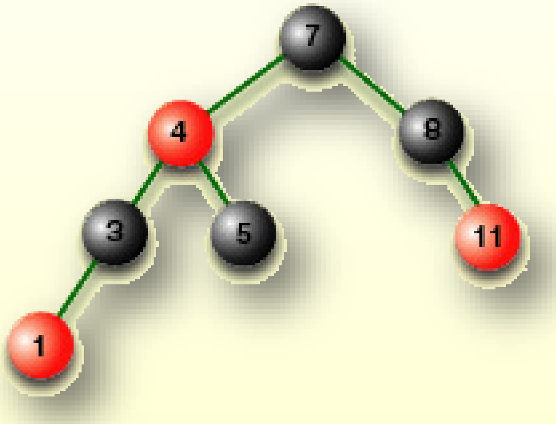
CS161:
Design and Analysis of Algorithms



Lecture 8
Leonidas Guibas

Outline

- Review of last lecture: Sorting Lower Bounds, Linear Time Sorting

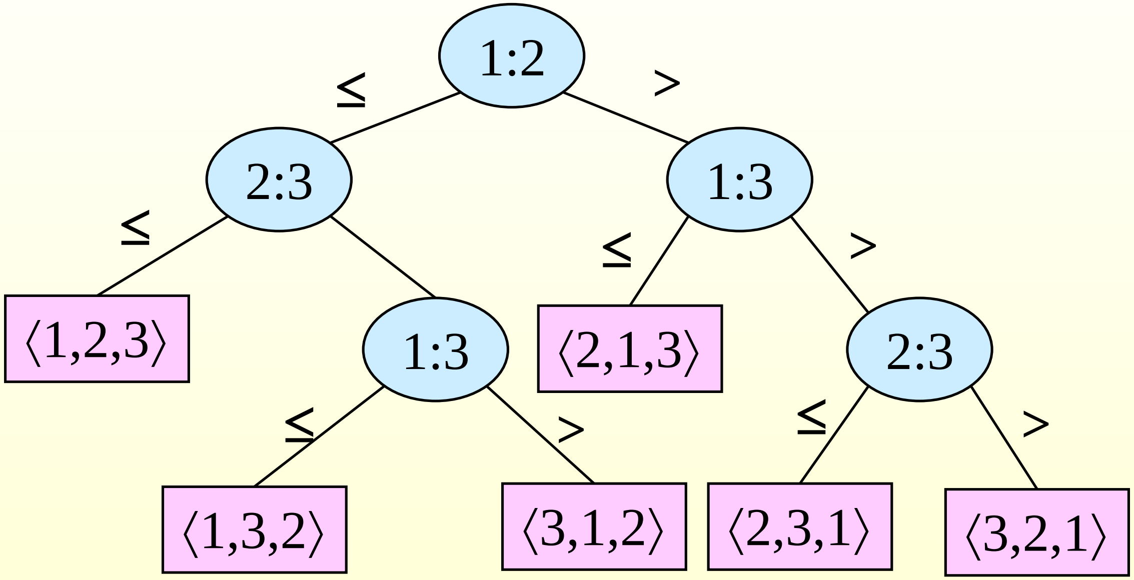
- Hashing
  - Chained hashing methods
  - Open addressing methods
  - Hash functions
  - Universal families

Slides modified from

- http://www.cs.unc.edu/~plaisted/comp122/00-intro.ppt
- http://school.eecs.wsu.edu/undergraduate/cpts/courses/223/

Decision Tree Approach

For InsertionSort operating on three elements.



Simply unroll all loops for all possible inputs.

Node *i*:*j* means compare A[*i*] to A[*j*].

Leaves show outputs;

No two paths go to same leaf!

Contains 3! = 6 leaves.

CountingSort: Example

Input A:

1	2	3	4	5	6	7	8	9	10
0	3	2	1	3	2	1	2	2	3

N=10
M=4

(all elements in input between 0 and 3)

Count array C:

0	**1**
1	**2**
2	**4**
3	**3**

Output sorted array B:

1	2	3	4	5	6	7	8	9	10
0	1	1	2	2	2	2	3	3	3

Time = $O(N + M)$

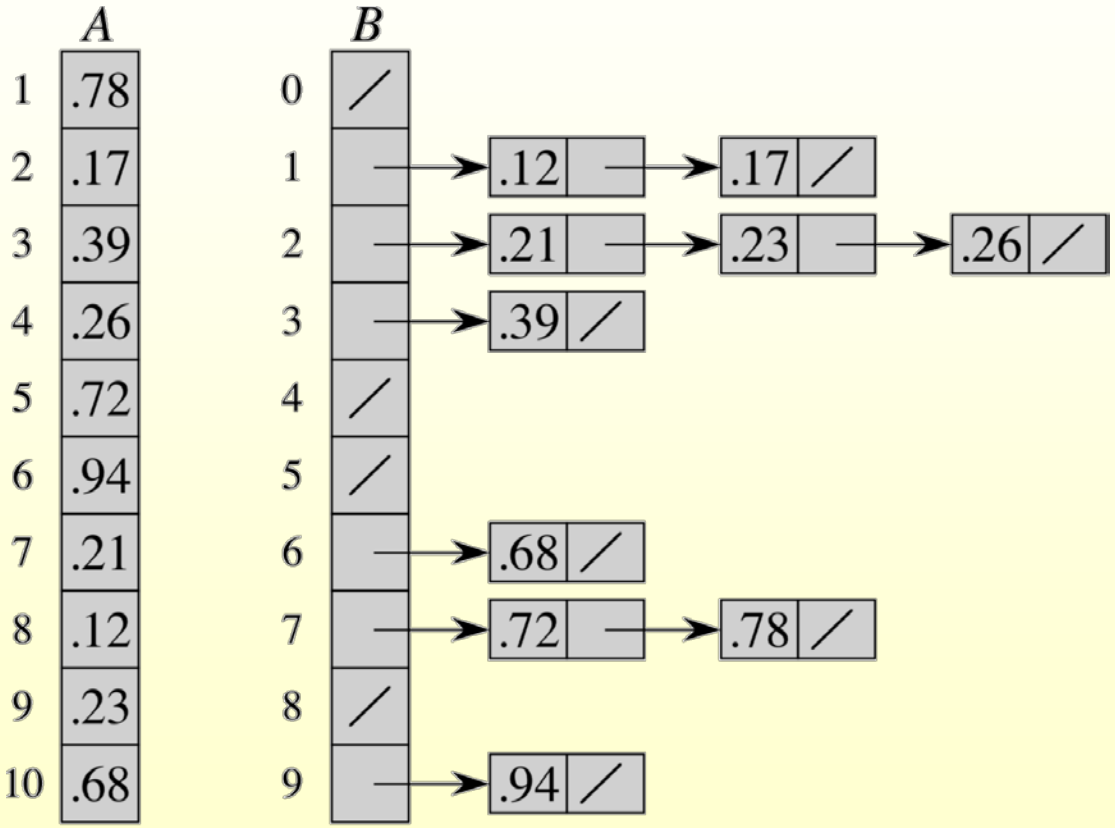
If $(M < N)$, Time = $O(N)$

RadixSort Example

Radix sort is the algorithm used by card-sorting machines that today may be found only in museums.

input			**output**
32**9**	720	**7**20	329
45**7**	355	**3**29	355
65**7**	436	**4**36	436
83**9**	457	**8**39	457
43**6**	657	**3**55	657
72**0**	329	**4**57	720
35**5**	839	**6**57	839

BucketSort Example



(a) (b)

Hashing and Hash Tables

Caller ID Problem Scenario

Consider a large phone company that wants to provide Caller ID service to its customers:

- Given a phone number, return the caller's name

Key	Element
phone number	caller's name

Assumption: Phone numbers are unique and are in the range 0..10^7 - 1. However, not all those numbers are current phone numbers.

How shall we store and look up our (phone number, name) pairs?

How do we update the record data base?

Abstract Data Structure: A Dictionary

- **Dictionary:**
  - Dynamic-set data structure for storing items, indexed using *keys*.
  - Supports operations: Insert, Search, and Delete.
  - Applications:
    - Symbol table of a compiler.
    - Memory-management tables in operating systems.
    - Large-scale distributed systems.
- **Hash Tables:**
  - Effective way of implementing dictionaries.
  - Generalization of ordinary arrays.

Hash Tables

- Hash table:
  - Given a table *T* and a record *x*, with a key and satellite data, we need to support:
    - Insert (*T*, *x*)
    - Delete (*T*, *x*)
    - Search(*T*, *x*)
  - We want these to be fast, but don't care about sorting the records, or about the relative order of the records
  - In this discussion we consider all keys to be (possibly large) natural numbers

One Solution: Direct Addressing

- Suppose:
  - The range of keys is 0..*M*-1
  - All keys are distinct
- The idea:
  - Set up an array T[0..*M*-1] in which
    - T[*i*] = *x* if $x \in T$ and key[*x*] = *i*
    - T[*i*] = NULL otherwise
  - This is called a *direct-address table*
    - All operations take O(1) time!

The Problem With Direct Addressing

- Direct addressing works well when the range *m* of keys is relatively small
- But what if the keys are 32-bit integers?
  - Problem 1: direct-address table will have 2^{32} entries, more than 4 billion
  - Problem 2: even if memory is not an issue, the time to initialize the elements to NULL may be significant
- Solution: map keys to smaller range 0..*m*-1 (*m* << *M*)
- This mapping is called a hash function

Hash Tables

- **Notation:**
  - *U* – Universe of all possible keys (of size M).
  - *K* – Set of keys actually stored in the dictionary.
  - $|K| = n$ (where $n \ll M$).
- When U is very large,
  - Arrays are not practical.
  - $|K| \ll |U|$.
- Use a table of size m, proportional to $|K|$ – The hash table.
  - However, we lose the direct-addressing ability.
  - Define functions that map keys to slots of the hash table.

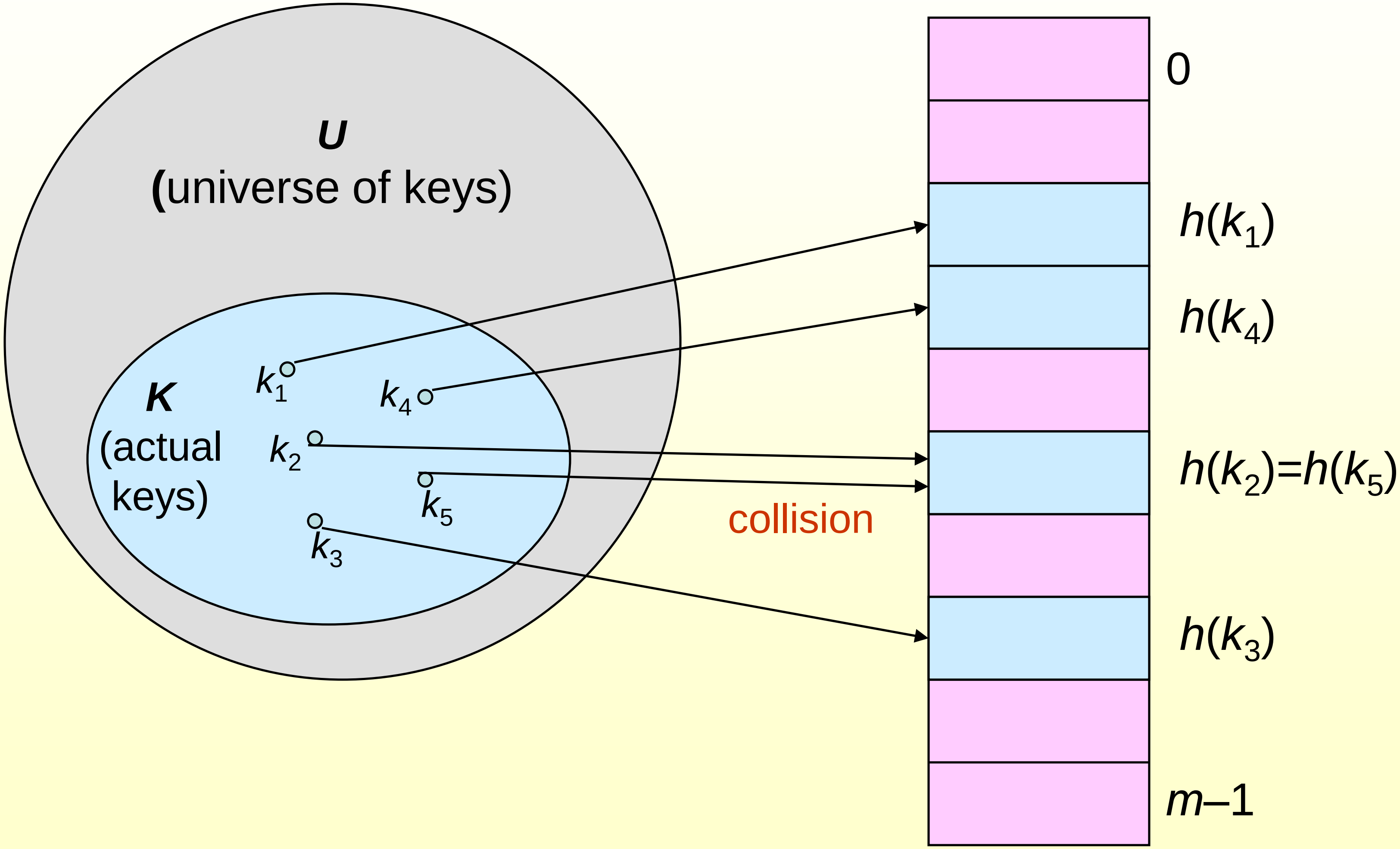
Hashing

- Hash function h: Mapping from U to the slots of a hash table $T[0..m–1]$.

$$h : U \to \{0,1,\ldots, m–1\}$$

- With arrays, key k maps to slot $A[k]$.
- With hash tables, key k maps, or "hashes", to slot $T[h[k]]$.
- $h[k]$ is the *hash value* of key k.

Hashing



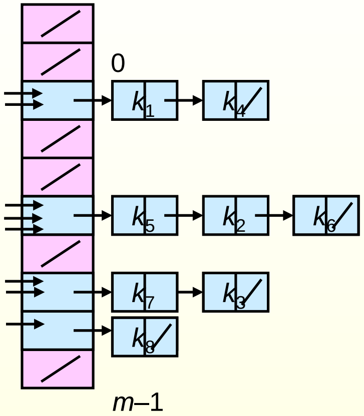
U
(universe of keys)
K
(actual keys)
k_1
k_4
k_2
k_5
k_3
collision
0
$h(k_1)$
$h(k_4)$
$h(k_2)=h(k_5)$
$h(k_3)$
$m–1$

Issues with Hashing

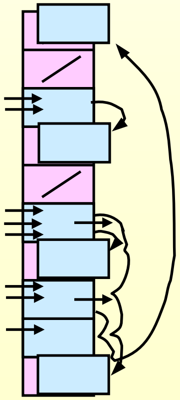
- Multiple keys can hash to the same slot – collisions are possible.
  - Design hash functions such that collisions are minimized.
  - But avoiding collisions is impossible.
    - Design collision-resolution techniques.
- Search will cost Θ(n) time in the worst case.
  - However, all operations can be made to have an expected complexity of Θ(1).

Methods of Collision Resolution

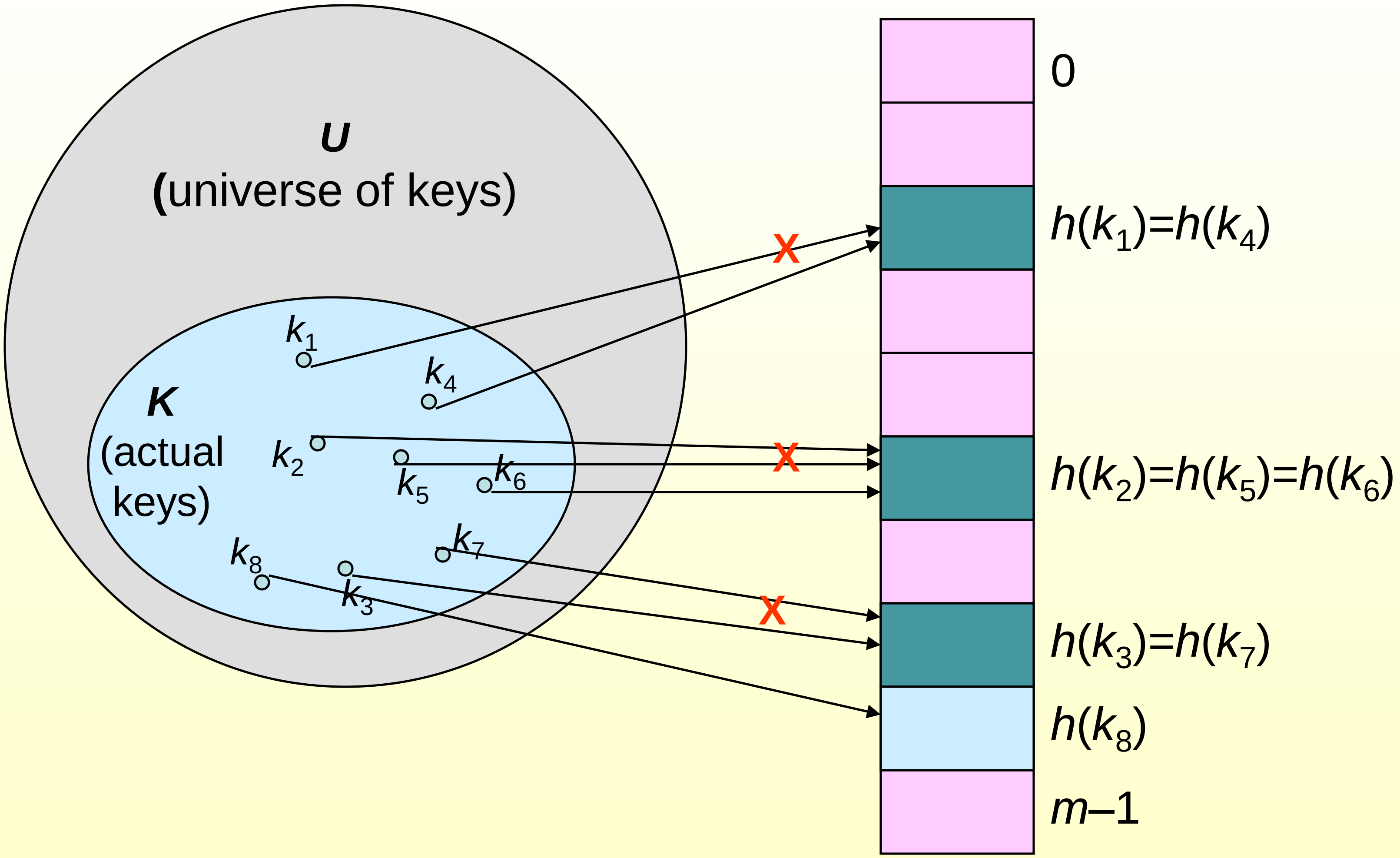
- Chaining:
  - Store all elements that hash to the same slot in a linked list.
  - Store a pointer to the head of the linked list in the hash table slot.



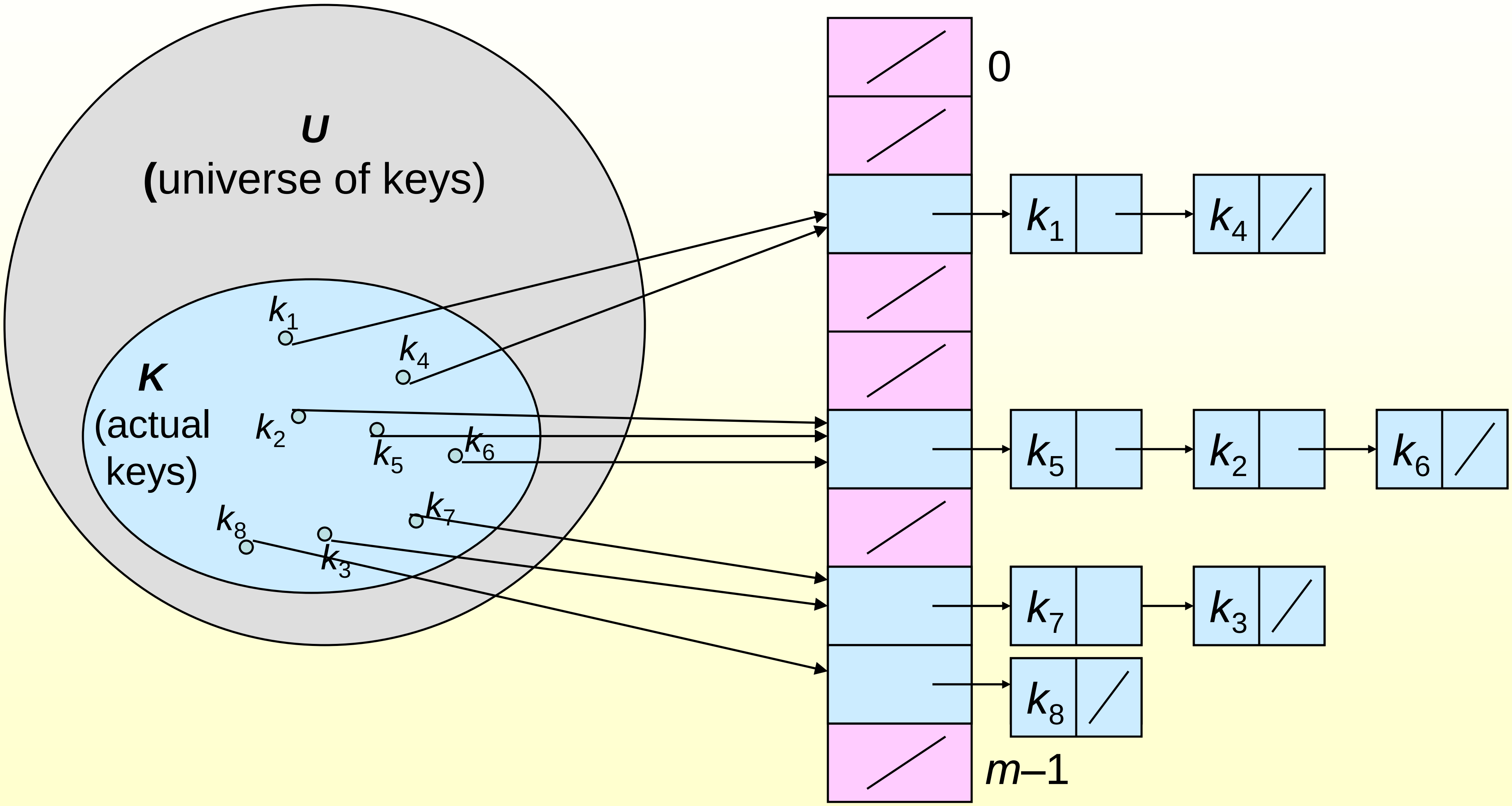
- Open Addressing:
  - All elements stored in hash table itself.
  - When collisions occur, use a systematic (consistent) procedure to store (and search for) elements in free slots of the table.



Collision Resolution by Chaining



Collision Resolution by Chaining



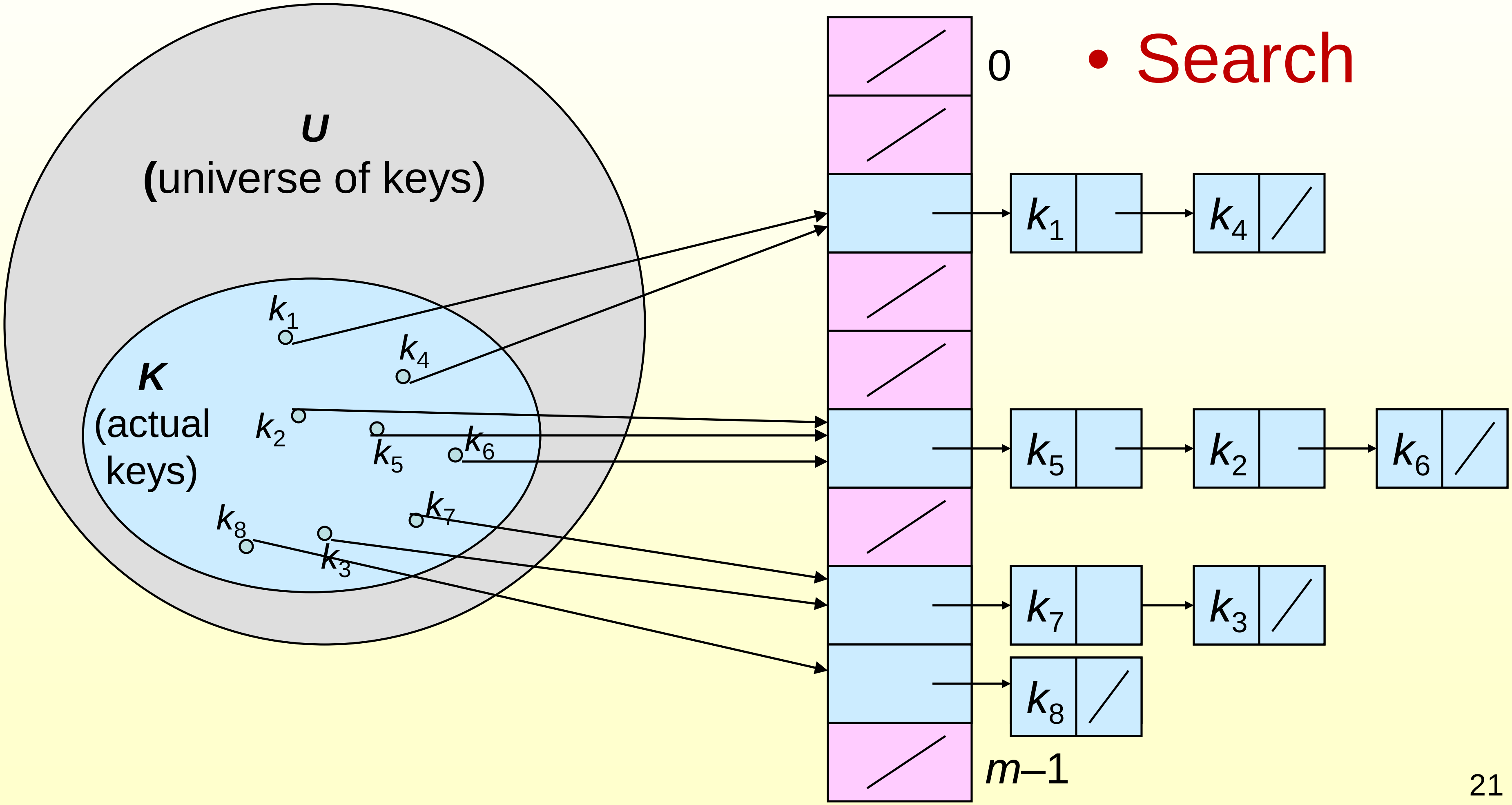
Hashing with Chaining

Dictionary Operations:

- Chained-Hash-Insert (*T*, *x*)
  - Insert *x* at the head of list *T*[*h*(*key*[*x*])].
  - Worst-case complexity – *O*(1).
- Chained-Hash-Delete (*T*, *x*)
  - Delete *x* from the list *T*[*h*(*key*[*x*])].
  - Worst-case complexity – proportional to length of list.
- Chained-Hash-Search (*T*, *k*)
  - Search an element with key *k* in list *T*[*h*(*k*)].
  - Worst-case complexity – proportional to length of list.

Operations in Chained Hash Table

- Insert
- Delete
- Search



Singly or doubly linked?

Analysis on Chained-Hash-Search

- Load factor $\alpha=n/m$ = average keys per slot.
  - m – number of slots (size of the table).
  - n – number of elements stored in the hash table.
- Worst-case complexity: $\Theta(n)$ + time to compute $h(k)$, resolve collisions
- Average depends on how h distributes keys among m slots.
- **Assume**
  - *Simple uniform hashing*.
    - Any key is equally likely to hash into any of the m slots, independent of where any other key hashes to.
  - $O(1)$ time to compute $h(k)$.
- Time to search for an element with key k is $\Theta(|T[h(k)]|)$.
- Expected length of a linked list = load factor = α = n/m.

Expected Cost of an Unsuccessful Search

Theorem:
An unsuccessful search takes expected time $\Theta(1+\alpha)$.

Proof:

- Any key not already in the table is equally likely to hash to any of the m slots.
- To search unsuccessfully for any key k, need to search to the end of the list $T[h(k)]$, whose expected length is α.
- Adding the time to compute the hash function, the total time required is $\Theta(1+\alpha)$.

Expected Cost of a Successful Search

Theorem:
A successful search takes expected time $\Theta(1+\alpha)$.

Assume each element present is equally likely to be searched for

Proof:

- The probability that a list is searched is proportional to the number of elements it contains.
- The number of elements examined during a successful search for an element *x* is 1 more than the number of elements that appear before *x* in *x*'s list.
  - These are the elements inserted ***after x*** was inserted.
- Goal:
  - Find the average, over the *n* elements *x* in the table, of how many elements were inserted into *x*'s list after *x* was inserted.

Expected Cost of a Successful Search

Theorem:
A successful search takes expected time Θ(1+α).

Proof (contd):

- Let x_i be the i^{th} element inserted into the table, and let $k_i = key[x_i]$.
- Define indicator random variables $X_{ij} = I\{h(k_i) = h(k_j)\}$, for all i, j.
- Simple uniform hashing ⇒ $\Pr\{h(k_i) = h(k_j)\} = 1/m$

  ⇒ $E[X_{ij}] = 1/m$.
- Expected number of elements examined in a successful search is:

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

No. of elements inserted after x_i into the same slot as x_i.

Proof – Contd.

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

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

(linearity of expectation)

$$=\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right)$$

$$=1+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right)$$

$$=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)$$

$$=1+\frac{n-1}{2m}$$

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

Expected total time for a successful search = Time to compute hash function + Time to search

= ***O*(2+α/2 – α/2*n*) = *O*(1+ α)**.

Expected Cost – Interpretation

- If $n = O(m)$, then $\alpha = n/m = O(m)/m = O(1)$.

$\Rightarrow$ Searching takes constant time on average.

- Insertion is $O(1)$ on the average.
- Deletion also takes $O(1)$ on the average.

- Hence, all dictionary operations take $O(1)$ time on average with hash tables with chaining.

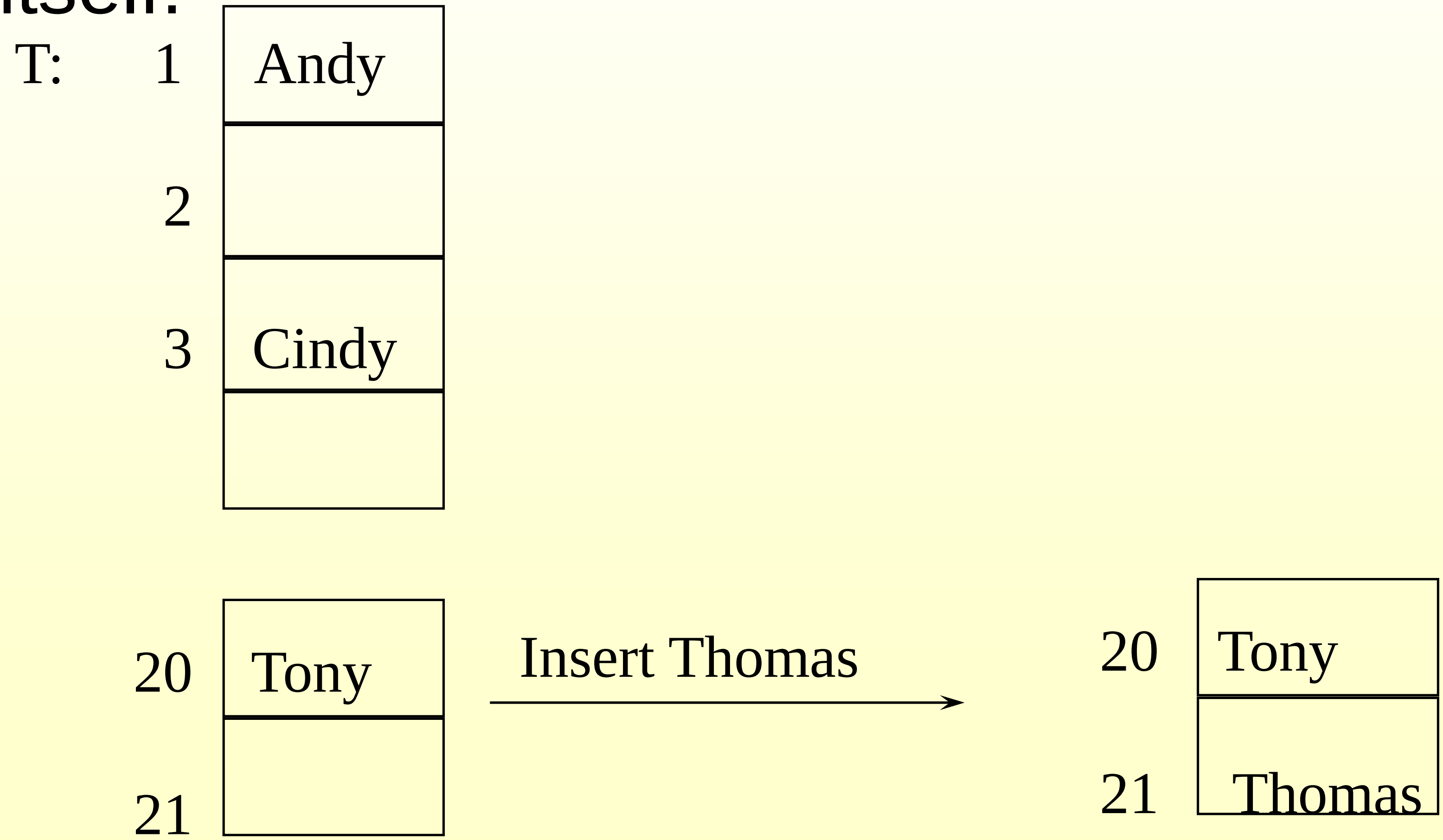
- But they are all $\Theta(n)$ in the worst-case.

Re-Hashing

- If n starts to become comparable to m, then we need another strategy.
- To grow: Whenever $\alpha \geq$ some threshold (e.g. 3/4), double the number of slots (double m).
- Requires rehashing everything into the new table—but by this cost can me amortized over the subsequent operations (wait till the amortized analysis lecture)

Collision Resolution by Open Addressing

- Store colliders in the hash table array itself:



T: 1 Andy

2

3 Cindy

20 Tony

21

Insert Thomas

20 Tony

21 Thomas

Collision Resolution by Open Addressing

- Advantages:

– No extra storage for lists

- Disadvantages:

– Harder to program, especially deletion

– Harder to analyze

– Table can overflow

– Performance is worse

Table Probing Strategies

- When there is a collision, where should the new item go?

- Many answers. It is crucial to put the key in a pace where we can find it later when we come looking for it.

- We generate a table probing sequence

Open Addressing Hashing Algorithms

```
INSERT(T, x)   ▷ in this version, we don't check
                 for duplicates
p ← the first probe
while T[p] is not empty do  ▷ assumes T is not full
    p ← the next probe
T[p] ← x
```

Search

```
SEARCH(T,k)
p ← the first probe
while T[p] is not empty do        ▷ again, assumes T is not full
   if T[p] is empty then
      return NIL
   else if key[T[p]] = k then
      return T[p]
else
     p ← next probe
```

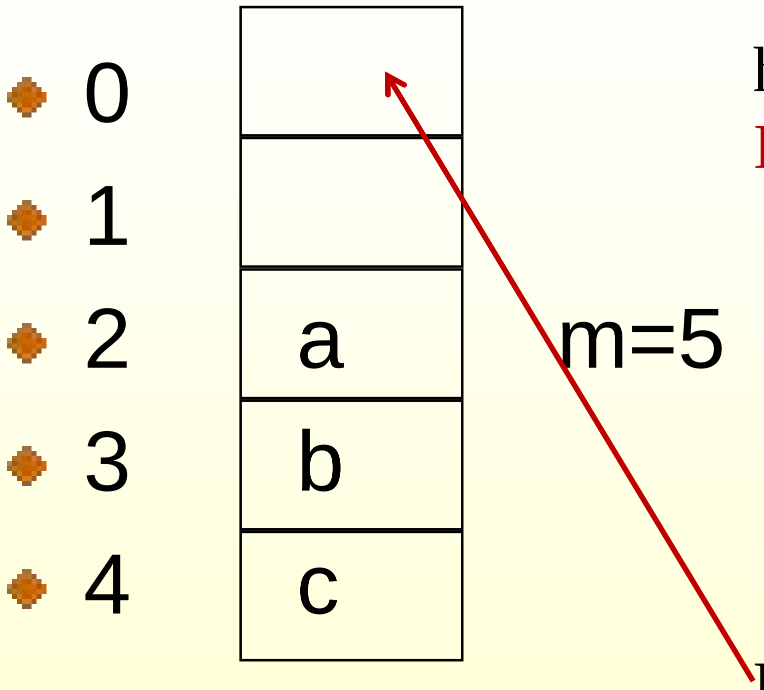
DELETE … is best avoided with open address hashing

Linear Probing

Linear probing: if a slot is occupied, just go to the next slot in the table. (Wrap around at the end.)

$$h(\underset{\text{key}}{k}, \underset{\text{probe \#}}{i}) = (\underset{\substack{\text{our original}\\ \text{hash function}}}{h'}(k) + i) \bmod \underset{\text{\# of slots in table}}{m}$$

Example of Linear Probing

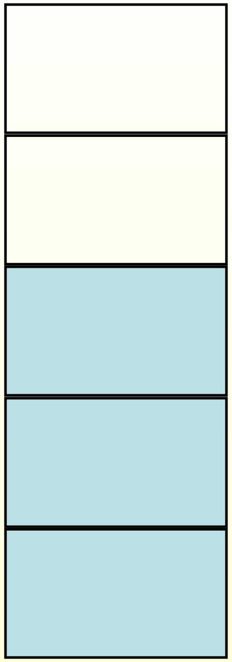


h(k,i) = (h'(k)+i) mod m

INSERT(d). h'(d) =3

i	h'(d,i)
0	3
1	4
2	0

Put d in slot 0



Problem: long runs of items tend to build up, slowing down the subsequent operations. (*primary clustering*)

Quadratic Probing

$$h(k,i) = (h'(k) + \underset{\uparrow}{c_1} i + \underset{\uparrow}{c_2} i^2) \bmod m$$

two constants, fixed at “compile-time”

Better than linear probing, but still leads to clustering, because keys with the same value for h’ have the same probe sequence.

Double Hashing

- Use one hash function to start, and a second to pick the probe sequence:

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

$h_2(k)$ must be relatively prime in m in order to sweep out all slots. E.g. pick m a power of 2 and make $h_2(k)$ always odd.

Uniform Hashing

- Use a sequence of *m* independent hash functions $h_1(k), h_2(k), \ldots h_m(k)$ to probe the table, with each table position equally likely to be chosen.

[Strictly speaking, each of the $m!$ permutations of table entries should be equally likely.]

Linear and quadratic probing give us m probe sequences, because each value $h'(k)$ results in a different, fixed sequence:

$h'(k) = 3 \rightarrow 3\ 4\ 5\ \ldots$ $\quad$ ($h'(k)$ has values from 0 to m-1)

$h'(k) = 8 \rightarrow\ 8\ 9\ 10$

Double hashing gives about m^2 sequences, because every pair $(h_1(k), h_2(k))$ yields a different probe sequence.

The analysis assumes *uniform hashing*, which holds that all of the $m!$ possible probe sequences are equally likely.

Though $m! >> m^2$, in practice double hashing has performance close to uniform hashing.

Analysis

Analysis of open address hashing (assuming uniform hashing model, n independent hash functions):

Recall load factor: $\alpha = \frac{\#\text{ of keys}}{\#\text{ of slots}}.$

Here $0 \le \alpha \le 1$. (with chaining, α can be >1.)

Time for unsuccessful search: we count probes.

worst case = n (you hit every key before you hit a blank slot)

avg case: assume a very large table.

Probability of doing a first probe: 1

Prob of 2nd probe = prob that 1st is occupied $\approx \alpha$

$$\text{Prob of 3rd probe} = (\text{prob of 2nd probe}) \times (\text{prob. 2nd is occ.}) \approx \alpha\alpha = \alpha^2$$

Analysis

$$\text{Expected \# of probes} = 1 + \alpha + \alpha^2 + \ldots$$

$$< \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1-\alpha}$$

Asymptotic Behavior

open address hashing, unsuccessful search: $\frac{1}{1\text{-}\alpha}$

chain hashing unsuccessful search: $1+\alpha$

Which is better?

Note: $\frac{1}{1\text{-}\alpha} = 1 + \frac{\alpha}{1\text{-}\alpha}$

When is $\dfrac{\alpha}{1-\alpha} < a$? When $0 \le \alpha \le 1$, $\dfrac{\alpha}{1-\alpha}$ is always > α.

It's only less when $\alpha > 1$ - but this cannot happen in open address hashing!

So chain hashing always wins an unsuccessful search.

Successful search: # of probes in open address hashing is at most

$\frac{1}{\alpha} \ln \frac{1}{1-\alpha}$. This is < 4 for $\alpha < 90\%$.

A successful search is like an "average" unsuccessful search

(you find something that was searched for earlier, not found, and inserted).

Asymptotic Behavior

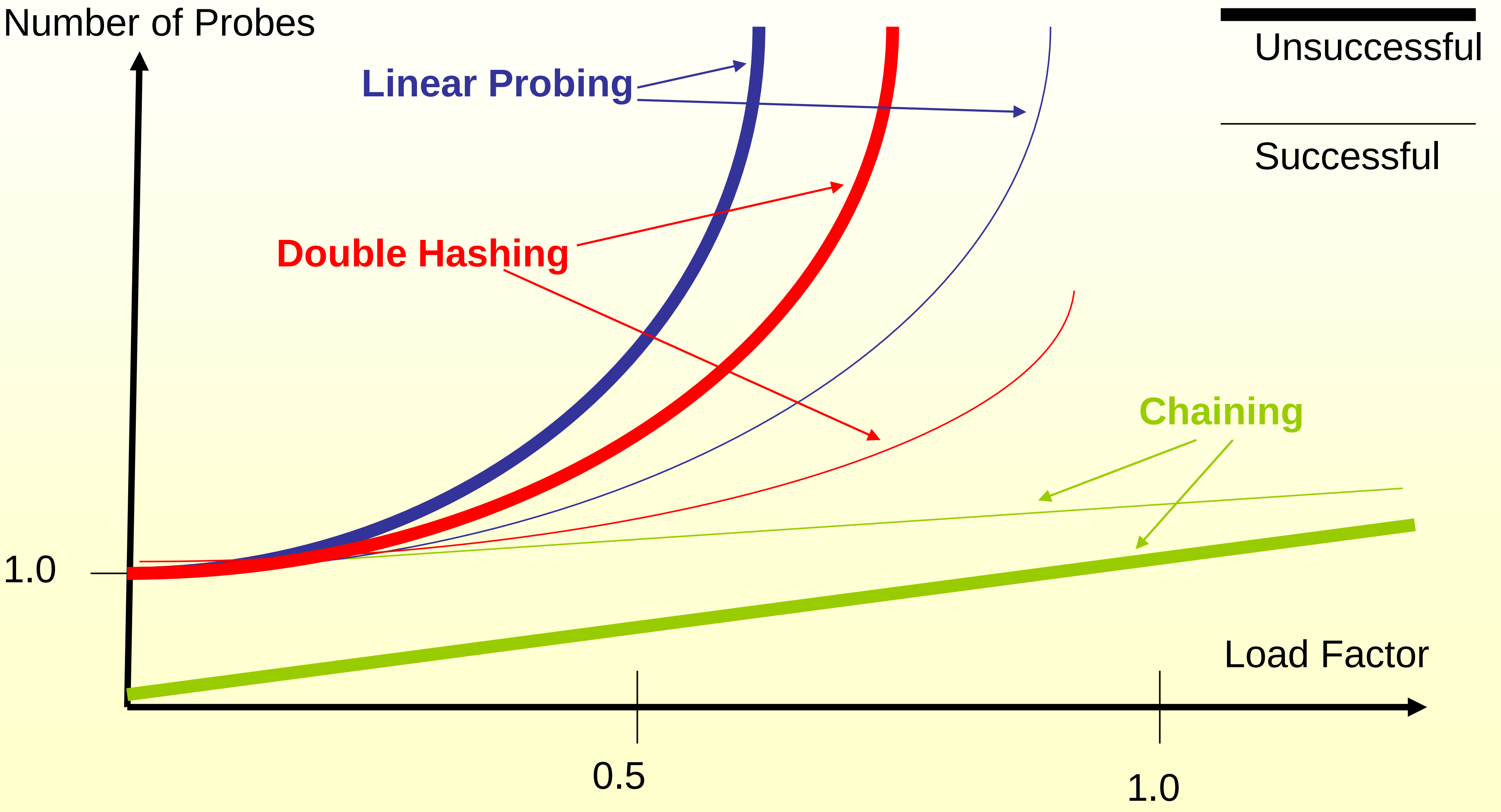
- If k was the $(i+1)$ –st key inserted , the expected number of probes is the search for it is at most $1/(1-i/m) = m/(m-i)$.

$$\frac{1}{n}\sum_{i=0}^{n-1} m/(m-i) = \frac{m}{n}\sum_{i=0}^{n-1} 1/(m-i) =$$

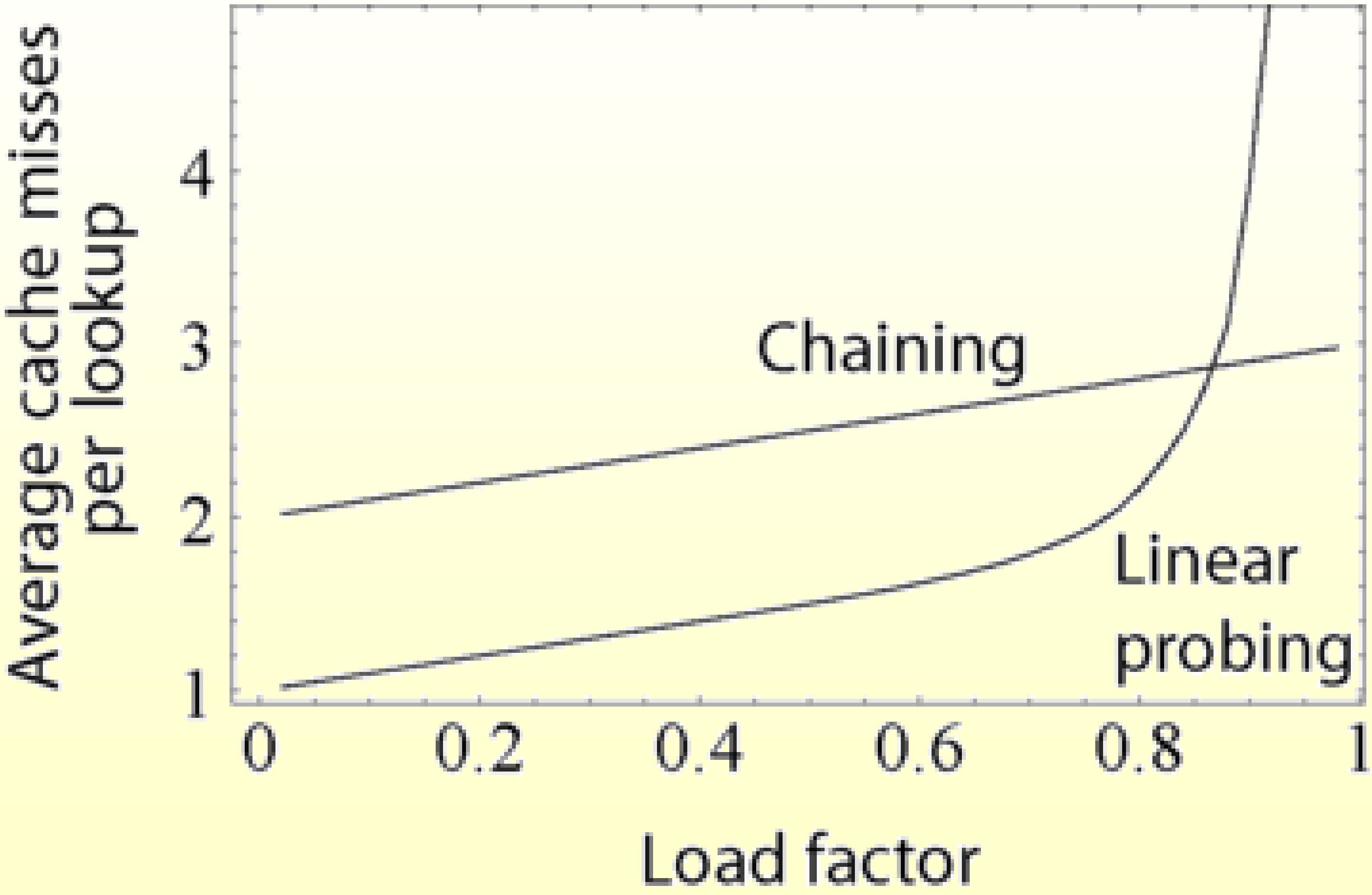
$$1/\alpha \sum_{k=m-n+1}^{m} 1/k < \frac{1}{\alpha}\int_{m-n}^{m} \frac{1}{x}\,dx = 1/\alpha \ln\frac{m}{m-n} =$$

$$1/\alpha \ln\frac{1}{1-\alpha}$$

Expected Number of Probes vs. Load Factor



Open Addressing vs. Chaining with Cashes



Choosing a Good Hash Function

- It should run quickly and distribute the keys up — each key should be equally likely to fit in any slot.

- General rules:

–Exploit known facts about the keys

–Try to use all bits of the key

Choosing A Good Hash Function (Continued)

- Although most commonly strings are being hashed, we'll assume *k* is an integer.
- Can always interpret strings (byte sequences) as numbers in base 256:

$$"cat" = 'c' \times 256^2 + 'a' \times 256 + 't'$$

The Division Method

- $h(k) = k \bmod m$
  - In words: hash k into a table with m slots using the slot given by the remainder of k divided by m
- Elements with adjacent keys hashed to different slots: good
- If keys bear relation to m: bad
  - E.g. if you're hashing decimal integers, then m= a power of ten means you're just taking the low-order digits.
  - If you're hashing strings, then $m = 256$ means you only use the last character.

- Upshot: pick table size m = prime number not too close to a power of 2 (or 10)

The Multiplication Method

- For a constant A, $0 < A < 1$:
- $h(k) = \lfloor m \underbrace{(kA - \lfloor kA \rfloor)}_{\text{Fractional part of } kA} \rfloor$
- Upshot:
  - Choose $m = 2^P$
  - Choose A not too close to 0 or 1
  - Knuth: Good choice for $A = (\sqrt{5} - 1)/2$

Hash Functions in Practice

- Almost all hashing is done on strings. Typically, one computes byte-by-byte on the string to get a non-negative integer, then takes that mod *m*.
- E.g. (sum of all the bytes) mod *m*.
- Problem: anagrams hash to the same value.
- Other ideas: xor, etc.
- Hash function in Microsoft Visual C++ class library:

```
$x = 0$
for $i \leftarrow 1$ to length[s] do
    $x \leftarrow 33x + \text{int}(s[i])$
```

Choosing A Hash Function

- Choosing the hash function well is crucial
  - Bad hash function puts all elements in same slot
  - A good hash function:
    - Should distribute keys uniformly into slots
    - Should not depend on patterns in the data
- We discussed two methods:
  - Division method
  - Multiplication method
- One more in worth mentioning
  - Universal hashing

Universal Hashing

- When attempting to foil a malicious adversary, randomize the algorithm
- *Universal hashing*: pick a hash function randomly when the algorithm begins (*not* upon every insert!)
  - Guarantees good performance on average, no matter what keys adversary chooses
  - Need a family of hash functions to choose from

Universal Hashing

- Let ς be a (finite) collection of hash functions
  - …that map a given universe *U* of keys…
  - …into the range {0, 1, …, *m* - 1}.
- ς is said to be *universal* if:
  - for each pair of distinct keys $x, y \in U$, the number of hash functions $h \in \varsigma$ for which $h(x) = h(y)$ is $|\varsigma|/m$
  - In other words:
    - With a random hash function from ς, the chance of a collision between *x* and *y* is exactly $1/m$ $(x \neq y)$

Universal Hashing

- Theorem 11.3:
  - Choose *h* from a universal family of hash functions
  - Hash *n* keys into a table of *m* slots, $n \le m$
  - Then the expected number of collisions involving a particular key *x* is less than 1
  - Proof:
    - For each pair of keys *y*, *z*, let $c_{yz} = 1$ if *y* and *z* collide, 0 otherwise
    - $E[c_{yz}] = 1/m$ (by definition)
    - Let C_x be total number of collisions involving key *x*
    - $E[C_x] = \sum_{\substack{y \in T \\ y \neq x}} E[c_{xy}] = \frac{n-1}{m}$
    - Since $n \le m$, we have $E[C_x] < 1$

A Universal Hash Function Family

- Choose table size m to be prime
- Decompose key x into r+1 bytes, so that $x = \{x_0, x_1, \ldots, x_r\}$
  - Only requirement is that max value of byte < m
  - Let $a = \{a_0, a_1, \ldots, a_r\}$ denote a sequence of r+1 elements chosen randomly from $\{0, 1, \ldots, m - 1\}$
  - Define corresponding hash function $h_a \in \varsigma$:

$$h_a(x) = \sum_{i=0}^{r} a_i x_i \bmod m$$

  - With this definition, ς has m^{r+1} members

A Universal Hash Function Family

- ς is a universal collection of hash functions (CLRS Theorem 11.4)
- How to use:
  - Pick *r* based on *m* and the range of keys in *U*
  - Pick a hash function by (randomly) picking the *a*’s
  - Use that hash function on all keys