COMP 251

Algorithms & Data Structures (Winter 2021)

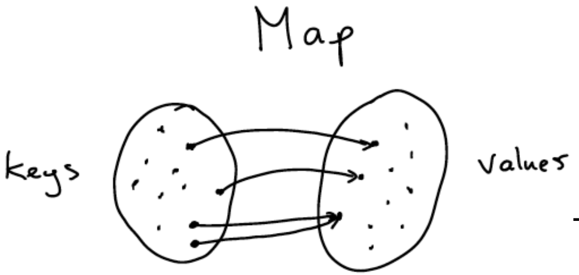
Hashing

School of Computer Science
McGill University

Based on Based on (Cormen *et al.*, 2002) & slides of (Waldispuhl,2020).

Hashing – Problem definition

- A map is a set of (key, value) pairs. Each key maps to at most one value.



Taken from Langer2014

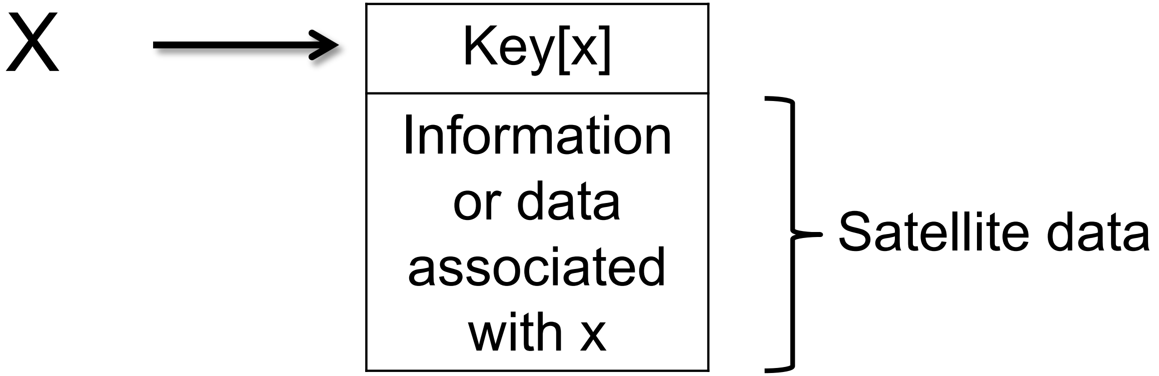
- This is also a map (‘function’ = ‘map’), but we will not be considering continuous functions/maps here.

e.g. $\{ (x, f(x)) \}$

Taken from Langer2014

Hashing – Problem definition

Table S with *m* records *x*:



We want a data structure to store and retrieve these data.

Operations:

- $insert(S,x): S \leftarrow S \cup \{x\}$
- $delete(S,x): S \leftarrow S \setminus \{x\}$
- $search(S,k)$

Dynamic set

Hashing – Real world Examples

- Compilers.
  - a compiler that translates a programming language maintains a symbol table, in which the keys of elements are arbitrary character strings corresponding to identifiers in the language.
- Password Authentication
  - Keys are usernames, and values are passwords (hash(passwords)).
  - http://www.md5.cz/
  - https://crackstation.net/hashing-security.htm
- Data consistency
  - For example, after downloading a file, you can compare the fingerprint of the downloaded file with the fingerprint of the original file. If the two fingerprints differ, the files are surely different, if they are the same, the files are equal with a very high probability.

Hashing – Why using hash

- Hashing is an extremely effective and practical technique.
- Many applications require a dynamic set that supports only the dictionary operations
  - INSERT, SEARCH, and DELETE.
  - A hash table is an effective data structure for implementing dictionaries.
    - Under reasonable assumptions, the average time to search for an element in a hash table is O(1).
- A hash table generalizes the simpler notion of an ordinary array.
  - Directly addressing into an ordinary array makes effective use of our ability to examine an arbitrary position in an array in O(1) time.
    - The array index is computed from the key.
    - To allocate an array that has one position for every possible key.

Direct-address tables – First attempt

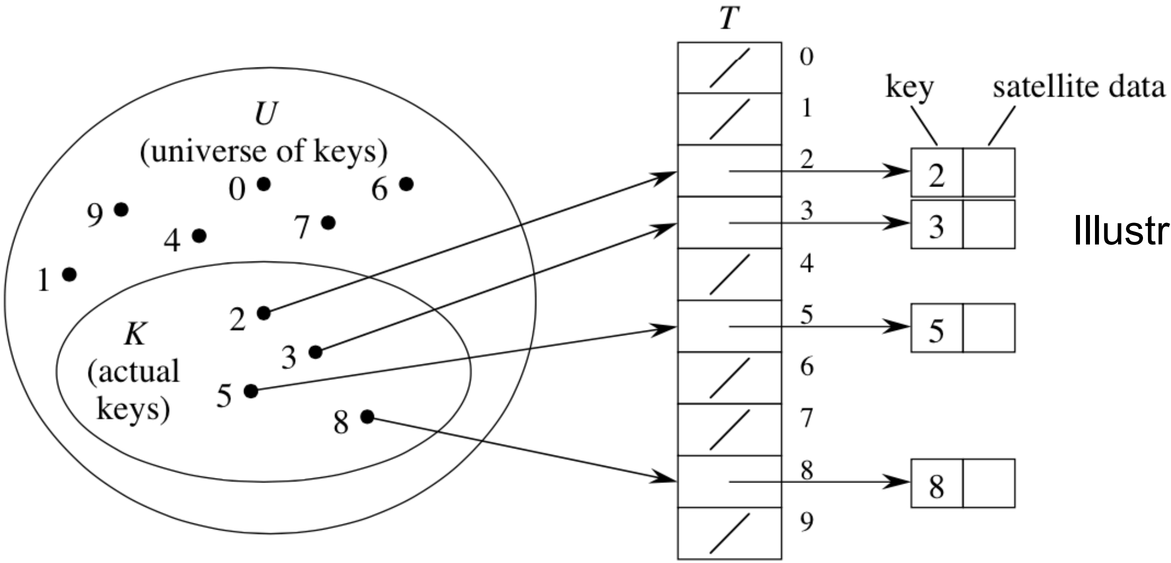


Illustration (CLR, 2005)

- Each slot, or position, corresponds to a key in U .
- If there is an element x with key k, then T[k] contains a pointer to x.
- If T [k] is empty, represented by NIL.
- All operations in O(1), but:
  - if n (#keys) < m (#slots), lot of wasted space.
  - m needs to be not too large (space constraint)

Hashing – Hash map

- Hash map (hash function + hash table).
  - Direct addressing:
    - an element with key k is stored in slot k.
  - Hash function:
    - an element with key k is stored in slot h(k).
      - That is, we use a hash function h to compute the slot from the key k.
      - Hash function: $h : U \rightarrow \{0,1,\ldots,m-1\}$

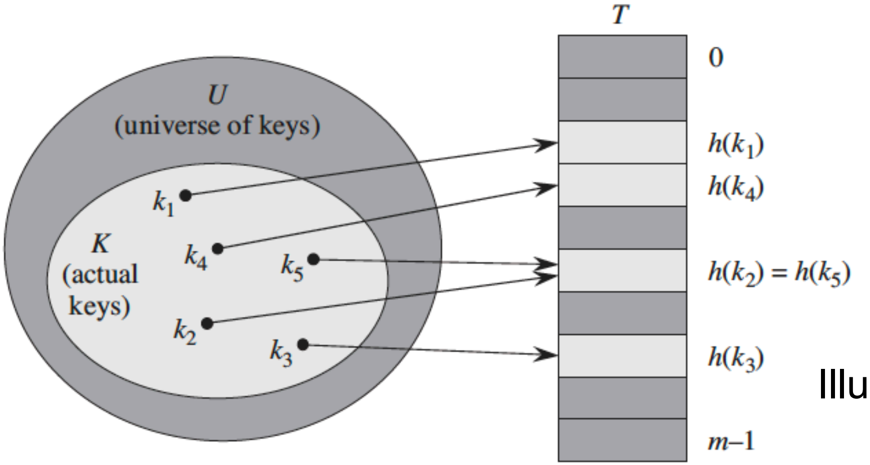
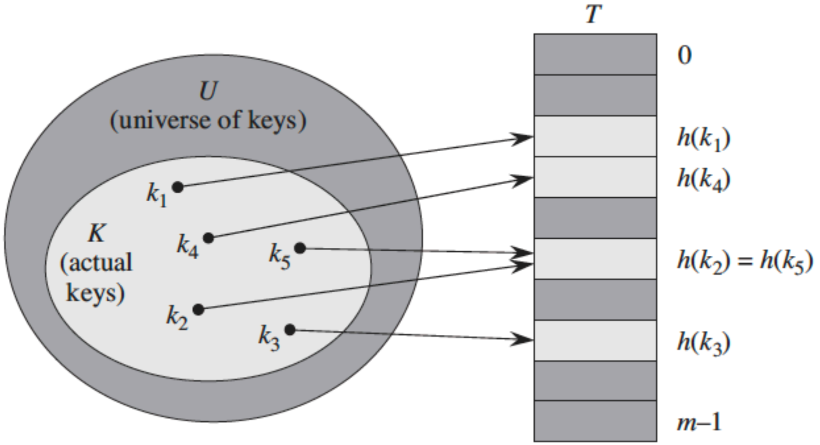


Illustration (CLR, 2009)

Hashing – Hash table

- The hash function reduces the range of array indices. Instead of a size of |U|, the array can have size m.
- The catch: We reduce the storage requirement while we maintain the benefit of searching in O(1).
  - This boundary is for the average-case time, whereas for direct addressing it holds for the worst-case time.
- The hitch: Two keys may hash to the same slot (i.e., collision).



→ collision

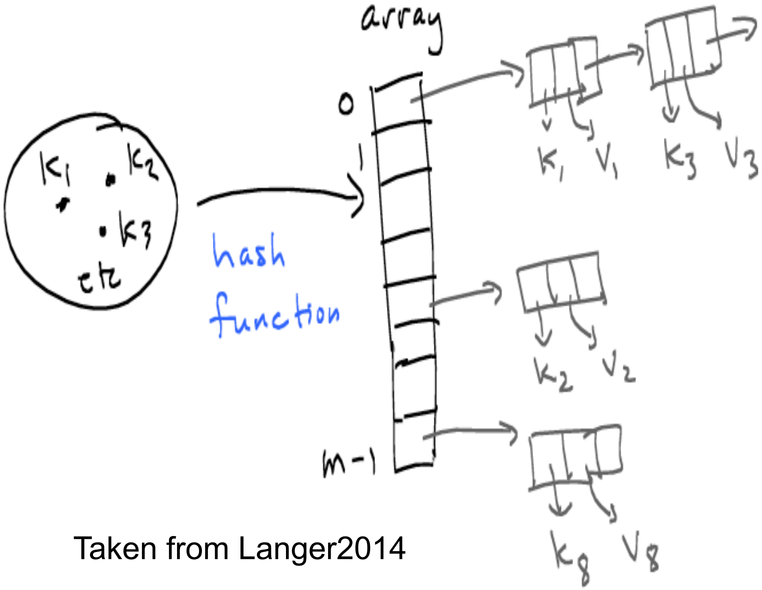
Illustration (CLR, 2009)

Outline

- Introduction.
- Hash functions.
- Collision resolution.

Hashing – Function

- A good hash function.
  - Satisfies the condition of simple uniform hashing.
    - Each key is equally likely to hash to any of the m slots.
  - Nearby/similar keys in U should map to different values.
    - http://www.md5.cz/
  - Independent of any patters of U.
  - Can be computed quickly O(1)



Taken from Langer2014

Hashing – Function

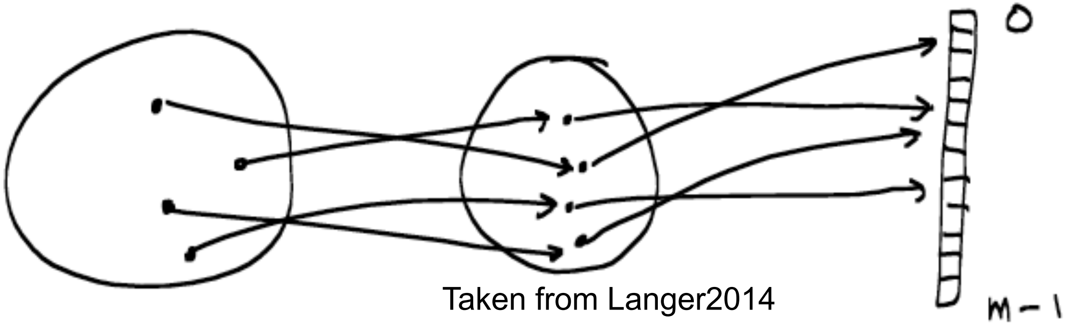
- h: U -> {0, 1, …. m-1}.
- Two steps

$$h : U \xrightarrow{\text{hash coding}} \{\text{integers}\} \xrightarrow{\text{compression}} \{0, 1, \ldots m-1\}$$

universe of keys

"hash codes"

"hash values"



Taken from Langer2014

Hashing – Function

- Collisions:
  - Two keys have same hash code
  - Two keys have different hash codes but are compressed to same hash value

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

$$h(key) = compression(hashCode(key))$$

hash code	hashcode % m (m = 7)
41	6
16	2
25	4
21	0
36	1
35	3
53	4

Taken from Langer2014

Hashing – Design

- **List of functions:**
- Division method
- Multiplication methods
- Universal Hashing

Hashing – Division Method

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

***d* must be chosen carefully!**

Example 1: *d* = 2^r and all keys are even?

Odd slots are never used…

Example 2: *d* = 2^r

$$k = 100010110101101011 \begin{cases} r = 2 \text{ -> } 11 \\ r = 3 \text{ -> } 011 \\ r = 4 \text{ -> } 1011 \end{cases}$$

keeps only r last bits…

Good heuristic: Choose *d* prime not too close from a power of 2.

Note: Easy to implement, but it depends on value d

Hashing – Multiplication Method

k=1

A= 0.11

$$h(k) = \lfloor m(\overset{\text{fractional part}}{\underbrace{k\cancel{s}A \bmod 1}})\rfloor$$

$$0 < A < 1$$

$$m = 2^p$$

$$w = word\ size\ computer$$

$$kA \bmod 1 = kA - \lfloor kA \rfloor$$

$\boxed{s} = A2^w = 2^w r_1 + r_0$ (2w-bit value)

assume integer

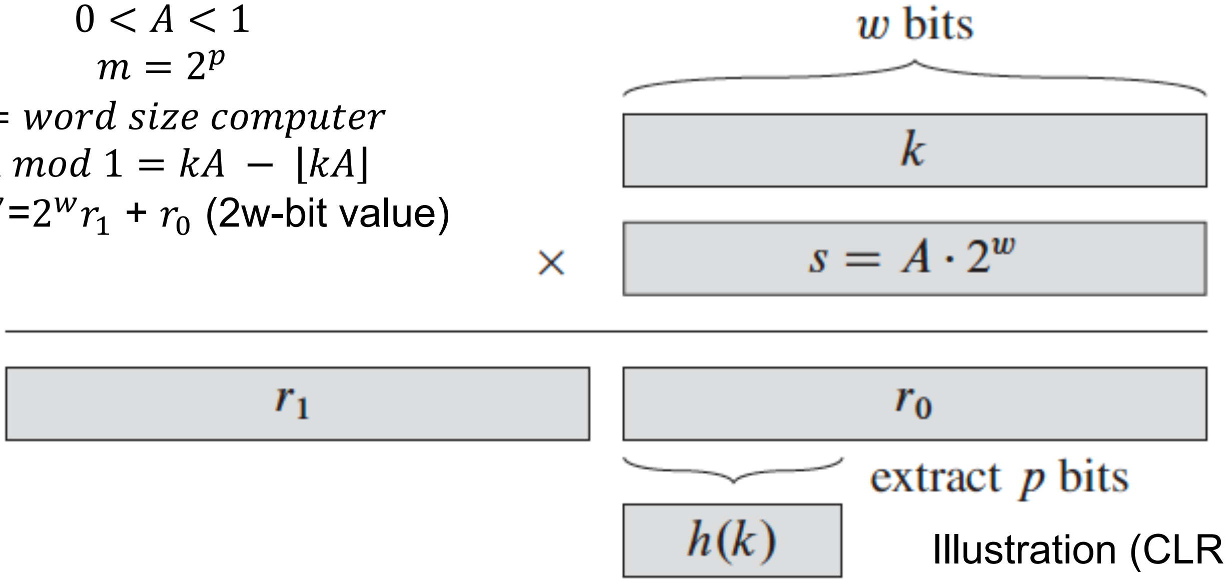


Illustration (CLR, 2009)

Note: A bit more complicated, but the value of m is not critical

Hashing – Universal Hashing

- A malicious adversary who has learned the hash function chooses keys that all map to the same slot, giving worst-case behavior.
- Defeat the adversary using **Universal Hashing**
    - Use a different random hash function each time.
        - No single input will always evoke worst-case behavior.
    - Ensure that the random hash function is independent of the keys that are actually going to be stored.
    - Ensure that the random hash function is “good” by carefully designing a class of functions to choose from:
        - Design a universal class of functions.

Hashing – Universal Hashing

- A finite collection of hash functions $\mathbf{H}$ that maps a universe $\mathbf{U}$ of keys into the range $\{0, 1, \ldots, m-1\}$ is **universal** if:

for each pair of distinct keys $x, y \in \mathbf{U}$,
the number of hash functions $h \in \mathbf{H}$
for which $h(x)=h(y)$ is $\leq |H|/m$.

for a hash function h chosen randomly
from $\mathbf{H}$, the chance of a collision
between two keys is $\leq 1/m$.

Universal hash functions give good hashing behavior.

Outline

- Introduction.
- Hash functions.
- Collision resolution.

Hashing – Hash table

- The hash function reduces the range of array indices. Instead of a size of |U|, the array can have size m.
- The catch: We reduce the storage requirement while we maintain the benefit of searching in O(1).
  - This boundary is for the average-case time, whereas for direct addressing it holds for the worst-case time.
- The hitch: Two keys may hash to the same slot (i.e., collision).

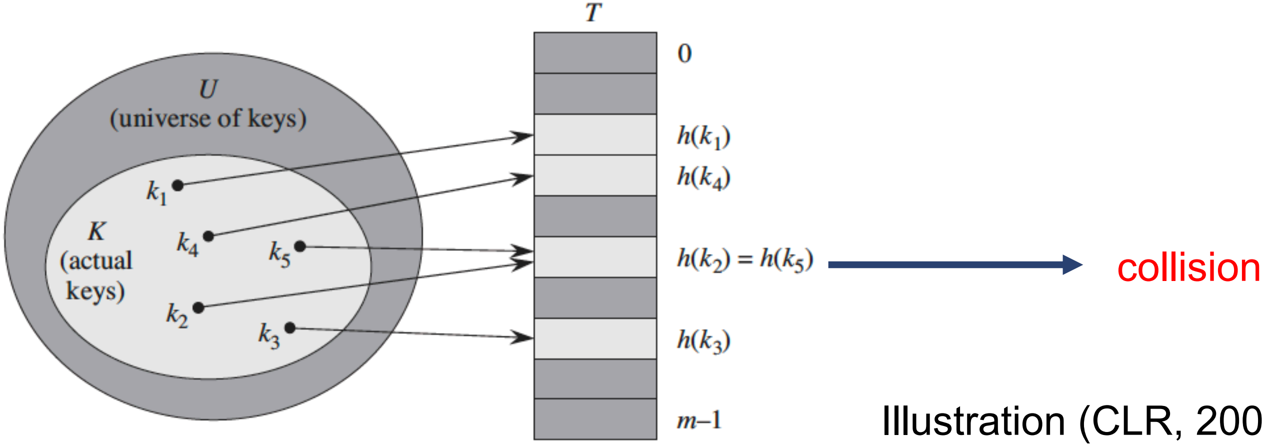
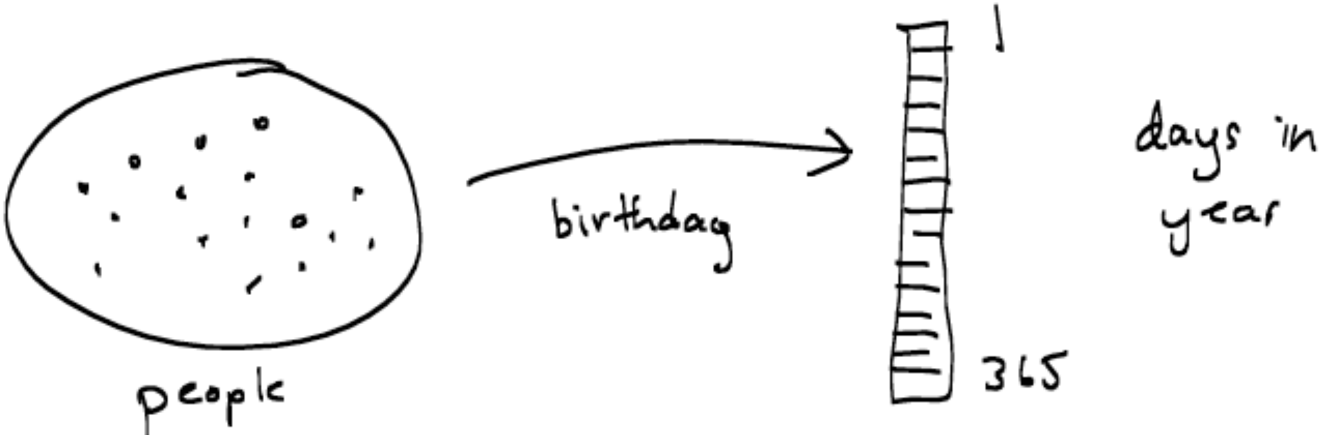


Illustration (CLR, 2009)

Hashing – Collisions – Birthday Problem

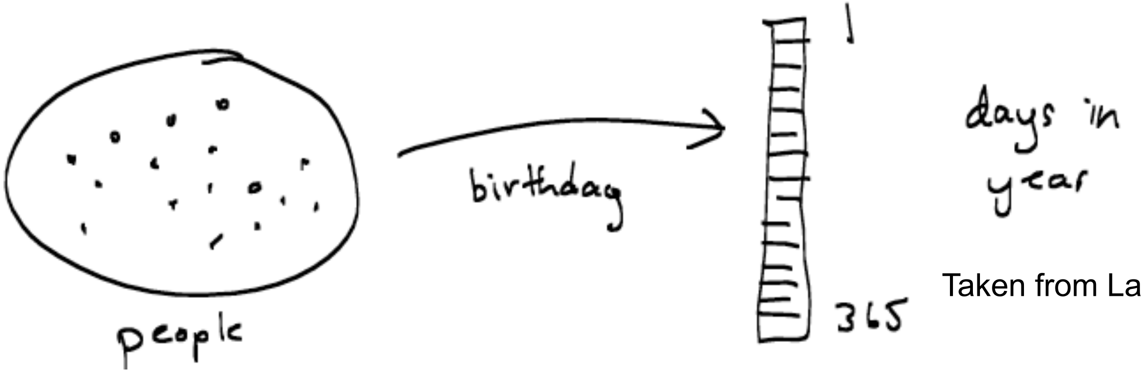
- We are 470 people in Comp251. What is the probability that at least one pair of us has the same birthday?



Taken from Langer2014

Hashing – Collisions – Birthday Problem

- We are 470 people in Comp251. What is the probability that at least one pair of us has the same birthday?



Taken from Langer2014

- |U| > m, then there must be at least two people that has the same birthday. In other words, |U| > m, then there must be at least two keys that have the same hash value.
  - For n = 23, probability ~ 0.5 (50% chance!)
- The home take message: collisions happen a lot.

Hashing – Collisions

- **List of :**
- Chaining method.
- Open addressing.
  - Linear
  - Quadratic
  - Double Hashing

Collision resolution – by chaining

- We place all the elements that hash to the same slot into the same linked list.

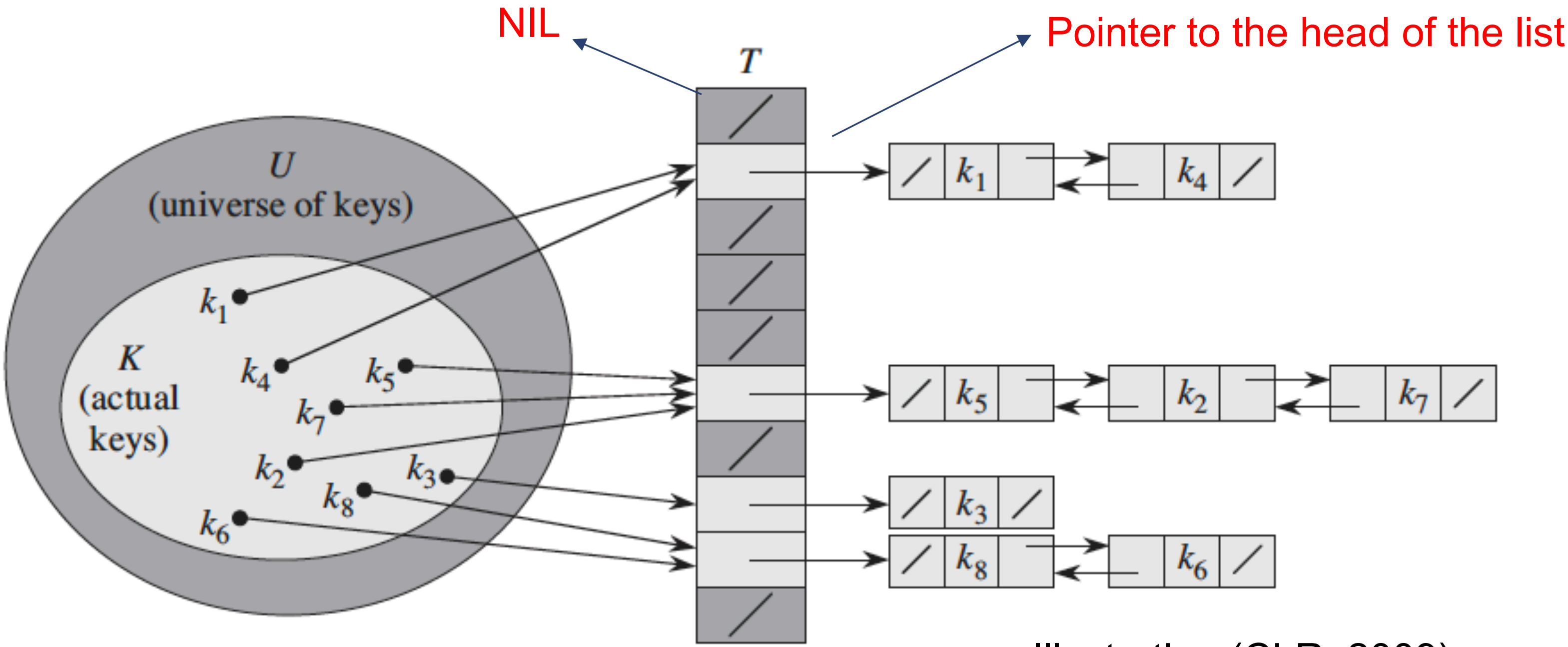


Illustration (CLR, 2009)

Collision resolution – by chaining

- Dictionary Operations.
- CHAINED-HASH-INSERT(T, x)
  - insert x at the head of list T[h(x.key)]
  - O(1) if we assume that that x is not already present in the table.
- CHAINED-HASH-SEARCH(T, k)
  - search for an element with key k in list T[h(k)]
  - proportional to the length of the list.
- CHAINED-HASH-DELETE(T, x)
  - delete x from the list T[h(x.key)]
  - O(1) if we assume that the list is doubly linked and a pointer to the object is given.
    - Proportional to the length of the list, otherwise.

Collision resolution – by chaining

- **Insertion:** O(1) time (Insert at the beginning of the list).
- **Deletion:** Search time + O(1) if we use a double linked list.
- **Search:**
  - Worst case: Worst search time is O(n).
    - Search time = time to compute hash function + time to search the list.
    - Assuming the time to compute hash function is O(1).
    - Worst time happens when all keys go the same slot (list of size n), and we need to scan the full list => O(n).
  - Average case: It depends how keys are distributed among slots.

Chaining – Average case Analysis

- Assume a **simple uniform hashing**: *n* keys are distributed uniformly among *m* slots.
- Let *n* be the number of keys, and *m* the number of slots.
- Average number of element per linked list?
- Load factor: $\alpha = \dfrac{n}{m}$
- ***Theorem:***
  - The expected time of a search is O(1 + α).
  - Note: O(1) if α < 1, but O(n) if α is O(n).

Chaining – Average case Analysis

- ***Theorem:***
  - The expected time of a search is O(1 + α).
    - expected number of elements examined to see whether any have a key equal to k.

- **Proof?**

- Distinguish two cases:
  - search is unsuccessful:
    - No element in the table has key k
  - search is successful
    - Finding an element with key k

Chaining – Unsuccessful search

- Assume that we can compute the hash function in *O*(1) time.
- An unsuccessful search requires to scan all the keys in the list.
- Average search time = O(1 + average length of lists)
- Let n_i be the length of the list attached to slot *i*.
- Average value of n_i ?

$$E(n_i) = \alpha = \frac{n}{m}$$ (Load factor)

$$\Rightarrow O(1) + O(\alpha) = O(1 + \alpha)$$

Chaining – Successful search

- Each list is not equally likely to be searched. Instead, the probability that a list is searched is proportional to the number of elements it contains.
- Assume the position of the searched key x is equally likely to be any of the elements stored in the list.
  - The number of elements examined during a successful search for an element x is one more than the number of elements that appear before x in x's list.
    - new elements are placed at the front of the list, elements before x in the list were all inserted after x was inserted
  - To find the expected number of elements examined, we take the average, over the n elements x in the table, of 1 plus the expected number of elements added to x's list after x was added to the list.

$$X_{ij} = I\left\{h(k_i) = h(k_j)\right\};\ E(X_{ij}) = \frac{1}{m}$$ (probability of a collision)

Chaining – Successful search

$$number\ of\ keys\ inserted\ after\ x\ =\ 1+\sum_{j=i+1}^{n} X_{ij}$$

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

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

By linearity of expectation

$\frac{1}{m}\sum_{i=1}^{n}\sum_{j=i+1}^{n} 1$

$\frac{n^2}{2}$

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

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

Search time:

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

Supplementary material

$$
\begin{aligned}
\mathrm{E}\left[\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}X_{ij}\right)\right] \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\mathrm{E}\left[X_{ij}\right]\right) \\
&= \frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}\frac{1}{m}\right) \\
&= 1+\frac{1}{nm}\sum_{i=1}^{n}(n-i) \\
&= 1+\frac{1}{nm}\left(\sum_{i=1}^{n}n-\sum_{i=1}^{n}i\right) \\
&= 1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right) \\
&= 1+\frac{n-1}{2m} \\
&= 1+\frac{\alpha}{2}-\frac{\alpha}{2n}\,.
\end{aligned}
$$

Chaining – Home take message

- If the number of hash-table slots is at least proportional to the number of elements in the table, we have n = O(m) and:

$$\alpha = \frac{n}{m} = \frac{O(m)}{m} = O(1)$$

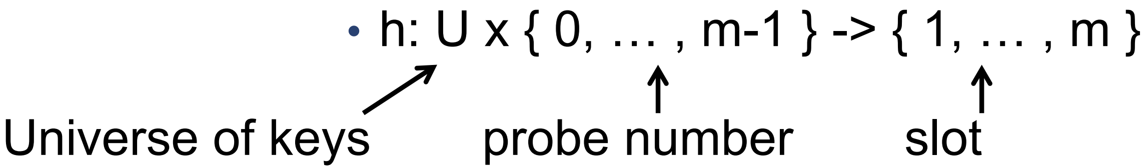
- **We can support all dictionary operations in O(1) time on average.**
  - Insertion => O(1) worst case time.
  - Deletion => O(1) worst case time (under assumptions).
  - Searching => O(1) on average time.

Hashing – Collisions

- **List of :**
- Chaining method.
- Open addressing.
  - Linear
  - Quadratic
  - Double Hashing

Hashing – Open Addressing

- No storage for multiple keys on single slot (i.e. no chaining).
- **Idea:** Probe the table.
  - Insert if the slot is empty,
  - Try another hash function otherwise.



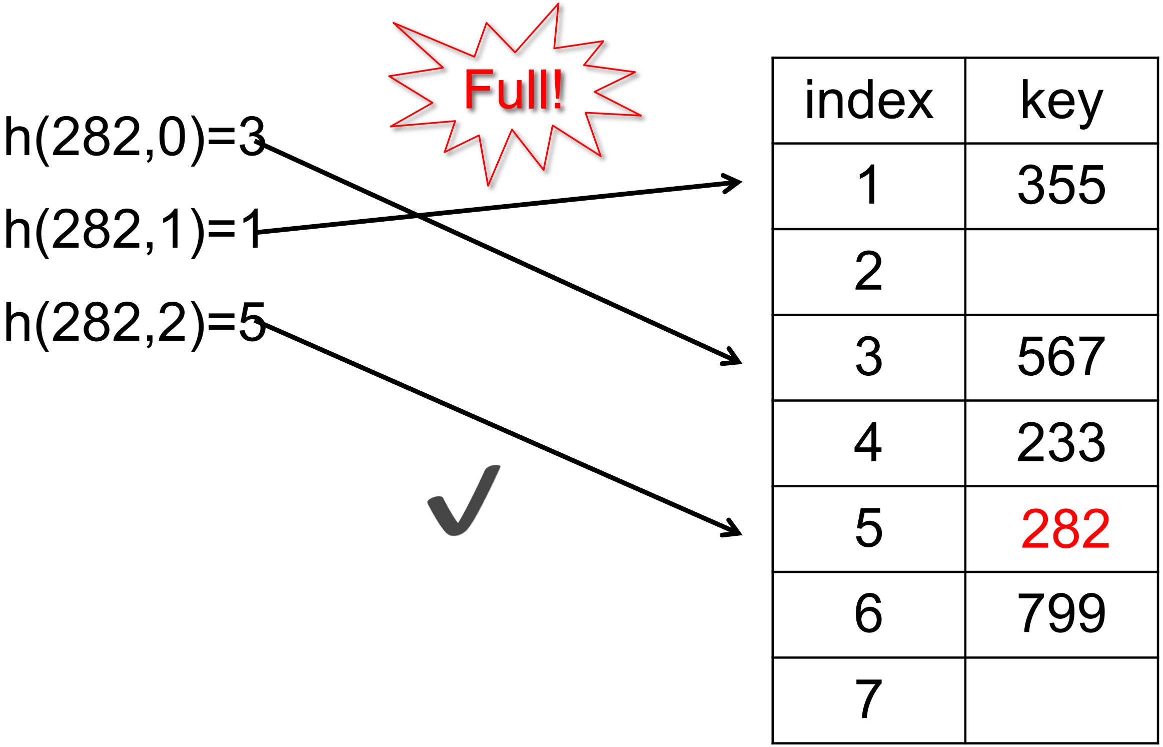
- h: U x { 0, … , m-1 } -> { 1, … , m }

Universe of keys probe number slot

- Constraints:
  - $n \leq m$ (i.e. more slots than keys to store)
  - Deletion is difficult
- Challenge: How to build the hash function?

Hashing – Open Addressing

Illustration: Where to store key 282?



$h(282,0)=3$

Full!

$h(282,1)=1$

$h(282,2)=5$

✓

index	key
1	355
2	
3	567
4	233
5	282
6	799
7	

Note1: Search must use the same probe sequence.

Note2: The probe sequence must be a permutation of the indexes

Hashing – Open Addressing

Linear probing:

$$h(k,i) = \left(h'(k) + i\right) \bmod m$$

Notes: primary clustering

- Tendency to create primary clusters (long runs of occupied slots build up).
- There are only m distinct probe sequences (the initial probe determines the entire probe sequence)

secondary clustering

Quadratic probing:

$$h(k,i) = \left(h'(k) + c_1 \cdot i + c_2 \cdot i^2\right) \bmod m$$

Notes:

- The values of c_1, c_2 and m must be constrained to ensure that we have a full permutation of ⟨ 0, … , m-1 ⟩.
- There are only m distinct probe sequences
- **Secondary clustering**: If 2 distinct keys have the same h′ value, then they have the same probe sequence.

Hashing – Open Addressing

Double hashing:

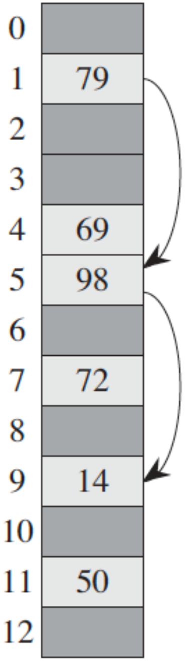
$$h(k,i) = \left(h_1(k) + i \cdot h_2(k)\right) \bmod m$$

- One of the best methods available.
  - More permutations.
    - Characteristics of randomly chosen permutations.
      - Alleviate the problem of clustering
- Must have $h_2(k)$ be “relatively” prime to m to guarantee that the probe sequence is a full permutation of $\langle 0, 1, \ldots, m - 1 \rangle$.
- Examples:
  - m power of 2 and h_2 returns odd numbers.
    - $O(m^2)$ probes are used.
      - Each possible $(h_1(k), h_2(k))$ pairs yields a distinct probe sequence.
  - m prime number and $1 < h_2(k) < m$

Hashing – Open Addressing

Double hashing:

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



Task: To insert k = 14 (key = value)

$h_1(k) = k \; mod \; m = k \; mod \; 13$

$h_2(k) = 1 + (k \; mod \; m') = 1 + (k \; mod \; 11)$

$i = 0$:

$h_1(14) = 14 \; mod \; 13 = 1$

$h_2(14) = 1 + (14 \; mod \; 11) = 4$

$h(14{,}0) = 1 + 0 * 4 = 1$

$i = 1$:

$h(14{,}1) = 1 + 1 * 4 = 5$

$i = 2$:

$h(14{,}2) = 1 + 2 * 4 = 9$

Hashing – Open Addressing

We assume **uniform hashing**: Each key equally likely to have anyone of the m's permutations as its probe sequence, independently of other keys.

Theorem 1: The expected number of probes in an unsuccessful search is at most $\frac{1}{1-\alpha}$.

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

Theorem 2: The expected number of probes in a successful search is at most $\frac{1}{\alpha}\cdot\log\left(\frac{1}{1-\alpha}\right)$

Reminder: $\alpha=\frac{n}{m}$ is the load factor

Open Addressing – supplemental material

Initial state: *n* keys are already stored in *m* slots.

Probability 1st slot is occupied: n/m.

Probability 2nd slot is occupied knowing 1st is too: (n-1)/(m-1).

Probability 3rd slot is occupied knowing 2nd is too : (n-2)/(m-2).

Let *X* be the number of unsuccessful probes.

$$\Pr\{X \geq i\} = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}$$

$n < m \Rightarrow (n-j)/(m-j) \leq n/m$, for $j \geq 0$

$$\Pr\{X \geq i\} \leq (n/m)^{i-1} = \alpha^{i-1}$$

$$E[X] = \sum_{i=1}^{\infty} \Pr\{X \geq i\} \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \sum_{i=0}^{\infty} \alpha^{i} = \frac{1}{1-\alpha}$$

Open Addressing – home take message

Corollary

The expected number of probes to insert is at most 1/(1 − α).

Interpretation:

- If α is constant, an unsuccessful search takes O(1) time.
- If α = 0.5, then an unsuccessful search takes an average of 1/(1 − 0.5) = 2 probes (1.387 for a successful search).
- If α = 0.9, takes an average of 1/(1 − 0.9) = 10 probes (2.559 for a successful search).

Proof of Theorem on successful searches: See [CLRS, 2009].

Open Addressing – home take message

Corollary

The expected number of probes to insert is at most $1/(1 - \alpha)$.

Interpretation:

- If α is constant, an unsuccessful search takes O(1) time.
- If $\alpha = 0.5$, then an unsuccessful search takes an average of $1/(1 - 0.5) = 2$ probes (1.387 for a successful search).
- If $\alpha = 0.9$, takes an average of $1/(1 - 0.9) = 10$ probes (2.559 for a successful search).

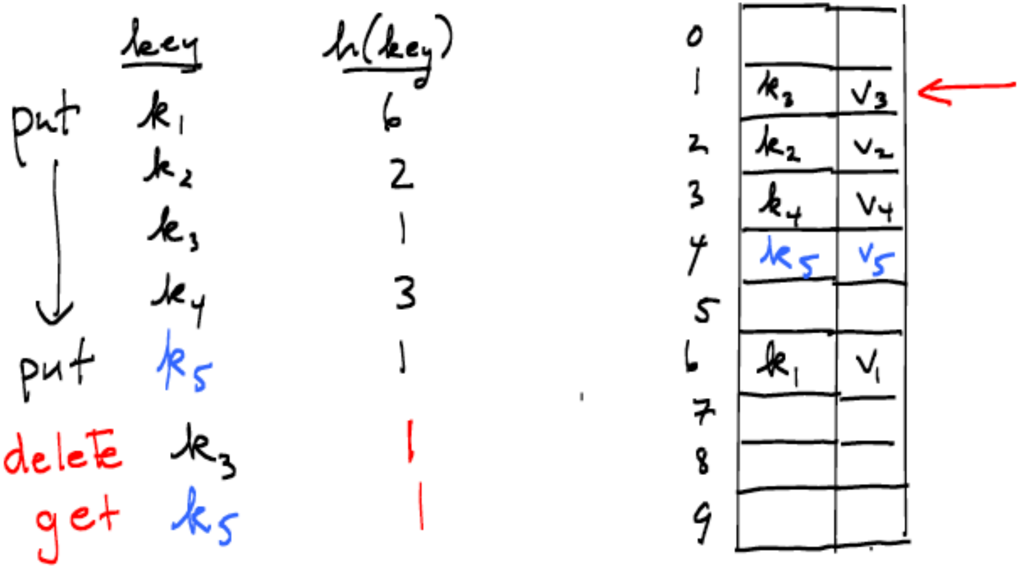
Proof of Theorem on successful searches: See [CLRS, 2009].

Open Addressing

- Dictionary Operations.
  - HASH-INSERT(T, x)
  - HASH-SEARCH(T, k)
  - HASH-DELETE(T, x)

Open Addressing - Problem

- Dictionary Operations.
  - HASH-DELETE(T, x) => Works poorly



	key	h(key)
put	k_1	6
	k_2	2
	k_3	1
	k_4	3
put	k_5	1
delete	k_3	1
get	k_5	1

0		
1	k_3	v_3
2	k_2	v_2
3	k_4	v_4
4	k_5	v_5
5		
6	k_1	v_1
7		
8		
9		

  - get(k_5) will fail because slot 1 will be empty even though k_5 is in the table

Outline

- Introduction.
- Hash functions.
- Collision resolution.

time for questions