CSE221

Lecture 11:
Maps and Hashing

Hyungon Moon

Acknowledgment: The content of this file is based on the slides of the textbook as well as the slides provided by Prof. Won-Ki Jeong.

ULSAN NATIONAL INSTITUTE OF SCIENCE AND TECHNOLOGY

Outline

- Maps ADT
- Static hashing
  - Division
  - Mid square
  - Folding
  - Digit analysis
- Overflow handling

Outline

- Maps ADT
- Static hashing
  - Division
  - Mid square
  - Folding
  - Digit analysis
- Overflow handling

Maps

- A map models a searchable collection of key-value entries
- The main operations of a map are for searching, inserting, and deleting items
- Multiple entries with the same key are not allowed
- Applications:
  - address book
  - student-record database

Entry ADT

- An entry stores a key-value pair (k,v)
- Methods:
  - key(): return the associated key
  - value(): return the associated value
  - setKey(k): set the key to k
  - setValue(v): set the value to v

The Map ADT

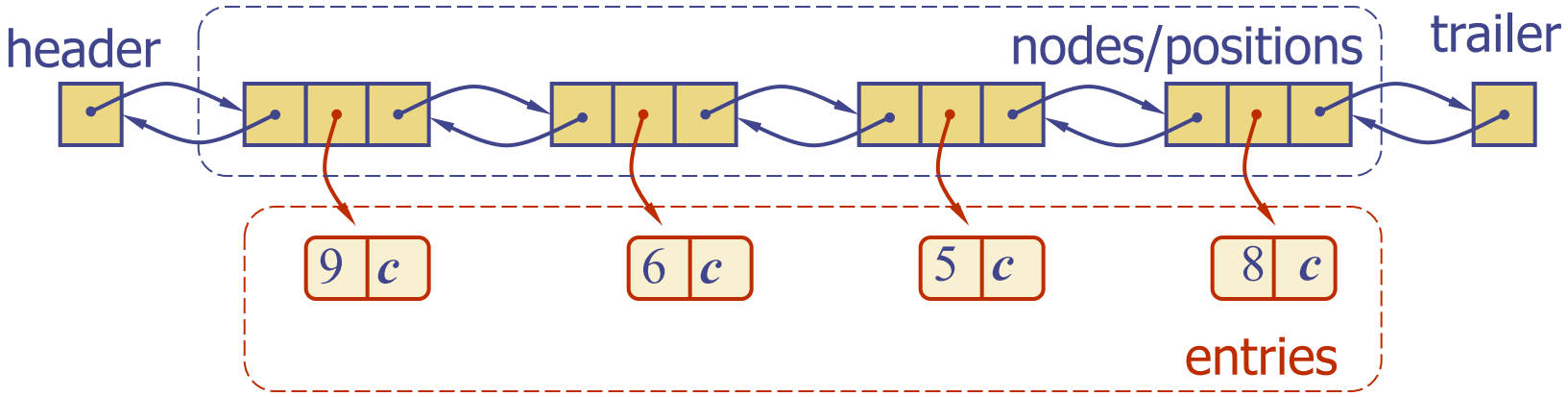
- find(k): if the map M has an entry with key k, return and iterator to it; else, return special iterator end
- put(k, v): if there is no entry with key k, insert entry (k, v), and otherwise set its value to v. Return an iterator to the new/modified entry
- erase(k): if the map M has an entry with key k, remove it from M
- size(), empty()
- begin(), end(): return iterators to beginning and end of M

Example

Operation	*Output*	*Map*
empty()	**true**	Ø
put(5,A)	[(5,A)]	(5,A)
put(7,B)	[(7,B)]	(5,A),(7,B)
put(2,C)	[(2,C)]	(5,A),(7,B),(2,C)
put(8,D)	[(8,D)]	(5,A),(7,B),(2,C),(8,D)
put(2,E)	[(2,E)]	(5,A),(7,B),(2,E),(8,D)
find(7)	[(7,B)]	(5,A),(7,B),(2,E),(8,D)
find(4)	end	(5,A),(7,B),(2,E),(8,D)
find(2)	[(2,E)]	(5,A),(7,B),(2,E),(8,D)
size()	4	(5,A),(7,B),(2,E),(8,D)
erase(5)	—	(7,B),(2,E),(8,D)
erase(2)	—	(7,B),(8,D)
find(2)	end	(7,B),(8,D)
empty()	**false**	(7,B),(8,D)

A Simple List-Based Map

- We can efficiently implement a map using an unsorted list
  - We store the items of the map in a list S (based on a doubly-linked list), in arbitrary order



The find Algorithm

```
Algorithm find(k):
    for each p in [S.begin(), S.end()) do
        if p->key() == k then
            return p
    return S.end() {there is no entry with key equal to k}
```

We use p->key() as a shortcut for (*p).key()

The put Algorithm

```
Algorithm put(k,v):
    for each p in [S.begin(), S.end()) do
        if p->key() == k then
            p->setValue(v)
            return p
    p = S.insertBack((k,v)) {there is no entry with key k}
    n = n + 1 {increment number of entries}
    return p
```

The erase Algorithm

```
Algorithm erase(k):
    for each p in [S.begin(), S.end()) do
        if p->key() == k  then
            S.erase(p)
            n = n – 1      {decrement number of entries}
```

Performance of a List-Based Map

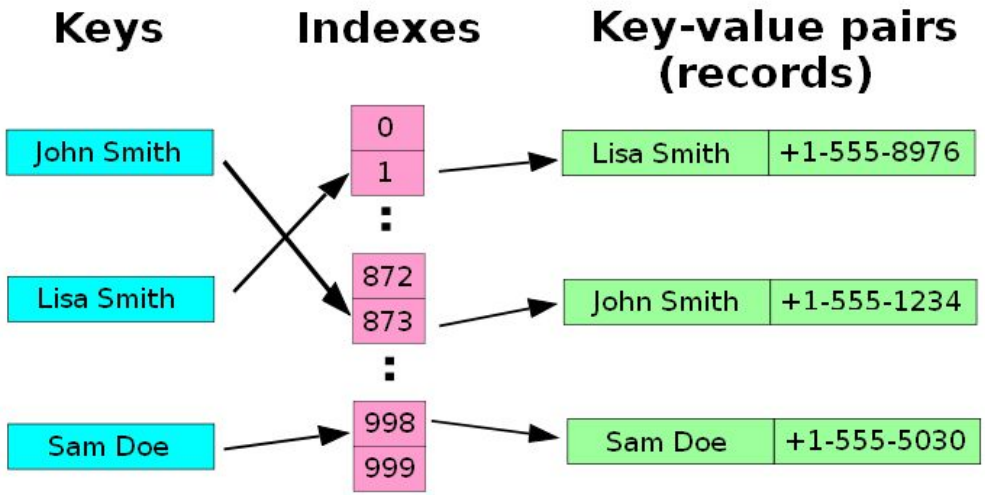
- Performance:
  - put takes $O(n)$ time since we need to determine whether it is already in the sequence
  - find and erase take $O(n)$ time since in the worst case (the item is not found) we traverse the entire sequence to look for an item with the given key
- The unsorted list implementation is effective only for maps of small size or for maps in which puts are the most common operations, while searches and removals are rarely performed (e.g., historical record of logins to a workstation)

Outline

- Maps ADT
- Static hashing
  - Division
  - Mid square
  - Folding
  - Digit analysis
- Overflow handling

Hashing

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



Wikipedia

Search vs. Hashing

- Search tree methods: key comparisons
  - Time complexity: O(n) or O(log n)
- Hashing methods: hash functions
  - Expected time: O(1)
  - No sorting, input data is not known in advance
- Types
  - Static hashing vs. Dynamic hashing

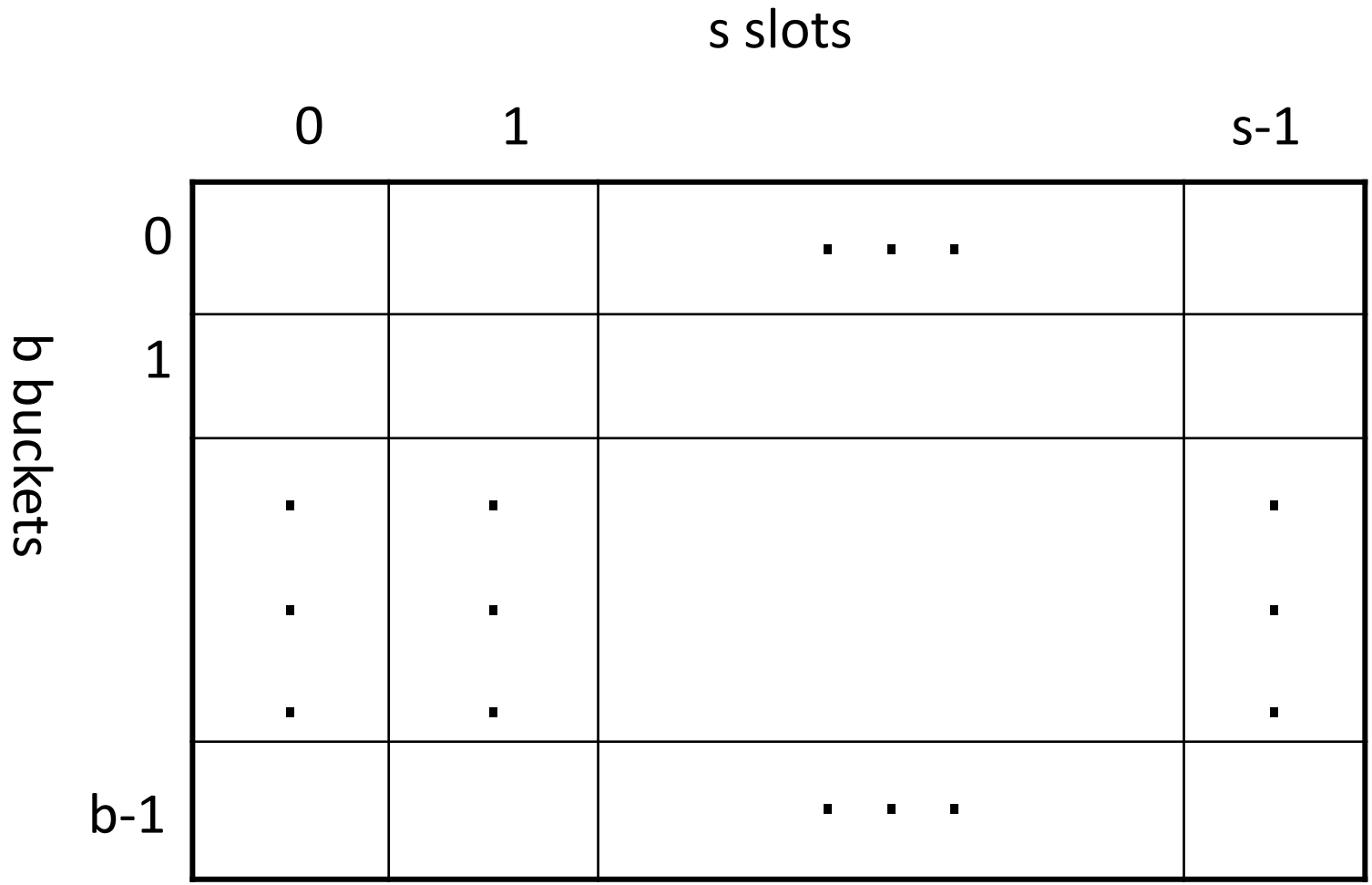
Static Hashing

- Key-value pairs are stored in a fixed size table called a *hash table*
  - A hash table is partitioned into many *buckets (b)*
  - Each bucket has many *slots (s)*
  - Each slot holds one record
  - A hash function h(k) transforms the identifier k (key) into an address in the hash table
  - n : current number of key-value pairs in the table
  - T : all possible keys

Static Hashing

- Key density : n/T
  - Fraction of keys in the table compared to the total number of possible keys
  - Usually very low b/c 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

Hash table



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
  - We need a bucket to store multiple keys
- Why this works?
  - Key density is usually very low, so there is a small chance for two keys map to a same location

Hash Table

- b = 26, s = 2, n = 10
- Keys : GA, D, A, G, L ,A2, A1, A3, A4, E
- Hash function : A~Z to 0~25, first character
  - 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		

Hash Table

- How about A1 and A3?
  - Synonyms : h(A1) = h(A3) = h(A) = 0, *collision*!
  - No slot left : *overflow*!

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

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

String To Integer

- `char` : 1 byte
- `int` : 4 bytes
- Two-character string key[0:1] may be converted into a unique 4 byte non-negative `int` using the code:

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

- Strings that are longer than 4 characters do not have a unique `int` representation

String To Integer

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

String To Integer

```
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 that h(k)=i to be 1/b for all buckets i
  - Distribute key values uniformly throughout the range
- Uniform hash function minimizes collision / overflow when keys are selected randomly
- 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 a hash table of size *b*, the number of integers that get hashed into bucket *i* is approximately $2^{31}/b$
- The division method maps approximately the same number of keys into each bucket
  - Uniform hashing function

Hash Function: Division

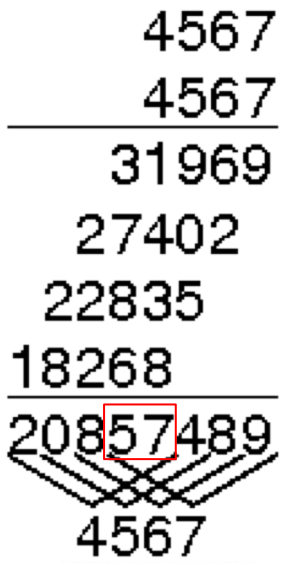
- In practice, keys tend to be biased
- If divisor is an even number, odd integers hash into odd buckets and even integers into even buckets (biased!)
  - 20%14 = 6, 30%14 = 2, 8%14 = 8
  - 15%14 = 1, 3%14 = 3, 23%14 = 9
- If divisor is an odd number, odd (even) integers may hash into any bucket
  - 20%15 = 5, 30%15 = 0, 8%15 = 8
  - 15%15 = 0, 3%15 = 3, 23%15 = 8

Hash Function: Division

- Similar biased distribution of buckets is seen in practice, when the divisor D is a multiple of small prime numbers such as 2, 3, 5, 7, …
- The degree of bias decreases as the smallest prime factor of D increases
- Choose D
  - Ideal : large prime number D <= b
  - Alternatives
    - Number that has no prime factors smaller than 20
    - Odd number

Hash Function: Mid-square

- Squaring the key and using r middle bits
  - Address 0 ~ 2^r -1 in binary
- Ex) k = 4567, k^2=20857489, h(k) = 57
  - r=2 digits in 10-base
  - Binary key is also possible
- Avoid division operation (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
  - 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

Hash Function: Digit Analysis

- Treat each key as a number using radix r
- Remove digits having the most skewed distributions

Keys	address
025452184	214
025453678	368
025458171	811
025159671	961
025151577	157
025554272	422

Outline

- Maps ADT
- Static hashing
  - Division
  - Mid square
  - Folding
  - Digit analysis
- Overflow handling
  - Open addressing
  - Chaining

Overflow Handling

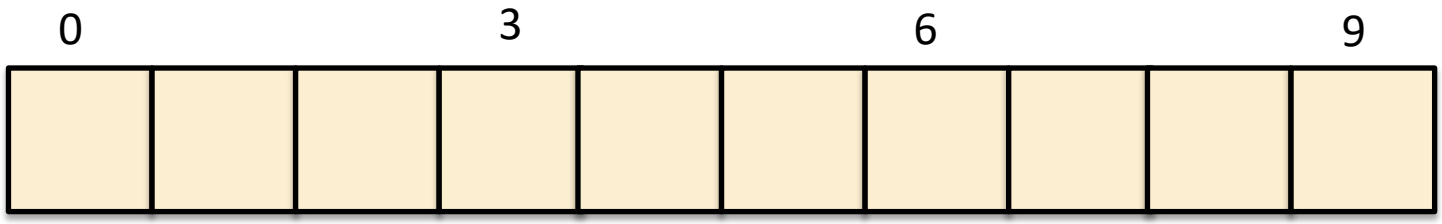
- An overflow occurs when bucket is full
- We may handle overflows by
  - Search the hash table in some systematic fashion for a bucket that is not full
    - Open addressing
  - Eliminate overflows by permitting each bucket to keep a list of all pairs for which it is the home bucket
    - Chaining

Linear Probing

- Find available bucket by examining ht[(h(k)+j)%b] for j=0, 1, 2, ..., b-1
- Insert
  - Find empty bucket
- Search
  - Find match key
  - If empty, key is not in the table
- Delete
  - Need to reorganize keys

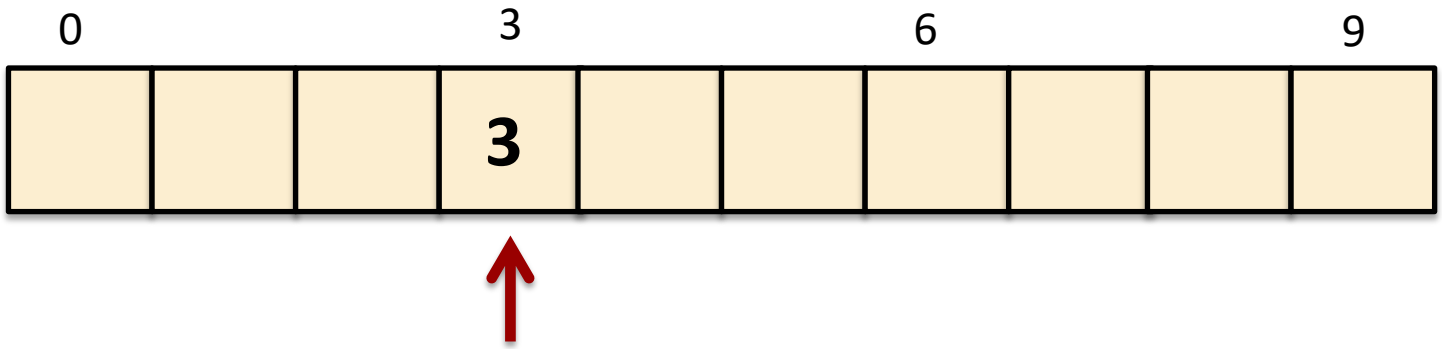
Linear Probing

- Divisor = b (# of buckets) = 10
- h(k) = k % 10



Linear Probing

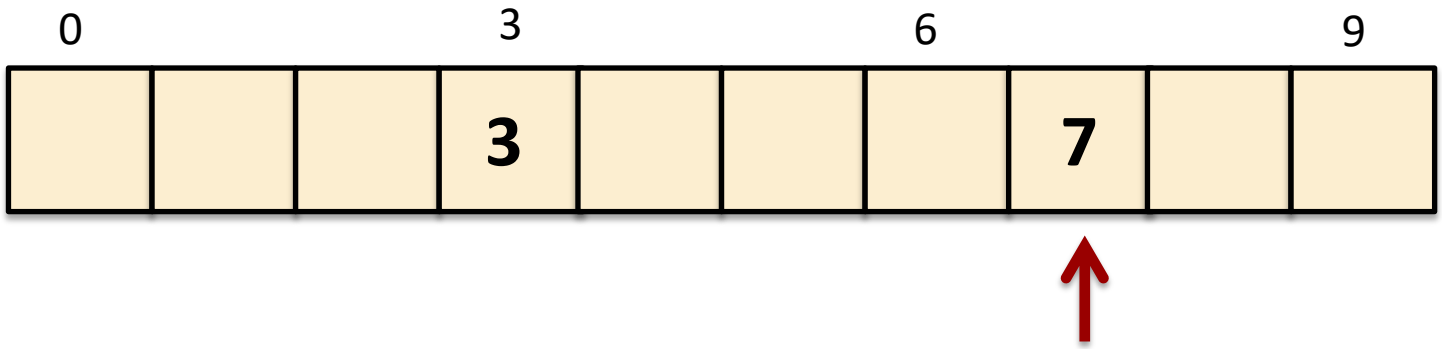
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 3
  - 3%10=3

Linear Probing

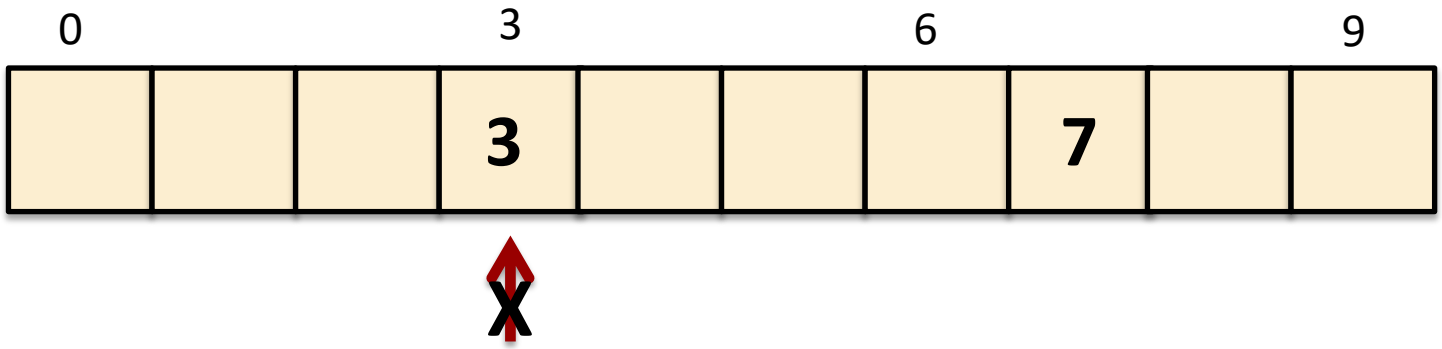
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 7
  - 7%10=7

Linear Probing

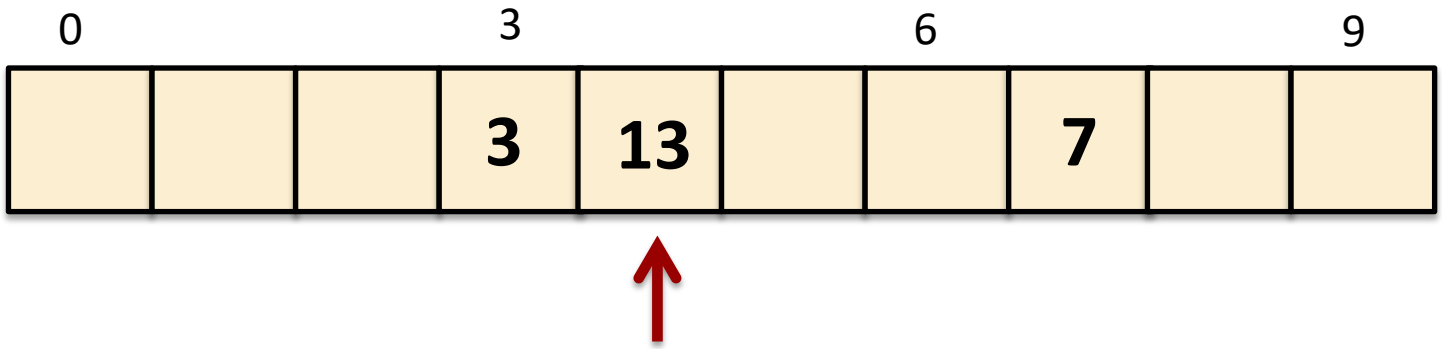
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 13
  - 13%10=3=collision & overflow!

Linear Probing

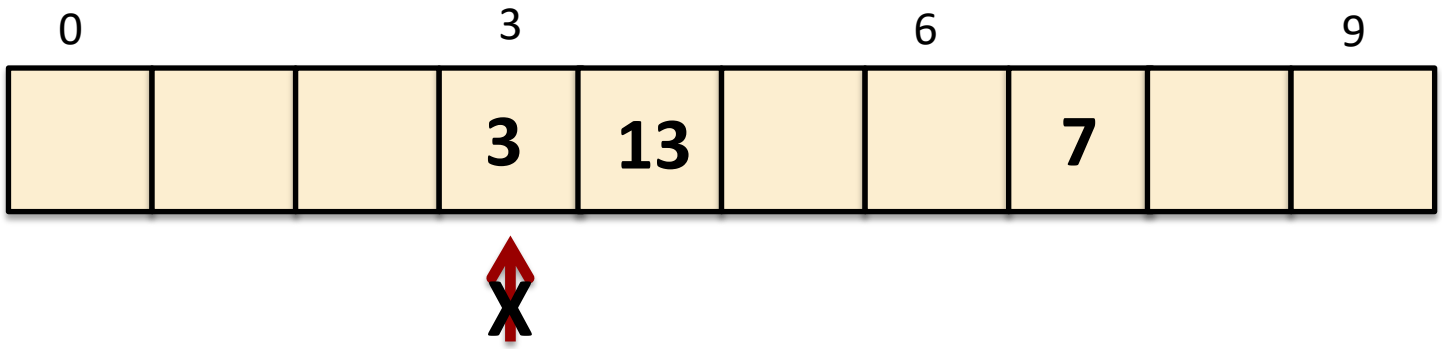
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 13
  - (13+1)%10=4

Linear Probing

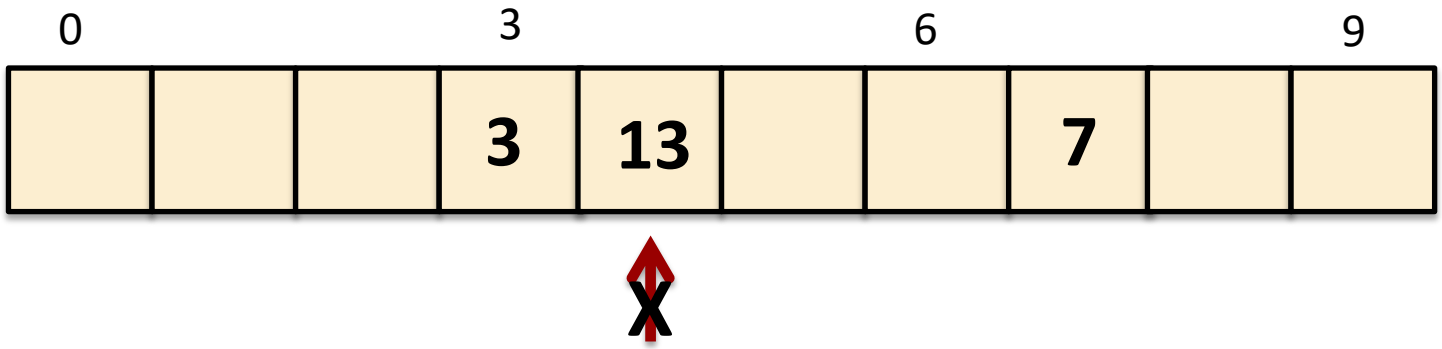
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 23
  - 23%10=3=collision & overflow!

Linear Probing

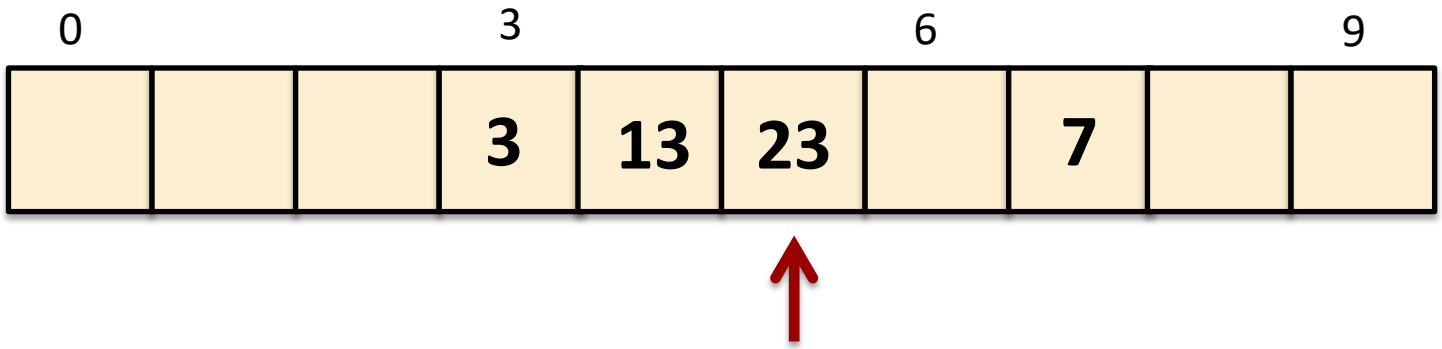
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 23
  - (23+1)%10=4=collision & overflow!

Linear Probing

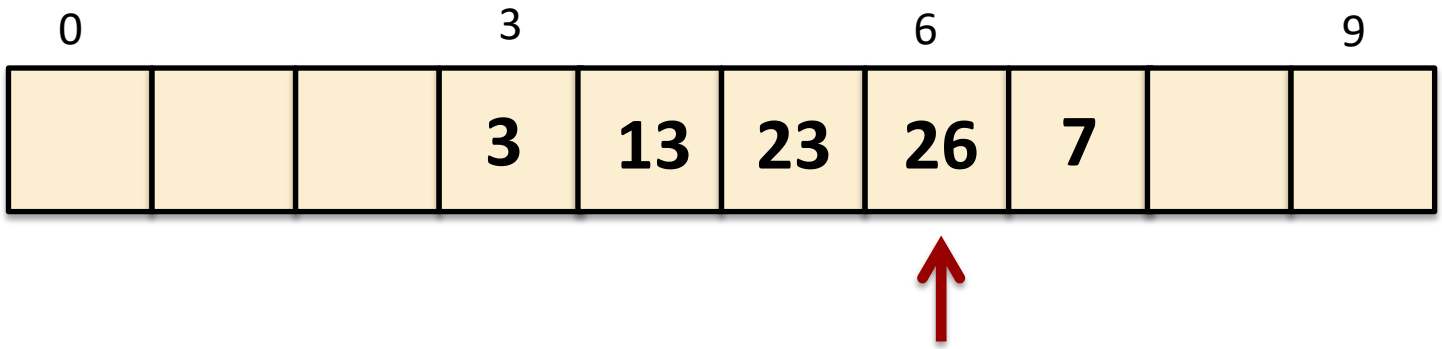
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 23
  - (23+2)%10=5

Linear Probing

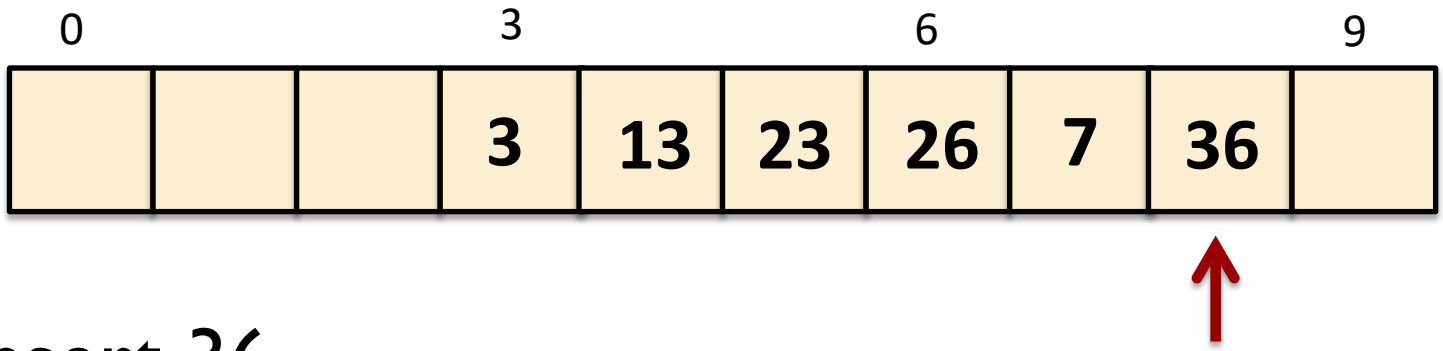
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 26
  - 26%10=6

Linear Probing

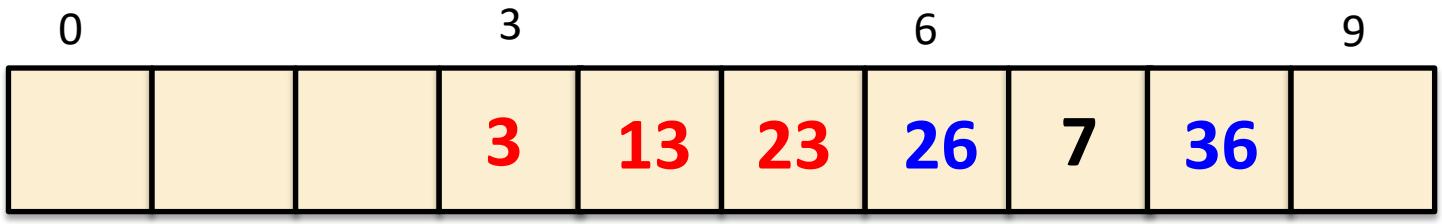
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Insert 36
  - 36%10=6=collision & overflow!
  - Next available bucket: 8

Linear Probing

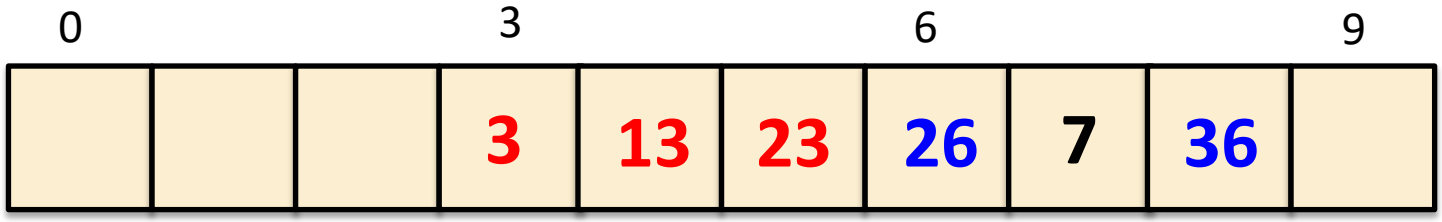
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Same color : same hash value group

Linear Probing

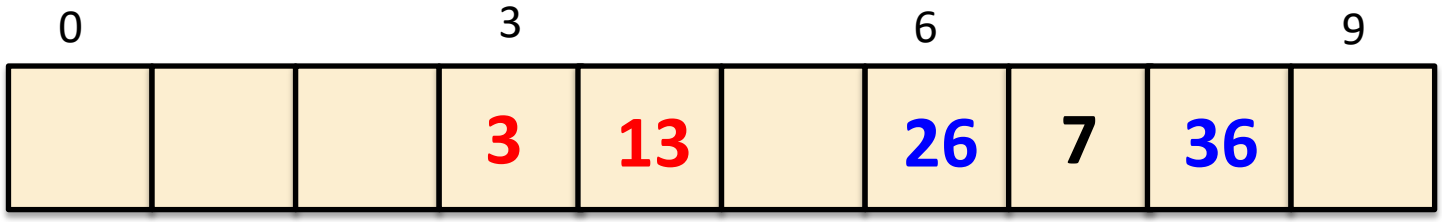
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Delete 23
  - Search the right cluster if there is a key to shift to left

Linear Probing

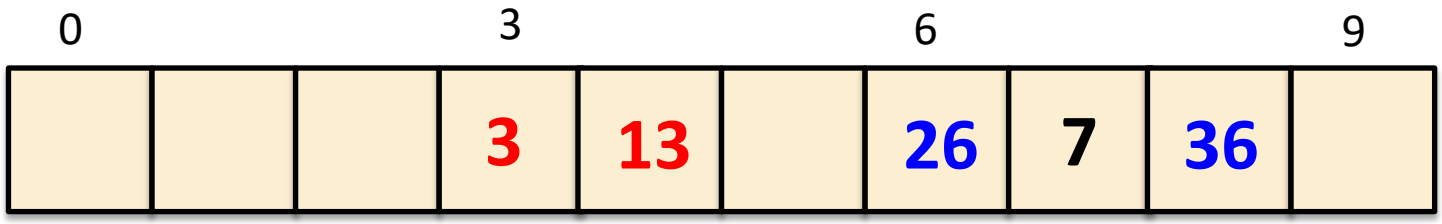
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Delete 23
  - h(26) = 6, h(7) = 7, h(36) = 8 (due to collision at 6) : no shifting required

Linear Probing

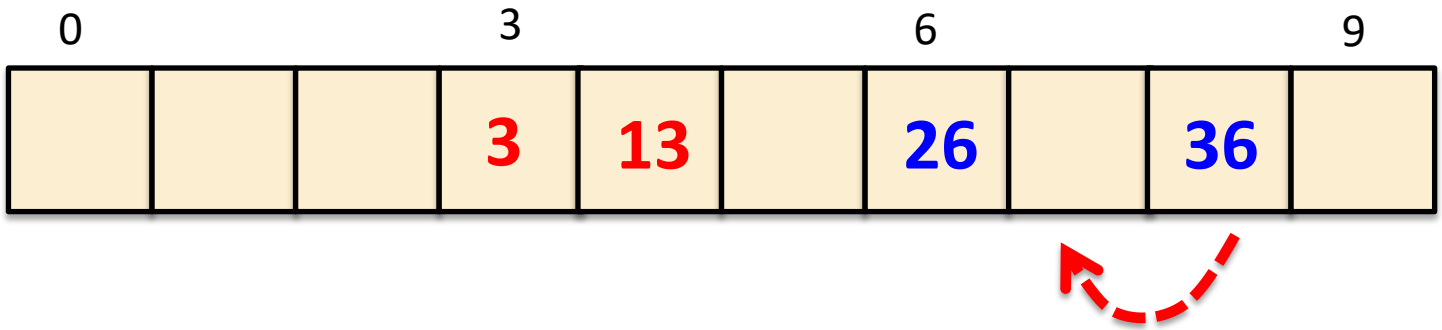
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Delete 7
  - Search right cluster (which is 36)

Linear Probing

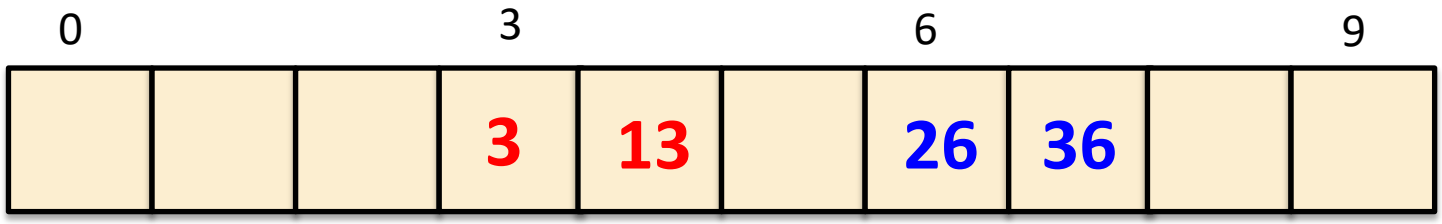
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Delete 7
  - h(36) = 7 (since 6 is collision and 7 is empty)

Linear Probing

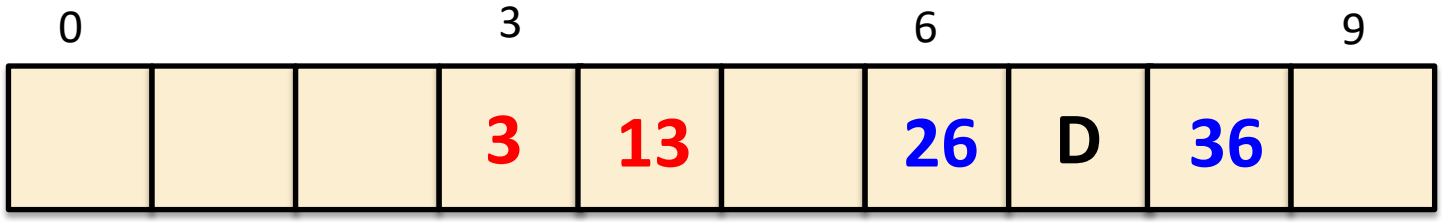
- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Delete 7
  - Shift 36 to left

Linear Probing

- Divisor = b (# of buckets) = 10
- h(k) = k % 10



- Delete without shifting
  - Mark as *deleted*, and a new key can be inserted to that location later (retain cluster)

Performance of Linear Probing

- Worst-case find/insert/delete time
  - O(n), when?
- S_n = expected number of buckets examined in a successful search when n is large
- U_n = expected number of buckets examined in an unsuccessful search when n is large
- Insert and delete time depend on U_n

Expected Performance

$$U_n \approx \frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right]$$

$$S_n \approx \frac{1}{2}\left[1+\frac{1}{1-\alpha}\right],$$

$$\alpha = \frac{n}{sb}: \text{ loading density}$$

α	S_n	U_n
0.50	1.5	2.5
0.75	2.5	8.5
0.90	5.5	50.5

α <= 0.75 is recommended

Discussion about Linear Probing

- Pros
  - Simple to compute
- Cons
  - Clustering
  - Worst case O(n)
  - Delete can be as expensive

Quadratic Probing

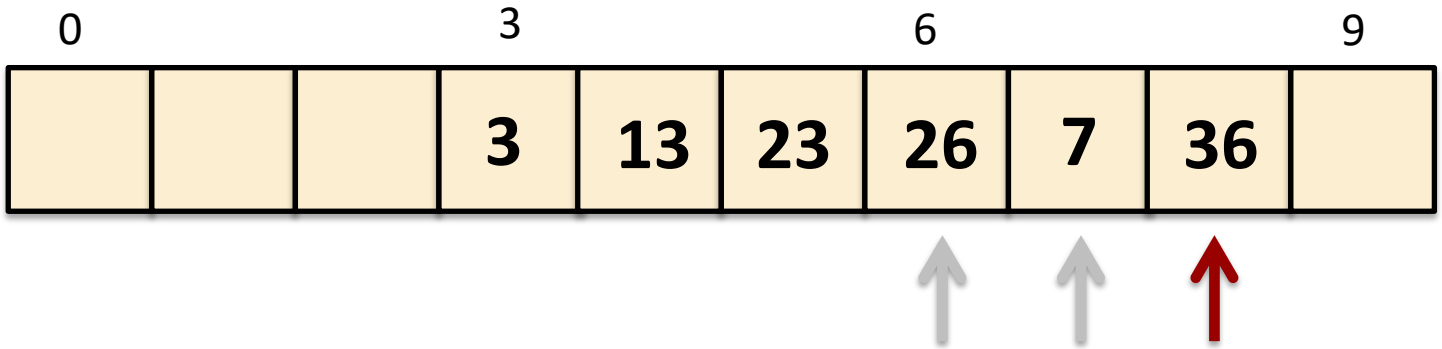
- Examining ht[(h(k)+j^2)%b] for j=0, 1, 2, ..., b-1
- Search the next available bucket from the original address by the distance 1, 4, 9, 16, ...
- Pros
  - Simple calculation, reduce clustering
- Cons
  - Not all buckets can be examined
  - Can be minimized if b is prime number

Other Open Addressing Methods

- Rehashing
  - Use a series of different hash functions $h_1, h_2, ..., h_m$
  - Examining $ht[h_j(k)]$ for $j=1, 2, ..., m$
  - Minimize clustering
- Random probing
  - Examining ht[(h(k)+s(i))%b] for i= 1, 2, ..., b-1
  - s(i) : pseudo random number between 1 to b-1, each number is generated only once

Chaining

- Problem of open addressing
  - Need to compare keys that have different hash values
- Ex) to find 36, comparing to 26 and 7 is required
  - h(7) != h(36), so this comparison is unnecessary

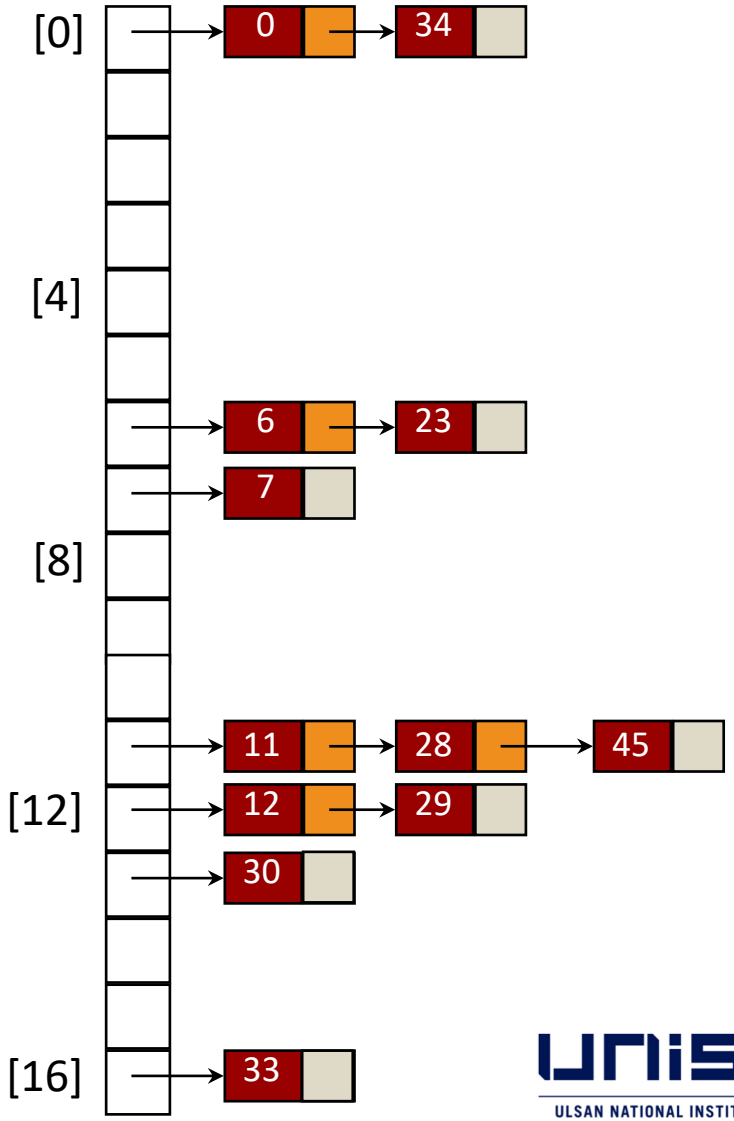


Chaining

- Linear list per each hash address
  - Chain (singly linked list) is often used
  - Sorted or unsorted
- ht[0:b-1] : has table with b buckets
- ht[i] : point to the first node of the chain for bucket i

Example: Sorted Chain

- Insert 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
- h[k] = k%17



Expected Performance

- Chaining

$$U_n \approx \alpha$$

$$S_n \approx 1 + \frac{\alpha}{2}$$

$$\alpha = \frac{n}{b} : \text{ loading density}$$

α	S_n	U_n
0.50	1.25	0.5
0.75	1.375	0.75
0.90	1.45	0.9

- Less calculation than open addressing
- Extra dynamic memory usage (pointer Ops)

Hash Table Design

- Maximum permissible loading density for given performance requirements
- e.g., linear probing
  - Max comparison for successful search : 10
    - $S_n \sim \frac{1}{2}(1 + 1/(1 - \alpha))$
    - $\alpha <= 18/19$
  - Max comparison for unsuccessful search : 13
    - $U_n \sim \frac{1}{2}(1 + 1/(1 - \alpha)^2)$
    - $\alpha <= 4/5$
  - Therefore, $\alpha <= \min\{18/19, 4/5\} = 4/5 = 0.8$

Hash Table Design

- Dynamic resizing of table
  - When loading density exceeds threshold
  - Array doubling (b to 2b+1)
  - Rehash old table into new large table (slow)
- Fixed table size
  - Maximum number of entries is known : x
  - Loading density <= y : b >= x/y
  - Pick b to be a prime number, or odd number with no prime divisor smaller than 20

Questions?