[SWE2015-41] Introduction to Data Structures (자료구조개론)

Hash Tables

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

Motivation

- Many databases often store **data information (values)** with their own **unique ID formats (keys)**
  - **Key → Value**
  - Mail Address → Email (e.g., hankook.lee@skku.edu)
  - URL → Web Page (e.g., https://skku.edu)
  - Student ID → Student Information (e.g., 202412345)
  - Social Security Number (SSN) → Citizen (e.g., 240612-0000000)

(Q) Which operations are necessary for such databases?

- **Existence** - check whether a key exists in the database
- **Retrieval** - extract data information associated with a key
- **Update** - insert, delete, or update data information associated with a key

Motivation

(Q) Which data structure is useful for such a database of key-value pairs?

- Example: A database of sports players (for simplicity, all their birthdays are distinct)

Birthday (key)	Name (value)
900905	Yuna Kim
880226	Yeon Koung Kim
920708	Heung Min Son
810225	Jisung Park
010219	Kangin Lee
960507	Sanghyeok Lee





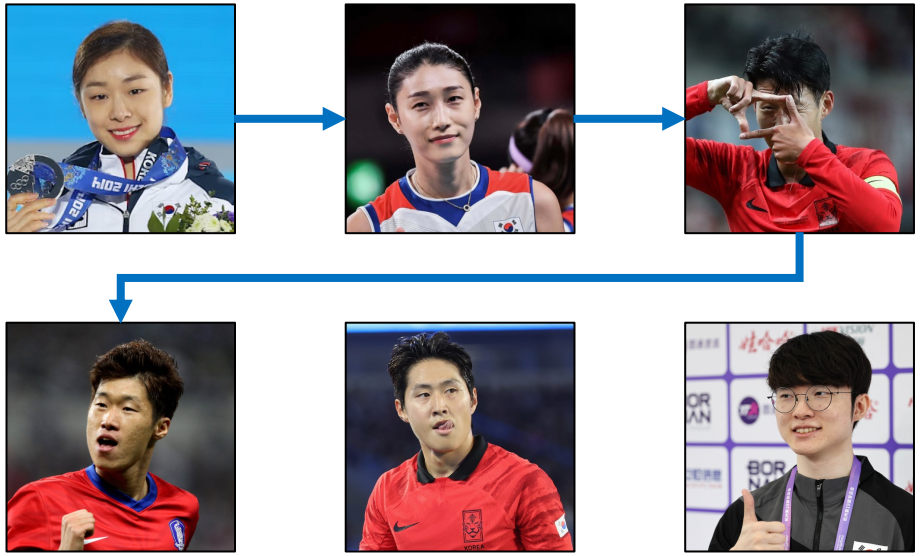

Motivation

(Q) Which data structure is useful for such a database of key-value pairs?

- Example: A database of sports players (for simplicity, all their birthdays are distinct)

(Option 1) Use an array or a linked list

index	Birthday (key)	Name (value)
1	900905	Yuna Kim
2	880226	Yeon Koung Kim
3	920708	Heung Min Son
4	810225	Jisung Park
5	010219	Kangin Lee
6	960507	Sanghyeok Lee



(Q) How to find a player whose birthday is `810025`? **(A)** Sequential search, O(N)

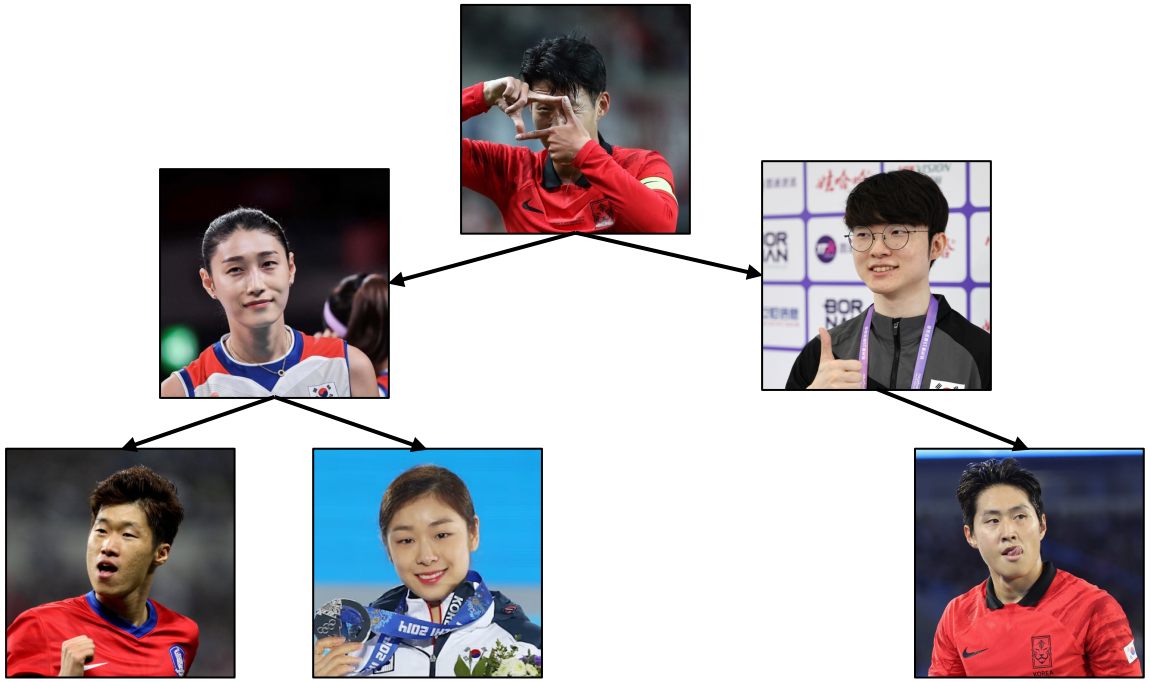
Motivation

(Q) Which data structure is useful for such a database of key-value pairs?

- Example: A database of sports players (for simplicity, all their birthdays are distinct)

(Option 2) Use a balanced BST (e.g., AVL/Red-Black Trees)

Birthday (key)	Name (value)
900905	Yuna Kim
880226	Yeon Koung Kim
920708	Heung Min Son
810225	Jisung Park
010219	Kangin Lee
960507	Sanghyeok Lee



(Q) How to find a player whose birthday is `810025`?

(A) Tree search, O(log N)

Motivation

(Q) Which data structure is useful for such a database of key-value pairs?

- Example: A database of sports players (for simplicity, all their birthdays are distinct)

(Option 3) Can we do better than BSTs?

Birthday (key)	Name (value)
900905	Yuna Kim
880226	Yeon Koung Kim
920708	Heung Min Son
810225	Jisung Park
010219	Kangin Lee
960507	Sanghyeok Lee

Advantage of BSTs:
BSTs support advanced search operations: minimum, maximum, predecessor, successor, and smallest/largest k-th items

If we just need **exact-match** search, then **BSTs** might be a bad choice

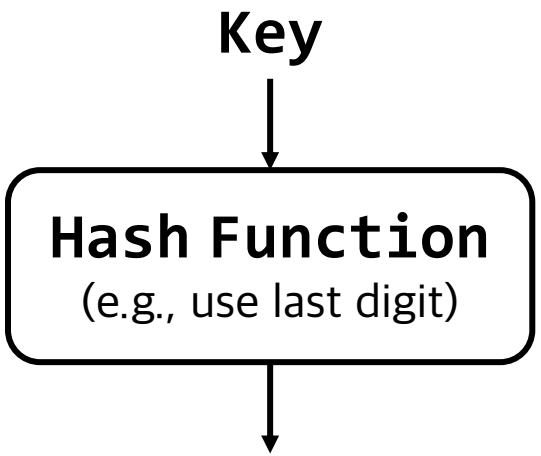
Motivation

(Q) Which data structure is useful for such a database of key-value pairs?

- Example: A database of sports players (for simplicity, all their birthdays are distinct)

(Option 3) Use **Hash Tables**

- Support O(1) search!



index	Data
0	NULL
1	NULL
2	NULL
3	NULL
4	NULL
5	(900905, Yuna Kim) (810225, Jisung Park)
6	(880226, Yeon Koung Kim)
7	(960507, Sanghyeok Lee)
8	(920708, Heung Min Son)
9	NULL

What is Hash Table?

- A hash table consists of two components: Table & Hash Function
  - **Table** is an array of a fixed size (each item is often called a **bucket** or a **slot**)
  - **Hash Function** $h(\cdot)$ converts an arbitrary key to some array index

Keys
900905
880226
920708
810225
010219
960507

Hash Function

index	Data
0	NULL
...	NULL
4	NULL
5	(900905, Yuna Kim) (810225, Jisung Park)
6	(880226, Yeon Koung Kim)
7	(960507, Sanghyeok Lee)
8	(920708, Heung Min Son)
9	NULL

Table (or **Buckets**)

Hash Functions

- Example: `[112, 23, 64, 205, 99, 47, 8, 31]`
  - Hash function: $h(x) = x \; mod \; 10$

Hash Functions

- Example: `[112, 23, 64, 205, 99, 47, 8, 31]`
  - Hash function: $h(x) = x \; mod \; 10$
  - We have 10 buckets because $h(x)$ produces 10 numbers

Index	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions

- Example: `[112, 23, 64, 205, 99, 47, 8, 31]`
  - Hash function: $h(x) = x \; mod \; 10$
  - We have 10 buckets because $h(x)$ produces 10 numbers
  - Put a number into the bucket whose index is $h(x)$

```
buckets[h(x)] ← x
```

Index	Data
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Hash Functions

- Example: `[112, 23, 64, 205, 99, 47, 8, 31]`
  - Hash function: $h(x) = x \; mod \; 10$
  - We have 10 buckets because $h(x)$ produces 10 numbers
  - Put a number into the bucket whose index is $h(x)$

    `buckets[h(x)] ← x`

    - `buckets[h(112)] = buckets[2] ← 112`

Index	Data
0	
1	
2	112
3	
4	
5	
6	
7	
8	
9	

Hash Functions

- Example: `[112, 23, 64, 205, 99, 47, 8, 31]`
  - Hash function: $h(x) = x \; mod \; 10$
  - We have 10 buckets because $h(x)$ produces 10 numbers
  - Put a number into the bucket whose index is $h(x)$

    `buckets[h(x)] ← x`

    - `buckets[h(112)] = buckets[2] ← 112`
    - `buckets[h(23)]  = buckets[3] ← 23`

Index	Data
0	
1	
2	112
3	23
4	
5	
6	
7	
8	
9	

Hash Functions

- Example: `[112, 23, 64, 205, 99, 47, 8, 31]`
  - Hash function: $h(x) = x \; mod \; 10$
  - We have 10 buckets because $h(x)$ produces 10 numbers
  - Put a number into the bucket whose index is $h(x)$

    `buckets[h(x)] ← x`

    - `buckets[h(112)] = buckets[2] ← 112`
    - `buckets[h(23)]  = buckets[3] ← 23`
    - `...`

Index	Data
0	
1	31
2	112
3	23
4	64
5	205
6	
7	47
8	8
9	99

Hash Functions

- Example: `[112, 23, 64, 205, 99, 47, 8, 31]`
  - Hash function: $h(x) = x \; mod \; 10$

(Q1) Is there 981?
(A1) No. `h(981)=1`, but 981 does not exist in `buckets[1]`

(Q2) Is there 99?
(A2) Yes. `h(99)=9`, and 99 exists in `buckets[9]`

(Q3) Time complexity for checking whether `x` exists or not
(A3) O(1)
  - Evaluate `h(x)` - O(1)
  - Look at `buckets[h(x)]` - O(1)

Index	Data
0	
1	31
2	112
3	23
4	64
5	205
6	
7	47
8	8
9	99

Hash Functions - Collision

- Example: [112, 23, 64, 205, 99, 47, 8, 31, 94]
  - Hash function: $h(x) = x \; mod \; 10$

- **Collision**: two keys map to the same index
  - Given a hash function $h(x)$, two different keys x and y may have the same value $h(x) = h(y)$
  - Example: h(64) = h(94) = 4

- How to solve the problem of collision?
  - Open Addressing
  - Chaining

Index	Data
0	
1	31
2	112
3	23
4	64, 94
5	205
6	
7	47
8	8
9	99

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	81	72	63	24		36	27		

If we want to insert 92 …

1. $h_0(92) = (92 + 0)\ mod\ 10 = 2 \rightarrow$ `buckets[2]` is full
2. $h_1(92) = (92 + 1)\ mod\ 10 = 3 \rightarrow$ `buckets[3]` is full
3. $h_2(92) = (92 + 2)\ mod\ 10 = 4 \rightarrow$ `buckets[4]` is full
4. $h_3(92) = (92 + 3)\ mod\ 10 = 5 \rightarrow$ `buckets[5]` is empty

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	92	36	27		

If we want to insert 92 …

1. $h_0(92) = (92 + 0)\ mod\ 10 = 2 \rightarrow$ `buckets[2]` is full
2. $h_1(92) = (92 + 1)\ mod\ 10 = 3 \rightarrow$ `buckets[3]` is full
3. $h_2(92) = (92 + 2)\ mod\ 10 = 4 \rightarrow$ `buckets[4]` is full
4. $h_3(92) = (92 + 3)\ mod\ 10 = 5 \rightarrow$ `buckets[5]` is empty $\rightarrow$ insert 92 at index 5

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	92	36	27		

If we want to insert 101 …

1. $h_0(101) = (101 + 0)\ mod\ 10 = 1 \rightarrow$ `buckets[1]` is full
2. …
3. $h_6(101) = (101 + 6)\ mod\ 10 = 7 \rightarrow$ `buckets[7]` is full
4. $h_7(101) = (101 + 7)\ mod\ 10 = 8 \rightarrow$ `buckets[8]` is empty

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	81	72	63	24	92	36	27	101	

If we want to insert 101 …

1. $h_0(101) = (101 + 0)\ mod\ 10 = 1$ → `buckets[1]` is full
2. …
3. $h_6(101) = (101 + 6)\ mod\ 10 = 7$ → `buckets[7]` is full
4. $h_7(101) = (101 + 7)\ mod\ 10 = 8$ → `buckets[8]` is empty → insert 101 at 8

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
					25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
			3		25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1		3		25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1	31	3		25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1	31	3	11	25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1	31	3	11	25	62			

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1	31	3	11	25	62	51		

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Linear Probing**: move to the **next** bucket
  - $h_i(x) = (h(x) + i)\ mod\ 10 = (x + i)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1	31	3	11	25	62	51		

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

- Inefficient search → its cost could be O(N) …
  - Buckets are **linearly** searched
  - Elements may **not be scattered**
  - Elements are **easily merged**

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Quadratic Probing**: move to the **far** bucket
  - $h_i(x) = (h(x) + i^2)\ mod\ 10 = (x + i^2)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Quadratic Probing**: move to the **far** bucket
  - $h_i(x) = (h(x) + i^2)\ mod\ 10 = (x + i^2)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
					25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h_0(x) = 5$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Quadratic Probing**: move to the **far** bucket
  - $h_i(x) = (h(x) + i^2)\ mod\ 10 = (x + i^2)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
			3		25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h_0(x) = 3$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Quadratic Probing**: move to the **far** bucket
  - $h_i(x) = (h(x) + i^2)\ mod\ 10 = (x + i^2)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1		3		25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h_0(x) = 1$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Quadratic Probing**: move to the **far** bucket
  - $h_i(x) = (h(x) + i^2)\ mod\ 10 = (x + i^2)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1	31	3		25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h_0(x) = 1$
    - $h_1(x) = 2$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Quadratic Probing**: move to the **far** bucket
  - $h_i(x) = (h(x) + i^2)\ mod\ 10 = (x + i^2)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
11	1	31	3		25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h_0(x) = 1$
    - $h_1(x) = 2$
    - $h_2(x) = 5$
    - $h_3(x) = (1 + 9)\ mod\ 10 = 0$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Quadratic Probing**: move to the **far** bucket
  - $h_i(x) = (h(x) + i^2)\ mod\ 10 = (x + i^2)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
11	1	31	3		25	62			

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h_0(x) = 2$
    - $h_1(x) = 3$
    - $h_2(x) = 6$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Quadratic Probing**: move to the **far** bucket
  - $h_i(x) = (h(x) + i^2)\ mod\ 10 = (x + i^2)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
11	1	31	3		25	62	51		

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h_0(x) = 1$
    - $h_1(x) = 2$
    - $h_2(x) = 5$
    - $h_3(x) = (1 + 9)\ mod\ 10 = 0$
    - $h_4(x) = (1 + 16)\ mod\ 10 = 7$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Double Hashing**: use **another hash** function
  - $h_i(x) = \left(h(x) + i \times h'(x)\right) mod\ 10 = \left(x + i \times \left(7 - (x\ mod\ 7)\right)\right)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Double Hashing**: use **another hash** function
  - $h_i(x) = \left(h(x) + i \times h'(x)\right) mod\ 10 = \left(x + i \times \left(7 - (x\ mod\ 7)\right)\right)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1		3		25				

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Double Hashing**: use **another hash** function
  - $h_i(x) = \left(h(x) + i \times h'(x)\right) mod\ 10 = \left(x + i \times \left(7 - (x\ mod\ 7)\right)\right)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1		3		25				31

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h'(x) = 7 - (31\ mod\ 7) = 4$
    - $h_0(x) = 1$
    - $h_1(x) = 5$
    - $h_2(x) = 9$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Double Hashing**: use **another hash** function
  - $h_i(x) = \left(h(x) + i \times h'(x)\right) mod\ 10 = \left(x + i \times \left(7 - (x\ mod\ 7)\right)\right)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1		3	11	25				31

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h'(x) = 7 - (11\ mod\ 7) = 3$
    - $h_0(x) = 1$
    - $h_1(x) = 4$

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Double Hashing**: use **another hash** function
  - $h_i(x) = \left(h(x) + i \times h'(x)\right) mod\ 10 = \left(x + i \times \left(7 - (x\ mod\ 7)\right)\right)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1	62	3	11	25				31

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

Hash Functions - Open Addressing

- **Idea:** If the bucket is **full**, find another **empty** bucket iteratively
- **Double Hashing**: use **another hash** function
  - $h_i(x) = \left(h(x) + i \times h'(x)\right) mod\ 10 = \left(x + i \times \left(7 - (x\ mod\ 7)\right)\right)\ mod\ 10$

0	1	2	3	4	5	6	7	8	9
	1	62	3	11	25	51			31

  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …
    - $h'(x) = 7 - (51\ mod\ 7) = 5$
    - $h_0(x) = 1$
    - $h_1(x) = 6$

Hash Functions - Chaining

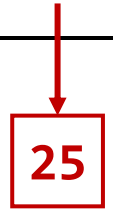
- **Idea:** Maintain a **linked list** for each bucket
  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

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

Hash Functions - Chaining

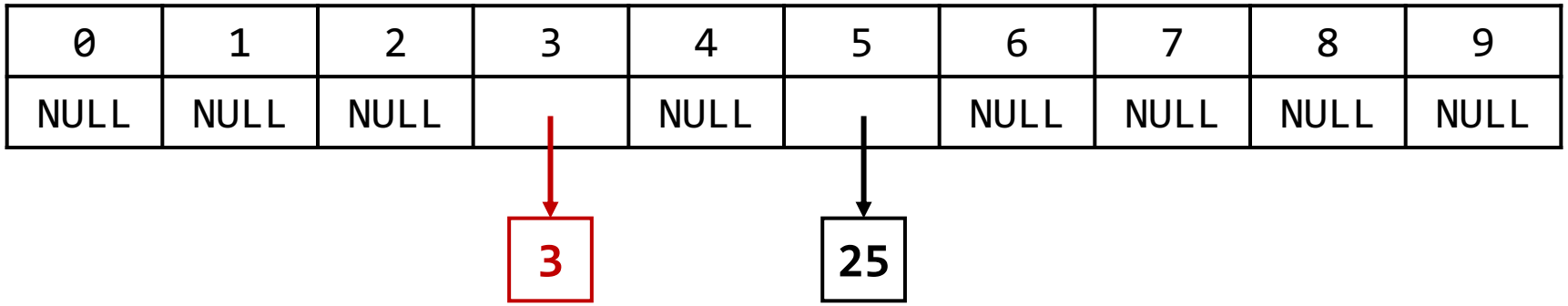
- **Idea:** Maintain a **linked list** for each bucket
  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …

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



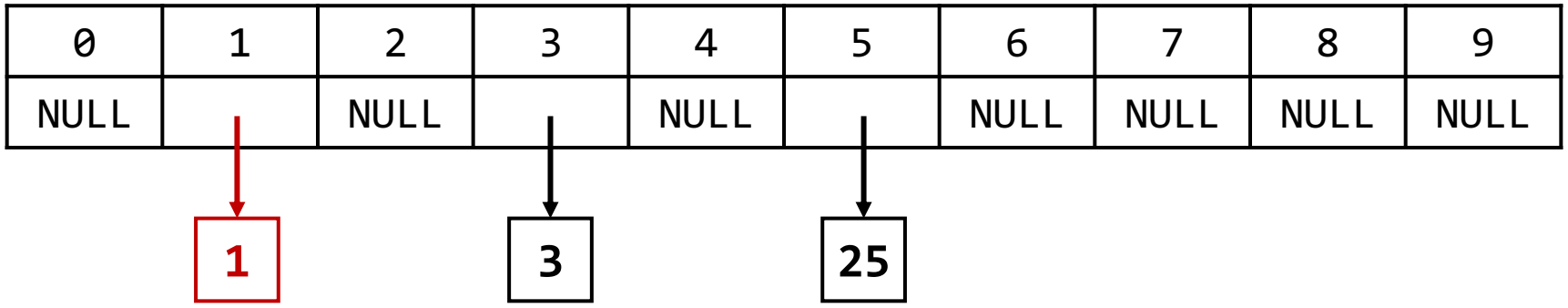
Hash Functions - Chaining

- **Idea:** Maintain a **linked list** for each bucket
  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …



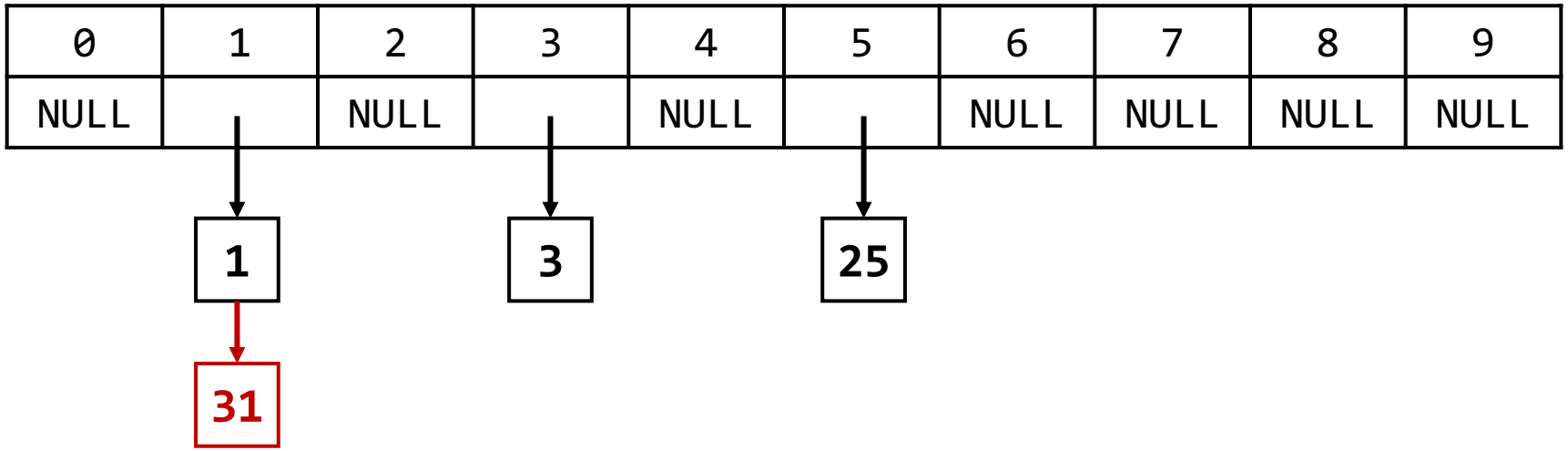
Hash Functions - Chaining

- **Idea:** Maintain a **linked list** for each bucket
  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …



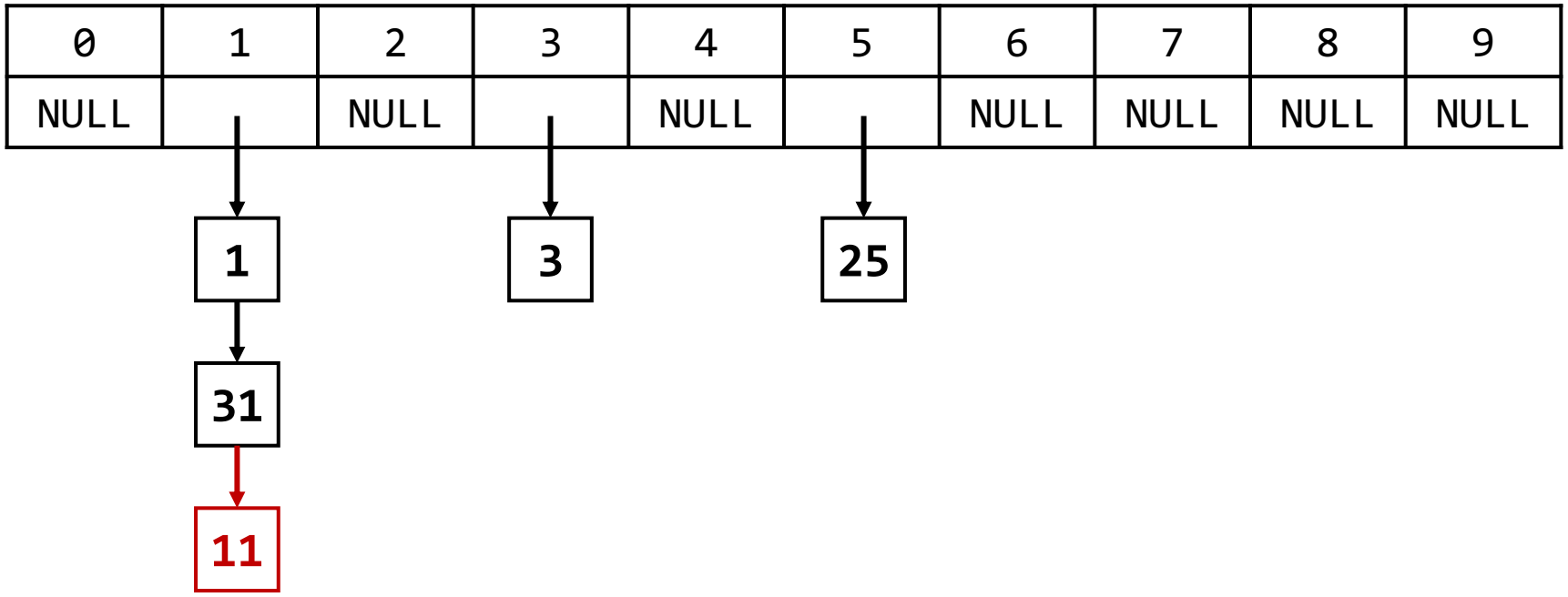
Hash Functions - Chaining

- **Idea:** Maintain a **linked list** for each bucket
  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …



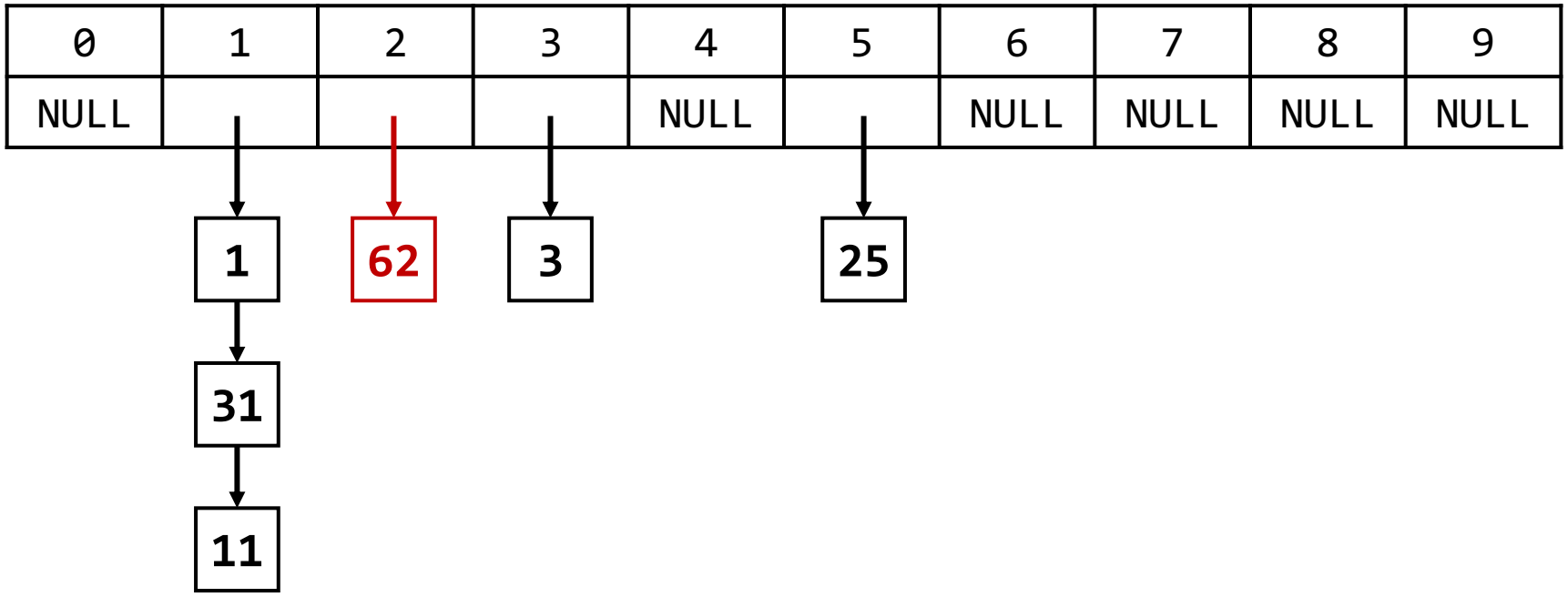
Hash Functions - Chaining

- **Idea:** Maintain a **linked list** for each bucket
  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …



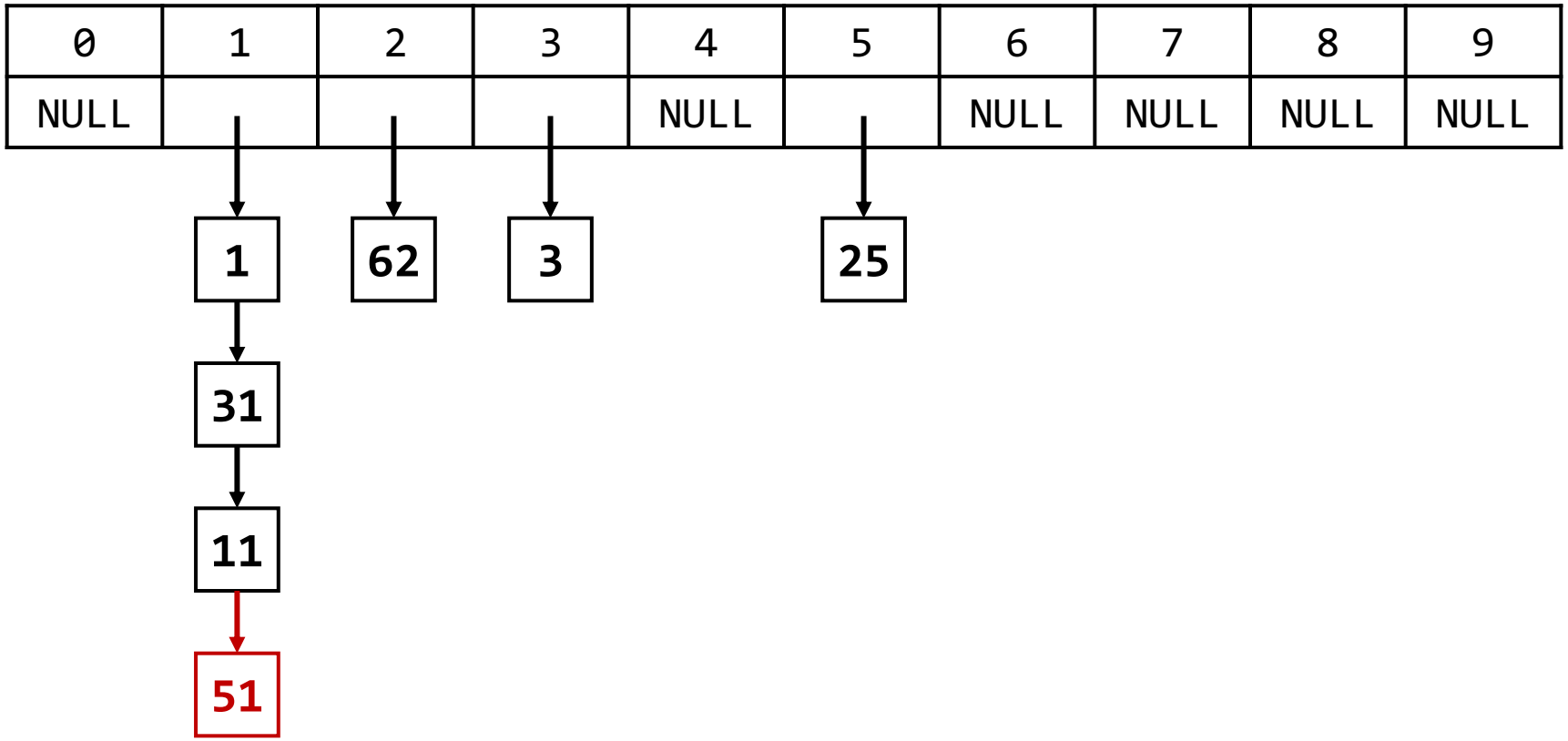
Hash Functions - Chaining

- **Idea:** Maintain a **linked list** for each bucket
  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …



Hash Functions - Chaining

- **Idea:** Maintain a **linked list** for each bucket
  - Example: 25, 3, 1, 31, 11, 62, 51 are sequentially added in this hash table …



What is a Good Hash Function?

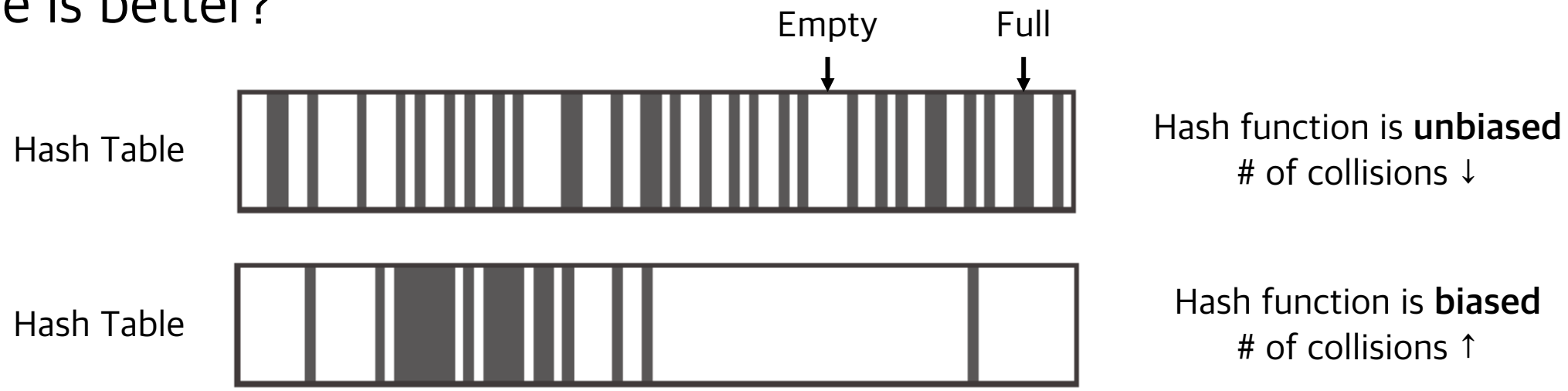
- Which one is better?

Hash function is **unbiased**
# of collisions ↓

Hash function is **biased**
# of collisions ↑

What is a Good Hash Function?

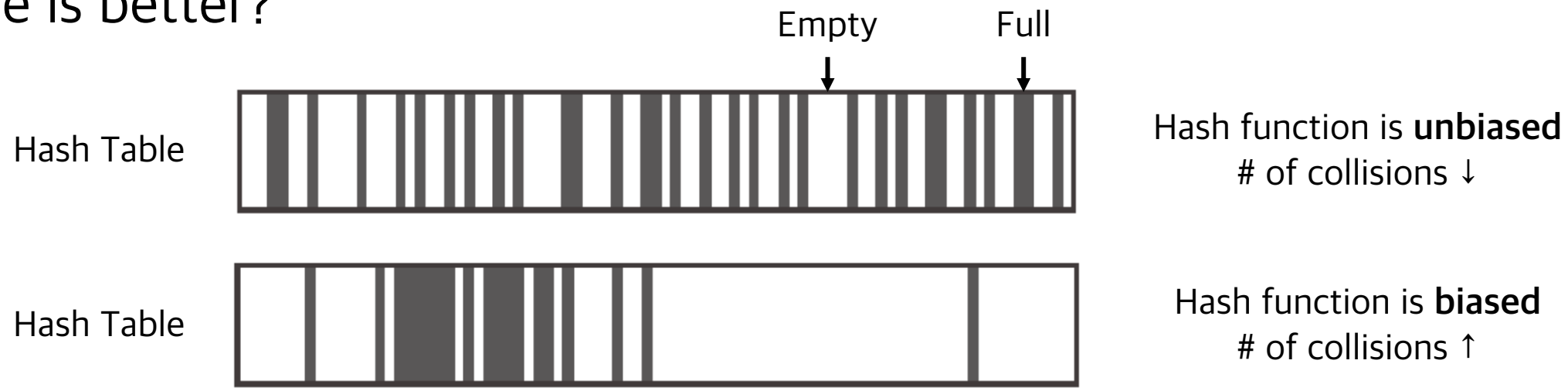
- Which one is better?



- Properties of a Good Hash Function
  - **Low-cost** - The cost of executing a hash function must be small
  - **Determinism** - A hash function must be deterministic
  - **Uniformity** - A good hash function must map the keys as evenly as possible over its output range
  - **All bits in data** should be utilized

What is a Good Hash Function?

- Which one is better?



Hash function is **unbiased**
# of collisions ↓

Hash function is **biased**
# of collisions ↑

- Properties of a Good Hash Function
  - **Low-cost** - The cost of executing a hash function must be small
  - **Determinism** - A hash function must be deterministic
  - **Uniformity** - A good hash function must map the keys as evenly as possible over its output range → This minimizes the number of collisions
  - **All bits in data** should be utilized → This minimizes the number of collisions

Different Hash Functions

- **Division Method:** $h(x) = x \; mod \; M$ where M is the size of a hash table
  - How to select a suitable value for M?
    - Choose M as a **prime number**
    - This increases likelihood that the keys are mapped with uniformity in the output range of values

    - Example: if x is a string and M=2^8 (# of ASCII characters), then $h(x)$ use only the last character
    - `"ABC" mod M = [A(65)*256² + B(66)*256 + C(67)] mod 256 = 67`
    - `"ZYC" mod M = [Z(90)*256² + Y(89)*256 + C(67)] mod 256 = 67`

Different Hash Functions

- **Mid-Square (middle of square):** $h(x)$ take $\boldsymbol{r}$ bits from the middle of $x * x$
  - The size of this hash table = 2^r

x	$x * x$	Binary number of $x * x$	7, 8, 9 bits	$h(x)$
8	64	000000000001000000	001	**1**
612	374544	101101101100010000	100	**4**
13	169	000000000010101001	010	**2**
15	225	000000000011100001	011	**3**
235	55225	000110101110111001	110	**6**

- Why does this algorithm work well?
  - The resultant index $h(x)$ is not dominated by the distribution of the bottom digit or the top digit of the original key value

Different Hash Functions

- **Folding**
  1. Divide the key value into a number of parts: $k \rightarrow [k_1, k_2, \ldots, k_n]$ where each part has the same number of digits except the last part
  2. Add the individual parts: $h(k) = k_1 + k_2 + \cdots + k_n$
     - The hash value is produced by ignoring the last carry from the sum of the parts

- **Example**: Given a hash table of 100 locations,

Key	Parts	Sum	Hash Value
5678	56 / 78	134	34
321	32 / 1	33	33
34567	34 / 56 / 7	97	97

Different Hash Functions

- **Folding**
  1. Divide the key value into a number of parts: $k \rightarrow [k_1, k_2, \ldots, k_n]$ where each part has the same number of digits except the last part
  2. Add the individual parts: $h(k) = k_1 + k_2 + \cdots + k_n$
     - The hash value is produced by ignoring the last carry from the sum of the parts

- **Example**: Given a hash table of 1000 locations,

Key	Parts	Sum	Hash Value
56789	567 / 78	645	645
3210	321 / 0	321	321
345678	345 / 678	1023	23

Comparison between Search Algorithms

- There are various data structures for search

Method	Search	Insertion	Deletion
Linear Search	O(N)	O(N)	O(N)
BSTs	O(N)	O(N)	O(N)
Balanced BSTs	O(logN)	O(logN)	O(logN)
Hash Table (Best)	O(1)	O(1)	O(1)
Hash Table (Worst)	O(N)	O(N)	O(N)

- In general, hashing is the best way for search
  - The hash function should be carefully designed for efficiency

Any Questions?