

# 4. Non linear Structure

- Limit of Divide and Conquer
- Hash table
- Hash function
- Examples of hash function
- Collision resolution of hashing
- Deletion in hash table
- Managing the size of hash table

# Hash

Il-Chul Moon  
Dept. of Industrial and Systems Engineering  
KAIST

[icmoon@kaist.ac.kr](mailto:icmoon@kaist.ac.kr)

### Weekly Objectives

- ◇ This week, we study the hash table
- ◇ Objectives are
  - ◇ Understanding why we use hash tables
  - ◇ Understanding the structure and the operation of hash tables
    - ◇ Hash functions
      - ◇ Modulo based hash function
      - ◇ Square based hash function
      - ◇ Digit based hash function
    - ◇ Insertion
      - ◇ Chaining
      - ◇ Open addressing
        - ◇ Linear probing
        - ◇ Quadratic probing
    - ◇ Deletion
  - ◇ Understanding the performance of hash tables

# Limit of Divide and Conquer

## Limit of divide and conquer

◆ The limitation of divide and conquer

$$O(N^2) \rightarrow O(N \log N)$$

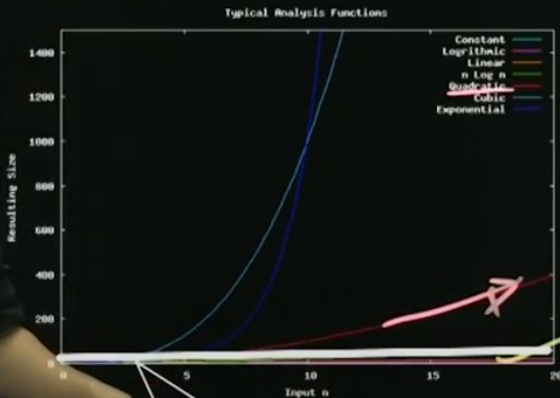
◆ If the divide and conquer is based upon a comparison

◆ The efficiency is limited to the logarithm of the problem size

◆ Search:  $O(N) \rightarrow O(\log N)$

◆ Sorting:  $O(N^2) \rightarrow O(N \log N)$

Linked list  
 $O(N^2)$



What-if  $O(1)$ ?  
You can't achieve this  
through comparisons...

divide and conquer를 통해  
비교해야할 대상이 줄어서  
 $O(N^2)$  ->  $O(N \log N)$  로  
되었다




그래도 결국 나중에는 점점  
값이 커지기 때문에 이것이  
해결해야할 문제이다

# Limit of Divide and Conquer

## Cheap storage cost and expensive time cost

- ❖ Searching and sorting without comparisons.
  - ❖ Because time is gold
  - ❖ With cheap storage spaces, memory and disk
- ❖ Let's imagine that we are storing the information of the whole population of Korea.
  - ❖ About 50,000,000 individuals
  - ❖ Are they uniquely identifiable?
    - ❖ Yes, by the registration number system
    - ❖ Similarly, bank accounts are identifiable by account numbers
- ❖ Then, why not store the individual's information in
  - ❖ A dictionary in Python
  - ❖ Key and value
    - ❖ Key: registration numbers?
    - ❖ Value: my information...

Dictionary size =  
50,000,000

Key	Value
....	....
Key <sub>1</sub>	
....	....
Key <sub>2</sub>	
....	....
Key <sub>3</sub>	
....	....!

Key?

8011171234567

Too many unused  
keys...

Hence...

f(8011171234567)

한국에 있는 모든 인구를 저장한다  
5천만명에 모두 유니크한  
아이디가있다(주민번호)  
모든 사람을 주민번호로 저장하면  
되는거아닌가?  
딕셔너리에 담으면?  
사이즈가 5천만인데  
주민등록번호는 13자리이고  
5천만은 8자리면 되는데  
손해아닌가? 쓸모없는 메모리가  
생긴다  
배열도 똑같이 주민등록번호로  
만들면 너무 커서 감당할 수 없다  
그러면 키를 실제 정보로  
변환해주는 과정이 필요하다  
키를 넣으면 인덱스를 리턴해주는  
함수를 만든다  
키에 연결된 값이 없을수 있지만  
그정도 빈 공간은 감당할 수 있다

# Hash table

## Hash table

### Hash table

◆ A large size of tables consisting of keys and values

◆ Hash function is used to find an array index for a key

#### Definition

◆ An array index: an index in the table

◆ A key: a unique identifier of a value, a parameter to find its array index through hash functions

◆ A value: a stored value

#### Characteristic


◆ One key can be associated with one index

◆ One index can be associated with multiple keys

◆ Why?

◆ An array element = a slot = a bucket

Hash Table

Key	→ dx	Value
....	...	....
Key <sub>1</sub>	50	
....	...	....
Key <sub>2</sub> from 801117 1234567	70	
121118 1234567	...	....
Key <sub>3</sub>	97	
....	...	....

## Hash table

key와 value pair가 존재하는 아주 큰 테이블이고

hash function이 있다

key를 통해서 index를 알아내는게 해쉬 테이블이다  
key를 index로 바꿔주는 함수가 hash function

## hash table은

하나의 키가 하나의 인덱스랑 연결되어있다

하나의 인덱스는 여러개의 키로 있을 수 있다

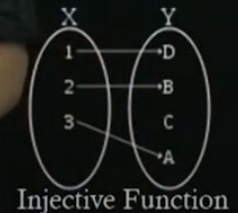
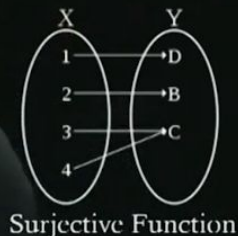
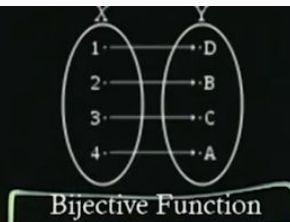
해쉬테이블 행을 a slot또는 a bucket이라고도 부른다



# Hash function

## Hash function

- ◆ Hash function is used to convert
  - ◆ Value's key  $\rightarrow$  Array's index
  - ◆ For example,  $f(8011171234567) = 3$ 
    - ◆ 8011171234567 = a person's register number : *key*
    - ◆ 3 = an index where the person is stored
    - ◆  $f$  is a hash function
- ◆ Perfect hash function
  - ◆ Either, if we have an unlimited space
  - ◆ Or, if we have a limited input size
  - ◆ We might have a chance to create a bijective hash function
  - ◆ However, you have to know the characteristic of keys
- ◆ Good hash function
  - ◆ Hash function resulting in a uniform distribution
    - ◆ Why is this a good hash function?
    - ◆ Why is this difficult?
    - ◆ Pseudo-random....
    - ◆ Any relation to the password encryption algorithm?
  - ◆ low cost, determinism, variable range



맵핑할때 사용된다.

키를 해쉬평선에 넣어서 인덱스가 나오면 된다.

perfect 해쉬 평선은  
x에 있는 유니크한 값이 y에 유니크한 값에 모두 연결되면 **Bijjective Function**이라 한다  
사이즈가 줄어드는게 아니기때문에 쓸모는 잘 모르겠다

good 해쉬 평선  
uniform distribution균등 분포가 좋은 해쉬평선이고  
한쪽 인덱스에 쏠려있는 것은 안좋은 해쉬평선이다

low cost빠른계산, determinism값을 넣으면 값이 나와야하고, variable range 범위에 맞는 결과



# Examples of hash function

## Modulo based hash function

$$f(\text{key}) = \text{key} \% \text{num}$$

$$= \text{key} \% \text{num}$$

$$[0, \text{num}-1]$$

$$\text{Index} = \text{Key} \bmod \text{Size}$$

$$34567 = 8011171234567 \bmod 50000$$

$$\text{But, } 34567 = 8011171284567 \bmod 50000$$

$$\text{Why is this often used?}$$

$$\text{Good variable range}$$

$$\text{Appears to be uniform}$$

$$\text{But, actually it is not}$$

$$\text{Still, surjective function}$$

Modulo based hash function

◊ We divide a numeric key with a number

◊ Often, a number = the size of the hash table

◊ We take the remainder as an array index

◊ For example,

◊  $\text{Index} = \text{Key} \bmod \text{Size}$

◊  $34567 = 8011171234567 \bmod 50000$

◊ But,  $34567 = 8011171284567 \bmod 50000$

◊ Why is this often used?

◊ Good variable range

◊ Appears to be uniform

◊ But, actually it is not

◊ Still, surjective function

low cost  
range  
uniform

key % num

num = size of the hash table

modulo based hash function은 low cost이고 variable range에 딱 맞는다

uniform distribution은 어려우나 위에 두가지를 만족하기때문에 많이 사용한다  
그리고 키를 유니크하게 사용 못할경우도 많다

# Examples of hash function

## Square based hash function

### ◊ Square based hash function

#### ◊ Or, mid-square hash function

#### ◊ The problem of the modulo operation

##### ◊ Under-utilizing available information

◊ Quotient is not used...

#### ◊ Therefore, we perform a mid-square method

◊ First, square the key value

◊ Second, take the N digits in the middle

◊ Third, apply the selected digits to the modulo operation

#### ◊ For example,

◊ Key = 123, Table size = 100, Mid-square digits = 3

◊  $f(123) = \text{MidSquare}(123) \bmod 100 = 512 \bmod 100 = 12$

#### ◊ Still utilizing the modulo operation at the end

◊ Why?

$$f(123)$$

$$\begin{array}{r} 123 \\ \times 123 \\ \hline \end{array}$$

369

246

023

$$\begin{array}{r} 123 \\ \times 123 \\ \hline \end{array}$$

$$f(\text{key}) = (\text{key}^2) \bmod 100$$

중간 3자리

modulo의 문제점을  
해결하기 위해 나왔다

### Under-utilizing available

아래것들이 무시되고있으니  
잘 활용하자

키값을 제공하고 몇개의  
데이터를 사용할 것인지  
선택해서 중간에서 데이터  
추출해서 사용하고 앞뒤의  
남는값은 버린다

low cost 가 약간 올라갔다  
range 마지막에 mod로  
처리하니 괜찮다  
더 유니폼해진다

# Examples of hash function

## Digit analysis based hash function

### Digit analysis

- ◇ A.k.a. checksum hash function
- ◇ Looks pretty stupid way
- ◇ But works pretty well

### Procedure of the function calculation

- ◇ One checksum method
  - ◇ First, take all the digits and sum of them
  - ◇ Second, apply the modulo operation to the summation of the digits
- ◇ For example,
  - ◇ Key = 8011171234567, Table size = 100
  - ◇  $f(8011171234567) = (8+0+1+1+1+7+1+2+3+4+5+6+7) \bmod 100$   
 $= 46$

checksum hash  
function이라고도 한다

digit analysis  
sorting 이 다 digit  
analysis이다

모든 키의 값을 다 더한다  
다 더한값에 나머지에  
**modulo based**를  
실행하는게 **digit analysis  
based**이다

low cost  
range 마지막에 mod로  
처리하니 괜찮다  
키에서 쓰지않는게 하나도  
없기때문에 **uniform**하다

# Collision resolution of hashing

## Collision resolution of hashing

### ◆ Load factor

- ◆ Load factor is often the determinant of the hash performance

$$= \frac{N}{S}$$

- ◆  $N$  = Size of the stored entries

- ◆  $S$  = Size of the hash table

### ◆ Why is this important?

- ◆ Related to one of the qualities of the hash function, or uniformity

### ◆ Because of collision

- ◆ Different keys with the same index

### ◆ Closed addressing

### ◆ Open addressing

Surjective Function

Key	Idx	Value
....	...	....
Key <sub>1</sub>	50	
....	...	....

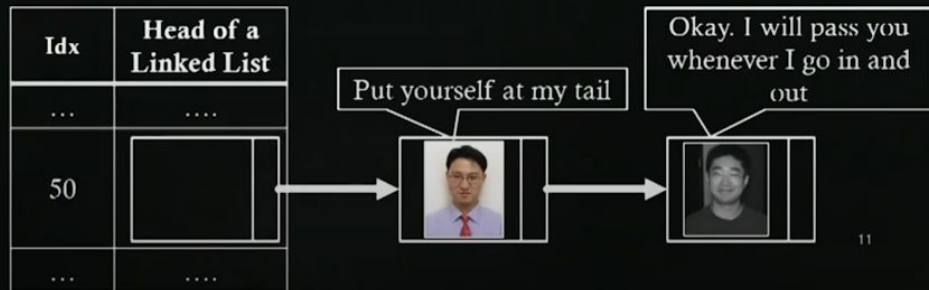
여전히 테이블 사이즈는 한정이고 **key**는 커서 **collision**이 많이 일어난다 **collision**이란 **x**에서 **y**로 연결할때 하나의 인덱스에 여러개가 바라보는걸 **collision**이라 한다

이걸 없애기위해 알아야하는 개념 **Load factor**  
얼마만큼 테이블에 붐비게 들어가있는가

# Collision resolution of hashing

## Collision resolution by closed addressing

- Collision resolution by closed addressing
  - Separate chaining
  - Live together approach
- The worst case scenario
  - Every entries have the same index from a stupid hash function
  - Just another linked list
- Considering the load factor
  - Load factor  $> 1$  is possible
  - This case means that every index has one or more entries
  - Then? Only use the linked list for each bucket? Trees can be used as well...



정해진 인덱스에게서 어떻게든 저장하는것

collision이 발생했을때 동일한 인덱스를 가지고있는 값이 들어오면 linked list를 사용하여 chaining된다 worst case는 모든 key의 인덱스가 같은 인덱스여서 결국 linked list가 되버린다

linked list 말고 다른 structure를 사용할 수 도 있지만 더 무거워지기 때문에 linked list를 주로 사용한다




# Collision resolution of hashing

## Collision resolution in open addressing

- Collision resolution by open addressing
  - Resolution by probing
    - See where an empty bucket is
    - I don't want to live with you, so get out and find your own place* approach
  - Various probing
    - Linear probing: see whether the next bucket is empty or not
      - Index =  $(f(\text{Key}) + i) \bmod S$
    - Quadratic probing
      - Index =  $(f(\text{Key}) + i + i^2) \bmod S$
      - $i$  = number of trials
      - $S$  = size of the hash table
  - Why quadratic probing?
    - Uniformity of a hash function
    - Because of a cluster

**Linear probing:**  
Okay... I will go to the next bucket

**Quadratic probing:**  
Okay... I will go to a bucket far away...

Key	Idx	Value
....	...	....
$f(\text{Key}_1)=50$	50	
$f(\text{Key}_2)=50$	51	Empty
....	....	....
....	....	....
....	....	....
$f(\text{Key}_2)=50$	99	Empty
....	....	....

한 테이블에 하나의 밸류만 저장한다

collision을 막을 순 없으니 근처에있는 다른 인덱스로 보낸다(probing)

linear probing 바로 다음 빈 인덱스에 넣는다

quadratic probing  
기존 인덱스에서 멀리 보낸다  
cluster가 발생했을때 근처로 가봐야 다 값이 있기때문에 아예 멀리 보내기위해 사용

# Deletion in hash table

## Deletion in open addressing hash table

### Deletion

#### Closed addressing

##### Simple

- Go to the bucket, and follow the linked list, and delete it

#### Open addressing

##### Complicated

##### Intuitively...

- Visit the index from the hash function
- If it is there, delete it
- If not, keep following the probing method, find and delete it

#### What is wrong with this idea?

#### Problem scenario?

- Insert an 1, 6, 11 with the same index from hash functions, resolved by linear probing
- Deleting 6
- And, searching 11 ← Problem!

### Then?

#### Lazy deletion

- You just mark it and do not delete it
- Keep adding entries and no deletion
  - Why? Cheap storage cost
- But, always there is a limit

$$f(K) = K \bmod 5$$

Open addressing  
Linear probing

0	1	2	3	4
		6	11	

Delete 6

0	1	2	3	4
	1		11	

Search 11

0	1	2	3	4
	1		11	

closed

쉽다 linked list에서 지우면된다

open

어렵다

왜 어렵냐면 지우고자 하는 인덱스로가서 지우면되는데 한 테이블에 하나씩 들어가있는데 probing때문에 하나 지우고나면 다음꺼 찾는게 힘들(그림 보면서 설명)

Lazy deletion

그래서 지우지않고 지웠다는 표시를 해준다  
그러면 이 값은 검색해도 안나오지만 다음 인덱스를 찾을때는 애도 이용해서 찾는다  
어느정도 꽉 차게되면 새로 큰 테이블로 옮기는데 그 때는 이제 옮기지 않고간다

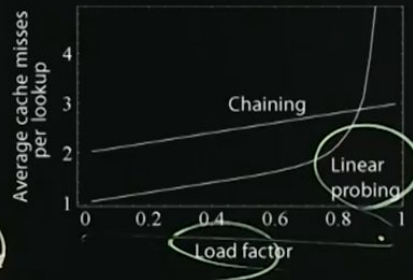


# Managing the size of hash table

## Managing the hash table size

Re-hashing

- ◆ There is always a limit
  - ◆ Even though the storage is cheap
  - ◆ You don't have an infinite storage
- ◆ If you don't delete entries
- ◆ Keep adding more entries
- ◆ Then, the table's load factor becomes higher
  - ◆ What this mean?
    - ◆ More probing to insert
- ◆ Hence, sometimes
  - ◆ You need to extend the storage space
  - ◆ And insert the entries to the new space with more buckets



처음에 아무리 크게  
만들어도 결국은 한계  
다다르기 마련

새로운 해시테이블에  
인서트하는 과정을 가진다  
이걸 Re hashing이라고  
한다

# Managing the size of hash table

## Performance of hash table

	Linked List	Stack	Queue	Binary Search Tree in Average Case	Binary Heap	Hash Table in Average Case
Search	$O(N)$			$O(\log N)$		$O(1)$
Insert	$O(1)$	$O(1)$	$O(1)$	$O(\log N)$	$O(\log N)$	$O(1)$
Delete	$O(1)$	$O(1)$	$O(1)$	$O(\log N)$	$O(\log N)$	$O(1)$
Type	Linked List Based			Tree Based		Hash Based
Major Paradigm	Linked list, chaining, referencing			Divide and conquer		Array and key

No silver bullet, always pros and cons  
Some smart ideas that works in an average case  
But, in the extreme case, most of them works similarly....

그때그때 상황에 따라 잘  
사용하는게 실력이다  
항상 좋은점과 나쁜점이  
공존하고있다  
특성에 맞춰서 결정할 수 있는  
능력을 키우자