CSE221

# Lecture 10:
# Hashing

Fall 2021

Young-ri Choi

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided in former lectures at UNIST.

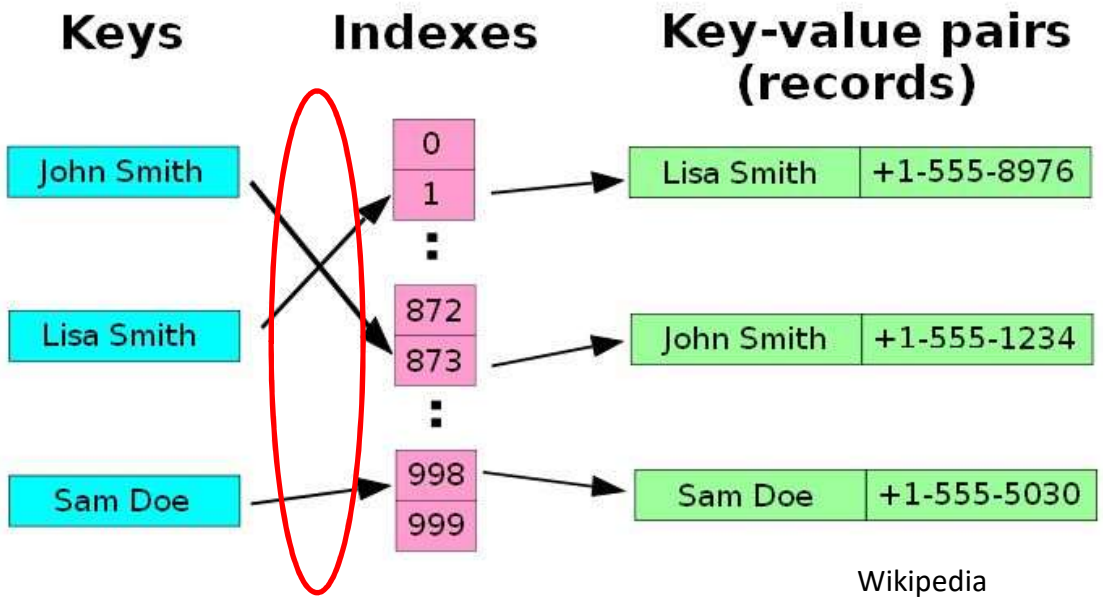
ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

# Outline

- Static hashing
  - Division
  - Mid square
  - Folding
- Overflow handling

# Hashing

- Hash table or hash map is a data structure that associates values with keys
- Example: phone book




Wikipedia

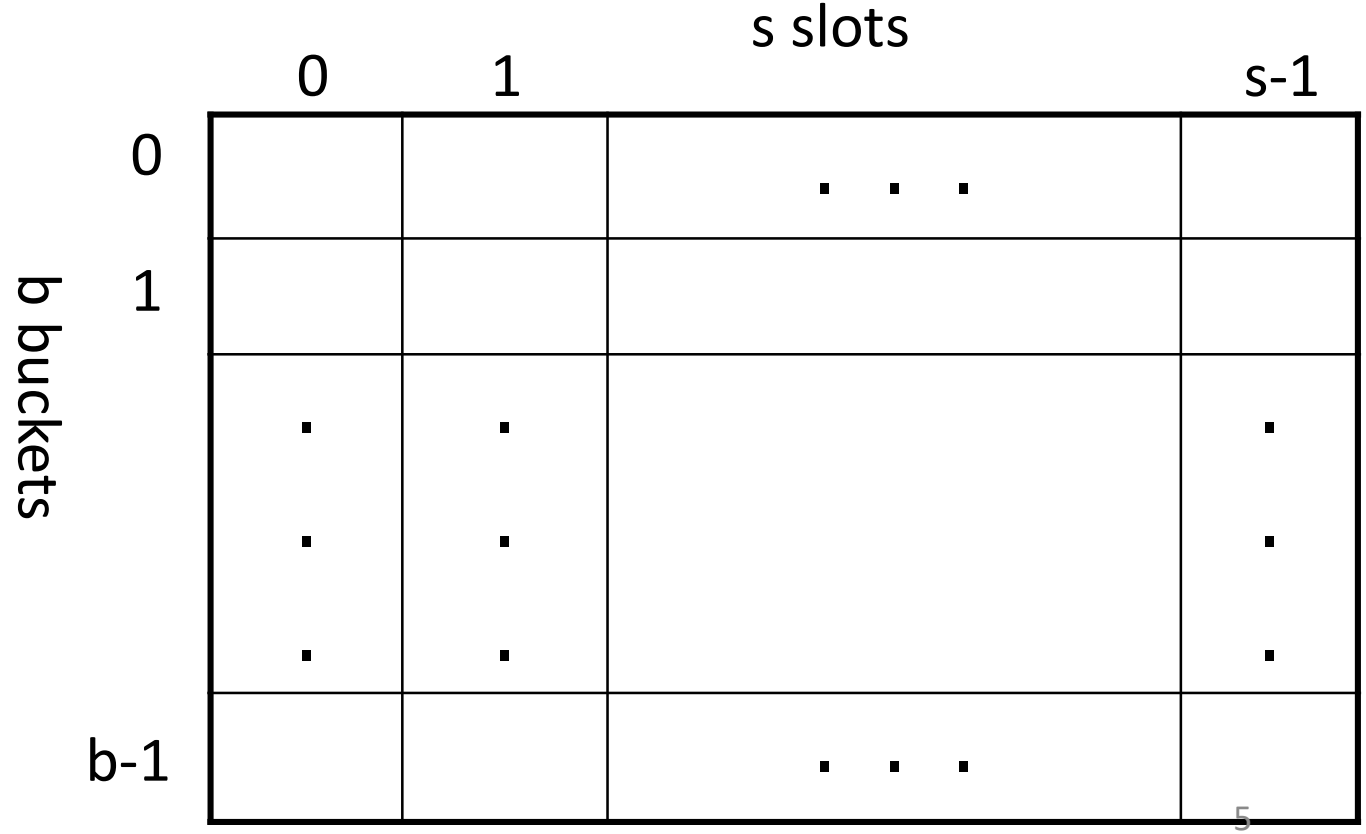
Hash function

# Hashing

- Hash function: hashing method
  - Calculate indexes from keys
  - Expected time: O(1)
  - Input data is not known in advance
    - Should consider all possible key values
- Types
  - Static hashing vs. Dynamic hashing

# Static Hashing

- Key-value pairs are stored in a fixed size hash table



# Static Hashing

- Key-value pairs are stored in a fixed size hash table
  - 2-D structure:
    - A hash table is partitioned into *b buckets*
    - Each bucket has *s slots*
    - Each slot holds one record
  - Hash function *h(k)* transforms *key k* into an address (or index) to a bucket in the hash table
    - Each key appears only once in the table

# Static Hashing

- n: current number of key-value pairs in the table
- T: all possible keys
- Key density : n/T
  - Fraction of keys in the table
  - Usually very low because not all keys are used
- Loading density(factor) : $\alpha = n/(sb)$
  - How much the hash table is used
  - 1 : table is full, 0 : table is empty

# Hashing

- Hash function *h* computes hash table address of a key without any other information
  - *h*(*k*) is a function of *k*
- Hash function may generate identical addresses for different keys
  - Why? Key density is usually very low
    - There is a chance for two keys map to a same location
  - A bucket may store multiple keys in slots

# Example

- b = 26, s = 2, n = 10
- Key: a string starting with a character
- Hash function: A~Z to 0~25, from the first character of key
  - A -> 0
  - A2 -> 0
  - D -> 3
  - G -> 6
  - GA -> 6

| | Slot 1 | Slot 2 |
|---|---|---|
| 0 | A | A2 |
| 1 | | |
| 2 | | |
| 3 | D | |
| 4 | | |
| 5 | | |
| 6 | GA | G |
| . | . | . |
| . | . | . |
| . | . | . |
| 25 | | |

# Example

- How about additionally inserting A1 and A3?
  - *Collision*: h(A1) = h(A3) = h(A) = h(A2) = 0
  - *Overflow*: no slot left

| | Slot 1 | Slot 2 |
|---|---|---|
| 0 | A | A2 |
| 1 | | |
| 2 | | |
| 3 | D | |
| 4 | | |
| 5 | | |
| 6 | GA | G |
| . | . | . |
| . | . | . |
| . | . | . |
| 25 | | |

A, A2, A1, A3 : collisions

A1, A3 : overflows

# Hash Table Issues

- Choice of hash function
  - Easy to compute
  - Avoid collision as much as possible
- Overflow handling method
  - Should handle when there is no space in the bucket for the new pair
- Size of hash table
  - If too small, the collision occurs often

# Two Steps for Hashing with Strings

- In many cases, keys are given as strings
- We first transform a string to an integer
- We then produce an address from the integer

# String To Integer

- `char` : 1 byte, `int` : 4 bytes
- A character has a unique value (i.e., ASCII code)

- We can pack and convert multiple characters in a string into a single integer
  - E.g. two-character string key[0:1] can be converted into a unique 4 byte non-negative `int`

```
number = key[0];
number += ((int) key[1]) << 8;
```

# String To Integer

- Example
  - Key: SA
  - S: ASCII code 83 = 01010011
  - A: ASCII code 65 = 01000001
  - S + A<<8
    = 0000000001010011 + 0100000100000000
    = 0100000101010011 (Binary)
    = 16723 (Decimal)

# Generalization

```
unsigned int stringToInt(char *key)
{
    int number = 0;
    while(*key)
    {
        number += *key++;
        if (*key)
            number += ((int) *key++) <<8;
    }
    return number;
}
```

This code generates 16bit integer for a string of arbitrary length (every two characters are converted into a 16bit integer and added together)

# Uniform Hash Function

- If *k* is a key chosen randomly, then we want probability of *h*(*k*)=*i* to be *1/b* for all buckets
  - Distribute key values uniformly over the range
- Uniform hash function minimizes collision / overflow when keys are selected randomly
- E.g., division, mid-square, folding, etc

# Hash Function: Division

- Domain is all nonnegative integers
- h(k)=k % D  (D is usually b)
- Generated address: 0 ~ D-1
- For *b buckets*, the number of integers that get hashed into bucket *i* is approximately $2^{31}/b$
- The division maps approximately the same number of keys into each bucket
  - Uniform hashing function

# Hash Function: Division

- Issue: keys tend to be biased
- If divisor (D) is an even number
  - Odd integers hash into odd buckets
  - Even integers into even buckets
  - Example
    - 20%14 = 6, 30%14 = 2, 8%14 = 8
    - 15%14 = 1, 3%14 = 3, 23%14 = 9
- If divisor is an odd number
  - Odd (even) integers may hash into any bucket
  - Example
    - 20%15 = 5, 30%15 = 0, 8%15 = 8
    - 15%15 = 0, 3%15 = 3, 23%15 = 8

# Hash Function: Mid-square

- Squaring the key and using r middle bits
- Example: r=2 digits in 10-base
  - k = 4567, $k^2$=20857489, h(k) = 57
- Avoid division operation, but expensive
- All bits of the key contribute the result

# Hash Function: Folding

- Partition the key x into several parts, and add the parts together to obtain the hash address
  - Part's size matches the size of the required address
- ex) x=12320324111220, 1000 addresses
  - partition x into 123,203,241,112,20
  - Shift folding
    - return the address 123+203+241+112+20=699
  - Folding at the boundaries
    - return the address 123+302+241+211+20=897