

빅 데이터 혁신 공유 대학

# 파이썬으로 배우는 데이터 구조

---

한동대학교 전산전자공학부

김영섭 교수



교육부



한국연구재단



# Data Structures in Python

## Chapter 6

- Hash Table
- **Collision Resolutions**
- Double Hashing & Rehashing
- Hash Implementation

# Agenda & Readings

---

- Collision Resolution
  - Separate chaining
  - Open addressing
    - Linear Probing
    - Quadratic Probing
    - Double Hashing
- Reference:
  - Problem Solving with Algorithms and Data Structures
  - Chapter 5 - Hashing

# Collision Resolution

---

- **Perfect hash functions** are hard to come by, especially if you do not know the input keys beforehand.
- If multiple keys map to the same hash value this is called **collision**.
  - For non-perfect hash functions we need systematic way to handle collisions.
- Handling collisions systematically is required - collision resolution.

# Collision Resolution

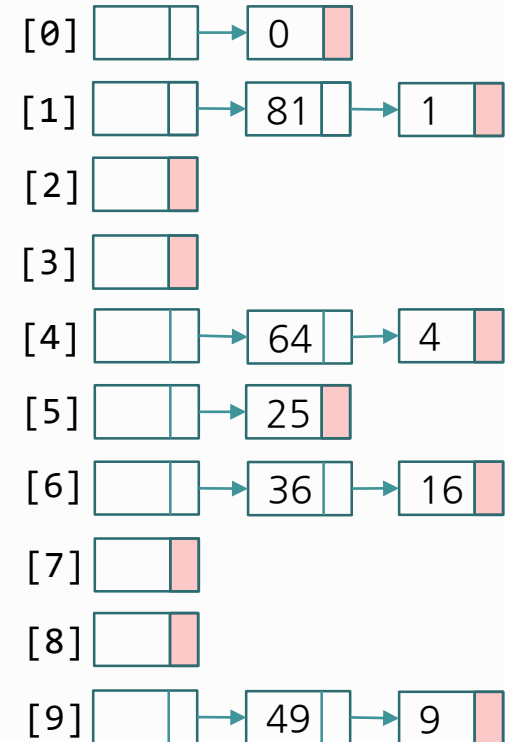
---

- Collision resolution methods
  - **Chaining** - Store colliding keys in a linked list at the same hash table index
  - **Open addressing** - Store colliding keys elsewhere in the table
    - Linear Probing
    - Quadratic Probing
    - Double hashing.

# Collision Resolution by Chaining

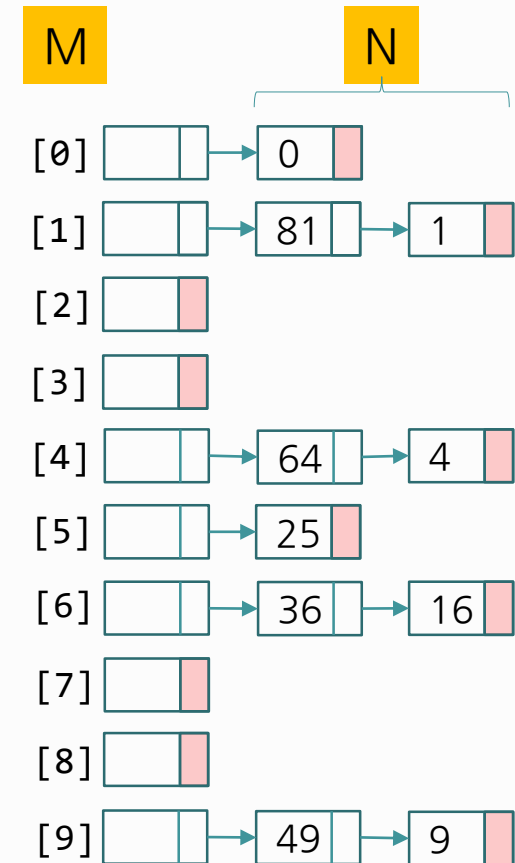
- Maintains a linked list at every hash index for collided elements
  - Hash table T is a vector of linked lists.
    - Insert element at the head (as shown here) or at the tail.
  - Key k is stored in list a  $\text{HashTable}[h(k)]$
  - For example,
    - $\text{TableSize} = 10$
    - $h(k) = k \% 10$
    - Insert first 10 perfect squares

Insertion sequence:  
{ 0 1 4 9 16 25 36 49 64 81 }



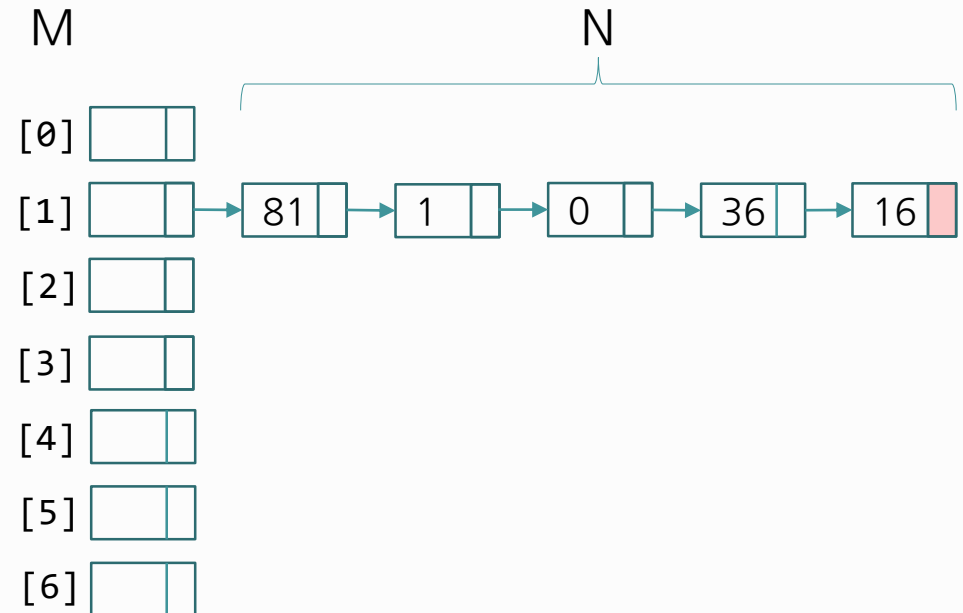
# Collision Resolution by Chaining

- Load factor  $\lambda$  of a hash table T is defined as follows:
  - Element size:  $N$  = number of elements in T
  - Table size:  $M$  = size of T
  - Load factor:  $\lambda = N/M$  (적재율)
    - i.e.,  $\lambda$  is the average length of a chain
- Unsuccessful search time:  $O(\lambda)$ 
  - Same for insert time
- Successful search time average:  $O(\lambda/2)$
- Ideally, want  $\lambda \leq 1$  (then, not a function of N)



# Collision Resolution by Chaining

- Potential disadvantages of Chaining
  - Linked lists could get long
    - Especially when  $N \gg M$
    - Longer linked lists could negatively impact performance
  - More memory because of pointers
  - Absolute worst-case (even if  $N \ll M$ ):
    - All  $N$  elements in one linked list!  
Typically the result of a bad hash function





# Collision Resolution by Open Addressing

---

1. Linear Probing (선형조사법)
2. Quadratic Probing (이차조사법)
3. Double Hashing (이중해싱법)

# Collision Resolution by Open Addressing

---

- When a collision occurs, look elsewhere in the table for an empty slot.
- Advantages over chaining
  - No need for list structures
  - No need to allocate/deallocate memory during insertion/deletion (slow)
- Disadvantages
  - Slower insertion - May need several attempts to find an empty slot.
  - Table needs to be bigger (than chaining-based table) to achieve average-case constant-time performance.
    - Load factor  $\lambda \approx 0.5$

# Collision Resolution by Open Addressing

- A "**Probe sequence**" is a sequence of slots in hash table while searching for an element  $k$ 
  - $h_0(k), h_1(k), h_2(k), \dots$
  - Needs to visit each slot exactly once
  - Needs to be repeatable (so we can find/delete what we've inserted before)
- Hash function
  - **$h_i(k) = (h(k) + f(i)) \% \text{TableSize}$**
  - $f(0) = 0 \rightarrow$  position for the 0<sup>th</sup> probe
  - **$f(i)$**  is "the distance to be traveled relative to the 0<sup>th</sup> probe position, during the  **$i^{\text{th}}$  probe**". It can be linear, quadratic etc.

# Collision Resolution by Open Addressing - Linear Probing

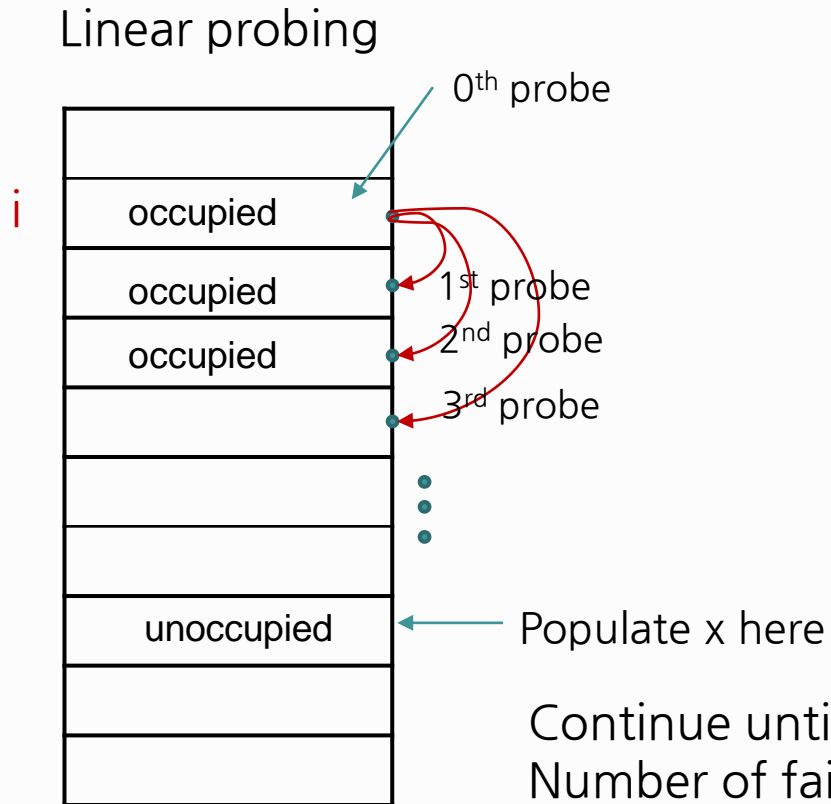
- $f(i)$  = is a linear function of  $i$ , e.g.,  $f(i) = i$

$$h_i(k) = (h(k) + i) \% \text{TableSize}$$

$i^{\text{th}}$  probe index       $0^{\text{th}}$  probe index       $f(i)$

Probe sequence:  $+0, +1, +2, +3, +4, \dots$

linear



Continue until an empty slot is found

Number of failed probes is a measure of performance

# Collision Resolution Example - Linear Probing

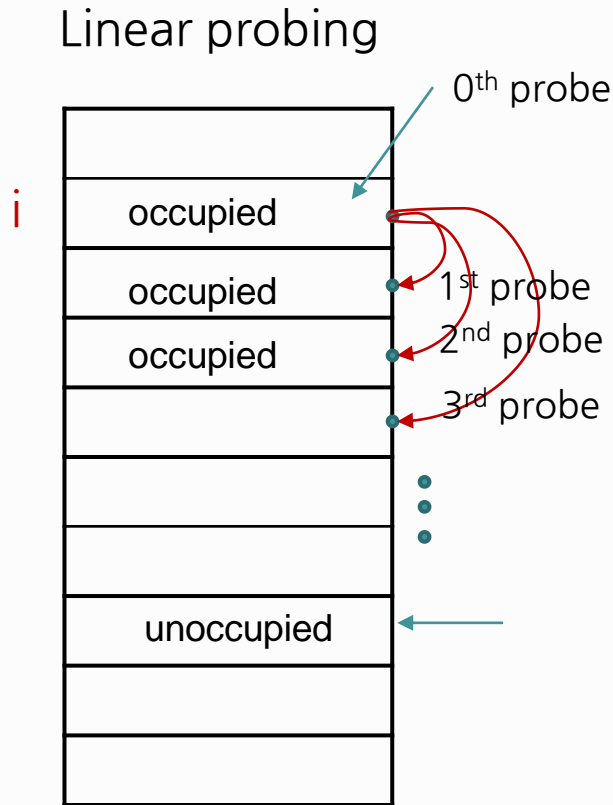
- $f(i)$  = is a **linear** function of  $i$ , e.g.,  $f(i) = i$

$$h_i(k) = (h(k) + i) \% \text{TableSize}$$

$i^{\text{th}}$  probe index       $0^{\text{th}}$  probe index       $f(i)$

Probe sequence:  $+0, +1, +2, +3, +4, \dots$

linear



- Example:  $h(k) = k \% \text{TableSize}$ 
  - $h_0(89) = (h(89) + 0) \% 10 = 9$
  - $h_0(18) = (h(18) + 0) \% 10 = 8$
  - $h_0(49) = (h(49) + 0) \% 10 = 9$  (collision)  
 $h_1(49) = (h(49) + 1) \% 10 = (h(49) + 1) \% 10 = 0$

# Collision Resolution Example - Linear Probing

Insert sequence: 8, 1, 9, 6, 15

$$h(k) = k \% 7$$

	Empty Table	After 8	After 1	After 9	After 6	After 15
0						
1		8	8			
2			1			
3						
4						
5						
6						

$h_0(8) = 8 \% 7 = 1$

$h_0(1) = 1 \% 7 = 1$

$h_1(1) = (h(1)+1) \% 7 = 2$

# Collision Resolution Example - Linear Probing

Insert sequence: **8, 1, 9, 6, 15**

$$h(k) = k \% 7$$

	Empty Table	After 8	After 1	After 9	After 6	After 15
0						
1		8	8	8	8	
2			1	1	1	
3				9	9	
4						
5						
6					6	

$h_0(8) = 8 \% 7 = 1$

$h_0(1) = 1 \% 7 = 1$   
 $h_1(1) = (h(1)+1) \% 7 = 2$

$h_0(9) = 9 \% 7 = 2$   
 $h_1(9) = (h(9)+1) \% 7 = 3$

$h_0(6) = 6 \% 7 = 6$

# Collision Resolution Example - Linear Probing

Insert sequence: 8, 1, 9, 6, 15

$$h(k) = k \% 7$$

	Empty Table	After 8	After 1	After 9	After 6	After 15
0						
1		8	8	8	8	8
2			1	1	1	1
3				9	9	9
4						15
5						
6					6	6

$$h_0(8) = 8 \% 7 = 1$$

$$h_0(1) = 1 \% 7 = 1$$
$$h_1(1) = (h(1)+1) \% 7 = 2$$

$$h_0(9) = 9 \% 7 = 2$$
$$h_1(9) = (h(9)+1) \% 7 = 3$$

$$h_0(6) = 6 \% 7 = 6$$

$$h_0(15) = (h(15) + 0) \% 7 = 1 \text{ (collision)}$$
$$h_1(15) = (h(15) + 1) \% 7 = 2 \text{ (collision)}$$
$$h_2(15) = (h(15) + 2) \% 7 = 3 \text{ (collision)}$$
$$h_3(15) = (h(15) + 3) \% 7 = 4$$

probe sequence  
and it is linear



# Collision Resolution Exercise - Linear Probing

Insert sequence: 89, 18, 49, 58, 69

$$h(k) = k \% 10$$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	
1						
2						
3						
4						
5						
6						
7						
8			18	18	18	
9		89	89	89	89	
Unsuccessful no. of probes		0	0	1	3	3

For example, linear probing for 58

$h_0(58) = (h(58) + f(0)) \% 10$   
 $= (8 + 0) \% 10 = 8$  (collision)

$h_1(58) = (h(58) + 1) \% 10 = 9$  (collision)

$h_2(58) = (h(58) + 2) \% 10 = 0$  (collision)

$h_3(58) = (h(58) + 3) \% 10 = 1$

probe sequence

# Collision Resolution Exercise - Linear Probing

Insert sequence: 89, 18, 49, 58, 69

$$h(k) = k \% 10$$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89
Unsuccessful no. of probes		0	0	1	3	3

probe sequence

For example, linear probing for 58  
 $h_0(58) = (h(58) + f(0)) \% 10$   
 $= (8 + 0) \% 10 = 8$  (collision)  
 $h_1(58) = (h(58) + 1) \% 10 = 9$  (collision)  
 $h_2(58) = (h(58) + 2) \% 10 = 0$  (collision)  
 $h_3(58) = (h(58) + 3) \% 10 = 1$

Complete the linear probing for 69  
 $h_0(69) =$

# Collision Resolution - Linear Probing Issues

---

- Probe sequences can get longer with time.
- Primary clustering
  - Keys tend to cluster in one part of table.
  - Keys that hash into cluster will be added to the end of the cluster (making it even bigger).
  - Side effect:  
Other keys could also get affected if mapping to a crowded neighborhood.

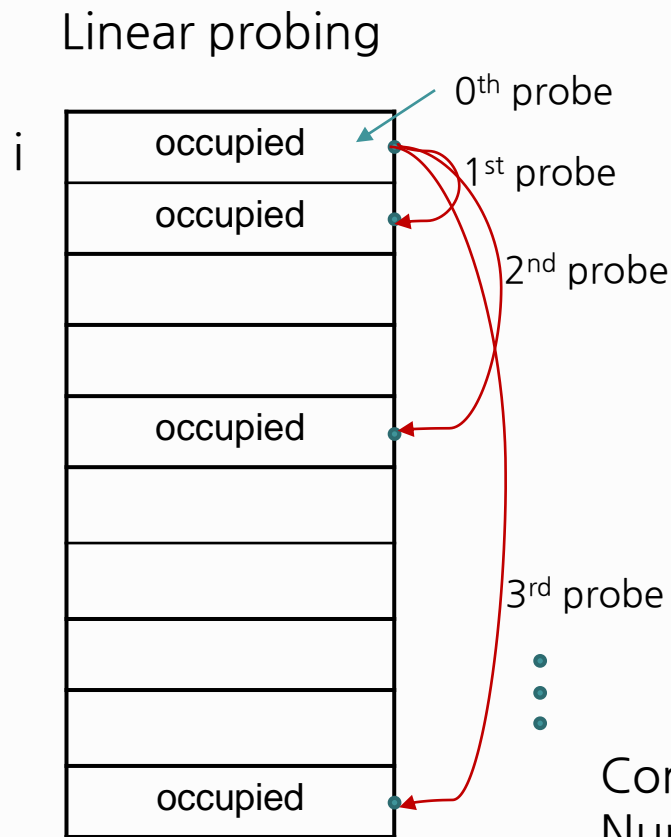
# Collision Resolution - Quadratic Probing<sup>이차조사법</sup>

- Avoids primary clustering
- $f(i)$  is quadratic in  $i$ , e.g.,  $f(i) = i^2$

$$h_i(k) = (h(k) + i^2) \% \text{TableSize}$$

$i^{\text{th}}$  probe index       $0^{\text{th}}$  probe index       $f(i)$

Probe sequence:  $+0, +1, +4, +9, +16, \dots$



Continue until an empty slot is found  
Number of failed probes is a measure of performance

# Collision Resolution - Quadratic Probing<sup>이차조사법</sup>

- Avoids primary clustering
- $f(i)$  is quadratic in  $i$ , e.g.,  $f(i) = i^2$

$$h_i(k) = (h(k) + i^2) \% \text{TableSize}$$

$i^{\text{th}}$  probe index       $0^{\text{th}}$  probe index       $f(i)$

Probe sequence: +0, +1, +4, +9, +16, ...      ← quadratic

- Example:
  - $h_0(58) = (h(58) + 0^2) \% 10 = 8$  (collision)
  - $h_1(58) = (h(58) + 1^2) \% 10 = 9$  (collision)
  - $h_2(58) = (h(58) + 2^2) \% 10 = 2$

# Collision Resolution Exercise - Quadratic Probing이차조사법

Insert sequence: 89, 18, 49, 58, 69

$$h(k) = k \% 10$$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89
Unsuccessful no. of probes		0	0	1	2	2

For example, quadratic probing for 58

$$h_0(58) = (h(58) + f(0)) \% 10 = (8 + 0) \% 10 = 8 \text{ (collision)}$$

$$h_1(58) = (h(58) + 1) \% 10 = 9 \text{ (collision)}$$

$$h_2(58) = (h(58) + 4) \% 10 = 2$$

probe sequence

# Collision Resolution Exercise - Quadratic Probing이차조사법

Insert sequence: 89, 18, 49, 58, 69

$$h(k) = k \% 10$$

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89
Unsuccessful no. of probes		0	0	1	2	2

probe sequence

For example, quadratic probing for 58

$$h_0(58) = (h(58) + f(0)) \% 10 = (8 + 0) \% 10 = 8 \text{ (collision)}$$
$$h_1(58) = (h(58) + 1) \% 10 = 9 \text{ (collision)}$$
$$h_2(58) = (h(58) + 4) \% 10 = 2$$

Complete quadratic probing for 69

$$h_0(69) =$$

# Collision Resolution - Quadratic Probing Analysis

---

- Difficult to analyze
- Theorem:
  - New element can always be inserted into a table that is at least half empty and TableSize is prime.
  - Otherwise, may never find an empty slot, even if one exists.
- Ensure table never gets half full.
  - If close, then expand (rehash) it.
- May cause "secondary clustering"



# Summary

---

- Table size prime
- Table size larger than number of inputs (to maintain  $\lambda \ll 1.0$ )
- Two types of collision resolution
  - Separate chaining
  - Open addressing - probing
  - Tradeoffs between chaining vs. probing
- Collision chances decrease in this order:  
linear probing  $\rightarrow$  quadratic probing  $\rightarrow$  double hashing
- Rehashing recommended to resize hash table at a time when  $\lambda$  exceeds 0.5

# Data Structures in Python

## Chapter 6

- Hash Table
- **Collision Resolutions**
- Double Hashing & Rehashing
- Hash Implementation