Hashing

Defining a Hash Function

1. **Uniform hash function**
2. **Division method**
3. **Multiplication method**
4. **Hashing of strings**

Uniform Hash Functions

- Distributes keys uniformly in the hash table
- If keys are uniformly distributed in [0, *X*), map them to a hash table of size ***m*** (*m* < *X*) using hash function:

$$k \in [0, X)$$

$$\mathbf{hash}(\mathbf{k}) = \left\lfloor \frac{\mathbf{km}}{\mathbf{X}} \right\rfloor$$

k: **key value**

[ ]: close interval

( ): open interval

Hence, 0 ≤ *k* < *X*

⌊ ⌋: *floor* function

Division Method (mod operator)

- Map into a hash table of m slots
- Use the modulo operator (%) to map an integer to a value between 0 and $m-1$
  - $(n \% m)$ = remainder of n divided by m, where n and m are positive integers
  - $$hash(k) = k \% m$$
- One of the most popular method

A Good Choice of “*m*”

- If m is power of two (say 2^n)
  - ($key \bmod m$): extracting the last n bits of the key
- If m is 10^n:
  - ($key \bmod m$): the last n digits of the key
- Both are not a good choice!
- **Rule of thumb:** Pick a **prime number**, *close to power of 2* as *key*

Multiplication Method

1. Multiply key by a fraction ***F*** (0 < F < 1) // kF
2. Extract the fractional part // $(kF - \lfloor kF \rfloor)$
3. Multiply by *m*, the hash table size

$$hash(k) = \lfloor m(kF - \lfloor kF \rfloor) \rfloor$$

Common example of F: The reciprocal of the golden ratio, i.e.

A = (sqrt(5) – 1) / 2 = 0.618033

Hashing of Strings: Example

```
hash1(str) {
  int sum = 0
  for each character c in str {
    sum += c
    // sum up the ASCII values of all characters
   }
  return (sum % H_SIZE)// H_SIZE: hash table size
  }
```

Hashing of Strings: Example

```
hash1("Tic-tac-toe")
= ('T' + 'i' + 'c' + '-' +
   't' + 'a' + 'c' + '-' +
   't' + 'o' + 'e') % H_SIZE
```

= (84 + 105 + 99 + 45 +
116 + 97 + 99 + 45 +
116 + 111 + 101 + 107) % H_SIZE

= 1125 % H_SIZE

= 4 (assuming H_SIZE = 19)

Hashing of Strings: Example

- All 3 strings below have the same hash value.
  - **`"Lee Chin Tan"`**
  - **`"Chen Le Tian"`**
  - **`"Chan Tin Lee"`**
- **What is the reason?**

- **Problem**: Hash value is independent of the positions of characters

An Improved Hashing

- Better to "*shift*" the sum before adding the next character, so that its position affects the hash code

```
hash2(str)
{  sum = 0
   for each character c in str
   {  sum = sum * 41 + c
   }
   return sum % m
}
```

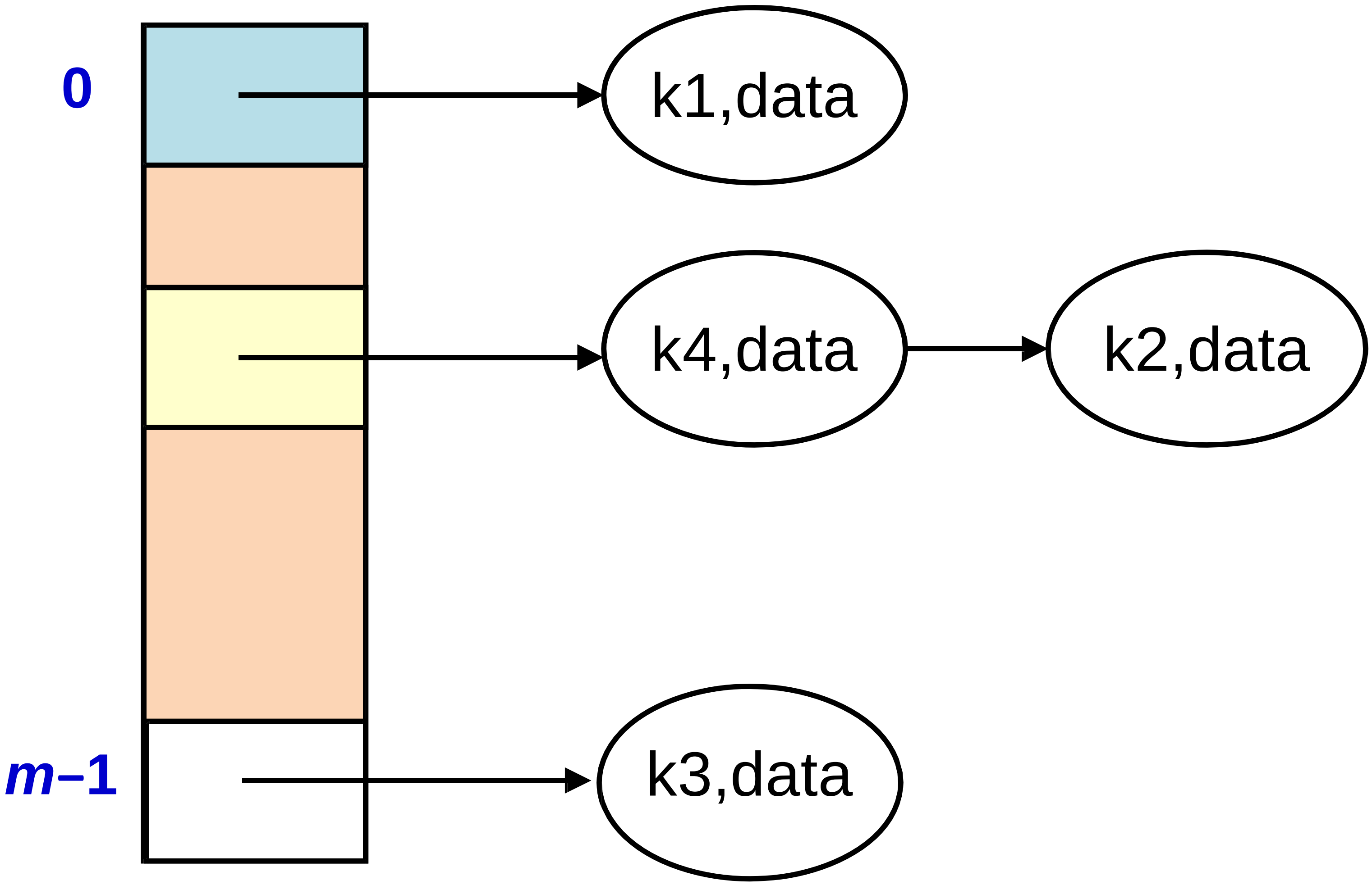
Prime Number

Resolving Collisions

Collision Resolution Techniques

- Separate Chaining
- Linear Probing
- Quadratic Probing
- Double Hashing

Separate Chaining



- **Use a linked-list to store collided keys.**
- **Always insert at the beginning (or back) of list.**

Separate Chaining: Performance

- **insert** (key, data)
  - Insert data into list a[h(key)], **Complexity**: **O(1)**
- **find** (key)
  - Find key from list a[h(key)], **Complexity**: **O(1+α)** on average
- **delete** (key)
  - Delete data from list a[h(key)], **Complexity**: **O(1+α)** on average

- **α = Number of keys / Capacity**
- **If α ≤ constant, Complexity of all three operations: O(1)**

Note: Address given to a key is fixed **=> Close addressing**

Linear Probing

- **hash(*k*) = *k* mod 7,** (i.e. table-size (m) = 7)
- **Note: 7 is prime**

0	1	2	3	4	5	6

Collision: Scan forward for next empty slot (wrapping around after reaching the last slot)

Linear Probing: Insert 25

hash(k) = k mod 7

hash (25) = 25 mod 7
= 4

0	1	2	3	4	5	6
				25		

Linear Probing: Insert 15

hash(k) = k mod 7

hash (15) = 15 mod 7
= 1

0	1	2	3	4	5	6
	15			25		

Linear Probing: Insert 1

hash(k) = k mod 7

hash (1) = 1 mod 7
= 1

0	1	2	3	4	5	6
	15	1		25		

Collision! Look for the next empty slot

Linear Probing: Insert 35

hash(k) = k mod 7

hash (35) = 35 mod 7
= 0

0	1	2	3	4	5	6
35	15	1		25		

Linear Probing: Insert 50

hash(k) = k mod 7

hash (50) = 50 mod 7
= 1

0	1	2	3	4	5	6
35	15	1	50	25		

Again Collision! Look for the next empty slot

Linear Probing: Search 50

hash(k) = k mod 7

hash (50) = 50 mod 7
= 1

0	1	2	3	4	5	6
35	15	1	50	25		

Found after 3 probes (index: 1, 2 and 3)

Linear Probing: Search 64

hash(k) = k mod 7

hash (64) = 64 mod 7
= 1

0	1	2	3	4	5	6
35	15	1	50	25		

Not Found. Required 3 probes!

Linear Probing: Delete

- Cannot simply **delete** a value, as it can affect **search**!
- **Example**: **Delete 1**
  - **hash (1) = 1 mod 7 = 1**

0	1	2	3	4	5	6
35	15		50	25		

- Now **Search (50)** will produce **wrong result**, as
  - hash (50) = 50 mod 7 **= 1**, and 50 cannot be found from 1

0	1	2	3	4	5	6
35	15		50	25		

NOT FOUND!

Correct “Delete” Operation

- Use **three** different **states** at each slot
  1. **Occupied**
  2. **Deleted**
  3. **Empty**

- **Deletion**: Mark the status of the slot as “deleted”, instead of actually emptying the slot

- Needs a **state array** of same size as the hash table

Linear Probing: Delete

- Cannot simply **delete** a value, as it can affect **search**!
- **Example**: **Delete 1**
  - **hash (1) = 1 mod 7 = 1**

0	1	2	3	4	5	6
35	15	X	50	25		

- Now **Search (50)** will produce **correct result**, as
  - hash (50) = 50 mod 7 **= 1**, and 50 can be found from 1

Linear Probing: Search and Insert

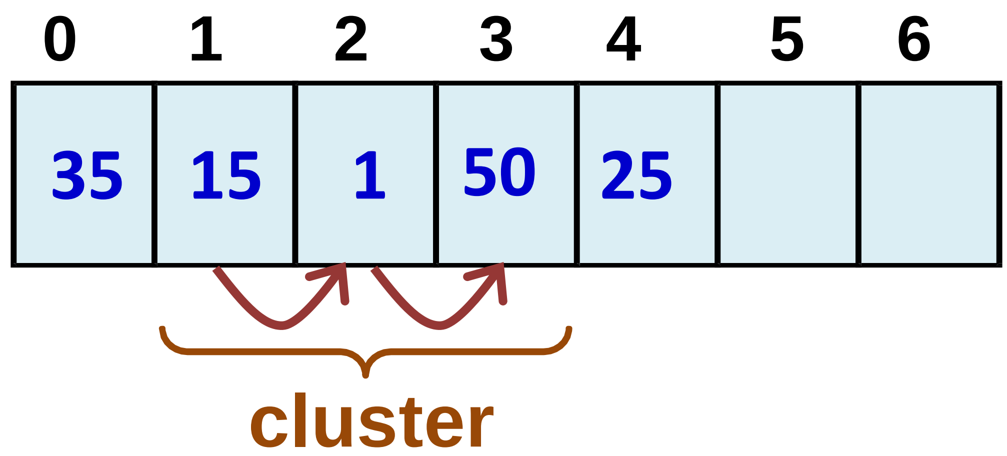
0	1	2	3	4	5	6
35	15	X	50	25		

- Slot 2 is marked as deleted.
- **Operation**: **Insert (99)**

1. **Search (99):** Check if 99 is already in the hash table
   - **Result**: **Not found**
2. Now, insert the new value (99) into the first available entry where $h(99) = 99 \bmod 7 = 1$
   - New value 99: Inserted in the slot, marked as deleted

0	1	2	3	4	5	6
35	15	99	50	25		

Problem 1: Primary Clustering



- **Cluster:** A collection of consecutive occupied slots
- **Primary Cluster**: Cluster covering home address of a key
- Linear probing can create large primary clusters that increases the running time of operations

Linear Probing: Probe Sequence

- The **probe sequence** of this linear probing is
  - hash(key)
  - (hash(key) + **1**) % *m*
  - (hash(key) + **2**) % *m*
  - (hash(key) + **3**) % *m*

    ⋮
- **Empty slot**: Sure to find it
- Conflict is resolved, but primary cluster is expanded
- Size of primary cluster: May grow up to a very large value

Improved Linear Probing

- Reduce primary clustering: Update probe sequence:
  - hash(key)
  - (hash(key) + **1 * *d***) % *m*
  - (hash(key) + **2 * *d***) % *m*
  - (hash(key) + **3 * *d***) % *m*

⋮

(d is some constant integer >1 and co-prime to *m)*

- As ***d*** and ***m*** are **co-primes**, probe sequence covers all slots in the hash table

Quadratic Probing

Probe sequence of **quadratic probing**:

- hash(key)
- (hash(key) + $\mathbf{1^2}$) % m = (hash(key) + **1**) % m
- (hash(key) + $\mathbf{2^2}$) % m = (hash(key) + **4**) % m
- (hash(key) + $\mathbf{3^2}$) % m = (hash(key) + **9**) % m

$\vdots$

- (hash(key) **+** $\mathbf{k^2}$) % m

Quadratic Probing: Insert 25, 15

Recall our example:

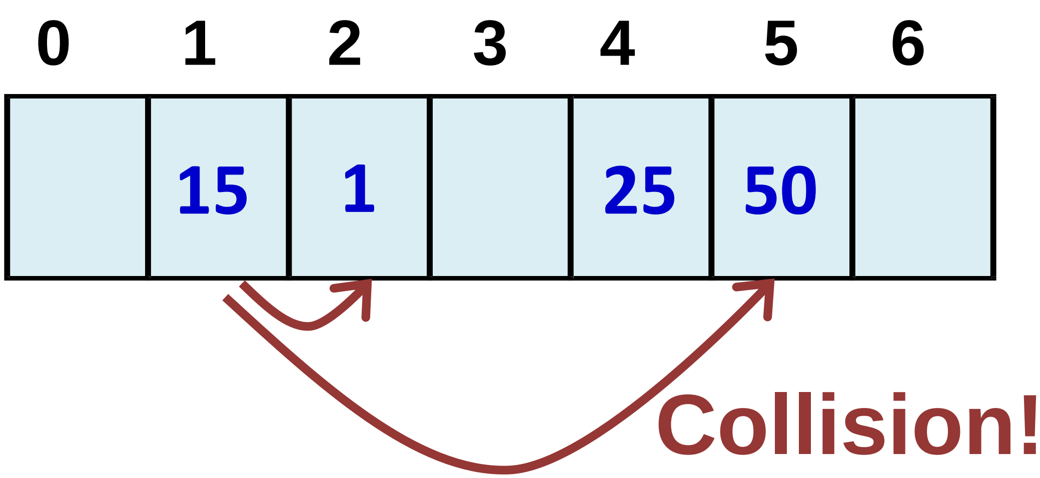
- hash(*k*) = *k* mod 7
- **hash (25) = 25 mod 7 = 4**
- **hash (15) = 15 mod 7 = 1**

0	1	2	3	4	5	6
	15			25		

Quadratic Probing: Insert 1, 50

Recall: hash(k) = k mod 7, (hash(key) + $\mathbf{k^2}$) % m

- **hash (1) = 1 mod 7 = 1 => Collision**
  - **=>** (1 + $\mathbf{1^2}$) % 7 = 2
- **hash (50) = 5 mod 7 = 1 => Collision**
  - **=>** (1 + $\mathbf{2^2}$) % 7 = **5**



Secondary Clustering: Problems

- Clusters, called **secondary clusters**, are formed along the path of probing, not around the home location
- **Reason**: Using of same pattern in probing by all keys
- Not as bad as primary clustering in linear probing

Double Hashing

- Reduce secondary clustering, by using a **second hash function** to generate different probe sequences for different keys
  - hash(key)
  - (hash(key) + **1** * $\mathbf{hash_2}$(key)) % *m*
  - (hash(key) + **2** * $\mathbf{hash_2}$(key)) % *m*
  - (hash(key) + **3** * $\mathbf{hash_2}$(key)) % *m*

    ⋮
- $\mathbf{hash_2}$: Secondary hash function
- What happens when $hash_2(k) = 1$?
  - Answer: Linear probing

Double Hashing: Insert 25, 15

Recall our example:

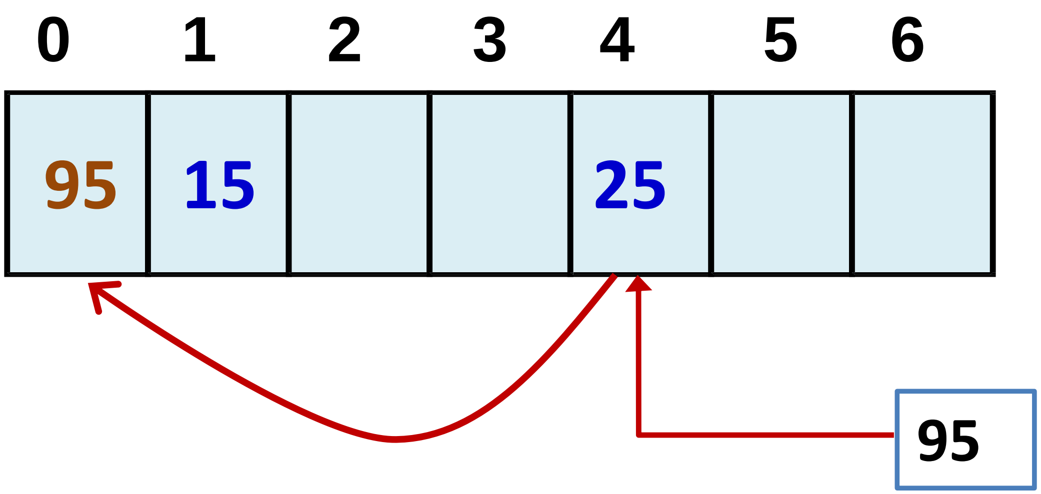
- hash(*k*) = *k* mod 7
  - **hash (25) = 25 mod 7 = 4**
  - **hash (15) = 15 mod 7 = 1**
  - **No collision: $hash_2$ is not needed**

0	1	2	3	4	5	6
	15			25		

Double Hashing: Insert 95

Recall: hash(k) = k mod 7, hash$_2$(k) = k mod 5

- **hash (95) = 95 mod 7 = 4 => Collision**
  - **=> hash$_2$ (95) = 95 mod 5 = 0**



Thank you!