

DATA STRUCTURES & ALGORITHMS

Lecture 5: Hash Table

Lecturer: Dr. Nguyen Hai Minh

CONTENT

- The ADT Dictionary
- Hash tables
 - Direct-address Tables
 - Hash Table
 - Hash Functions
- More Reading

THE ADT DICTIONARY

6/21/2023

nhminh@FIT

Introduction

- *Lecture 2 – Sorting:* the notion of a **sort key**.
- *Lecture 3 – Searching:* the notion of a **search key**.
- Applications that require **value-oriented** operations are **extremely prevalent**.
- Example: the tasks involve values instead of positions.
 - Find the phone number of John Smith
 - Delete all information about the employee with ID number 12908.

Dictionary: A value-oriented ADT

The ADT Dictionary – Example

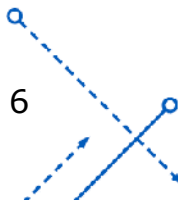
- Data of some major cities in the world in alphabetical order.
- How to get the population of London?
→ Binary Search
- How to find out which cities are in Spain?
→ Sequential Search

<u>City</u>	<u>Country</u>	<u>Population</u>
Buenos Aires	Argentina	13,170,000
Cairo	Egypt	14,450,000
Cape Town	South Africa	3,092,000
London	England	12,875,000
Madrid	Spain	4,072,000
Mexico City	Mexico	20,450,000
Mumbai	India	19,200,000
New York City	U.S.A.	19,750,000
Paris	France	9,638,000
Sydney	Australia	3,665,000
Tokyo	Japan	32,450,000
Toronto	Canada	4,657,000

The ADT Dictionary

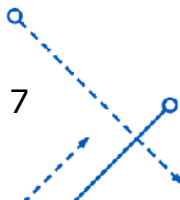
The **ADT dictionary**, or **map**, or **table** allows us to look up information easily based on a specified **search key**.

- For simplicity, we assume that all items in the dictionary have **distinct** search keys.
- Example:
 - In Dictionary of Cities, City is designed as the search key



ADT Dictionary Data & Operations

- **Data:** A finite number of objects, each associated with a search key.
- **Operations:**
 1. Check whether a dictionary is empty
 2. Get the number of items in a dictionary
 3. Insert a new item into a dictionary
 4. Remove the item with a given search key from a dictionary
 5. Remove all items from a dictionary
 6. Search for an item with a given search key from a dictionary
 7. Traverse the items in a dictionary in sorted search-key order



Possible Implementations

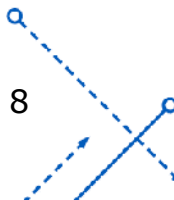
□ Linear implementations:

- Sorted (by search key), array-based
- Sorted (by search key), link-list-based
- Unsorted, array-based
- Unsorted, link-list-based

□ Non-linear implementations:

- Binary Search Tree (Lecture 6)

Which implementation should you choose?



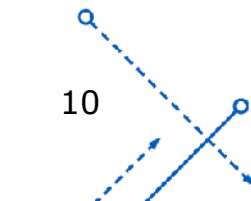
Which implementation should be chosen?

- Consider several factors:
 - What operations are needed for your application?
 - How often the application will perform each operation?
- Example: in dictionary of cities, many more retrieval operations than additions or removals
- The average-case of the ADT dictionary operations:

	Insertion	Removal	Retrieval	Traversal
Unsorted array-based	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Unsorted link-based	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Sorted array-based	$O(n)$	$O(n)$	$O(\log n)$	$O(n)$
Sorted link-based	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Which implementation should be chosen?

- New data structure to implement the ADT dictionary:
 - Hash Table



The background of the slide is a solid blue color. Overlaid on this background is a complex, abstract pattern of white lines and dots. The pattern consists of several intersecting straight lines, some solid and some dashed, creating a grid-like structure. Scattered throughout this grid are numerous small white circles, some of which are connected by dashed lines, suggesting a network or a data structure. The overall effect is a technical, geometric aesthetic.

HASH TABLES

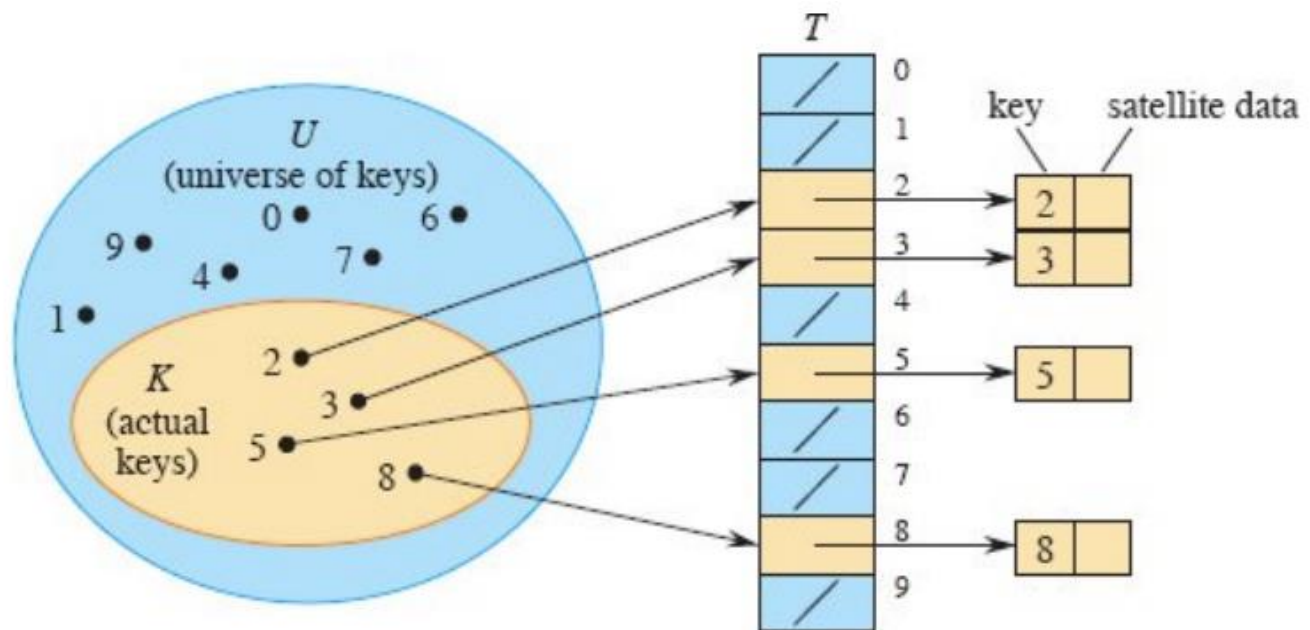
Direct-Address Table

Hash Tables Definition

Hash Functions

Direct-address table

- Generalizes a simpler notion of an ordinary array:
 - **Directly addressing** into an array takes advantage of the **$O(1)$** access time for any array element.
- A dynamic set implemented by a direct-address table T



Direct-address table

□ **IDEA:** Suppose that the keys are drawn from the set $U \in \{0, 1, \dots, m - 1\}$, and keys are distinct. Set up an array $T[0, 1, \dots, m - 1]$ so that:

$$T[k] = \begin{cases} x & \text{if } x \in K \text{ and } \text{key}[x] = k, \\ \text{NIL} & \text{otherwise.} \end{cases}$$

□ Then, all operations take **$O(1)$** time:

```
DIRECT-ADDRESS-SEARCH(T, k)
```

```
    return T[k]
```

```
DIRECT-ADDRESS-INSERT(T, x)
```

```
    T[x.key] = x
```

```
DIRECT-ADDRESS-DELETE(T, x)
```

```
    T[x.key] = NIL
```

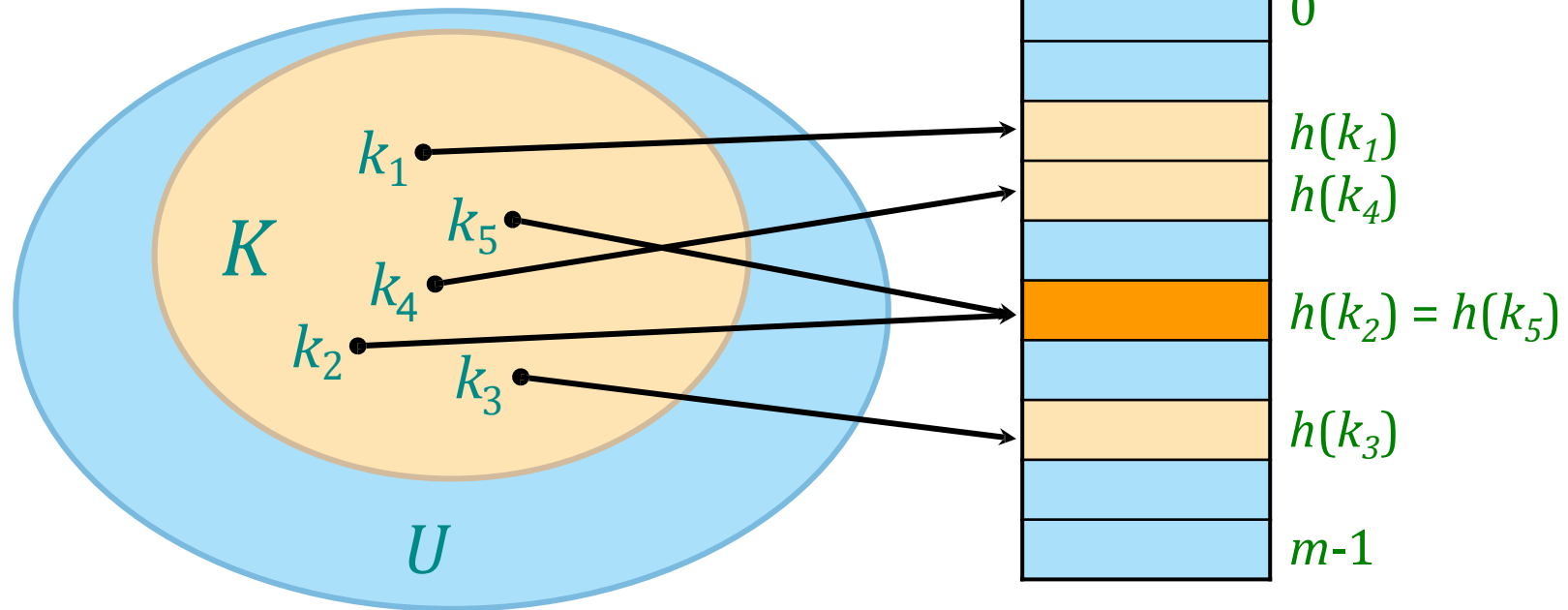
Direct-access table

- To use direct addressing, you must be able to allocate an array that contains a position for every possible key.
- **Problem:**
 - The range of keys can be large:
 - 64-bit numbers (which represent 18,446,744,073,709,551,616 different keys),
 - character strings (even larger!).
 - When K is much smaller than U , most of the space allocated for T would be wasted.



Solution: Hash tables

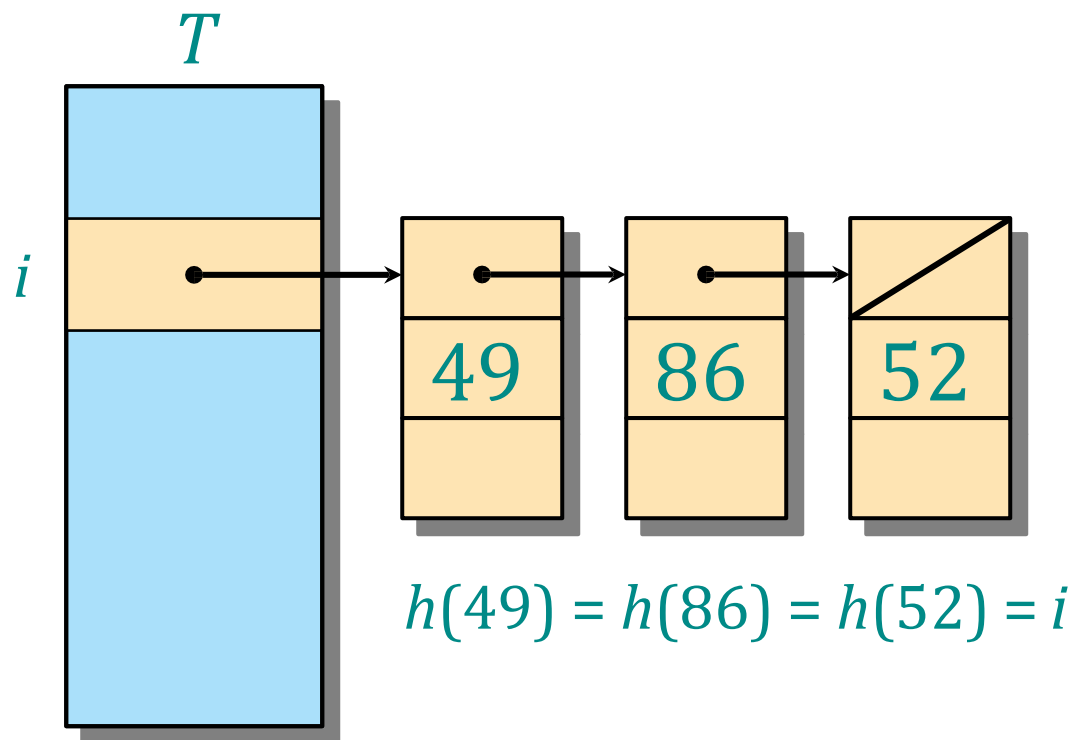
□ **Solution:** Use a **hash function** h to map the universe U of all keys into $\{0, 1, \dots, m-1\}$:



□ When a record to be inserted maps to an already occupied slot in T , a **collision** occurs.

Resolving collisions by chaining

- Link records in the same slot into a list.



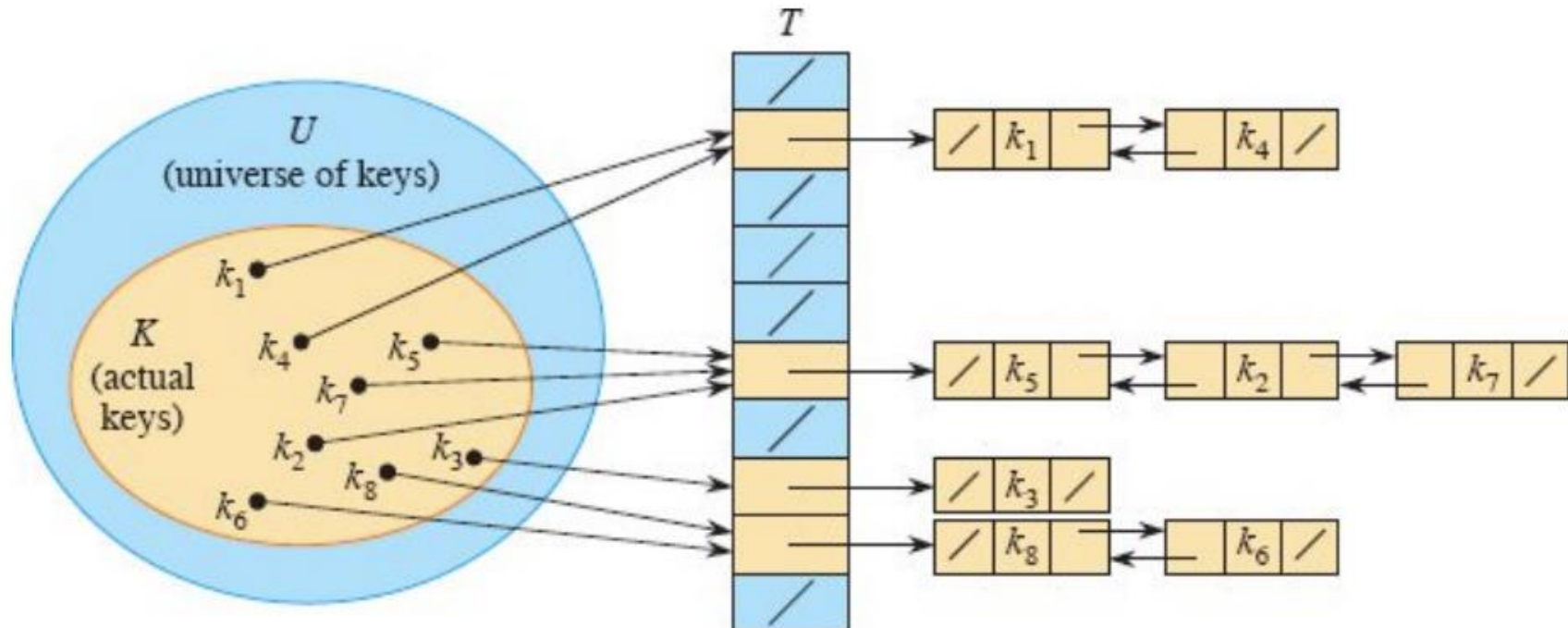
Worst case:

- Every key hashes to the same slot.
- Access time:

$O(n)$

(similar to a linked list)

Resolving collisions by chaining



CHAINED-HASH-INSERT(T, x) $\leftarrow O(1)$

LIST-PREPEND($T[h(x.key)], x$) //add x to the front of linked list

CHAINED-HASH-SEARCH(T, k) $\leftarrow ?$

return LIST-SEARCH($T[h(k)], k$) //find element with key k in list

CHAINED-HASH-DELETE(T, x) $\leftarrow O(1)$

LIST-DELETE($T[h(x.key)], x$) //remove x from a linked list

Analysis of hashing with chaining

- **Average case**: depends on how well the hash function h distributes the set of keys to be stored among the m slots.
- We make the assumption of ***simple uniform hashing***
 - Each key $k \in K$ is equally likely to be hashed to any slot of table T , independent of where other keys are hashed.
 - Let n be the number of keys in the table, and let m be the number of slots. Define the ***load factor*** of T to be

$\alpha = n/m$ = average number of keys per slot.



Search cost

□ The expected time for an **unsuccessful** search for a record with a given key is $O(1 + \alpha)$

*apply hash function
and access slot*

*search the
list*

□ Expected search time: $O(1)$ if $\alpha = O(1)$, or equivalently, if $n = O(m)$.

□ A **successful** search has same asymptotic bound, but a rigorous argument is a little more complicated. (See textbook 1.)

Hash functions

□ What is a good hash function?

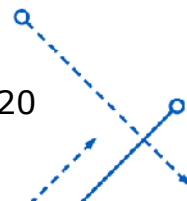
- A good hash function should distribute the keys **uniformly** into the slots of the table.

→ *However, it is difficult to check as we don't know the key distribution.*

- Sometimes, we do know:

- Ex: keys are random real numbers drawn independently and uniformly from $[0,1)$

→ Hash function: $h(k) = \lfloor km \rfloor$ satisfies the simple uniform hashing



Hash functions

□ What is a good hash function?

- In practice, we use heuristics to create hash functions
 - May not satisfy simple uniform hashing, but perform well.
- General idea: **avoid** the hash value to be **dependent on the patterns** that might exist in the key.



Hash functions – Division method

- Assume all keys are integers, and define

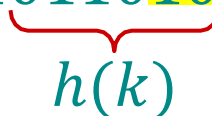
$$h(k) = k \bmod m$$

■ Example: $m = 12, k = 100$ $h(k) = 4$

- **Deficiency:** Don't pick an m that has a small divisor d . A preponderance of keys that are congruent modulo d can adversely affect uniformity.

- **Extreme deficiency:** If $m = 2^r$, then the hash doesn't even depend on all the bits of k :

□ If $k = 10110001110110102$ and $r = 6$, then
 $h(k) = 0110102$.





Hash functions – Division method

- Pick m to be a prime not too close to a power of 2 and not otherwise used prominently in the computing environment.
 - Example: $n = 2000$, we may choose $m = 701$
- **Annoyance:** Sometimes, making the table size a prime is inconvenient.
- However, this method is popular, although the next method we'll see is usually superior.



Hash functions – Multiplication method

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

□ Where:

- $0 < A < 1$
- $(kA \bmod 1)$: fractional part of kA ($kA - \lfloor kA \rfloor$)
- $\lfloor x \rfloor$: floor(x)

□ Advantage:

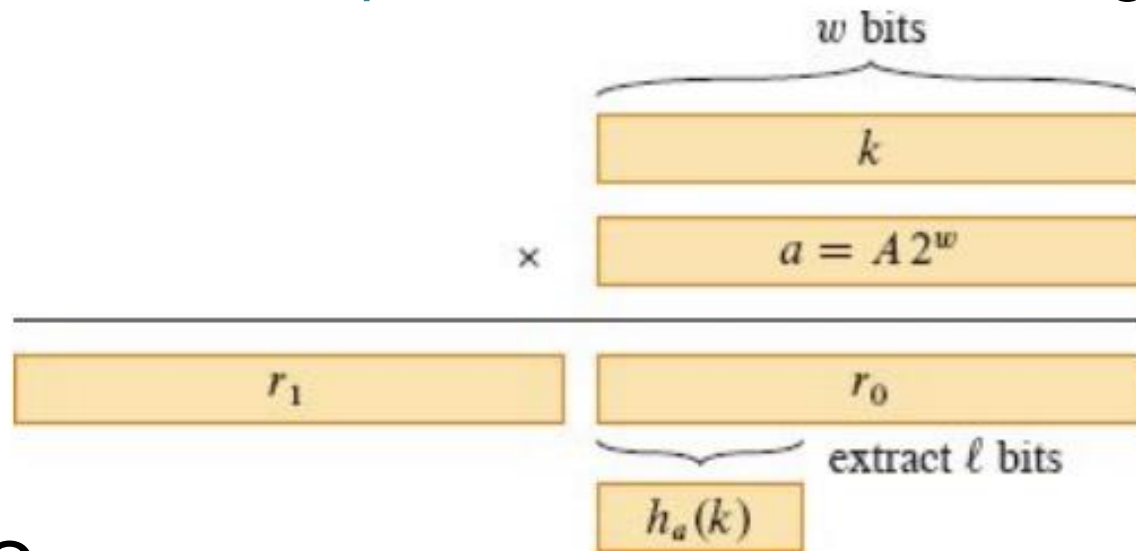
- The value of m is not critical.
- Typically: $m = 2^l$ (for some integer l)

□ Knuth suggests that:

$$A \approx \frac{(\sqrt{5} - 1)}{2} = 0.6180339887 \dots$$

Hash functions – Multiplication method

- Suppose that the word size of the machine is w bits and k fits into a single word.
- Restrict $A = a/2^w$, where a is an integer $0 < a < 2^w$



- In C++:

$$h_a(k) = (ka \bmod 2^w) \gg (w - l);$$

Hash functions – Multiplication method

□ Example:

- $k = 123456, l = 14, m = 2^{14} = 16384, w = 32$
- We choose $a = 2654435769$ which is the integer closet to $Ax232$
- We choose $ka = 327706022297664$
 $= 76300 \cdot 2^{32} + 17612864$
- So, $r_1 = 76300$ and $r_0 = 17612864$
- The 14 most significant bits of r_0 is $h_a(k) = 67$



Resolving collisions by open addressing

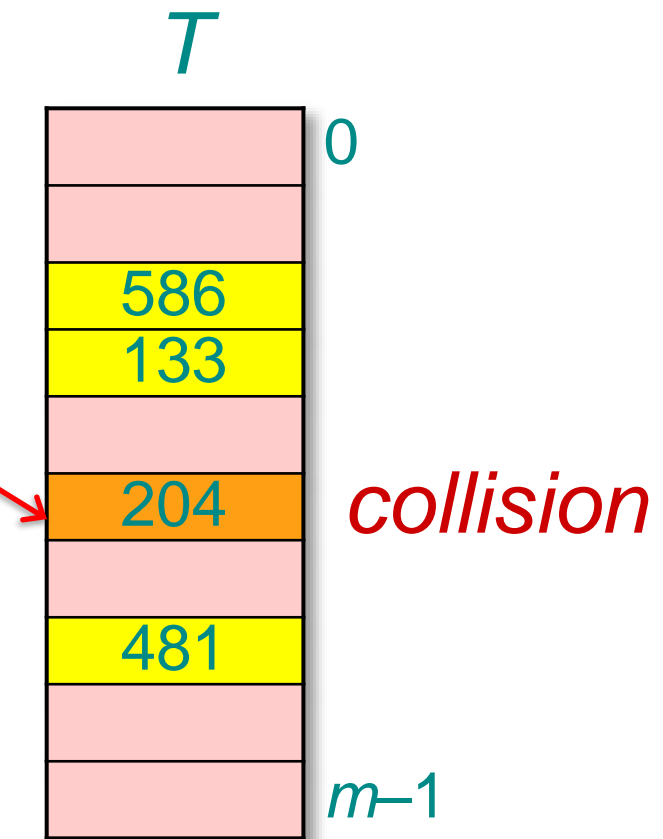
- ❑ No storage is used **outside** of the hash table itself.
- ❑ Insertion **systematically probes** the table until an empty slot is found.
- ❑ The hash function depends on both the key and probe number:
- ❑ $h : U \times \{0, 1, \dots, m-1\} \rightarrow \{0, 1, \dots, m-1\}$.
 - The probe sequence $\langle h(k, 0), h(k, 1), \dots, h(k, m-1) \rangle$ should be a **permutation** of $\{0, 1, \dots, m-1\}$.
 - The table may fill up, and deletion is difficult (but not impossible).



Example of open addressing

Insert key $k = 496$:

0. Probe $h(496, 0)$

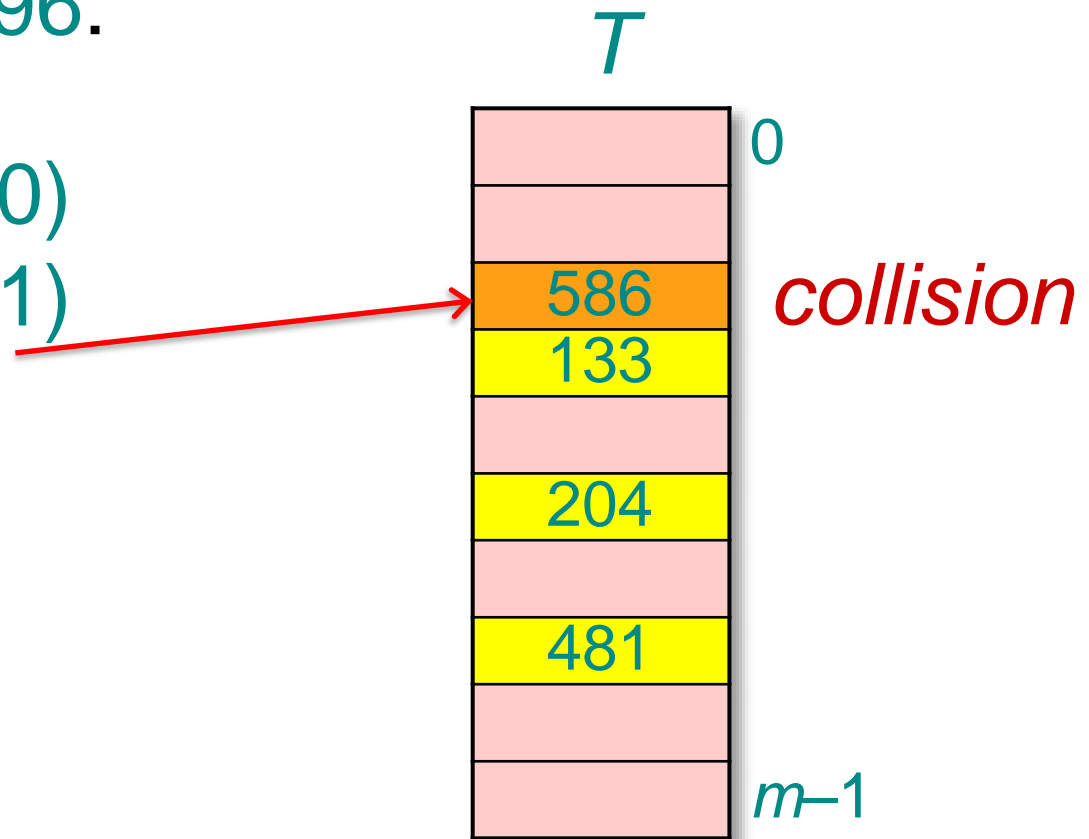


Example of open addressing

Insert key $k = 496$:

0. Probe $h(496, 0)$

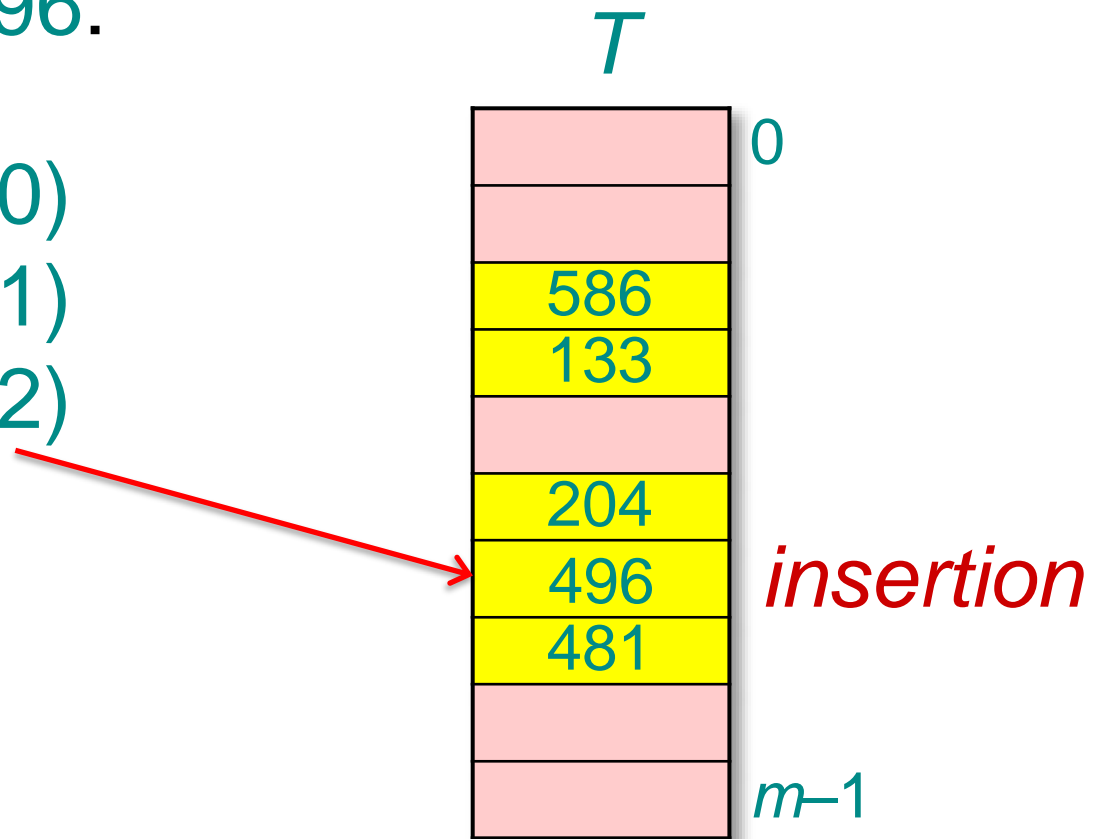
1. Probe $h(496, 1)$



Example of open addressing

Insert key $k = 496$:

0. Probe $h(496, 0)$
1. Probe $h(496, 1)$
2. Probe $h(496, 2)$



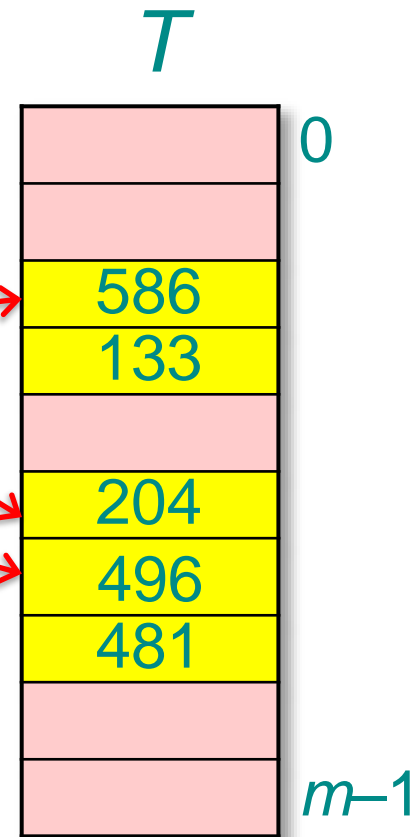
Example of open addressing

Insert key $k = 496$:

0. Probe $h(496, 0)$
1. Probe $h(496, 1)$
2. Probe $h(496, 2)$

Search uses the same probe sequence, terminating successfully if it finds the key

and unsuccessfully if it encounters an empty slot.



Probing strategies

□ Linear probing:

- Given an ordinary hash function $h_1(k)$, linear probing uses the hash function

$$h(k, i) = (h_1(k) + i) \bmod m \text{ for } i = 0, 1, 2, \dots, m - 1$$

- This method, though simple, suffers from ***primary clustering***

- Long runs of occupied slots build up, increasing the average search time.
- Moreover, the long runs of occupied slots tend to get longer.



Probing strategies

□ Quadratic probing:

$$h(k, i) = (h_1(k) + c_1i + c_2i^2) \bmod m$$

- Where $h_1(k)$ is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants.
 - To make full use of the hash table, c_1 , c_2 , and m are constrained.
- This method, works much better than linear probing, but it suffers from **secondary clustering**.
 - 2 keys have the same collision chain if their initial position is the same.



Probing strategies

□ Double hashing:

- Given two ordinary hash functions $h_1(k)$ and $h_2(k)$, double hashing uses the hash function

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

- This method generally produces excellent results, but $h_2(k)$ must be relatively prime to m . One way is to make m a power of 2 and design $h_2(k)$ to produce only odd numbers.



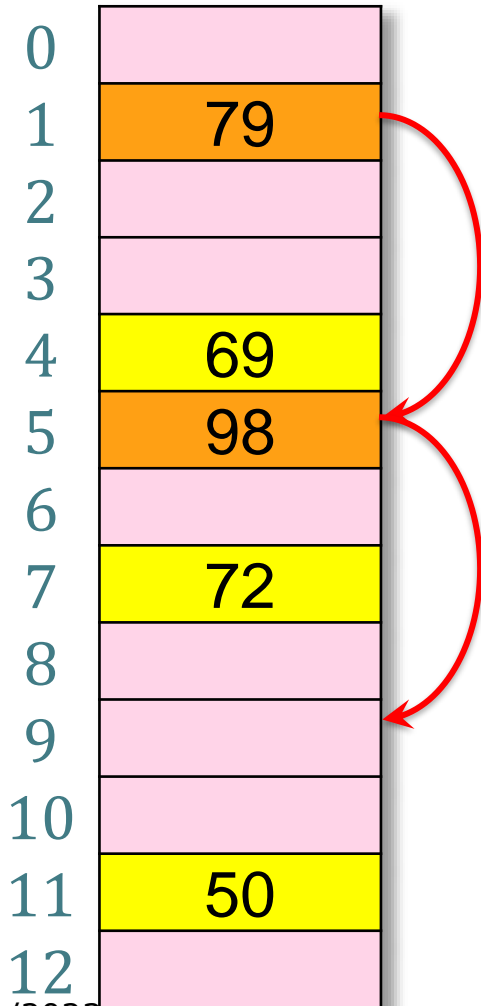
Double hashing – Example

□ Insert 14 to the hash table T by double hashing

■ $m = 13$

■ $h_1(k) = k \bmod 13$

■ $h_2(k) = 1 + (k \bmod 11)$



0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	
10	
11	50
12	

Double hashing – Example

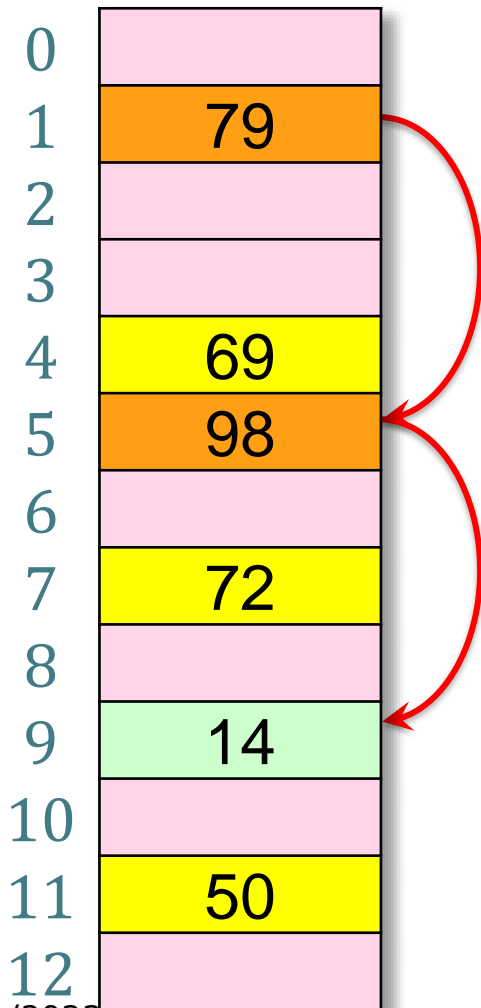
□ Insert 14 to the hash table T by double hashing

■ $m = 13$

■ $h_1(k) = k \bmod 13$

■ $h_2(k) = 1 + (k \bmod 11)$

□ Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot **9**, after slots 1 and 5 are examined and found to be occupied.



0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Probing strategies

□ Double hashing:

■ More examples:

□ $h_1(k) = k \bmod m$; $h_2(k) = 1 + (k \bmod m')$ where m' is slightly less than m (eg, $m-1$):

■ $m = 701$, $m' = 700$, $k = 123456$

■ $h_1(k) = 80$, $h_2(k) = 257$

■ 1st position: 80, next every 257th slot (mod m) is examined



Probing strategies

□ Double hashing:

- $O(m^2)$ probe sequence are used, rather than $O(m)$ (linear/quadratic probing)
 - Each possible $(h_1(k), h_2(k))$ pair yields a distinct probe sequence.
- The performance of double hashing appears to be very close to the performance of the “*ideal*” scheme of uniform hashing.



Analysis of open addressing

- We make the assumption of ***uniform hashing***:
 - Each key is equally likely to have any one of the $m!$ permutations as its probe sequence.
- **Theorem.** Given an open-addressed hash table with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.



Proof of the theorem (1)

- At least one probe is always necessary.
- With probability n/m , the first probe hits an occupied slot, and a second probe is necessary.
- With probability $(n-1)/(m-1)$, the second probe hits an occupied slot, and a third probe is necessary.
- With probability $(n-2)/(m-2)$, the third probe hits an occupied slot, etc.

Observe that $\frac{n-i}{m-i} < \frac{n}{m} = \alpha$ for $i = 1, 2, \dots, n$.



Proof of the theorem (2)

Therefore, the expected number of probes is:

$$\begin{aligned} & 1 + \frac{n}{m} \left(1 + \frac{n-1}{m-1} \left(1 + \frac{n-2}{m-2} \left(\dots \left(1 + \frac{1}{m-n+1} \right) \dots \right) \right) \right) \\ & \leq 1 + \alpha \left(1 + \alpha \left(1 + \alpha \left(\dots \left(1 + \alpha \right) \dots \right) \right) \right) \\ & \leq 1 + \alpha + \alpha^2 + \alpha^3 + \dots \\ & = \sum_{i=0}^{\infty} \alpha^i \\ & = \frac{1}{1-\alpha} \end{aligned}$$

The textbook has a more rigorous proof and an analysis of successful searches.

Implications of the theorem

- If α is constant, then accessing an open-addressed hash table takes constant time.
- If the table is half full, then the expected number of probes is $1/(1-0.5) = 2$.
- If the table is 90% full, then the expected number of probes is $1/(1-0.9) = 10$.



Hash table applications

□ Dictionary problems:

- Problems that usually need only 2 operations:
Insert & Search
- Example:



□ Passwords storing and matching

- Example: MD5 checksum (128 bit – 16 bytes hash value)

The background of the slide is a solid blue color. Overlaid on this background is a complex, abstract pattern of white lines and arrows. The pattern consists of several intersecting straight lines, some of which are dashed and others solid. There are also curved dashed lines and arrows pointing in various directions, creating a sense of movement and connectivity. The overall effect is a technical or mathematical aesthetic.

MORE READING

- Universal hashing
- Perfect hashing

A weakness of hashing

Problem: For any hash function h , a set of keys exists that can cause the average access time of a hash table to skyrocket.

- An adversary can pick all keys from $\{k \in U: h(k) = i\}$ for some slot i .
- Example: $h(k) = k \bmod m$
→ Set of keys: $\{k, k+m, k+2m, \dots\}$



A weakness of hashing

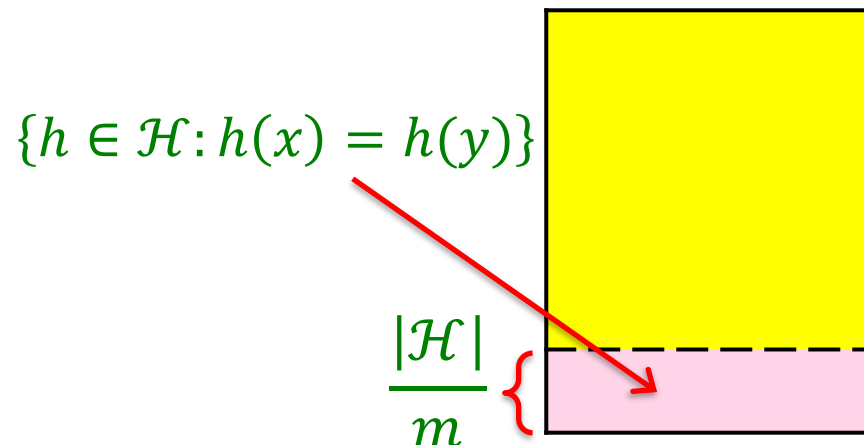
IDEA: Choose the hash function at random, independently of the keys.

- Even if an adversary can see your code, he or she cannot find a bad set of keys, since he or she doesn't know exactly which hash function will be chosen.
- This approach is called ***universal hashing***.

Universal hashing

Definition. Let U be a universe of keys, and let \mathcal{H} be a finite collection of hash functions, each mapping U to $\{0, 1, \dots, m - 1\}$. We say \mathcal{H} is *universal* if for all $x, y \in U$, where $x \neq y$, we have $|\{h \in \mathcal{H} : h(x) = h(y)\}| = |\mathcal{H}|/m$.

That is, the chance of a collision between x and y is $1/m$ if we choose h randomly from \mathcal{H}



Universality is good

Theorem. Let h be a hash function chosen (uniformly) at random from a universal set of hash functions. Suppose h is used to hash n arbitrary keys into the m slots of a table T . Then, for a given key x , we have

$$E[\text{\#collisions with } x] < n/m.$$

Proof of theorem

Proof. Let C_x be the random variable denoting the total number of collisions of keys in T with x , and let

$$c_{xy} = \begin{cases} 1 & \text{if } h(x) = h(y) \\ 0 & \text{otherwise} \end{cases}$$

Note: $E[c_{xy}] = 1/m$ and $C_x = \sum_{y \in T - \{x\}} c_{xy}$

Proof of theorem

$$\begin{aligned} E[C_x] &= E \left[\sum_{y \in T - \{x\}} c_{xy} \right] \\ &= \sum_{y \in T - \{x\}} E[c_{xy}] \\ &= \sum_{y \in T - \{x\}} 1/m \\ &= \frac{n-1}{m} \end{aligned}$$

- Take expectation of both sides.
- Linearity of expectation.
- $E[c_{xy}] = 1/m$.
- Algebra.

Constructing a set of universal hash functions

Choose a prime number p large enough: every key k is in the range 0 to $p - 1$. We have $p > m$.

Let $Z_p = \{0, 1, \dots, p - 1\}$, $Z_p^* = \{1, 2, \dots, p - 1\}$,

Carter and Wegman's strategy:

For any $a \in Z_p^*$ and $b \in Z_p$, define:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m.$$

How big is $\mathcal{H} = \{h_{a,b}\}$?

$$|\mathcal{H}| = p(p - 1)$$

Universality of dot-product hash functions

Theorem. The set $\mathcal{H} = \{h_{a,b}\}$ is universal.

Proof. Suppose that x and y be distinct keys. For a given hash function $h_{a,b}$, we let $r = (ax + b) \bmod p$ and $s = (ay + b) \bmod p$. Then,

$$r - s \equiv a(x - y) \bmod p$$

p is prime, a and $(x - y)$ are both nonzero

$\rightarrow a(x - y) \bmod p \neq 0$.

Therefore, $r \neq s$, which means there are no collisions yet at the “ $\bmod p$ ” level.



Proof (continued)

$$r - s \equiv a(x - y) \pmod{p}$$

We can solve for a and b given r and s :

$$a = ((r - s)((x - y)^{-1} \pmod{p})) \pmod{p}$$

$$b = (r - ax) \pmod{p}$$

$(x - y)^{-1} \pmod{p}$ is unique multiplicative inverse, modulo p , of $(x - y)$.

The number of pairs (r, s) with $r \neq s$: $p(p - 1)$

The number of pairs (a, b) with $a \neq 0$: $p(p - 1)$

Therefore, there is a one-to-one correspondence between pairs (a, b) with $a \neq 0$ and pairs (r, s) with $r \neq s$.

Thus, each of the pair (a, b) picked randomly from $Z_p^* \times Z_p$ yields a different resulting pair (r, s) with $r \neq s$.



Proof (continued)

$$\Pr\{x, y \text{ collide}\} = \Pr\{r \equiv s \bmod m\}$$

For a given value of r , of the $p - 1$ possible values for s , the number of values s such that $s \neq r$ and $s \equiv r \bmod m$ is at most

$$\begin{aligned} \lfloor p/m \rfloor - 1 &\leq ((p + m - 1)/m) - 1 && \text{(inequality)} \\ &= (p - 1)/m \end{aligned}$$

$$\text{Therefore, } \Pr\{r \equiv s \bmod m\} \leq \frac{\frac{(p-1)}{m}}{p-1} = \frac{1}{m}.$$

Thus, for any pair of distinct values $x, y \in Z_p$:

$$\Pr\{h_{a,b}(x) = h_{a,b}(y)\} \leq 1/m$$

so that, \mathcal{H} is indeed universal.

Perfect hashing

- Given a set of n keys, construct a static hash table of size $m = O(n)$ such that **SEARCH** takes $O(1)$ time in the **worst case**.
- **IDEA:** Two-level scheme with universal hashing at both levels.
 - Level 1: Hashing with chaining
 - Level 2: For n_j keys hashing to slot j , use a secondary hash table S_j with an associated hash function h_j .

No collisions at level 2! if $m_j = n_j^2$



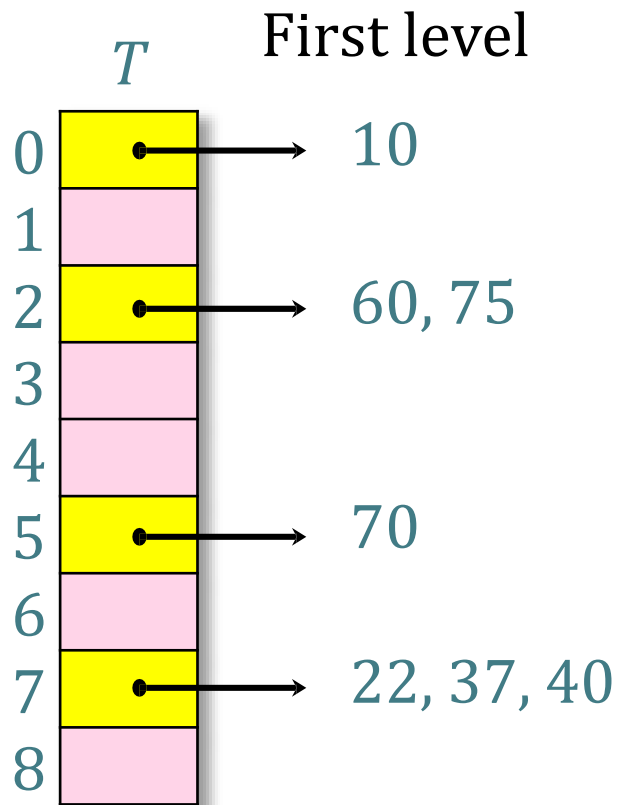
Perfect hashing

- Using perfect hashing to store the set $K=\{10, 22, 37, 40, 60, 70, 75\}$.
- Level 1 hash function is $h(k)=((ak+b) \bmod p) \bmod m$, where $a=3$, $b=42$, $p=101$, $m=9$.
 $\rightarrow \{h(k)\} = \{0, 7, 7, 7, 2, 5, 2\}$
- Level 2 hash function is $h_j(k)=((a_jk+b_j) \bmod p) \bmod m_j$, for each hash table S_j of slot j .
 $\rightarrow m_0=m_5=0^2=1, m_2=2^2=4, m_7=3^2=9$

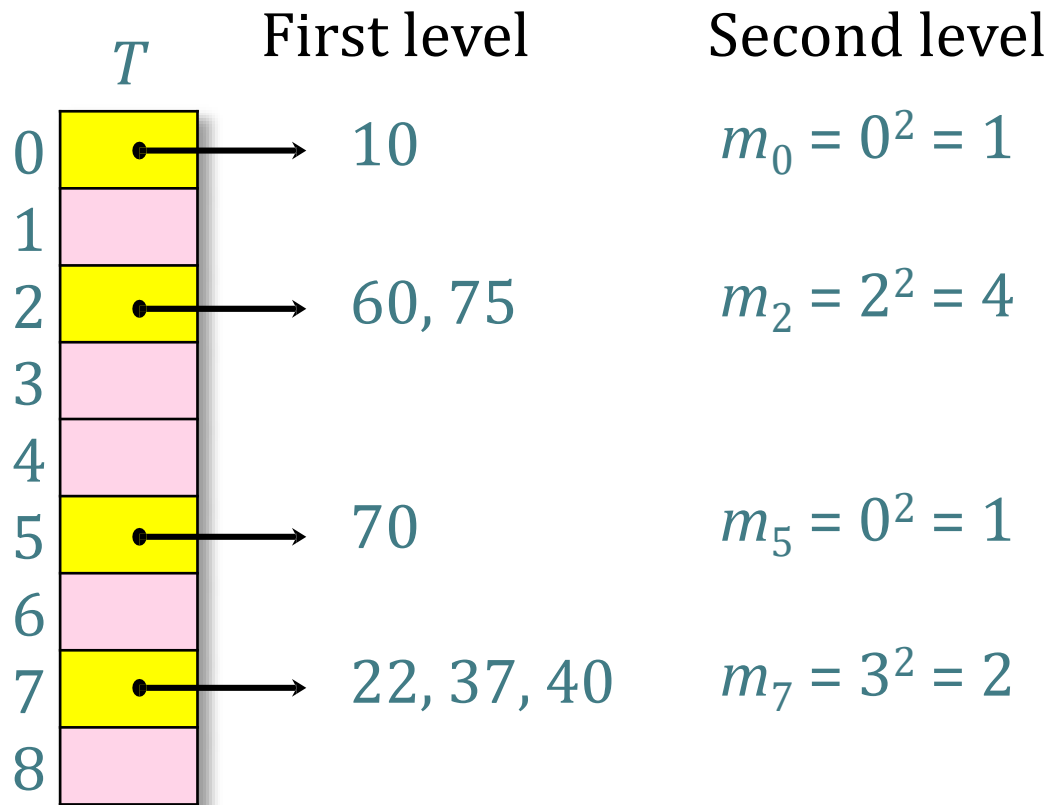
Perfect hashing



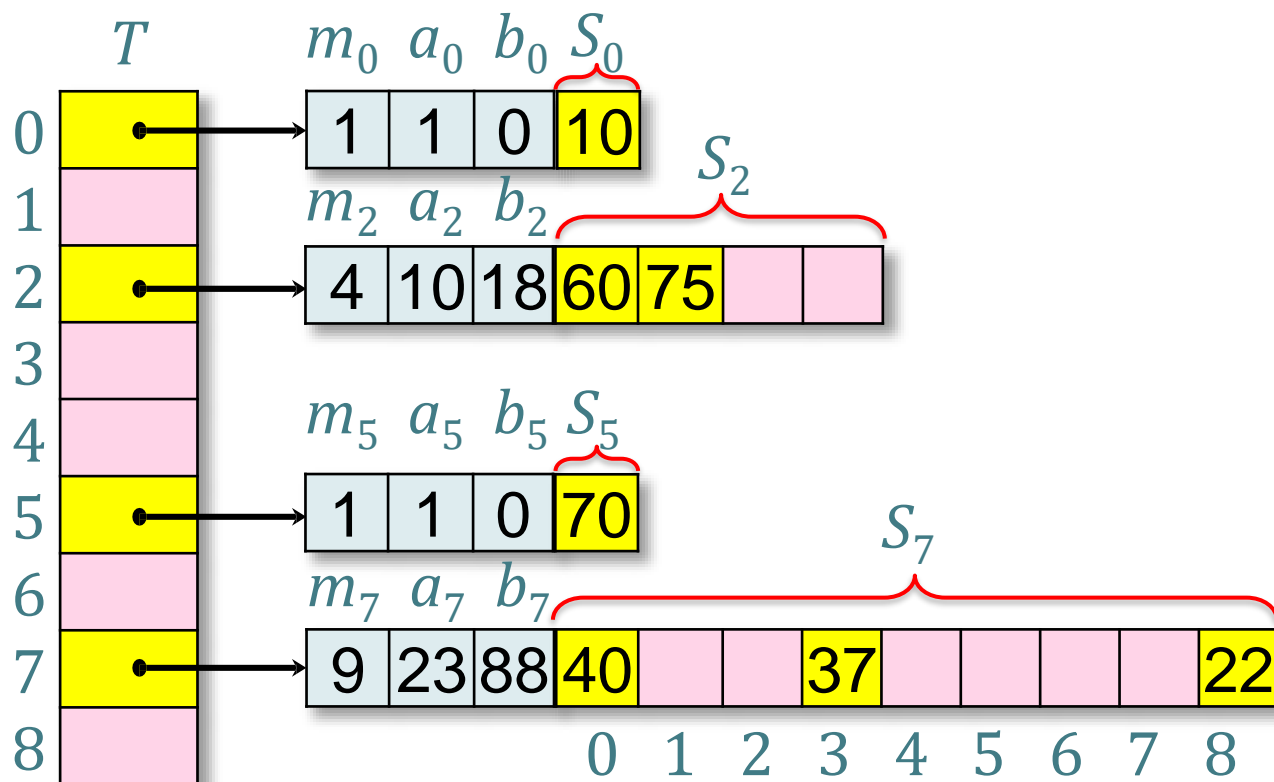
Perfect hashing



Perfect hashing



Perfect hashing



Collisions at level 2

Theorem. Let \mathcal{H} be a class of universal hash functions for a table of size $m = n^2$. Then, if we use a random $h \in \mathcal{H}$ to hash n keys into the table, the probability of there being any collisions is less than $1/2$.

Proof. By the definition of universality, the probability that 2 given keys in the table collide under h is $\frac{1}{m} = \frac{1}{n^2}$.

Since there are pairs $\binom{n}{2}$ of keys that can possibly collide, the expected number of collisions is

$$E[\text{\#collisions}] = \binom{n}{2} \cdot \frac{1}{n^2} = \frac{n(n-1)}{2} \cdot \frac{1}{n^2} < \frac{1}{2}$$

Proof (continued)

Markov's inequality says that for any nonnegative random variable X , we have

$$\Pr\{X \geq t\} \leq E[X]/t$$

Applying this inequality with $t = 1$, we find that the probability of 1 or more collisions is at most $1/2$.

Thus, just by testing random hash functions in \mathcal{H} , we'll quickly find one that works.



Analysis of storage

For the level-1 hash table T , choose $m = n$, and let n_i be random variable for the number of keys that hash to slot i in T . By using n_i^2 slots for the level-2 hash table S_i , the expected total storage required for the two-level scheme is therefore

$$E[\text{total storage}] = n + E\left[\sum_{i=0}^{m-1} O(n_i^2)\right] < n + 2n = O(n)$$

What's next?

□ After today:

- Read textbook 1 – Chapter 11 (page 368~)
- Read textbook 2 – Chapter 18 (page 552~)
- Do Homework 5

□ Next class:

- Individual Assignment 3 (topic: Hash Tables)
- Lecture 6: Tree Data Structures



Q&A