散列表
Hash Tables

钮鑫涛
Nanjing University

2023 Fall

The slides are mainly adapted from the original ones shared by Chaodong Zheng and Kevin Wayne. Thanks for their supports!

Efficient implementation of Ordered Dictionary

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

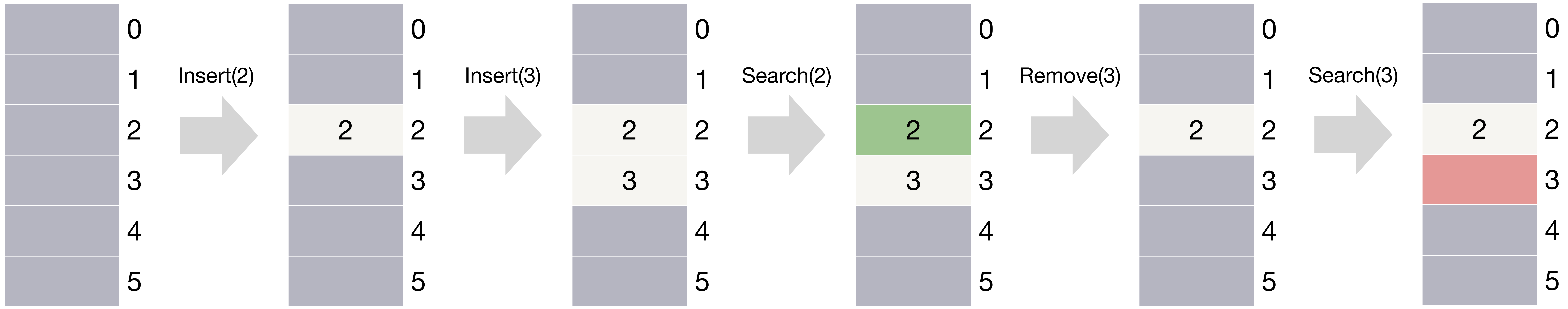
Can we be faster?
(if we only care about Search/Insert/Remove)

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

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

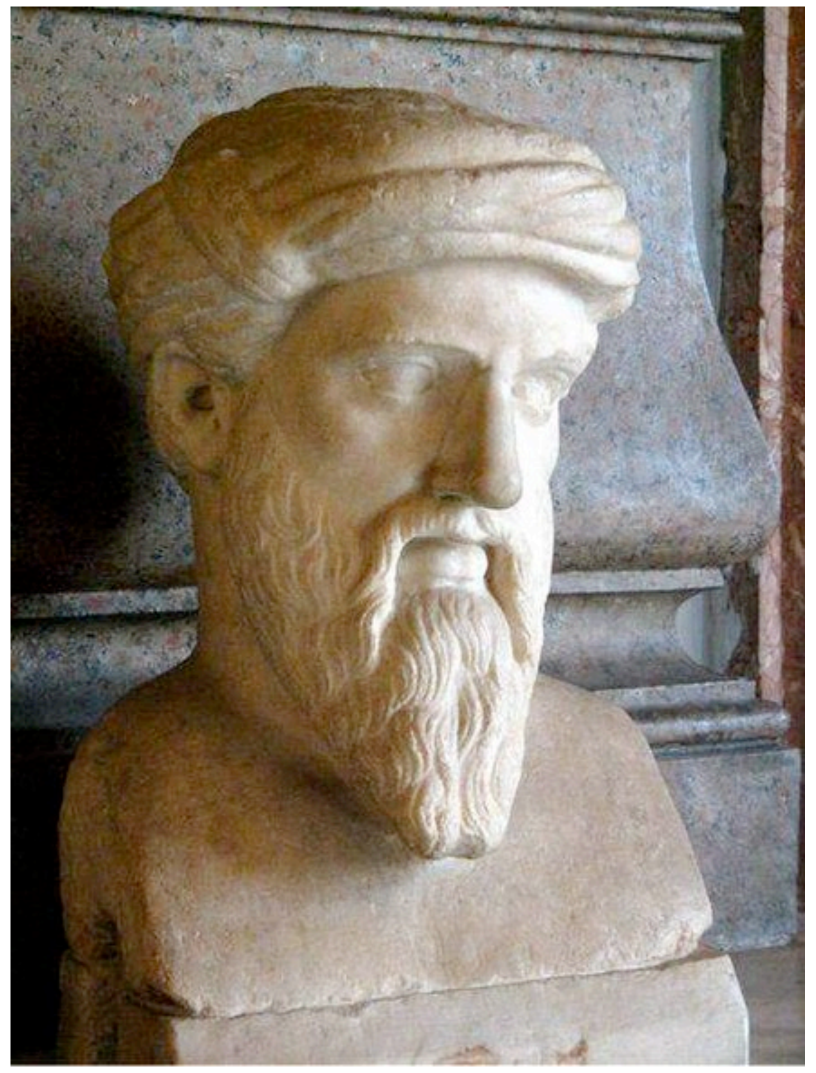
Direct-address Tables

Any potential issue?



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

- Assume keys are distinct integers from universe $U = \{0,1,\ldots,m-1\}$
- Easy, just allocate an array of size $m = |U|$.
- **Search/Insert/Remove** can be done in $O(1)$ time
- Potential Issues:
  - What if keys are distinct, but not integers (e.g., strings).
    - "Everything is number." This is especially true for modern computers…


Pythagoras

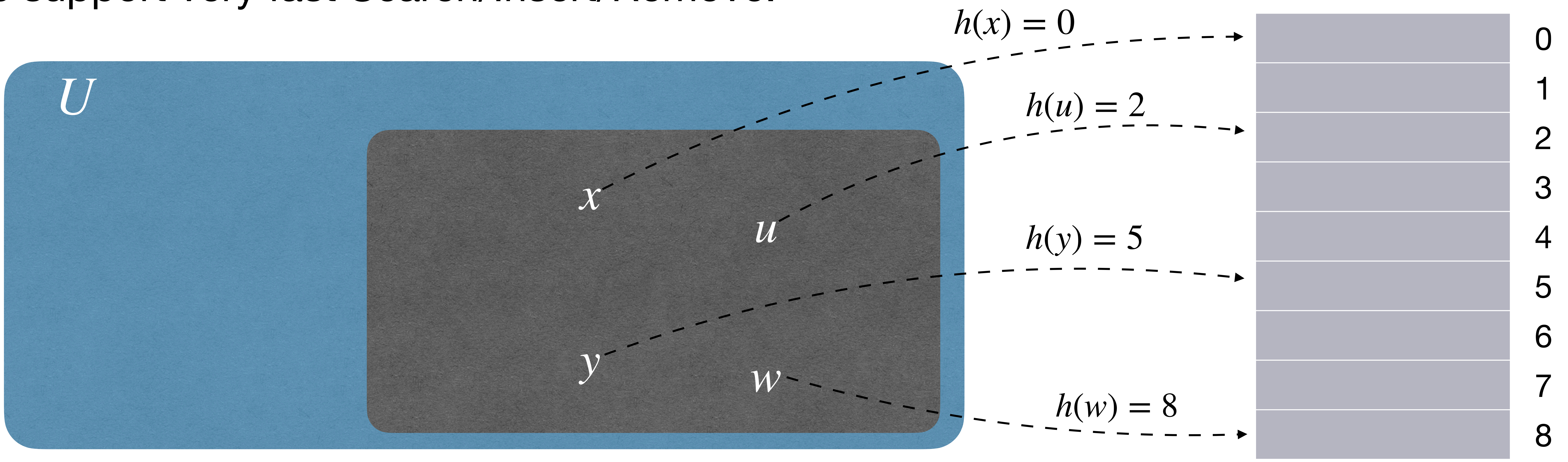
Direct-address Tables

- 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.

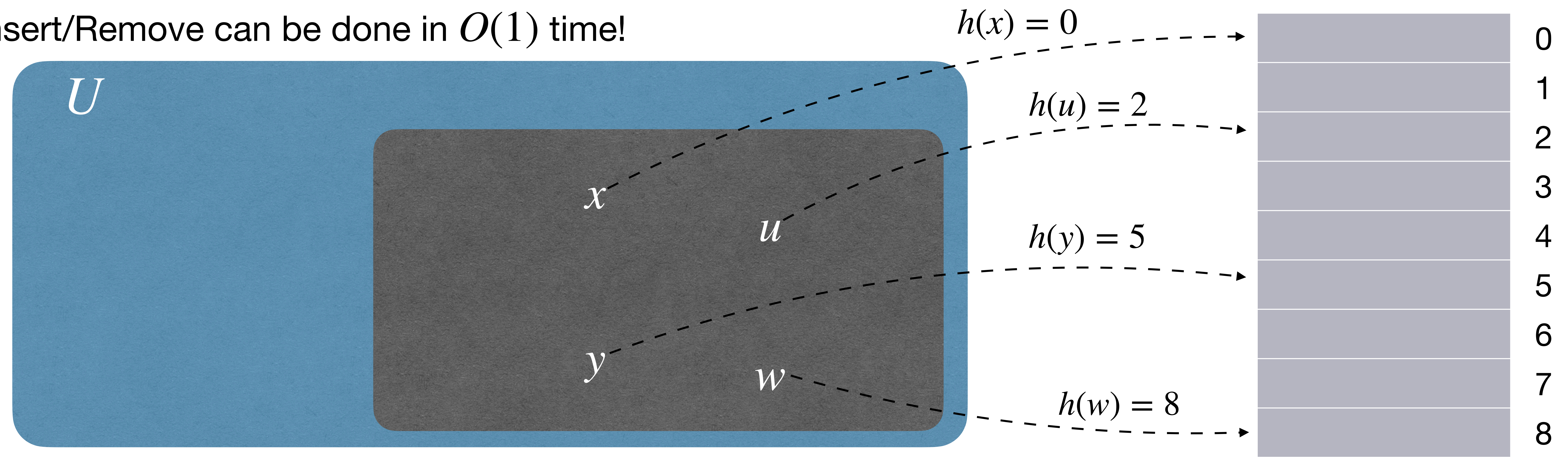
Hash function $h : U \to [m]$
$h(k)$ decides index of slot for storing key k



Hashing

- Design hash function $h : U \to [m]$
- Use $h(k)$ as the index of slog 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. → **NO!**
- Search/Insert/Remove can be done in $O(1)$ time!

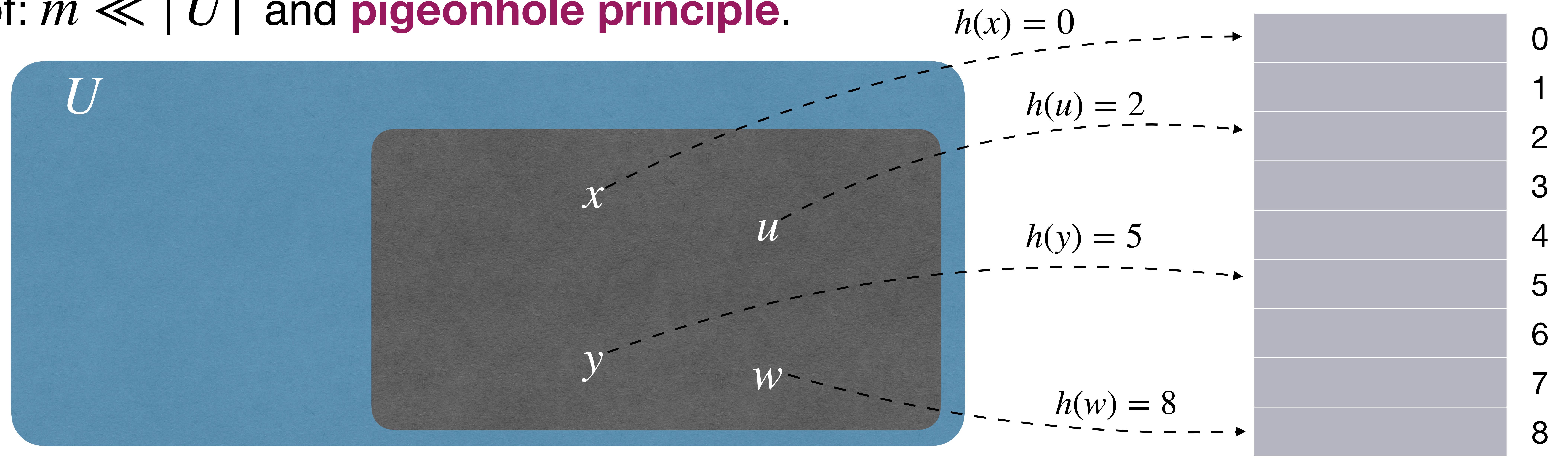
But is this possible?



Collisions in hashing

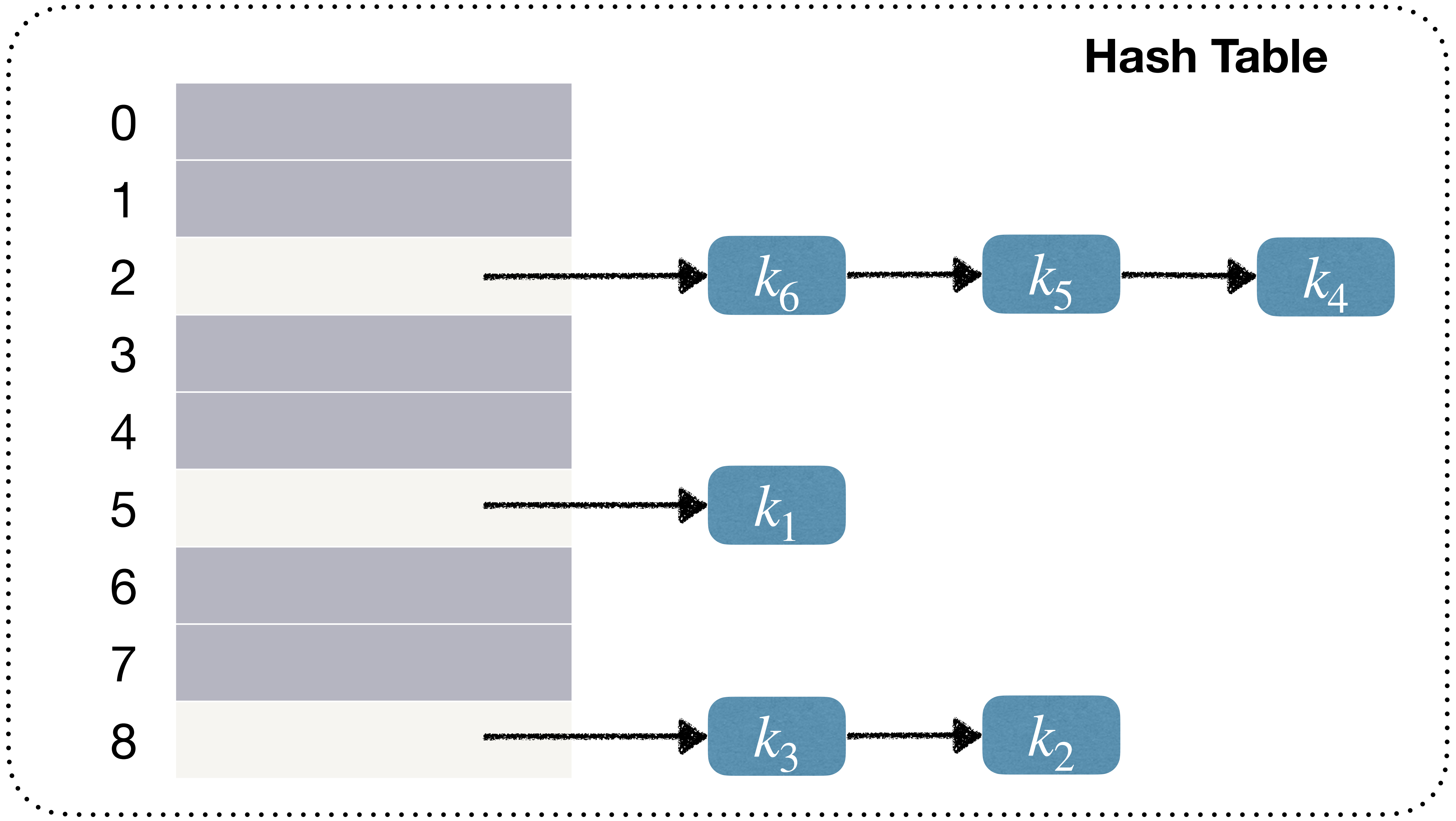
- Hash function $h : U \to [m]$
- Two distinct keys k_1 and k_2 **collide** if: $h(k_1) = h(k_2)$
- Collisions are unavoidable!
  - Proof: $m \ll |U|$ and **pigeonhole principle**.

How to cope with collisions?



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!
  - Search can cost $\Theta(n)$ in worst-case. (All keys hash to same value.)
  - 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 <u>head</u> of the corresponding list.
- `Remove(x)` where x is a pointer to an item. - - - - - - - - - $O(1)$ Note: we assume computing h takes $O(1)$ time.
  - Simply remove x from the linked list.

The “Simple Uniform Hashing” Assumption

- Let’s be optimistic (for now):
  - 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.

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.

Performance of hashing with chaining

- m buckets, storing n keys, **load factor** $\alpha = n/m$.
- 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.
  - $\Theta(1) + \Theta(\alpha) = \Theta(1+\alpha)$

Performance of hashing with chaining

- Expected cost of <u>successful</u> 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 $\frac{1}{n} \cdot \sum_{i=1}^{n} \mathbb{E}[C_i]$
    - Let X_{ij} be an indicator random variable taking value 1 if and only if $h(x_i.key) = h(x_j.key)$

$$\frac{1}{n} \cdot \sum_{i=1}^{n} \mathbb{E}[C_i] = \frac{1}{n} \cdot \sum_{i=1}^{n} \mathbb{E}[(1 + \sum_{j=i+1}^{n} X_{ij})]$$

By linearity of expectation

$$= \frac{1}{n} \cdot \sum_{i=1}^{n} (1 + \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]) = \frac{1}{n} \cdot \sum_{i=1}^{n} (1 + \sum_{j=i+1}^{n} \frac{1}{m})$$

WHY?

$$= 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1+\alpha)$$

Performance of hashing with chaining

- Consider a hash table containing m buckets, storing n keys, and load factor $\alpha = n/m$.

- Expected cost of 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**.

Performance of hashing with chaining

- *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, what is the ***max*** number of balls in a bin, in expectation?

If $m = \Theta(n)$, the answer is $\Theta(\frac{\lg n}{\lg \lg n})$.

Max-Load problem

Reality bites

- "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.
  - Once h is fixed and known, you can find a set of "bad" keys that hash to same value.

Design 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., $h(\text{"test"}) = t$
- Problem?
  - Many words start with s, few words start with x.

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., $h(\text{"hat"}) = 8 + 1 + 20 = 29$
- Problem?
  - Most of the words are short words.

The Division Method

- Common technique when designing hash functions
  - Hash function: $h(k) = k \mod m$
    - E.g., if $m = 13$, $h(24) = 11$
  - Two keys k_1 and would collide if $k_1 \equiv k_2 \pmod m$

The Division Method

- Hash function: $h(k) = k \mod m$
  - E.g., if $m = 13$, $h(24) = 11$
- How to pick m? (Say we want to store n keys)
  - Idea: let $r = \lceil \lg n \rceil$, set $m = 2^r$. - - - - - - - **Bad Idea!**
  - Computing $h(k)$ is very fast: $h(k) = k - \big((k \gg r) \ll r\big)$
  - But we are only using rightmost r bits of the input key.
    - Not good! For example, if all input keys are even, we use at most half space.

The Division Method

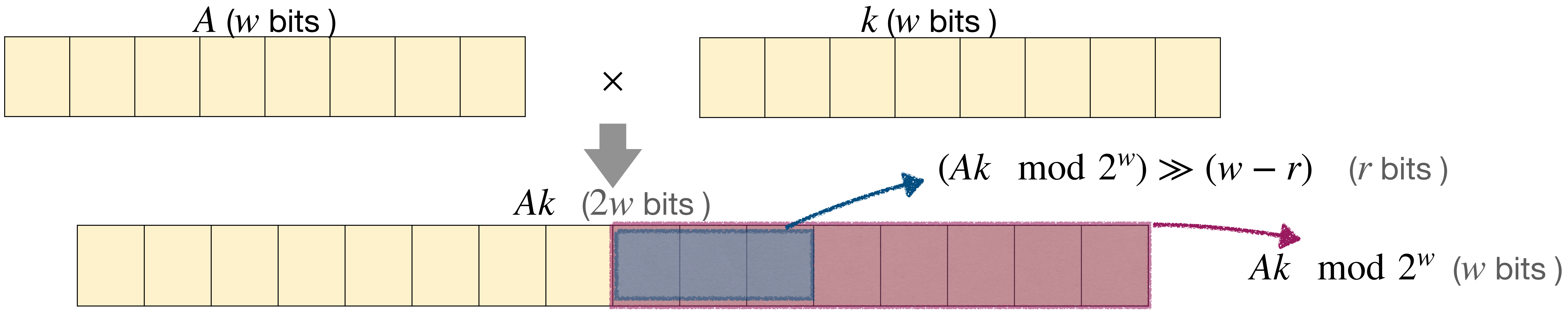
- Hash function: $h(k) = k \mod m$
- How to pick m? (Say we want to store n keys)
- In general, we don't want m to be a composite number.
    - Assume key k and m have **common divisor** d.

      $h(k)$ is also divisible by d, since $(k \mod m) + \lfloor \frac{k}{m} \rfloor \cdot m = k$.
    - If all input keys are divisible by d, we use at most $\frac{1}{d}$ space.

Rule of thumb: prime not too close to exact power of two

The Multiplication Method

- Another common technique when designing hash functions
  - 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: $h(k) = (Ak \mod 2^w) \gg (w - r)$


A (w bits)
k (w bits)
$\times$
Ak ($2w$ bits)
$(Ak \mod 2^w) \gg (w - r)$ (r bits)
$Ak \mod 2^w$ (w bits)

The Multiplication Method

- Another common technique when designing hash functions
  - 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: $h(k) = (Ak \mod 2^w) \gg (w - r)$
- Faster than the **Division Method.**
  - Recall in division method, $h(k) = k \mod m$
  - **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!

Solution: 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 $\mathscr{H}$ is universal if:
    - For any $x \neq y$, at most $\frac{|\mathscr{H}|}{m}$ hash functions in $\mathscr{H}$ lead to $h(x) = h(y)$
        - Therefore, $\Pr_{h \in \mathscr{H}}[h(x) = h(y)] \leq \frac{1}{m}$ for all $x \neq y$

"Simple Uniform Hashing" vs "Universal Hashing"

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

Performance of hashing with chaining

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

How to construct How to construct $\mathscr{H}$?

- Let $L_{h(k)}$ 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 $\mathbb{E}[L_{h(k)}] \le \alpha$.
    - For any key l, define indicator random variable $X_{kl} = I\{h(k)=h(l)\}$.

$$\mathbb{E}[L_{h(k)}] = \mathbb{E}[\sum_{l\in T, l\neq k} X_{kl}] = \sum_{l\in T, l\neq k} \mathbb{E}[X_{kl}] \le n \cdot \frac{1}{m} = \alpha$$

  - **Claim 2**: if key k in table T, then $\mathbb{E}[L_{h(k)}] \le 1+\alpha$.

$$\mathbb{E}[L_{h(k)}] = \mathbb{E}[\sum_{l\in T, l\neq k} X_{kl}] + 1 \le (n-1)\cdot\frac{1}{m} + 1 = 1+\alpha$$

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

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

A Typical Universal Hash Family

- $\mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}$ and $\mathbb{Z}_p^* = \{1, 2, \ldots, p-1\}$
- $h_{ab}(k) = \big((ak+b) \mod p\big) \mod m$, where $a \in \mathbb{Z}_p^*$ and $b \in \mathbb{Z}_p$.
- **Prove**: $\Pr_{h \in \mathscr{H}}[h(k) = h(l)] \le \dfrac{1}{m}$ for all $k \neq l$, where $k \in \mathbb{Z}_p$ and $l \in \mathbb{Z}_p$.

A Typical Universal Hash Family

- Let $r = (ak + b) \mod p$, and $s = (al + b) \mod p$.
- **Claim 1**: $r \neq s$.
- **Proof**:
  - $r - s \equiv a(k - l) \pmod{p}$
  - but $a \not\equiv 0 \pmod{p}$ and $k - l \not\equiv 0 \pmod{p}$
  - p is prime!

That is: h_{ab} does not generate collision at " $\mod p$ level"!

A Typical Universal Hash Family

- Let $r = (ak + b) \mod p$, and $s = (al + b) \mod p$.
- **Claim 2**: Fix k and l, there is a **1-to-1** mapping between (a, b) and (r, s) pairs.
  - Recall $r - s \equiv a(k - l) \pmod p$

    modular multiplicative inverse of $k - l$, unique since p is prime
  - $a = \left((r - s)\left((k - l)^{-1} \mod p\right)\right) \mod p$
  - $b = (r - ak) \mod p$

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

$a \in \mathbb{Z}_p^*, b \in \mathbb{Z}_p$

$0 \le r < p, 0 \le s < p$

A Typical Universal Hash Family

- Let $r = (ak + b) \mod p$, and $s = (al + b) \mod p$.
- **Claim 1**: $r \neq s$.
- **Claim 2**: Fix k and l, there is a 1-to-1 mapping between (a, b) and (r, s) pairs.
- Thus, for any given pair of distinct inputs of k and l, if we pick (a, b) uniformly at random from $\mathbb{Z}_p^* \times \mathbb{Z}_p$, the resulting pair (r, s) is equally likely to be any pair of distinct values modulo p.
- Therefore, the probability that distinct keys k and l collide is equal to the probability that $r \equiv s \pmod{m}$

A Typical Universal Hash Family

- Let $r = (ak + b) \mod p$, and $s = (al + b) \mod p$.
- **Claim 1**: $r \neq s$.
- **Claim 2**: Fix k and l, there is a 1-to-1 mapping between (a, b) and (r, s) pairs.
- **Lemma**: $\Pr_{h \in \mathscr{H}}[h(k) = h(l)] = \Pr_{0 \le r,s < p}[r \equiv s \pmod m)]$

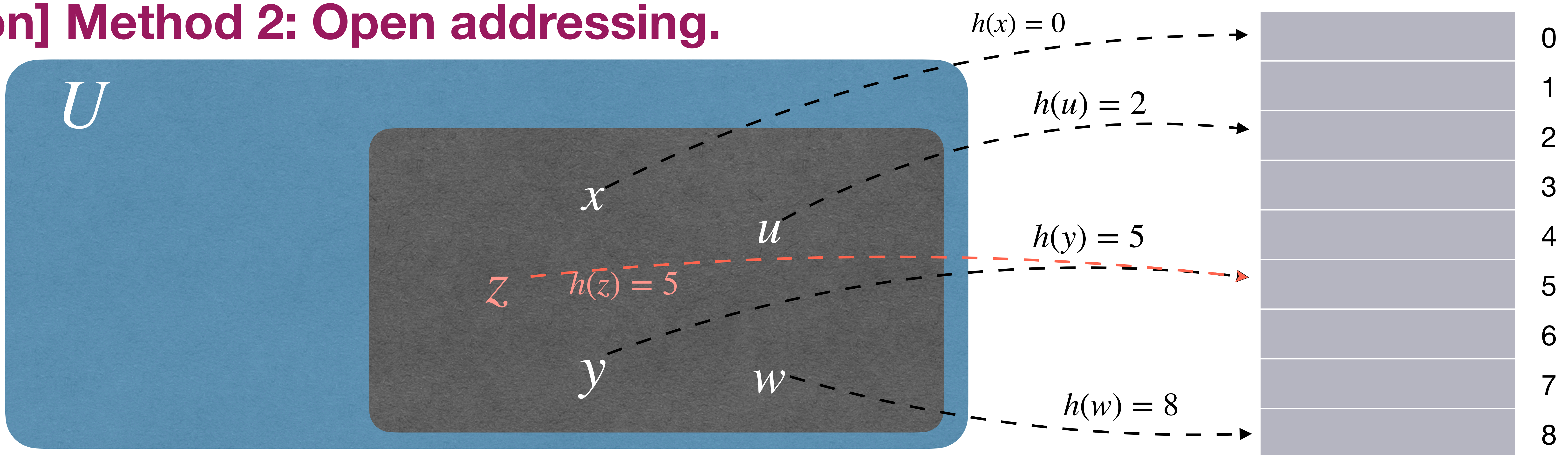
$= \Pr[r \equiv s \pmod m)$ when (r, s) are distinct values chosen from $\mathbb{Z}_p$ uniformly at random]

$$\le \frac{(\lceil \frac{p}{m} \rceil) - 1}{p - 1} \le \frac{\frac{p+m-1}{m} - 1}{p - 1} = \frac{(p-1)/m}{p - 1} = \frac{1}{m}$$

Open addressing

Quick Review

- Hash Tables
  - Store n keys from a huge universe U into a table of size $m \approx n$
  - Use a hash function $h : U \to [m]$ to decide where to put each key
- Collisions are inevitable!
  - [Collision Resolution] Method 1: Chaining.
  - **[Collision Resolution] Method 2: Open addressing.**



Open Addressing

- Basic idea:
  - No linked lists! All items store in the table, one item per bucket!
- Load factor $\alpha = \frac{n}{m} \leq 1$
- On collision?
  - Probe a **sequence** of buckets until an empty one is found!

But hash function h is a function on key, how is the probe sequence determined?

key = 321
$h(321) = 1$
245
366
111
571
533
0 1 2 3 4 5 6 7 8
key = 321
$h(321) = 7$
key = 321
$h(321) = 4$
321

Hash Function Re-defined

- In case we use open addressing for collision resolution,
  - $h : \underbrace{U}_{\text{key}} \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{table index}}$

key = 321
h(321,0) = 1
245
366
111
571
533
0
1
2
3
4
5
6
7
8
key = 321
h(321,1) = 7
245
366
111
571
533
key = 321
h(321,2) = 4
245
366
321
111
571
533

Hash Function Re-defined

- In case we use open addressing for collision resolution,
  - $h : \underbrace{U}_{\text{key}} \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \rightarrow \underbrace{\{0, 1, \ldots, m-1\}}_{\text{table index}}$

```
HashInsert(T, k):
i := 0
repeat
    $j := h(k, i)$
    if  $T[j] = \text{NULL}$
        $T[j] := k$
        return $j$
    else $i := i + 1$
until $i = m$
return "overflow"
```

```
HashSearch(T, k):
i := 0
repeat
    $j := h(k, i)$
    if  $T[j] = k$
        return $j$
    $i := i + 1$
until $i = m$ or $T[j] = \text{NULL}$
return NULL
```

The Remove Operation

HashInsert(T, k):

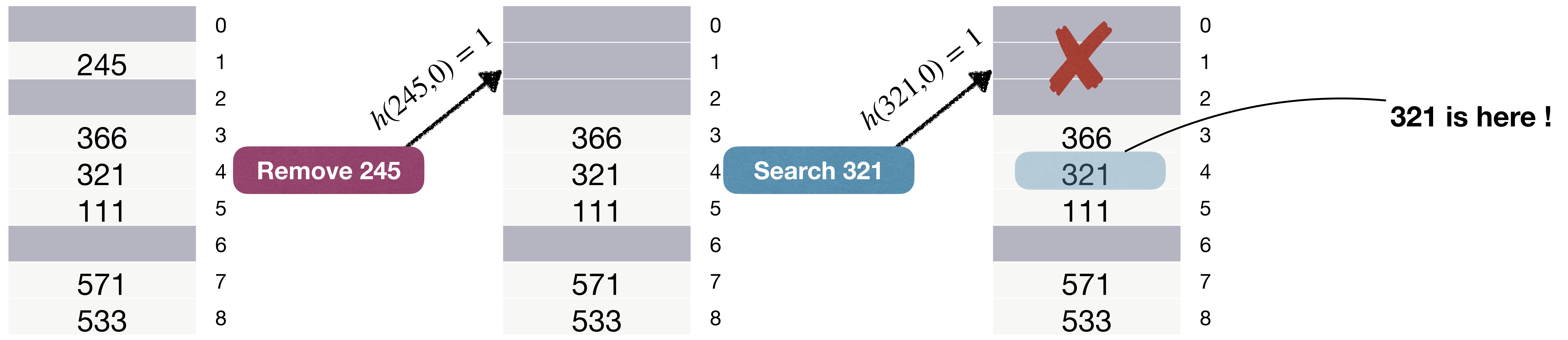
```
$i := 0$
repeat
    $j := h(k, i)$
    if  $T[j] = \text{NULL}$
        $T[j] := k$
        return $j$
    else $i := i + 1$
until $i = m$
return "overflow"
```

HashSearch(T, k):

```
$i := 0$
repeat
    $j := h(k, i)$
    if  $T[j] = k$
        return $j$
    $i := i + 1$
until $i = m$ or $T[j] = \text{NULL}$
return NULL
```

HashRemove(T, k):

```
$pos := HashSearch(T, k)$
if  $pos$ != NULL
    $T[pos] := \text{NULL}$
return $pos$
```



智能软件与工程学院
School of Intelligent Software and Engineering

The Remove Operation

```
HashInsert(T, k):
$i := 0$
repeat
    $j := h(k, i)$
    if $T[j] = \text{NULL}$ or $T[j] = \text{DEL}$
        $T[j] := k$
        return $j$
    else $i := i + 1$
until $i = m$
return "overflow"
```

```
HashSearch(T, k):
$i := 0$
repeat
    $j := h(k, i)$
    if $T[j] = k$
        return $j$
    $i := i + 1$
until $i = m$ or $T[j] = \text{NULL}$
return NULL
```

```
HashRemove(T, k):
$pos := HashSearch(T, k)$
if $pos \mathrel{!=} \text{NULL}$
    $T[pos] := \textbf{DEL}$
return $pos$
```

245
366
321
111
571
533
0
1
2
3
4
5
6
7
8
Remove 245
$h(245,0) = 1$
DEL
Search 321
$h(321,0) = 1$
Search 321
$h(321,1) = 7$
Search 321
$h(321,2) = 4$

Linear Probing

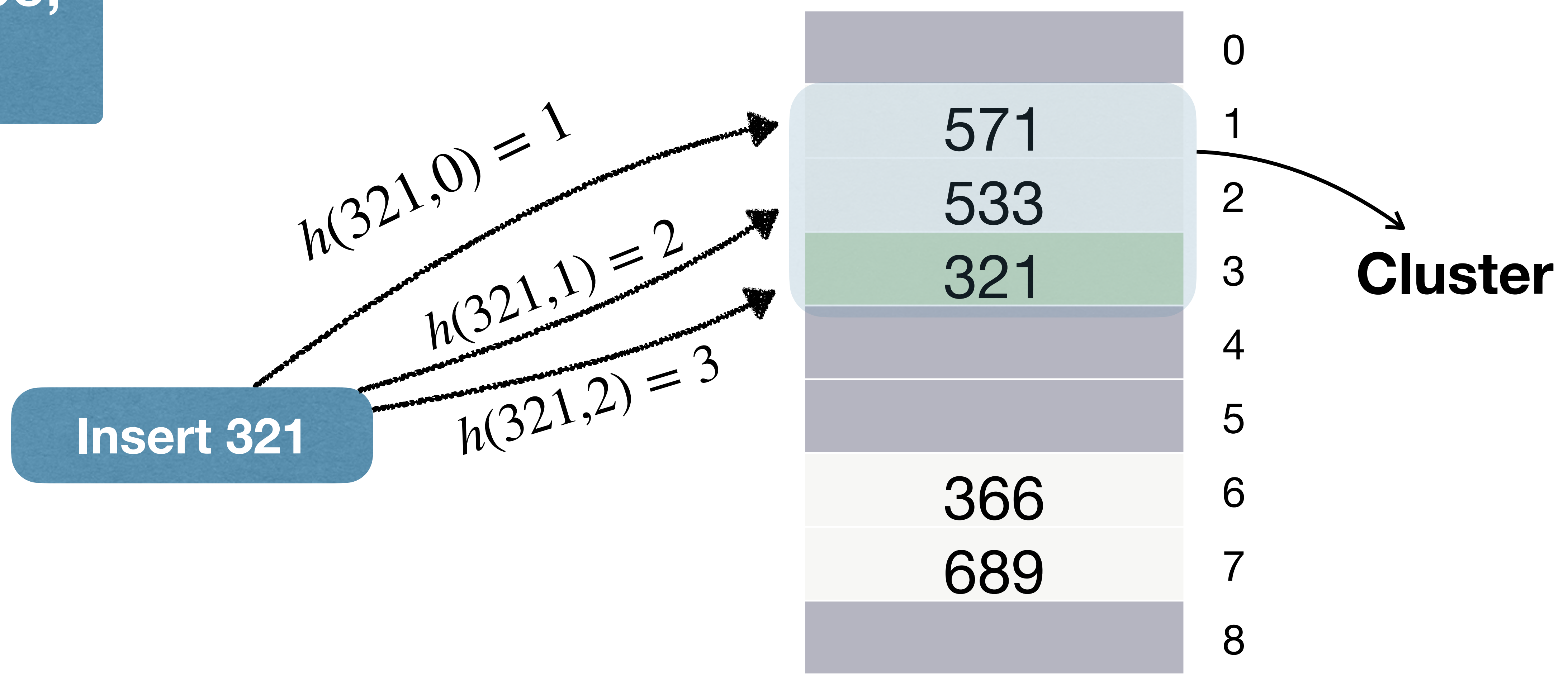
- $h(k,i) = \left(h'(k) + i\right) \mod m.$

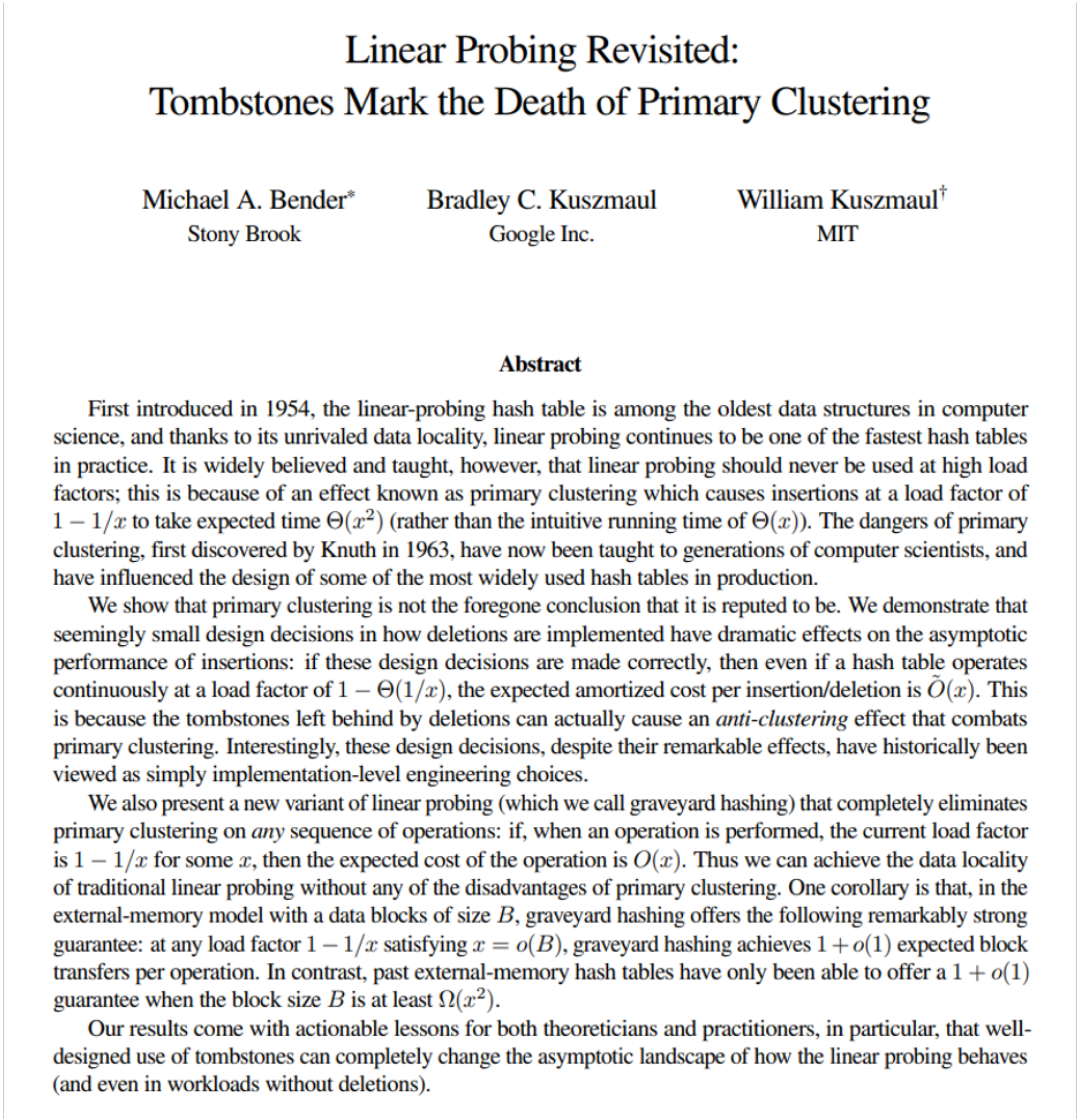
Here, h' is an "auxiliary hash function".

- The probe sequence is $h'(k), h'(k)+1, h'(k)+2, \ldots$

Since the initial probe position determines the entire probe sequence, only m distinct probe sequences are used with linear probing.

- Another problem with linear probing: **Clustering.**
  - Empty slot after a "cluster" has higher chance to be chosen.
  - "Cluster" grows larger and larger.
  - Cluster" leads to higher search time, in theory.





Linear Probing Revisited: Tombstones Mark the Death of Primary Clustering

Michael A. Bender[*]
Stony Brook

Bradley C. Kuszmaul
Google Inc.

William Kuszmaul[†]
MIT

Abstract

First introduced in 1954, the linear-probing hash table is among the oldest data structures in computer science, and thanks to its unrivaled data locality, linear probing continues to be one of the fastest hash tables in practice. It is widely believed and taught, however, that linear probing should never be used at high load factors; this is because of an effect known as primary clustering which causes insertions at a load factor of $1 - 1/x$ to take expected time $\Theta(x^2)$ (rather than the intuitive running time of $\Theta(x)$). The dangers of primary clustering, first discovered by Knuth in 1963, have now been taught to generations of computer scientists, and have influenced the design of some of the most widely used hash tables in production.

We show that primary clustering is not the foregone conclusion that it is reputed to be. We demonstrate that seemingly small design decisions in how deletions are implemented have dramatic effects on the asymptotic performance of insertions: if these design decisions are made correctly, then even if a hash table operates continuously at a load factor of $1 - \Theta(1/x)$, the expected amortized cost per insertion/deletion is $\tilde{O}(x)$. This is because the tombstones left behind by deletions can actually cause an *anti-clustering* effect that combats primary clustering. Interestingly, these design decisions, despite their remarkable effects, have historically been viewed as simply implementation-level engineering choices.

We also present a new variant of linear probing (which we call graveyard hashing) that completely eliminates primary clustering on *any* sequence of operations: if, when an operation is performed, the current load factor is $1 - 1/x$ for some x, then the expected cost of the operation is $O(x)$. Thus we can achieve the data locality of traditional linear probing without any of the disadvantages of primary clustering. One corollary is that, in the external-memory model with a data blocks of size B, graveyard hashing offers the following remarkably strong guarantee: at any load factor $1 - 1/x$ satisfying $x = o(B)$, graveyard hashing achieves $1 + o(1)$ expected block transfers per operation. In contrast, past external-memory hash tables have only been able to offer a $1 + o(1)$ guarantee when the block size B is at least $\Omega(x^2)$.

Our results come with actionable lessons for both theoreticians and practitioners, in particular, that well-designed use of tombstones can completely change the asymptotic landscape of how the linear probing behaves (and even in workloads without deletions).

The remove mechanism
(i.e., the DEL mark”)
causes “anti-clustering” effect,
improving the performance of
linear-probing hash tables.

Quadratic Probing

Here, h' is an “auxiliary hash function”.

- $h(k,i) = \left(h'(k) + c_1 i + c_2 i^2\right) \mod m.$
- Problem with quadratic probing: **(Secondary) Clustering.**
  - Keys having same h' values result in same probe sequences.
  - As in linear probing, the initial probe determines the entire sequence, so only m distinct probe sequences are used.

Double Hashing

- $h(k, i) = \left(h_1(k) + i \cdot h_2(k)\right) \mod m.$

Here, h_1 and h_2 are both “auxiliary hash functions”.

- Why “doubling” hashing?
  - **Observation 1**: If h_1 is good, $h(k,0)$ looks “random”.
  - **Observation 2**: If h_2 is good, probe sequence looks “random”.
  - Linear and quadratic probing does not give **Observation 2**.

Double Hashing

- $h(k,i) = \left(h_1(k) + i \cdot h_2(k)\right) \mod m.$

Here, h_1 and h_2 are both “auxiliary hash functions”.

- The value $h_2(k)$ must be relatively prime to m for the **entire hash table** to be searched. Conveniently, just let m be a prime number.
  - Otherwise, if m and $h_2(k)$ have greatest common divisor $d > 1$ for some key k, then a search for key k would examine only $\dfrac{1}{d}$ of the hash table.
- Each possible $\left(h_1(k), h_2(k)\right)$ pair yields a distinct probe sequence
  - As we vary the key, the initial probe position $h_1(k)$ and the offset $h_2(k)$ may vary independently.
  - Double hashing can use $\Theta(m^2)$ different probe sequences.

Performance of open-address hashing

- Recall the “Simple Uniform Hashing” Assumption:
  - Every key is equally likely to map to every bucket
  - Keys are mapped independently.
- The “**Uniform Hashing**” Assumption:
  - The probe sequence of each key is equally likely to be any of the $m!$ permutations of $< 0, 1, \ldots, m-1 >$.
- None of linear probing, quadratic probing, or double hashing fulfills the “uniform hashing” assumption!
  - But double hashing does better than the other two.

Performance of open-address hashing

Theorem Let random variable X be the number of probes made in an **unsuccessful** search, then $\mathbb{E}[X] \leq \dfrac{1}{1-\alpha}$. Here, $\alpha = \dfrac{n}{m} < 1$.

- Let event A_i be: The i^{th} probe leads to an occupied bucket.
- $\Pr[A_i \mid A_1 \cap A_2 \cap \ldots \cap A_{i-1}] = \dfrac{n-(i-1)}{m-(i-1)} \leq \dfrac{n}{m}$
- $\Pr[X \geq i] = \Pr[A_1 \cap A_2 \cap \ldots \cap A_{i-1}]$

$$= \Pr[A_1] \cdot \Pr[A_2 \mid A_1] \cdot \Pr[A_3 \mid A_1 \cap A_2] \ldots \Pr[A_{i-1} \mid A_1 \cap A_2 \cap \ldots \cap A_{i-2}] \leq (\frac{n}{m})^{i-1} = \alpha^{i-1}$$

- $\mathbb{E}[X] = \sum_{i=1}^{\infty} \Pr[X \geq i] \leq \sum_{i=1}^{\infty} \alpha^{i-1} = 1 + \alpha + \alpha^2 + \ldots. = \dfrac{1}{1-\alpha}$

Always make 1^{st} probe
Make 2^{nd} probe with probability $\approx \alpha$
Make 3^{rd} probe with probability $\approx \alpha^2$
...

Performance of open-address hashing

Theorem Let random variable X be the number of probes made in an **successful** search, then $\mathbb{E}[X] \leq \frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$. Here, $\alpha = \frac{n}{m} < 1$.

- Let N_i be: the expected number of probes made when searching the i^{th} inserted key.
- Due to previous analysis, $N_i \leq \frac{1}{1-\frac{i-1}{m}}$
- $\mathbb{E}[X] \leq \frac{1}{n} \cdot \sum_{i=1}^{n} N_i \leq \frac{1}{n} \cdot \sum_{i=1}^{n} \frac{m}{m-(i-1)} = \frac{m}{n} \cdot \sum_{i=0}^{n-1} \frac{1}{m-i} = \frac{1}{\alpha} \cdot \sum_{k=m-n+1}^{m} \frac{1}{k}$

$$\leq \frac{1}{\alpha} \cdot \int_{m-n}^{m} \frac{1}{x}dx = \frac{1}{\alpha} \cdot \ln\left(\frac{m}{m-n}\right) = \frac{1}{\alpha} \cdot \ln\left(\frac{1}{1-\alpha}\right)$$

Chaining vs Open-addressing

- **Good** parts of Open-addressing:
  - No memory allocation
    - Chaining needs to allocate list-nodes
  - Better cache performance
    - Hash table stores in a continuous region in memory
    - Fewer accesses brings table into cache

- **Bad** parts of Open-addressing:
  - Sensitive to choice of hash functions
    - Clustering is a common problem
  - Sensitive to load factor
    - Poor performance when $\alpha \approx 1$

Efficient implementation of Ordered Dictionary

	`Search(S,k)`	`Insert(S,x)`	`Remove(S,x)`
`Treap / SkipList`	$O(\log n)$ in expectation	$O(\log n)$ in expectation	$O(\log n)$ in expectation
`RB-Tree`	$O(\log n)$ in worst case	$O(\log n)$ in worst case	$O(\log n)$ in worst case
`Hashing (chaining)`	$O(1+\alpha)$ in expectation	$O(1)$ in worst case	$O(1)$ in worst case
`Hashing (open addressing)`	$O(\frac{1}{1-\alpha})$ (unsuccessful) $O\left(\frac{1}{\alpha}\ln(\frac{1}{1-\alpha})\right)$ (successful) in expectation	$O(\frac{1}{1-\alpha})$ in expectation	Same as searching

Further reading

- [CLRS] Ch.11