# Hashing II

# Rehashing

- **Rehashing** is a technique used to resolve collisions and handle the increased load factor in hash tables.
- When the table becomes too full (usually defined by a load factor threshold), a new hash function is applied, and all elements are reinserted into a larger table.

- $\boldsymbol{Load\ factor(\ \alpha) = \frac{number\ of\ elements}{table\ size}}$
- When $\alpha\ \ exceeds\ a\ threshod\ (commonly\ 0.7)\ rehashing\ is\ triggered.$

# Rehashing : steps

- Create a new, larger table
  - Usually double the size or next prime number
- Apply a new hash function
- Reinsert all existing elements into the new table

| Advantages | Disadvantages |
|---|---|
| • Reduces collisions.<br>• Improves efficiency for insertion, deletion, and search. | • Expensive operation because all keys must be rehashed.<br>• Increased memory usage temporarily during rehashing. |

# Rehashing : Example

- Hash table size (m) = 7
- Load factor threshold = 0.7
- Hash function h(k) = k mod m
- Keys to insert : 10,22,31,40,42,52,55

# Rehashing : Example

- Hash table size = 7
- Load factor threshold = 0.7
- Hash function h(k) = k mod m
- Keys to insert : 10,22,31,40,42,52,55
- h(10) = 10 mod 7 = 3
- h(22) = 22 mod 7 = 1
- h(31) =31 mod 7 = 3, collision , resolve using quadratic probing = $(3+1^2)$mod 7 = 4
- h(40) = 40 mod 7 = 5
- h(42) = 42 mod 7 = 0 ← When inserting element 42, load factor exceeds 0.71
- h(52) = 52 mod 7 = 3, collision = $(3+1^2)$mod 7 =4 collison, = $(3+2^2)$mod 7= 0, collision = $(3+3^2)$mod 7 = 5 collision, = $(3+4^2)$mod 7, collision, = $(3+5^2)$mod 7 = 0 collision, = $(3+6^2)$mod 7 =4 collision = $(3+7^2)$mod 7 cannot insert due to over crowding
- h(55) = 55 mod 7 = 6
- compute $\boldsymbol{Load\ factor}(\boldsymbol{\alpha}) = \frac{\boldsymbol{number\ of\ elements}}{\boldsymbol{table\ size}}$ = 5/7 ~ 0.71

# Rehashing : Example

Rehashing

- Create a new table of size 14 ( double the old size)
- New hash function h'(k) = k mod 14
- Reinsert all keys
- h(10) = 10 mod 14 = 10
- h(22) = 22 mod 14 = 8
- h(31) =31  mod 14 = 3
- h(40) = 40 mod 14 = 12
- h(42) = 42 mod 14 = 0

# Double Hashing

- **Double hashing** is one of the best methods for dealing with collisions.
- Double Hashing is a collision resolution method in open addressing. When a collision occurs, a second hash function is used to determine the step size for probing.
  - If the slot is full, then a second hash function is calculated and combined with the first hash function.
  - $h'(k,i) = (h(k) + i.h_2(k)) \bmod m$
  - $h(k)$ : primary hash function
  - $h_2(k)$ : secondary hash function
  - i: probe attempt number (i=0,1,2,3,….)
  - m : table size

# Double hashing

- Table size = 7
- Hash function
  - Primary $h(k) = k \bmod 7$
  - Secondary $h_2(k) = 5 - (k \bmod 5)$
- Keys : 10,22,31,40,52

$h'(k,i) = (h(k) + i.h_2(k)) \bmod m$

# Double hashing

- Table size = 7
- Hash function
  - Primary $h(k) = k \bmod 7$
  - Secondary $h_2(k) = 5 - (k \bmod 5)$
- Keys : 10,22,31,40,52
- h(10) = 10 mod 7 = 3
- h(22) = 22 mod 7 = 1
- h(31) = 31 mod 7 = 3 collision → use double hashing (3+ 1.(5-(31 mod 5))) = 3 + (5-1) = 7 mod 7 = 0
- h(40) = 40 mod 7 = 5
- h(52) = 52 mod 7 = 3 Collison -> double hashing (3+1.(5-(52 mod 5)) = 3+ 5-2 = 6 mod 7 = 6

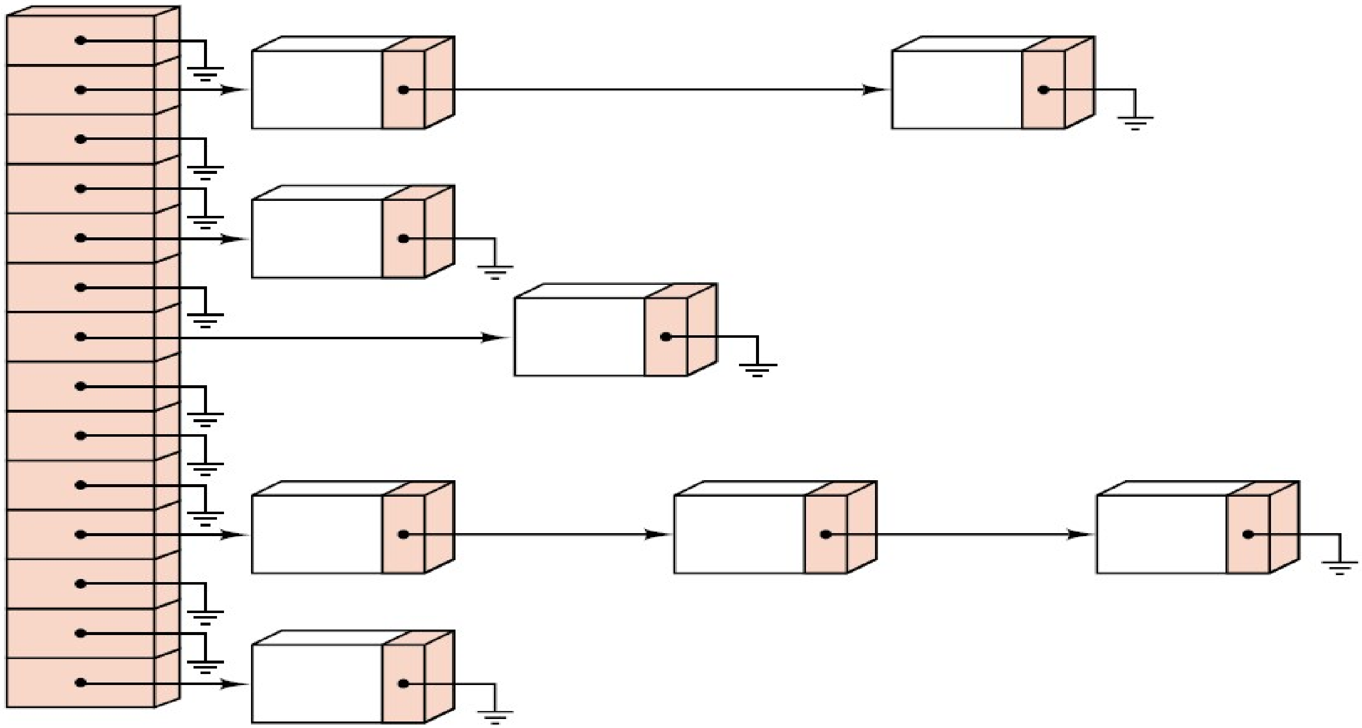
$h'(k,i) = (h(k) + i.h_2(k)) \bmod m$

# Data Structure for Chaining

```
#define MAX_CHAR 10
#define TABLE_SIZE 13
#define IS_FULL(ptr) (!(ptr))
typedef struct {
   char key[MAX_CHAR];
   /* other fields */
} element;
typedef struct list *list_pointer;
typedef struct list {
   element item;
   list_pointer link;
};
list_pointer hash_table[TABLE_SIZE];
```

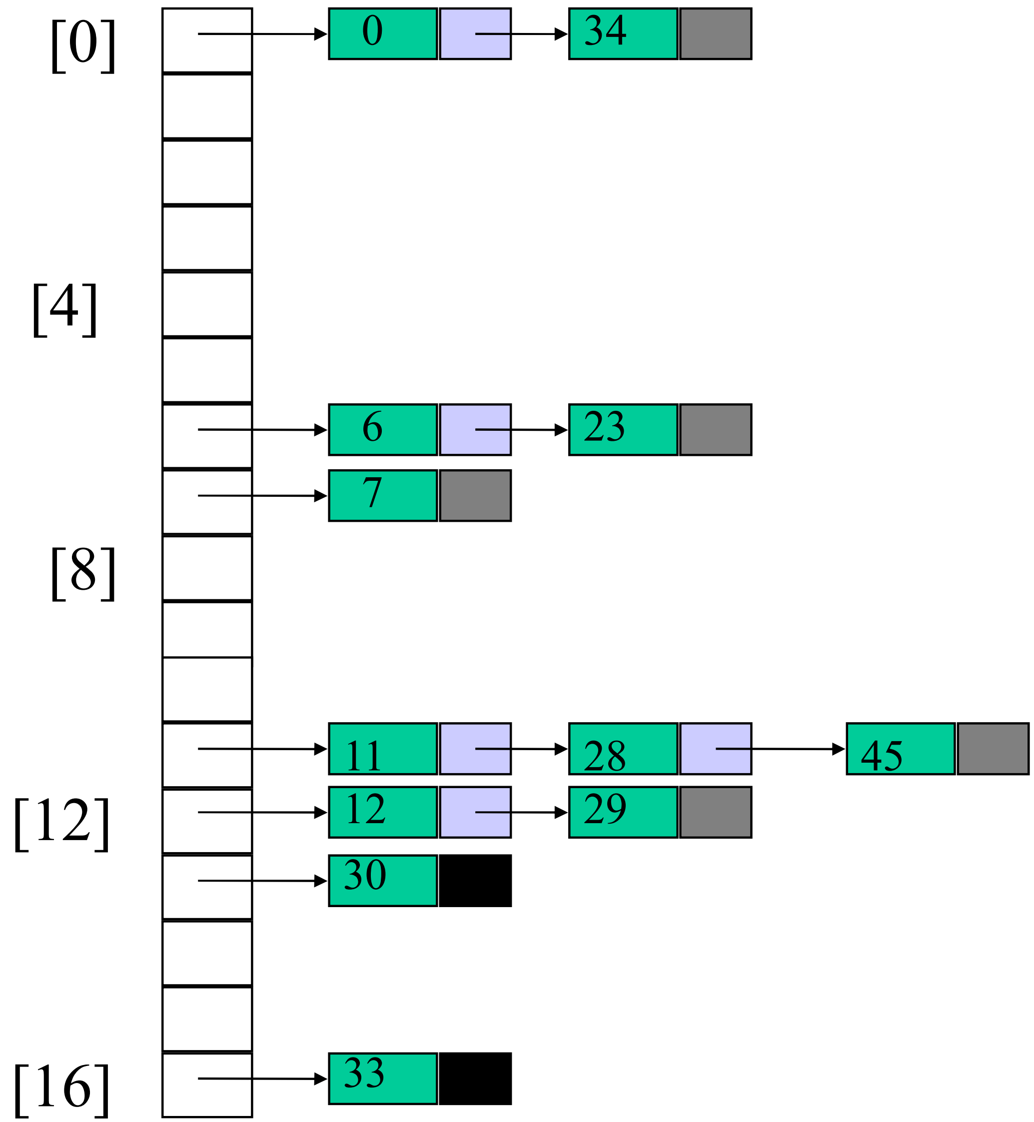
The idea of **Chaining** is to combine the linked list and hash table to solve the overflow problem.

# Figure of Chaining



# Sorted Chains

- Put in pairs whose keys are 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- Bucket = key % 17.



# Theory of Hashing

- Theorem : For any hash function h: U -> {0, 1, …, M}, there exists a set S of n keys that all map to the same location assuming |U| > nM.
    - So, in worst case, no hash function can avoid linear search complexity.

Proof:

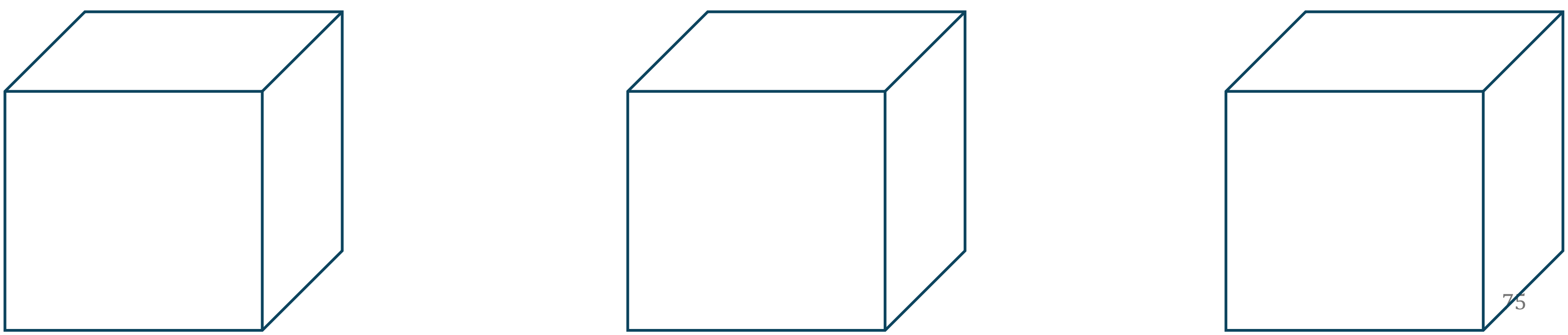
- Take any hash function h you wish to consider
- Map all the keys of U using h to the table of size M
- By the pigeonhole principle, at least one table entry will have n keys.
- Choose those n keys as input set S.
    - Now h will maps the entire set S to a single location, for worst example of hashing

# Pigeonhole Principle

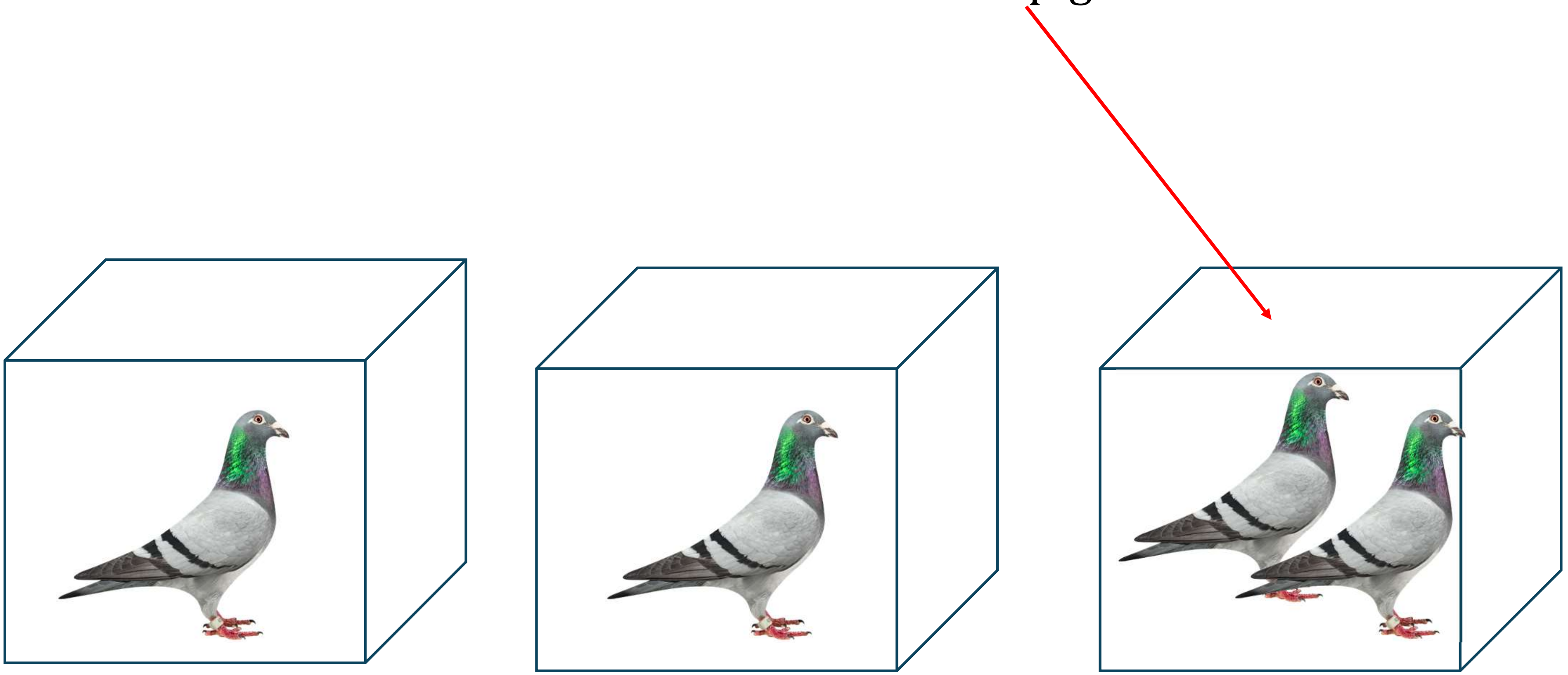



4 pigeons

3 pigeonholes

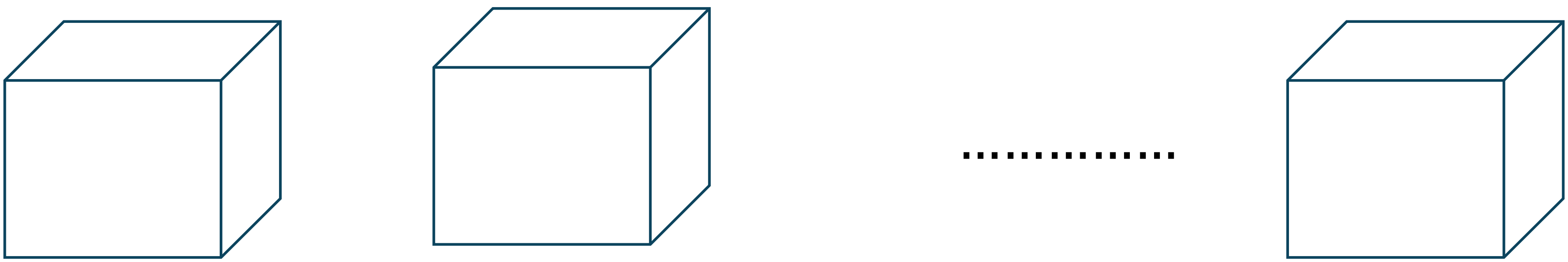


A pigeonhole must fit at least two pigeons.



n pigeons

m pigeonholes $n > m$



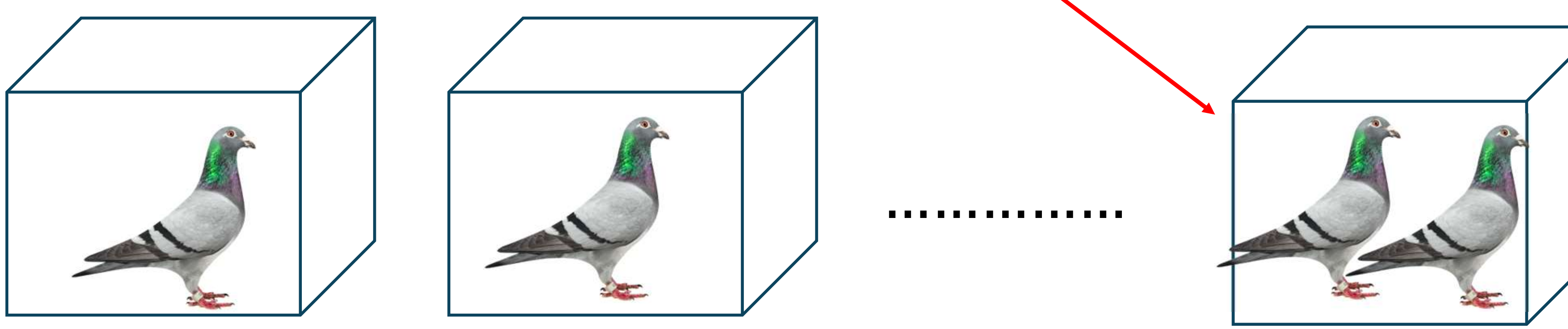
# The Pigeonhole Principle

n pigeons

m pigeonholes

$n > m$

There is a pigeonhole with at least 2 pigeons



# Theory of Hashing

- Theorem : For any hash function h: U -> {0, 1, …, M}, there exists a set S of n keys that all map to the same location assuming |U| > nM.
    - So, in worst case, no hash function can avoid linear search complexity.
- Proof:
- Take any hash function h you wish to consider
- Map all the keys of U using h to the table of size M
- By the pigeonhole principle, at least one table entry will have n keys.
- Choose those n keys as input set S.
    - Now h will maps the entire set S to a single location, for worst example of hashing

# Theory of Hashing: Birthday Paradox

- Suppose birth days are chance events:
  - Date of birth is purely random
  - Any day of the year just as likely as another
- What are the chances that in a group of 30 people, at least two have the same birthday?
- How many people will be needed to have at least 50% chance of same birthday?
- Its called a paradox because the answer appears to be counter-intuitive.
- There are 365 different birthdays, so for 50% chance, you expect at least 182 people.

# Birthday Paradox : math behind !

Suppose 2 people in the room.

- What is the probability that they have the same birthday ?
- Answer is 1/365
- All birthdays are equally likely, so B's birthday falls on A's birthday 1 in 365 times.

# Birthday Paradox : math behind !

Now suppose there are k people in the room.

- Its more convenient to calculate the probability X that no two have the same birthday is (1-x) ( the probability of at least one match).
- Let P ( no two have the same birthday) = X then
- P( at least one match) = 1-X

# Calculating X: Probability of No Shared Birthdays

- For k people, X can be computed as :
- For the first person, any birthday is fine $\frac{365}{365}$
- For the second person, their birthday must not match the first person $\frac{364}{365}$
- For the third person, their birthday must not match either of the first two $\frac{363}{365}$
- Continue this pattern for k people
- $X = \frac{365}{365}$ x $\frac{364}{365}$ x $\frac{363}{365}$ $x$ … x $\frac{365-k+1}{365}$
- $X = \prod_{i=0}^{k-1} \frac{365-}{365} = e^{\frac{k^2}{2n}}$
- For k =23, e $^{-0.69}$ <= 0.4999

# Key insight :

- For k =23, probability of at least one shared birthday is approximately 50%
  - Even 23 <<< 365
- Results show how collisions (shared birthdays) occur much sooner than intuitively expected, a concept directly relevant to hashing.

# Universal Hash function

- A universal hash function : To a family of hash functions H with the following key property:
  - for any two distinct elements x and y in the universe U
  - the probability that a randomly chosen hash function h∈H maps them to the same slot in the hash table is at most where M is the number of slots in the hash table.
- A set of hash functions H is universal if the likelihood of a collision between two distinct keys x and y is bounded by 1/M.

$$P(h(x) = h(y)) \leq \frac{1}{M}, \quad \forall x, y \in U, x \neq y$$

# Universal Hash function – expected search time

- When using a random hash function hhh from a universal family:
- The expected search time in the hash table is $O(1 + \frac{n}{M})$.
  - n is the number of elements in the hash table.
  - M is the number of slots in the table.
  - O(1) corresponds to the average cost of probing the hash table, while
  - n/M accounts for the expected collisions when n>M

# Perfect Hashing : Worst – Case O(1) lookup

- Universal hashing assures us that hashing has expected O(1) search time, assuming n/M is at most a constant.
- But what about worst case?
- There remains a small, but non-zero, prob. of unlucky random draw.
- A more sophisticated theory of Perfect Hashing shows that one can even achieve O(1) worst-case result, using a 2-level hashing table.

# Perfect Hashing

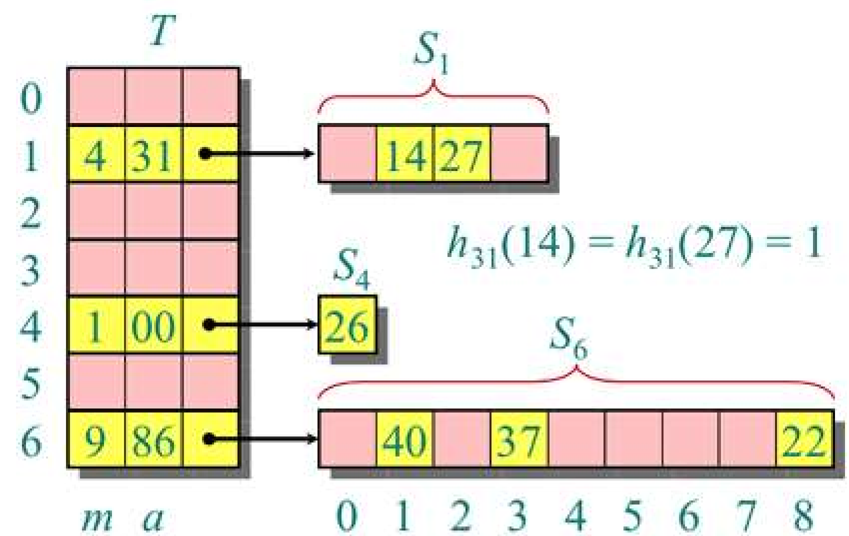
1. First-level hashing

- Use a hash function $h_1$ to divide keys into n buckets.
- Each bucket i contains $k_i$ keys such that:
- $h_1(x) = I$
- If the number of collisions in a bucket is small, resolving them in the second level becomes easier.

# Perfect Hashing

2. Second-level hashing

- For each bucket, design a second-level hash function $h_2$ such that all $k_i$ keys are mapped uniquely within that bucket.
- The second-level table size is often proportional to $k^2_i$ ensuring a collision-free mapping.



# Perfect Hashing

- Algorithm ( input: N data elements)
- Set the primary hash table size to M >= N, choose primary hash function $h_1$ : hash all elements into the primary table.
- For each bucket i, in the hash table that contains $b_i$ >1 data elements,
  - Build a secondary hash table with a size of $b_i^2$
  - Find a hash function $h_i^2$ that makes no collisions in the secondary hash table
  - Hash all $b_i$ elements into the secondary hash table : record $h_i^2$

# Perfect Hashing : Applications

- Compiler Design:
  - For keyword lookup, where the set of keywords is fixed.
- Databases:
  - For static datasets, such as indexing a fixed set of keys.
- Networking:
  - In routing tables, where the set of routes is known in advance.

# Example 1: Static Perfect Hashing

- Suppose we have a fixed set of keys :

| Data |
|---|
| bat |
| cat |
| dog |

h(key) = g[h(first_letter) + h(second_letter])] %3

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Letter | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| value (g) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |

# Example 1: Static Perfect Hashing

- Suppose we have a fixed set of keys :

| Data |
|---|
| bat |
| cat |
| dog |

h(key) = g[h(first_letter) + h(second_letter])] %3

h(bat) = g[h(b) + h(a])] %3 = (2+1)%3 = 0

h(cat) =( 3+1) % 3 = 1

H(dog) = (4+15) %3 = 1 → Collision

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Letter | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| value (g) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |

# Example 1: Static Perfect Hashing

- Suppose we have a fixed set of keys :

| Data | index |
|---|---|
| bat | 0 |
| cat | 1 |
| dog | 2 |

h(key) = g[h(first_letter) + h(second_letter])] %3

h(bat) = g[h(b) + h(a])] %3 = (2+1)%3 = 0

h(cat) =( 3+1) % 3 = 1

H(dog) = (5+15) %3 = 2 → fixed!

| Index | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Letter | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z |
| value (g) | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 |

5

# Example 2: Perfect Hashing

- Hash the numbers 2,12,4,5,23,13,3
- Choose M = 10; $h_1$ = x mod M

1. 2 mod 10 = 2
2. 12 mod 10 = 2
3. 4 mod 10 = 4
4. 5 mod 10 = 5
5. 23 mod 10 = 3
6. 13 mod 10 = 3
7. 3 mod 10 = 4

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 2,12 |
| 3 | 23,13,3 |
| 4 | 4 |
| 5 | 5 |
| …. | |
| 9 | |

# Example 2: Perfect Hashing

- Hash the numbers 2,12,4,5,23,13,3
- Choose M = 10; $h_1$ = x mod M

1. 2 mod 10 = 2
2. 12 mod 10 = 2
3. 4 mod 10 = 4
4. 5 mod 10 = 5
5. 23 mod 10 = 3
6. 13 mod 10 = 3
7. 3 mod 10 = 4

Handling collisions

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 2,12 |
| 3 | 23,13,3 |
| 4 | 4 |
| 5 | 5 |
| …. | |
| 9 | |

# Example 2: Perfect Hashing

Handling collisions

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | 2,12 |
| 3 | 23,13,3 |
| 4 | 4 |
| 5 | 5 |
| …. | |
| 9 | |

- Bucket 2 hash two elements ($b_2 = 2$ ).
- We will make secondary hash table with a size of 4 ($b_2^2 = 4$) for this bucket.
- Secondary hash function $h_2^2$ = x mod $b_2^2$ = x % 4
- Secondary hash function $h_3^2$ = x mod $b_3^2$ = x % 9

# Example 2: Perfect Hashing

Handling collisions

| | |
|---|---|
| 0 | |
| 1 | |
| 2 | x mod 4 |
| 3 | x mod 9 |
| 4 | 4 |
| 5 | 5 |
| …. | |
| 9 | |

- 2 mod 4 = 2
- 12 mod 4 = 0
- 23 mod 9 = 5
- 13 mod 9 =4
- 3 mod 9 = 3

| | |
|---|---|
| 0 | 12 |
| 1 | |
| 2 | 2 |
| 3 | |

| | |
|---|---|
| 0 | |
| 1 | |
| 3 | x mod 4 |
| 3 | 13 |
| 4 | 23 |
| 5 | 5 |
| …. | |
| 8 | |

# Dynamic hashing: Extendible hashing

- Fagin, R.; Nievergelt, J.; Pippenger, N.; Strong, H. R., "Extendible Hashing - A Fast Access Method for Dynamic Files", *ACM Transactions on Database Systems* **4** (3): 315–344, September, 1979.
- For a good overview, read:

  http://en.wikipedia.org/wiki/Extendible_hashing