

HASHING*

PART 1

Revised Fall 2019

OBJECTIVES

Be able to:

- Define terms related to hashing, including: hash function, double hashing, collision, collision resolution, mapping, perfect hash function, load factor, cluster.
- Define how the following hashing techniques work:
 - Open addressing with linear probing
 - Open addressing with double hashing
 - Chaining
 - Hashing with buckets

SEARCH ALGORITHMS

Linear search:

- Best case efficiency: $O(1)$
- Worst case efficiency: $O(n)$
- Average case efficiency: $O(n)$
- Works on sorted or unsorted data

Binary search:

- Efficiency (all cases): $O(\log n)$
- Requires that the data is already sorted

Can we do better?

SEARCHING FOR INFORMATION: WHEN SPEED IS ESSENTIAL



Report an emergency from 555-5555

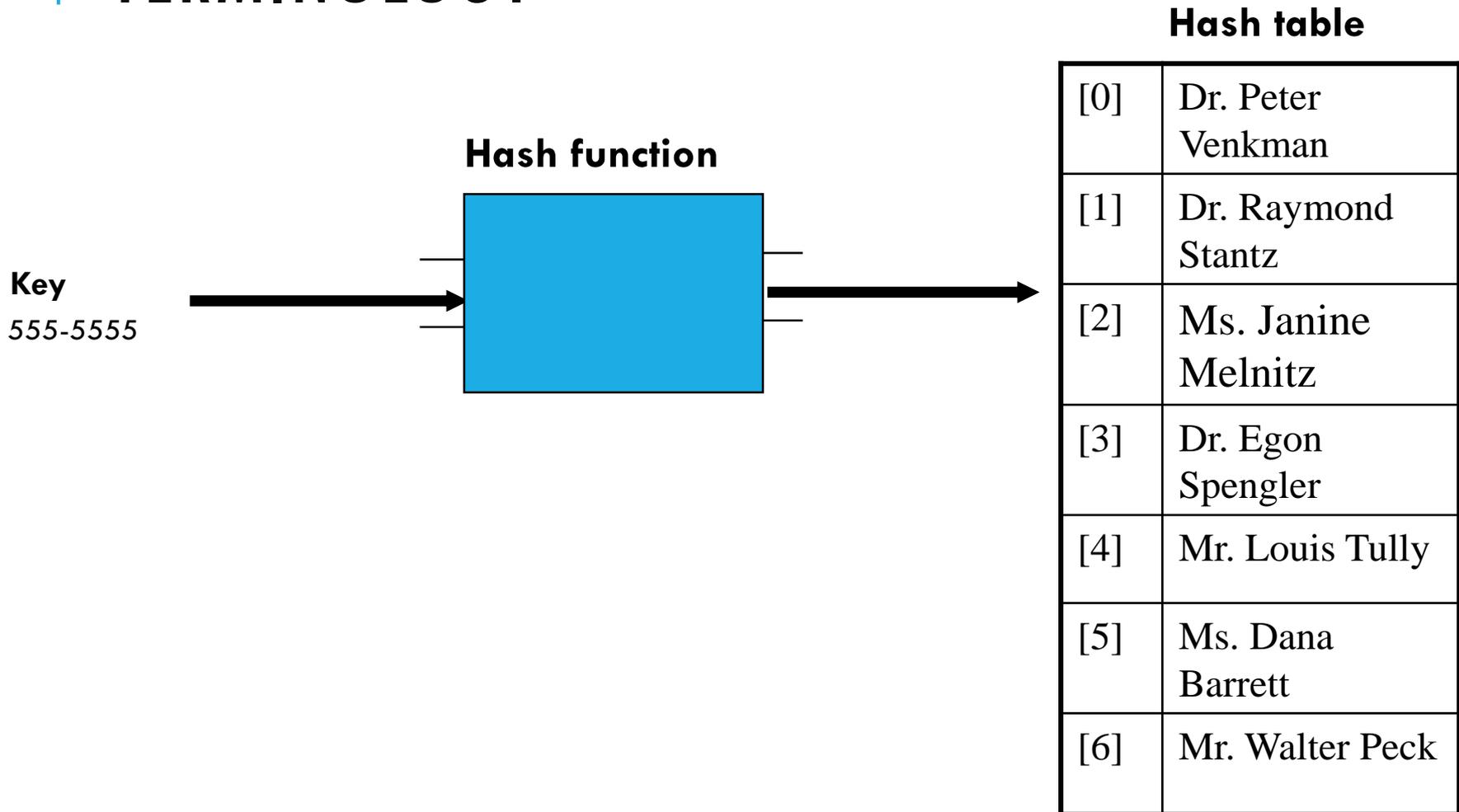


Map 555-5555 to a list element

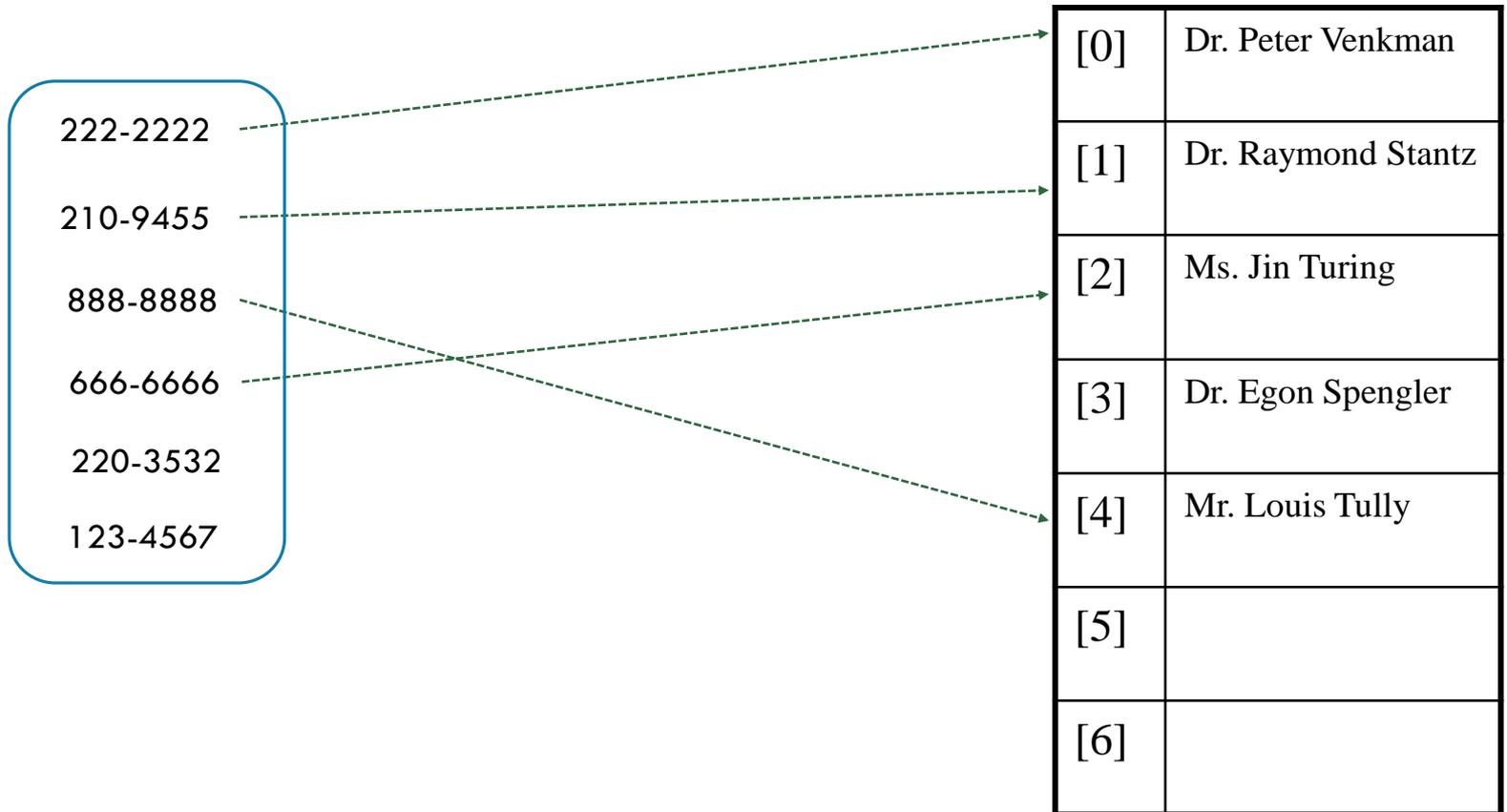


[0]	Dr. Peter Venkman
[1]	Dr. Raymond Stantz
[2]	Ms. Janine Melnitz
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully
[5]	Ms. Dana Barrett
[6]	Mr. Walter Peck
[7]	Mr. Winston Zeddemore
:	

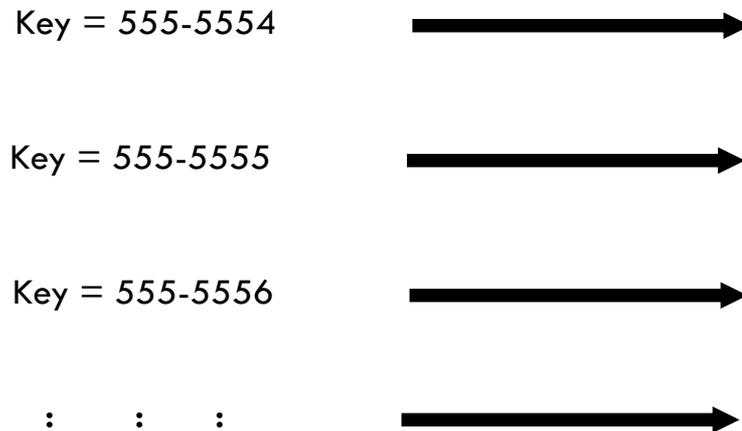
STORING/SEARCHING FOR INFORMATION: TERMINOLOGY



HASHING: MAPPING KEYS TO POSITIONS IN A TABLE (STORING OR SEARCHING)



AN EXAMPLE OF HOW HASHING CAN BE DONE: TAKE ONE

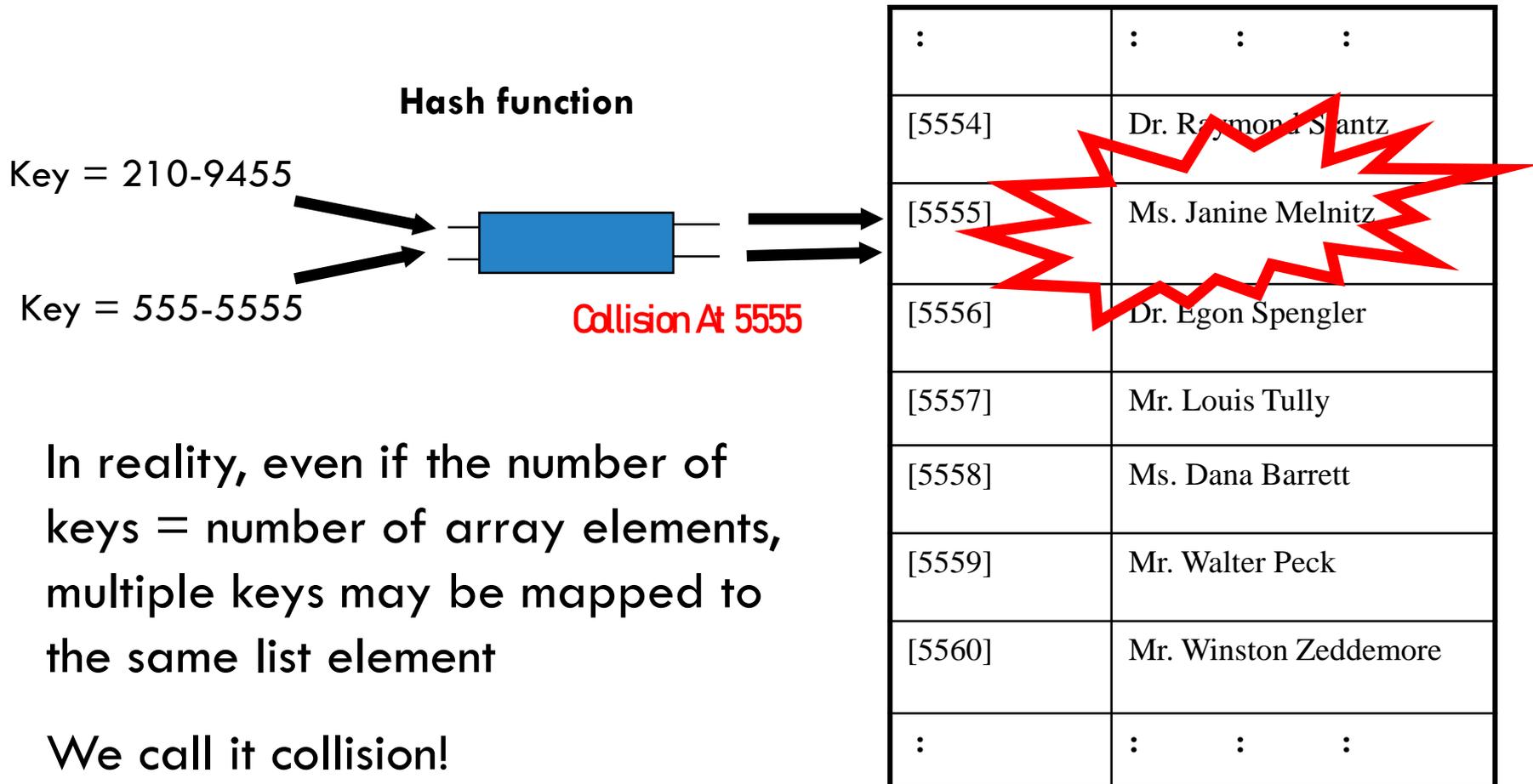


:	: : :
[5555554]	Dr. Raymond Stantz
[5555555]	Ms. Janine Melnitz
[5555556]	Dr. Egon Spengler
[5555557]	Mr. Louis Tully
[5555558]	Ms. Dana Barrett
[5555559]	Mr. Walter Peck
[5555560]	Mr. Winston Zeddemore
:	: : :

Perfect hash function: Each key (e.g., phone number) maps to a unique list entry (7 digit value)

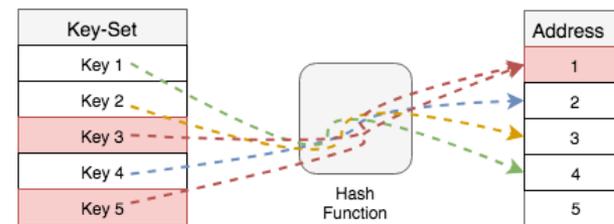
$O(1)$ search times but yields many empty elements

AN EXAMPLE OF HOW HASHING CAN BE DONE: TAKE TWO



COLLISION

- Two keys mapping to the same location in the hash table is called “Collision”
- Collisions can be reduced with a selection of a good hash function
- But it is not possible to avoid collisions altogether, unless we find a perfect hash function
- Perfect Hash Function
 - A hash function that maps every key to a unique location in the hash table
- Which is hard to do! Because:
 - All of the keys are not known in advance
 - e.g., flight numbers mapping to actual flights
 - Only a small percentage of the possible key combinations are used.
 - e.g., a company with 500 employees would not create a hash table mapped to the 1 billion combinations of SIN numbers



EXAMPLE HASH FUNCTIONS

Selecting digits

Folding

Modular arithmetic

Converting characters to integers

EXAMPLE HASH FUNCTION: SELECTING DIGITS

Select a portion of the key to use as the index into the hash table

Works only for keys that are positive integers.

Pro:

- The mapping of a key to an index is quick.

Con:

- It usually does not evenly distribute items through the hash table (may lead to many collisions or clustering around certain parts of the hash table).

403-210-9455



Hash function:
Select the even
position digits
starting with the 4th
digit

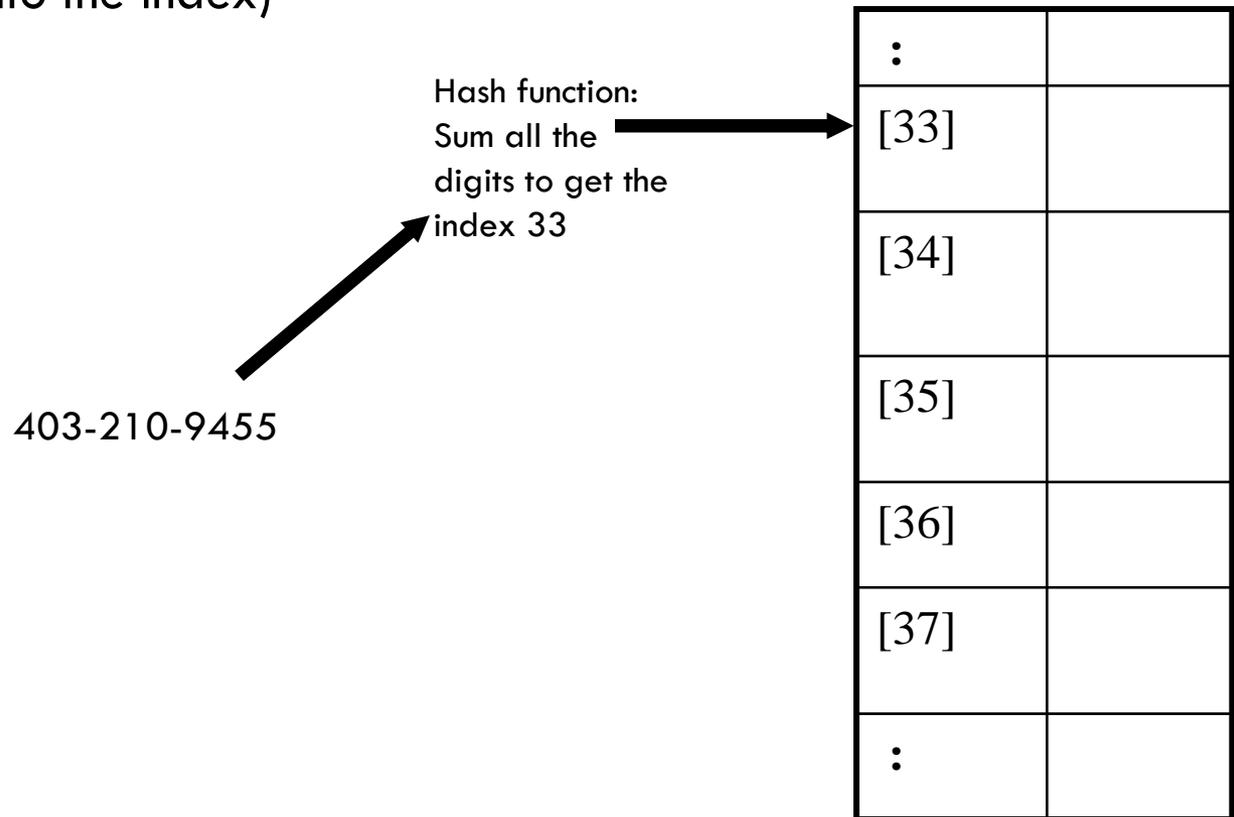


:	
[2045]	
[2046]	
[2047]	
[2048]	
[2049]	
:	

EXAMPLE HASH FUNCTION: FOLDING

An improvement of the previous method because the entire number is used (folded into the index)

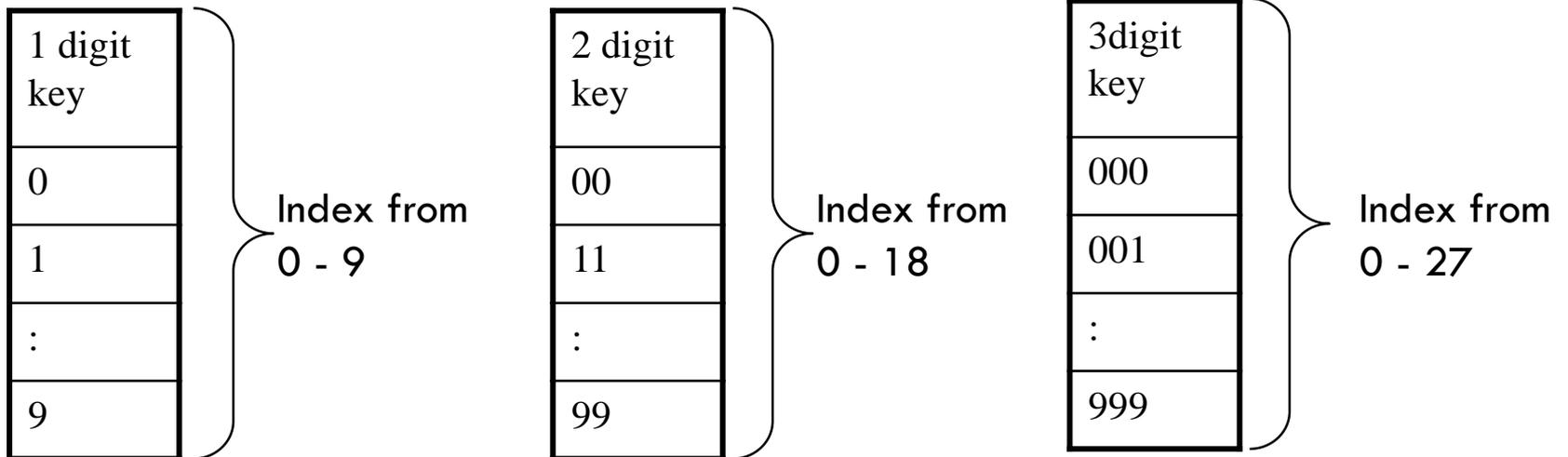
Example 1:



EXAMPLE HASH FUNCTION: FOLDING

Analysis of the example:

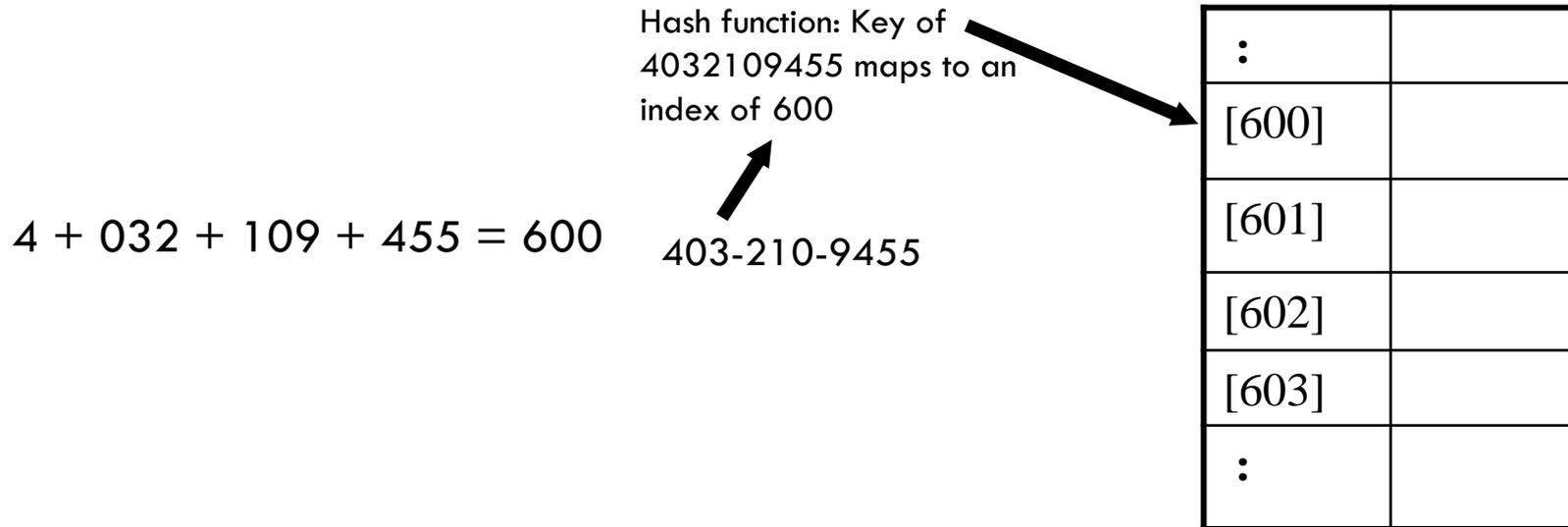
- Range of possible keys is limited to the number of digits of the key.



EXAMPLE HASH FUNCTION: FOLDING

To increase the size of the hash table (and increase the range of possible values generated by the hash function) groups of numbers can be added instead of individual numbers.

Example 2:



EXAMPLE HASH FUNCTION: FOLDING

Other examples of hashing algorithms that employ folding could combine selecting certain digits to be “folded” into a key e.g., sum only the odd positioned digits.

Other mathematical/bitwise operations could be employed e.g., multiplying digits together, bit shifting or bit rotating the numerical values.

The quality of the hash function using folding will vary.

EXAMPLE HASH FUNCTION: MODULAR ARITHMETIC

The index = (key) modulo (table size)

Example:

1250



Hash function:
1250 modulo
100 = 50



[0]	
:	
[50]	
[51]	
:	
[99]	

EXAMPLE HASH FUNCTION: MODULAR ARITHMETIC

The index = (key) modulo (table size)

- In Java the modulo operator is “%”.

To ensure an even distribution of keys to the different parts of the table, the table size should be prime number (e.g., 101)

Why?!

RATIONALE

If we are adding numbers $a_1, a_2, a_3 \dots a_4$ to a table of size m

- All values will be hashed into multiples of $\gcd(a_1, a_2, a_3 \dots a_4, m)$

- For example, if we are adding 64, 100, 128, 200, 300, 400, 500 to a table of size 8, all values will be hashed to 0 or 4

$$\gcd(64, 100, 128, 200, 300, 400, 500, 8) = 4$$

- When m is a prime $\gcd(a_1, a_2, a_3 \dots a_4, m) = 1$, all values will be hashed to anywhere

$$\gcd(64, 100, 128, 200, 300, 400, 500, 7) = 1$$

unless $\gcd(a_1, a_2, a_3 \dots a_4) = m$, which is rare.

EXAMPLE HASH FUNCTION: CONVERTING CHARACTERS TO INTEGERS

If the search key is a string of characters, computing the index could be a two step process:

- Convert the characters to an integer value e.g., Unicode

Character key: 'T' 'O' 'N' 'E'

Integer value: 84 79 78 69

- Apply one of the previous hash functions to the integer values

Note: To avoid having anagrams (e.g., “NOTE” and “TONE”) yielding the same integer value concatenate rather than add the results.

- TONE: $84\ 79\ 78\ 69 = 84797869$
- NOTE: $78\ 79\ 84\ 69 = 78798469$

OTHER METHODS

Truncation:

- e.g. 123456789 map to a table of 1000 addresses by picking the last 3 digits of the key: $H(\text{IDNum}) = \text{IDNum} \% 1000 = \text{hash value}$

Squaring:

- Square the key and then truncate

Radix conversion:

- e.g. 1 2 3 4 treat it to be base 11, truncate if necessary.

CHARACTERISTICS OF A GOOD HASH FUNCTION

It should be as uncomplicated as possible and fast to compute e.g., a single mathematical or bitwise operation.

It should scatter the data evenly throughout the hash table – collisions are unavoidable except for the case of perfect hashing but they should be minimized.

- The calculation of the index should involve the entire search key.
- If the hash function uses modulo arithmetic then the base should be prime
 - e.g., $\text{index} = \text{key} \text{ MODULO } \langle \text{base} \rangle$

COLLISION RESOLUTION TECHNIQUES

Open hashing/Separate chaining

- Restructuring addressing to the hash table: Open hashing/closed addressing
 - Separate (external) chaining
 - Buckets

Closed hashing/Open addressing

- Linear probing
- Quadratic probing
- Double hashing using key-dependent increments
- Increasing the size of the hash table: rehashing

CLOSED HASHING / OPEN ADDRESSING

When collisions occur, find a new table entry to make the insertion.

Each table entry can only store one key.

Closed Hashing: all keys are stored in the **hash** table itself without the use of linked lists. AKA **Open Addressing**; allocation within the hash table is open when collision occurs

Index	Key

Closed hashing:
Can't look outside the table for new places to insert hashed keys.

CLOSED HASHING / OPEN ADDRESSING

When collisions occur, find a new table entry to make the insertion.

Each table entry can only store one key.

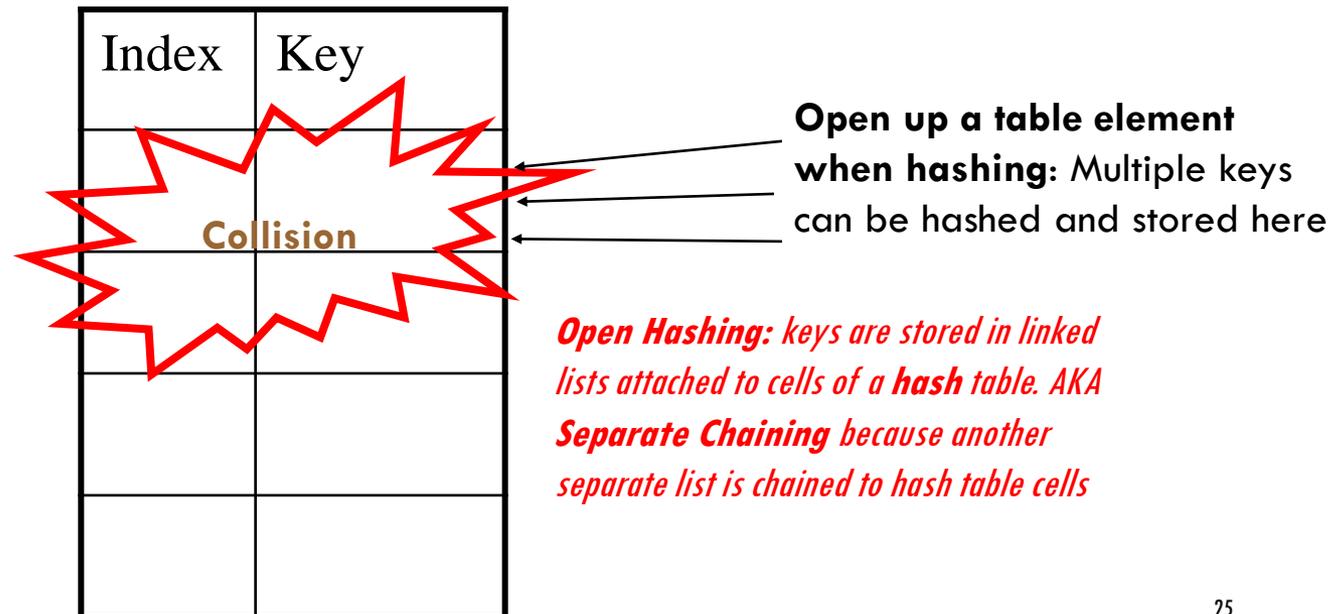
Open addressing: When a collision occurs, the other addresses in the table are “opened up” as possible locations to hash to.

Index	Key
[1]	
[2]	
[3]	

OPEN HASHING / CLOSED ADDRESSING

When a collision occurs, accommodate the additional key by adding additional keys at the same location in the table.

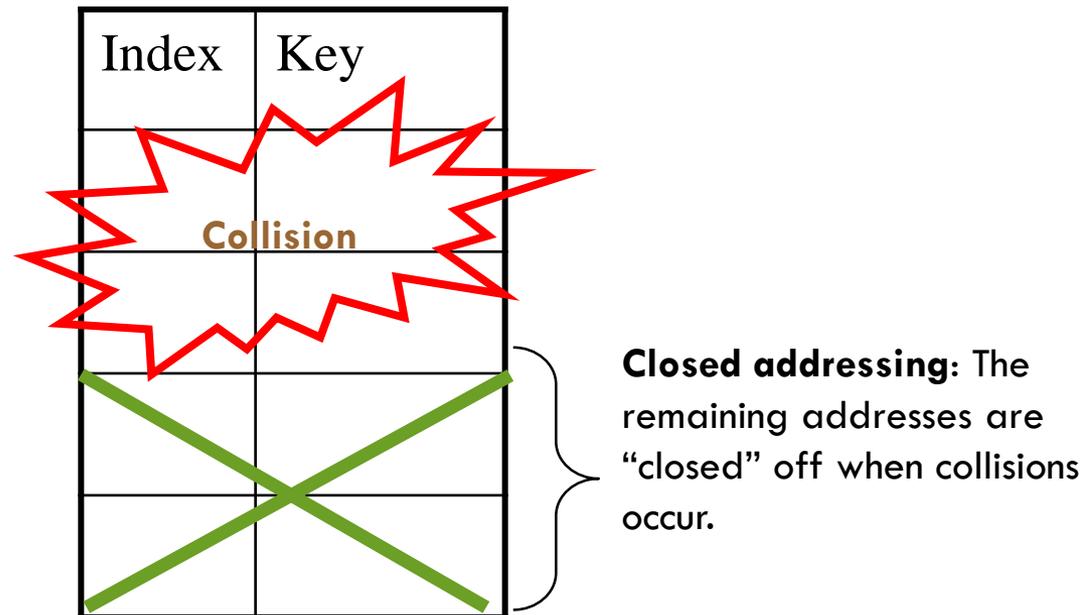
Each table entry can only store multiple keys.



OPEN HASHING / CLOSED ADDRESSING

When a collision occurs, accommodate the addition key by adding additional keys at the same location in the table.

Each table entry can only store multiple keys.



OPEN HASHING/SEPARATE CHAINING

Change the structure of the hash table so that if collisions occur, each location in the hash table can accommodate multiple keys.

- The entries of the hash table are pointers/references to lists.
- A new item is inserted to the list if it hashes to the location.

Advantages:

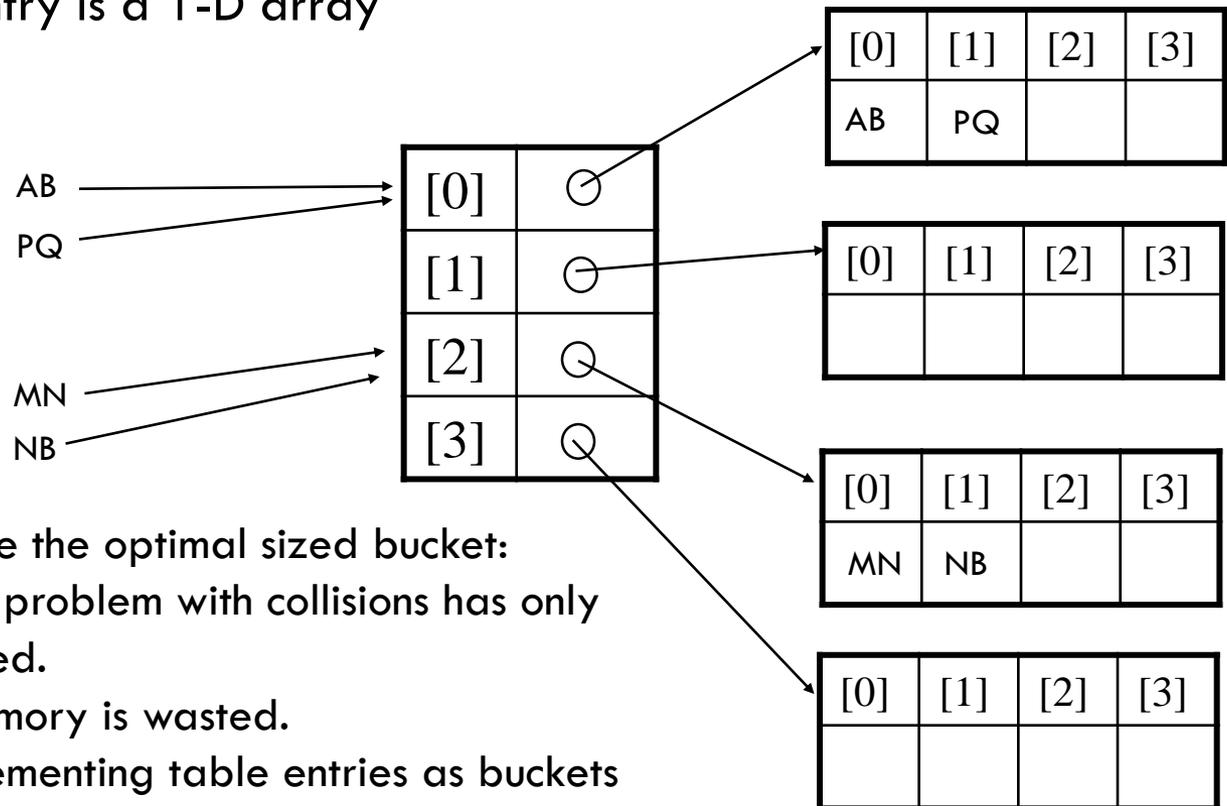
- Better space utilization for large items.
- Simple collision handling: searching linked list.
- Overflow: we can store more items than the hash table size.
- Deletion is quick and easy: deletion from the linked list.

Implementation:

- Buckets(arrays)
- Linked lists

SEPARATE CHINNING- BUCKETS

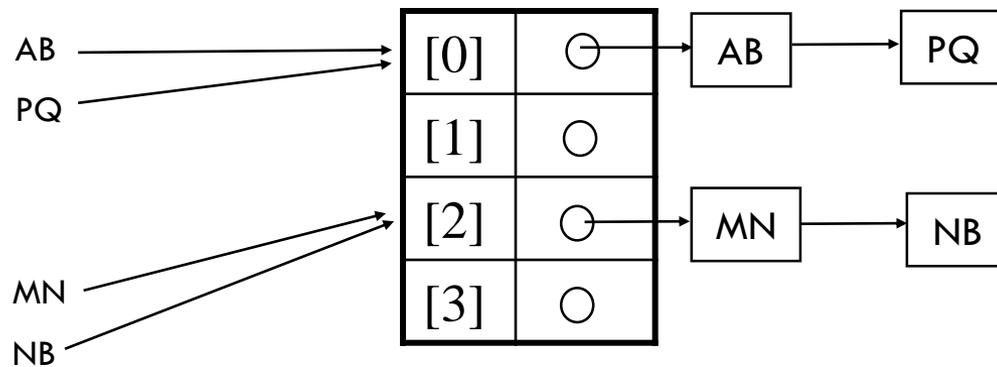
Each hash table entry is a 1-D array



- Issue: How to choose the optimal sized bucket:
 - Too small: The problem with collisions has only been postponed.
 - Too large: Memory is wasted.
- Consequence: Implementing table entries as buckets is seldom done in actual practice.

SEPARATE CHAINING- LINKED LISTS

Each table entry is a reference to a linked list.



SEPARATE CHAINING- EXAMPLE

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/
9	/

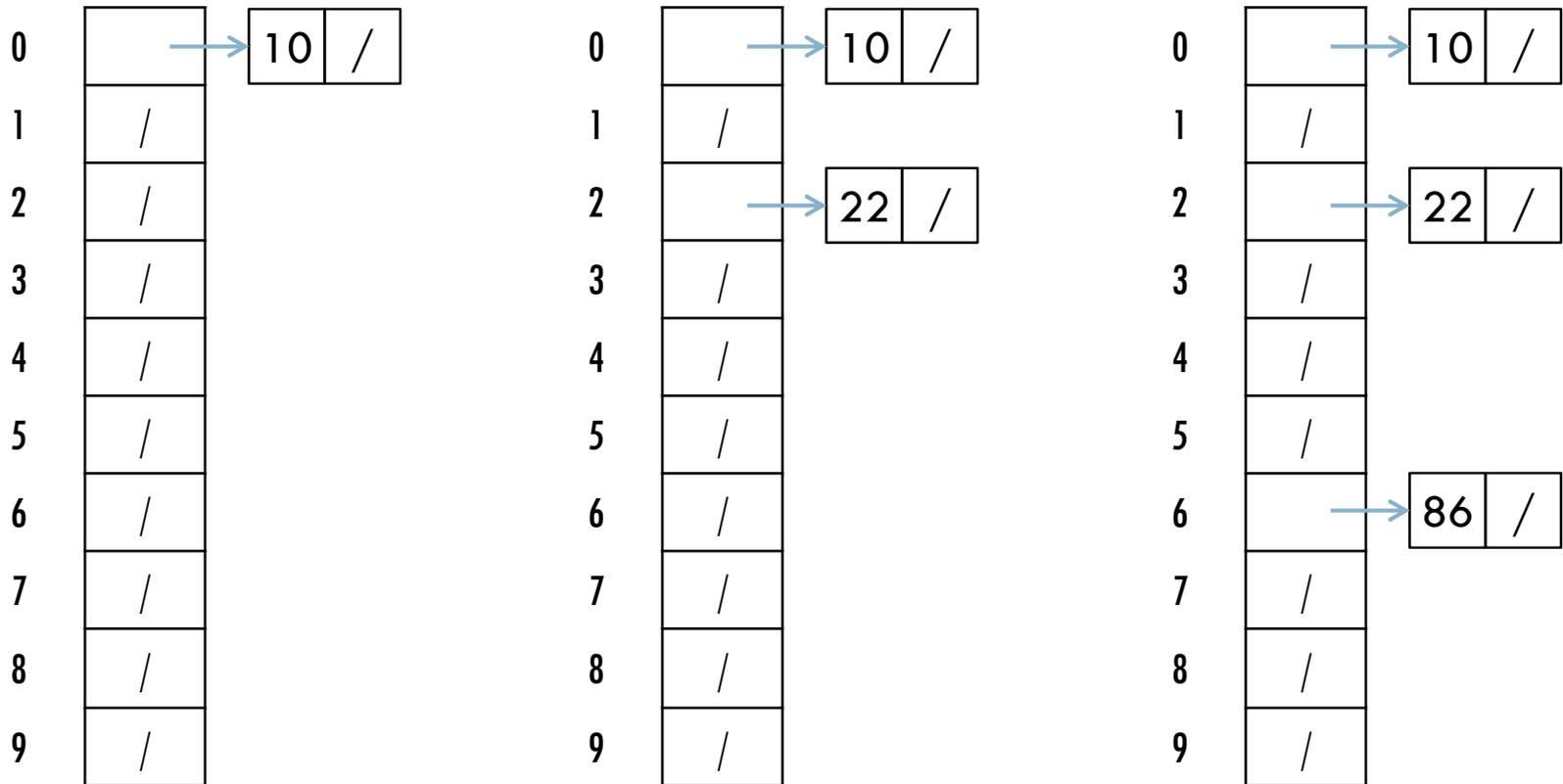
All keys that map to the same table location are kept in a linked list (a.k.a. a "chain" or "bucket")

As easy as it sounds

Example:

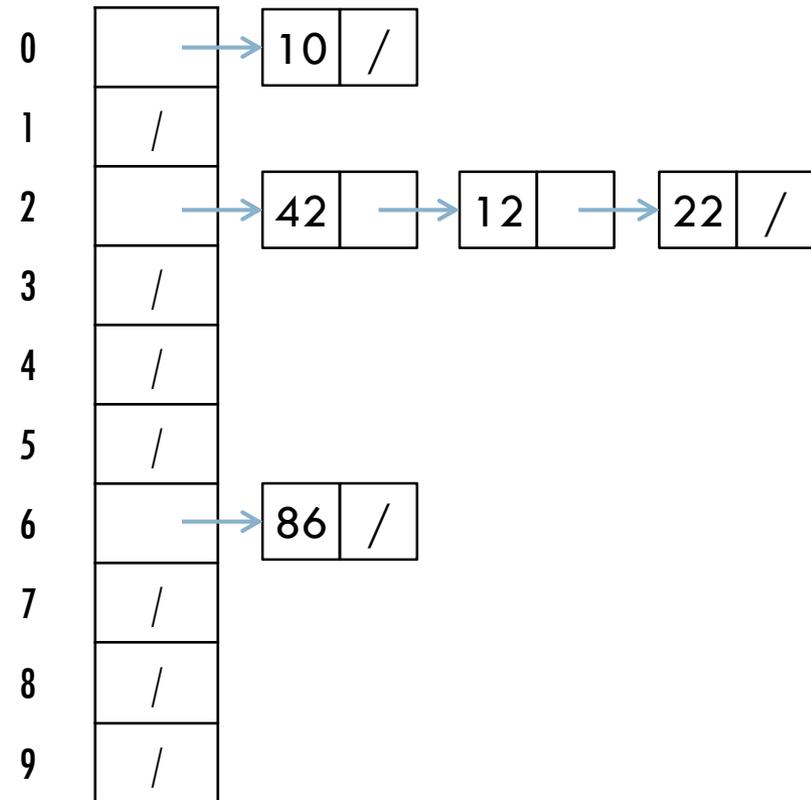
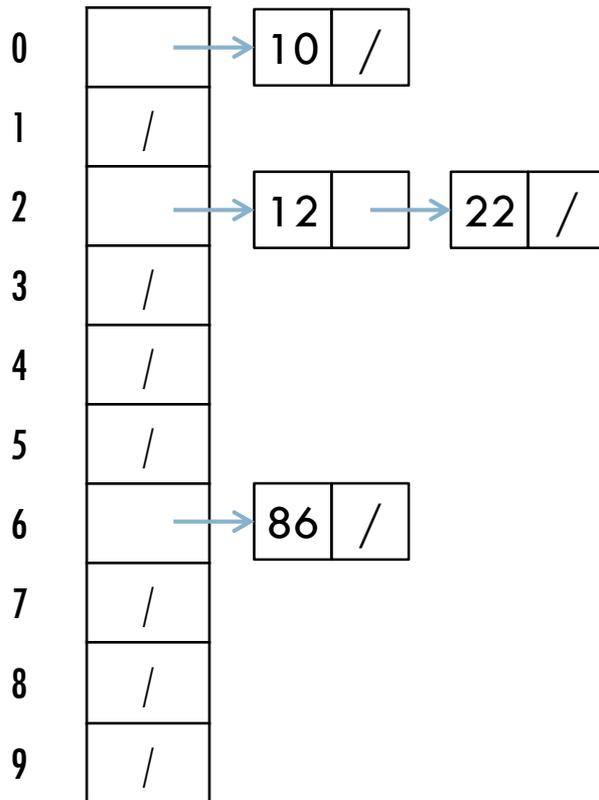
insert 10, 22, 86, 12, 42
with $h(x) = x \% 10$

SEPARATE CHAINING



Insert 10, 22, 86, 12, 42 with $h(x) = x \% 10$

SEPARATE CHAINING



Insert 10, 22, 86, 12, 42 with $h(x) = x \% 10$

THOUGHTS ON SEPARATE CHAINING

Separate chaining does not use all the space in the table

Worst-case time for search/find?

- Linear
- But only with really bad luck or bad hash function
- Not worth avoiding (e.g., with balanced trees at each bucket)
 - Keep small number of items in each bucket
 - Overhead of tree balancing not worthwhile for small n

Beyond asymptotic complexity, some "data-structure engineering" can improve constant factors

- Linked list, array, or a hybrid
- Insert at end or beginning of list
- Sorting the lists gains and loses performance
- Splay-like: Always move item to front of list



NEXT

Closed hashing- open addressing.