

# 파이썬으로 배우는 데이터 구조



한동대학교  
전산전자공학부  
김영섭 교수



# 학습 목표

---

이중 해싱(Double hashing)과  
재해싱(Rehashing)을 적용할 수 있다

# Data Structures in Python

## Chapter 6

- Hash Table
- Collision Resolution
- **Double Hashing & Rehashing**
- HashMap Coding

# Agenda & Readings

---

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

# Collision Resolution by Open Addressing

---

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

# Collision Resolution - Double Hashing 이중해싱법

- Keep two hash functions,  $h(x)$  and  $h'(x)$ .
- Use **a second hash function** for all tries  $i$  other than 0

$$f(i) = i * h'(x)$$

- Good choices for  $h'(x)$ ?
  - Should never evaluate to 0.
  - $h'(x) = R - (x \% R)$ 
    - $R$  is prime number less than TableSize.

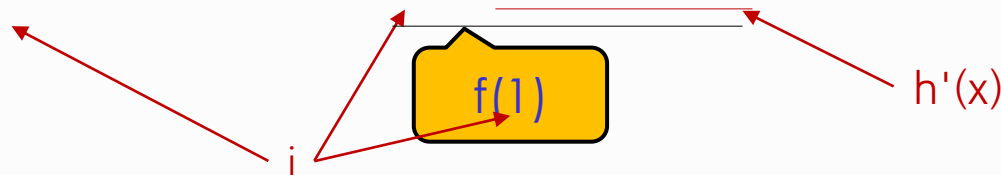
Hash function

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

- For example,  $h(x) = k \% 10$  with  $R = 7$

$$h_0(49) = (h(49) + f(0)) \% 10 = 9 \text{ (collision)}$$

$$h_1(49) = (h(49) + 1 * (7 - 49 \% 7)) \% 10 = 6$$



If we assume that  $h_1(49) = 6$  ends up a collision, the next probing is ...

$$h_2(49) = (h(49) + 2 * (7 - 49 \% 7)) \% 10 = 3$$

## Collision Resolution Example - Double Hashing 이중해싱법

- Insert keys 43, 25 into the hash table below and find the probe sequence for each:
- Use  $h(k) = k \% 13$  with  $R = 7$ .

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	None	None	None	None	17

$h_0(43) = h(43) = 43 \% 13 = 4$  (collision)

$h_1(43) = (h(43) + 1 * (7 - 43 \% 7)) \% 13 = (4 + 6) \% 13 = 10$ , Probe sequence: 4, 10

$h_0(25) = h(25) = 25 \% 13 = 12$  (collision)

$h_1(25) = (h(25) + 1 * (7 - 25 \% 7)) \% 13 = (12 + 3) \% 13 = 2$

$h_2(25) = (h(25) + 2 * (7 - 25 \% 7)) \% 13 = (12 + 6) \% 13 = 5$

$h_3(25) = (h(25) + 3 * (7 - 25 \% 7)) \% 13 = (12 + 9) \% 13 = 8$ , Probe sequence: 12, 2, 5, 8

0	1	2	3	4	5	6	7	8	9	10	11	12
26	None	54	94	17	31	None	None	25	None	43	None	17

# Collision Resolution Exercise - Double Hashing 이중해싱법

Insert sequence: 89, 18, 49, 58, 69, 23

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

$$h'(x) = R - (x \% R)$$

R is prime number less than TableSize

	Empty Table	After 89	After 18	After 49	After 58	After 69	After 23
0							
1							
2							
3							
4							
5							
6				49	49	49	49
7							
8			18	18	18	18	18
9		89	89	89	89	89	89
Unsucessful no. of probes		0	0	1	1	1	

$$h_0(49) = (h(49) + f(0)) \% 10 = 9 \text{ (collision)}$$

$$h_1(49) = (h(49) + 1 * (7 - 49 \% 7)) \% 10 = 6$$

$$h_0(58) =$$

$$h_1(58) =$$

$$h_0(69) =$$

$$h_1(69) =$$

$$h_0(23) =$$


$$h_1(23) =$$

:



# Collision Resolution Analysis - Double Hashing

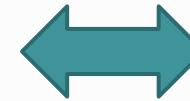
---

- Imperative that **TableSize is prime**
  - e.g., insert 23 into previous table
- Empirical tests show **double hashing** close to random hashing.  Is it good or bad?
- Extra hash function takes extra time to compute.

# Rehashing

- Rehashing is the reconstruction of the hash table:

0	6
1	15
2	
3	24
4	
5	
6	13



0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

# Rehashing

---

- Rehashing is the reconstruction of the hash table:
  - All the elements in the container **are rearranged** according to their hash value into the new set of buckets. This may alter the order of iteration of elements within the container.
- Increases the size of the hash table when load factor becomes "too high" (defined by a cutoff)
  - Anticipating that collisions would become higher
- Typically expand the table to **twice** its size (**but still prime**)
  - $\text{TableSize}_{\text{new}} = \text{nextprime}(2 * \text{TableSize}_{\text{old}})$
  - e.g.,  $2 \rightarrow 5$ ,  $5 \rightarrow 11$ ,  $11 \rightarrow 23$
- Need to **reinsert all existing elements** into new hash table

# Rehashing Example

$$h(x) = x \% 7$$
$$\lambda = 0.57$$

0	6
1	15
2	
3	24
4	
5	
6	13

Insert 23



$$\lambda = 0.71$$

0	6
1	15
2	23
3	24
4	
5	
6	13

TableSize = 7

$$\lambda_{max} = 0.6$$

Rehashing  
since  $\lambda > \lambda_{max}$



0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

TableSize = 17  
nextprime(TableSize \* 2)

$$h(x) = x \% 17$$
$$\lambda = 0.29$$

## Hashing Analysis

---

- The **load factor** ( $\lambda$ ) of the hash table is the number of items in the table divided by the size of the table.
- If  $\lambda$  is small then keys are more likely to be mapped to slots where they belong and searching will be  $O(1)$ .
- If  $\lambda$  is large then collisions are more likely and more comparisons (is the slot available or not) are needed to find an empty slot.

# Rehashing Analysis

---

- If the load factor goes high, the performance slows down significantly. In that case the easiest solution is to copy the entire hash table into a larger table. This process is called rehashing.
- When to rehash
  - For separate chaining, the load factor should not exceed **0.75**.  
For open addressing, the load factor should not exceed **0.5**.
- Rehashing a table is expensive (since elements must be inserted using the new hash function) - do only occasionally, e.g. double size of table each time, but make sure that the size is a prime number.

## Rehashing - Exercise

- Rehash the following table into a new hash table below using the hash function: Use  $\text{hash}(\text{key}) = \text{key} \% 13$  and quadratic probing to resolve the collisions. Show your computation, collision and resolution. Compute the load factors before and after rehashing .

0	1	2	3	4	5	6
56	43	30	None	None	26	13

0	1	2	3	4	5	6	7	8	9	10	11	12

# Hashing Applications

---

- Symbol table in compilers
- Accessing tree or graph nodes by name
  - e.g., city names in Google maps
- Maintaining a transposition table in games
  - Remember previous game situations and the move taken (avoid re-computation)
- Dictionary lookups
  - Spelling checkers
  - Natural language understanding (word sense)
- Heavily used in text processing languages
  - e.g., Perl, Python, etc.



# Summary

---

- The hash table size uses a prime number in general.
  - The table size is larger than number of inputs (to maintain  $\lambda \ll 1.0$ )
  - It helps its performance and prevents it from rehashing.
- The collision cannot be avoided.
  - Collision resolution strategies are required.
  - There are some trade-offs between chaining vs. probing
  - Collision chances decrease in this order:  
linear probing  $\rightarrow$  quadratic probing  $\rightarrow$  double hashing
- Rehashing is recommended when the load factor  $\lambda$  exceeds 0.5 in general.

# 학습 정리

- 1) 실증적 테스트를 통해 이중 해싱은 무작위(random) 해싱에 가깝다는 것을 알 수 있다
- 2) 재해싱(Rehashing)할 때는, 테이블 크기를 2배로 늘리되 크기는 소수(prime)를 유지하는 것이 중요하다
- 3) 해싱은 컴파일러의 심볼 테이블, 데이터베이스, 딕셔너리 등등에서 빠른 검색을 위하여 많이 활용된다

# 파이썬으로 배우는 데이터 구조

수고했습니다  
곧 다음 시간에  
다시 뵙겠습니다