We will now show you this is really not easy...

Designing hash functions

Some bad hash functions

- Assume keys are English words.
- One bucket for each letter (i.e., 26 buckets).
- Hash function: $h(w)$ = first letter in word w.
  - E.g., $h(\text{"}test\text{"}) = t$

- Problem?
- Many words start with s, few words start with x.

Designing hash functions

Some bad hash functions

- Assume keys are English words.
- One bucket for each number in $[26 \cdot 50]$.
- Hash function: $h(w)$ = sum of indices of letters in w.
  - E.g., $h(\text{"}hat\text{"}) = 8 + 1 + 20 = 29$

- Problem?
- Most of the words are short words.

Common technique when designing hash functions

The Division Method

- Hash function: $h(k) = k \bmod m$
  - E.g., if $m = 13$, $h(24) = 11$
- Two keys k_1 and k_2 would collide if $k_1 \equiv k_2 \pmod{m}$

- How to pick m? (Say we want to store n keys.)
- (Bad) Idea: let $r = \lceil \lg n \rceil$, set $m = 2^r$.
- Computing $h(k)$ is very fast: $h(k) = k - ((k \gg r) \ll r)$
- But we are only using rightmost r bits of the input key.
  (So if all input keys are even, we use at most half space.)

Common technique when designing hash functions

The Division Method

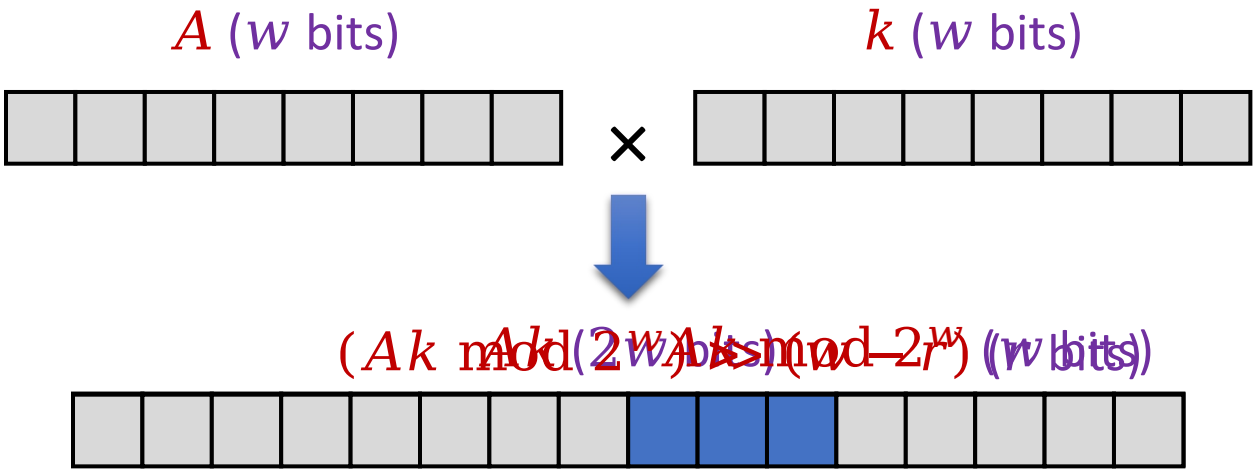
- Hash function: $h(k) = k \bmod m$
  - E.g., if $m = 13$, $h(24) = 11$
- Two keys k_1 and k_2 would collide if $k_1 \equiv k_2 \pmod{m}$

- How to pick m? (Say we want to store n keys.)
- In general, we don't want m to be a composite number.
  - Assume key k and m have common divisor d.
  - $h(k)$ is also divisible by d, as $(k \bmod m) + \lfloor k/m \rfloor \cdot m = k$.
  - If all input keys are divisible by d, we use at most $1/d$ space.
- Rule of thumb: prime not too close to exact power of two.

Common technique when designing hash functions

The Multiplication Method

- Assume key length is at most w bits.
- Fix table size $m = 2^r$ for some $r \leq w$.
- Fix constant integer $0 < A < 2^w$.
- Hash function: $h(k) = (Ak \bmod 2^w) \gg (w - r)$



Common technique when designing hash functions

The Multiplication Method

- Assume key length is at most w bits.
- Fix table size $m = 2^r$ for some $r \leq w$.
- Fix constant integer $0 < A < 2^w$.
- Hash function: $h(k) = (Ak \bmod 2^w) \gg (w - r)$

- Faster than the Division Method.
  - Recall in division method, $h(k) = k \bmod m$
  - Multiplication and bit-shifting faster than division.
- Works reasonably well with proper choice of A.

However...

- Once hash function h is *fixed* and *known*, there must exist a set of "bad" keys that hash to the same value.
- Such *adversarial input* will result in poor performance!

- Use randomization!

Universal Hashing

- Pick a *random* hash function h when the hash table is first built
  - Once chosen, h is fixed throughout entire execution.
  - Since h is randomly chosen, no input is always bad.

- A collection of hash functions $\mathcal{H}$ is *universal* if:
  $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$ for all $x \neq y$
  - For any $x \neq y$, at most $|\mathcal{H}|/m$ hash functions in $\mathcal{H}$ lead to $h(x) = h(y)$.

- "Simple Uniform Hashing" vs "Universal Hashing"
  - **Simple Uniform Hashing:**
    Uncertainty due to randomness of input.
  - **Universal Hashing:**
    Uncertainty due to choice of h (and potentially randomness of input).

Assuming "Universal Hashing"

Performance of hashing with chaining

- Universal hashing: $\Pr_{h \in \mathcal{H}}[h(x) = h(y)] \leq \frac{1}{m}$ for all $x \neq y$
- Load factor $\alpha = \frac{n}{m} = \frac{\text{num of inserted keys}}{\text{size of the table}}$

- **Question:** $L_{h(k)}$ be length of list at index $h(k)$, what's $\mathbb{E}[L_{h(k)}]$?
- **Claim 1:** If key k *not* in table T, then $\mathbb{E}[L_{h(k)}] \leq \alpha$.
  - For any key l, define i.r.v. $X_{kl} = \mathrm{I}\{h(k) = h(l)\}$.
  - $\mathbb{E}[L_{h(k)}] = \mathbb{E}\left[\sum_{l \in T,\ \ l \neq k} X_{kl}\right] = \sum_{l \in T,\ \ l \neq k} \mathbb{E}[X_{kl}] \leq n \cdot \frac{1}{m} = \alpha$
- **Claim 2:** If key k in table T, then $\mathbb{E}[L_{h(k)}] \leq 1 + \alpha$.
  - $\mathbb{E}[L_{h(k)}] = \mathbb{E}\left[\sum_{l \in T,\ \ l \neq k} X_{kl}\right] + 1 \leq (n-1) \cdot \frac{1}{m} + 1 < 1 + \alpha$

If the hash table is not overloaded (i.e., $\alpha = O(1)$), **Search/Insert/Remove** can be done in $O(1)$ expected time.

How to construct a universal hash family $\mathcal{H}$?

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

Strictly speaking, the definition here is actually “2-universal hash family”.

A Typical Universal Hash Family

- Proposed by Carter and Wegman in 1977
  - They introduced the notion of universal classes of hash functions. [“*Universal Classes of Hash Functions*”, STOC 77 and JCSS 79]

- Find a large prime p larger than the max possible key value, let $\mathbb{Z}_p = \{0,1,2, \ldots, p-1\}$ and $\mathbb{Z}_p^* = \{1,2, \ldots, p-1\}$

- Define $h_{ab}(k) = ((ak + b) \bmod p) \bmod m$, then $\mathcal{H}_{pm} = \{h_{ab} \mid a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}$ is a universal hash family.

A Typical Universal Hash Family

$$\mathcal{H}_{pm} = \{h_{ab} \mid a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}$$

- $\mathbb{Z}_p = \{0,1,2,\ldots,p-1\}$ and $\mathbb{Z}_p^* = \{1,2,\ldots,p-1\}$
- $h_{ab}(k) = ((ak+b) \bmod p) \bmod m$, where $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$
- **Goal:** $\Pr_{h \in \mathcal{H}}[h(k) = h(l)] \leq \frac{1}{m}$ for all $k \neq l$, where $k \in \mathbb{Z}_p$ and $l \in \mathbb{Z}_p$

- Let $r = (ak+b) \bmod p$, and $s = (al+b) \bmod p$
- **Claim:** $r \neq s$.
- **Proof:** $r - s \equiv a(k-l) \pmod p$, but $a \not\equiv 0 \pmod p$ and $k - l \not\equiv 0 \pmod p$
- **That is:** h_{ab} does not generate collision at "mod p level".

A Typical Universal Hash Family

Let S be a set, **closed** under binary operations $+$ (addition) and $\cdot$ (multiplication). It gives us the following algebraic structures if the corresponding set of axioms are satisfied.

Structures							Axioms	Operations
field	*commutative ring*	*ring*	*abelian group*	*group*	*monoid*	*semigroup*	1. **Addition** is **associative**: $\forall x, y, z \in S, (x + y) + z = x + (y + z)$.	$+$
							2. Existence of **additive identity 0**: $\forall x \in S, x + 0 = 0 + x = x$.	
							3. Everyone has an **additive inverse**: $\forall x \in S, \exists -x \in S,$ s.t. $x + (-x) = (-x) + x = 0$.	
							4. **Addition** is **commutative**: $\forall x, y \in S, x + y = y + x$.	
							5. Multiplication **distributes** over addition: $\forall x, y, z \in S, x \cdot (y + z) = x \cdot y + x \cdot z$ and $(y + z) \cdot x = y \cdot x + z \cdot x$.	$+, \cdot$
							6. **Multiplication** is **associative**: $\forall x, y, z \in S, (x \cdot y) \cdot z = x \cdot (y \cdot z)$.	$\cdot$
							7. Existence of **multiplicative identity 1**: $\forall x \in S, x \cdot 1 = 1 \cdot x = x$.	
							8. **Multiplication** is **commutative**: $\forall x, y \in S, x \cdot y = y \cdot x$.	
							9. Every non-zero element has a **multiplicative inverse**: $\forall x \in S \setminus \{0\}, \exists x^{-1} \in S,$ s.t. $x \cdot x^{-1} = x^{-1} \cdot x = 1$.	

- Let $r = (ak + b) \bmod p$, and $s = (al + b) \bmod p$
- **Claim:** $r \neq s$. (h_{ab} does not generate collision at "mod p level")
- **Claim:** Fix k and l, there is 1-to-1 mapping between (a, b) and (r, s) pairs.
  - Recall $r - s \equiv a(k - l) \pmod{p}$ — Unique since $\mathbb{Z}_p$ is a field.
  - $a = \left((r - s)((k - l)^{-1} \bmod p)\right) \bmod p$
  - $b = (r - ak) \bmod p$

  Given (r, s), we get *unique* (a, b)
  - There are $(p - 1)p$ pairs of (a, b), and $p(p - 1)$ pairs of (r, s) if $r \neq s$.

A Typical Universal Hash Family

$$\mathcal{H}_{pm} = \{h_{ab} \mid a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}$$

- $\mathbb{Z}_p = \{0,1,2,\ldots,p-1\}$ and $\mathbb{Z}_p^* = \{1,2,\ldots,p-1\}$
- $h_{ab}(k) = ((ak+b) \bmod p) \bmod m$, where $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$
- **Goal:** $\Pr_{h \in \mathcal{H}}[h(k) = h(l)] \leq \frac{1}{m}$ for all $k \neq l$, where $k \in \mathbb{Z}_p$ and $l \in \mathbb{Z}_p$

- Let $r = (ak+b) \bmod p$, and $s = (al+b) \bmod p$
- **Claim:** $r \neq s$. (h_{ab} does not generate collision at "mod p level")
- **Claim:** Fix k and l, there is 1-to-1 mapping between (a,b) and (r,s) pairs.
- **Lemma:** $\Pr_{h \in \mathcal{H}}[h(k) = h(l)] = \Pr_{h \in \mathcal{H}}[(r \bmod m) = (s \bmod m)]$
- $= \Pr[r \equiv s \pmod m$ when (r,s) are distinct values chosen from $\mathbb{Z}_p$ u. a. r
- $\leq \frac{\left(\lceil \frac{p}{m} \rceil - 1\right)}{p-1} \leq \frac{(p-1)/m}{p-1} = \frac{1}{m}$

Reading

- [CLRS] Ch.11 (11.1-11.3)