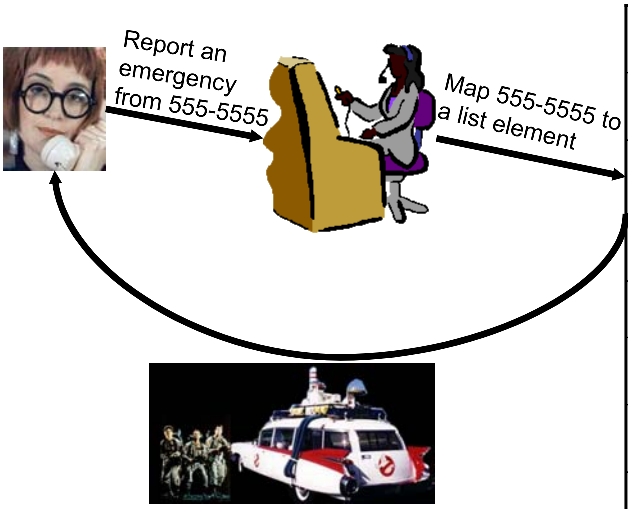
Introduction To Hashing

- In this section of notes you will learn an approach for organizing information that allows for searches in constant time

Searching For Information: Algorithms You Know

- Linear search:
  - Best case efficiency: $O(1)$
  - Worse case efficiency: $O(N)$
  - Average case efficiency: $O(N)$
  - Works on sorted or unsorted data
- Binary search:
  - Efficiency (all cases): $O(\log_2 N)$
  - Requires that the data is already sorted
- Interpolation search:
  - Best case efficiency: $O(1)$
  - Worse case efficiency: $O(N)$
  - Average case efficiency: $\sim O(\log_2(\log_2 n))$
  - Requires that the data is already sorted
  - Works most efficiently when the data is uniformly distributed

Searching For Information: When Speed Is Essential

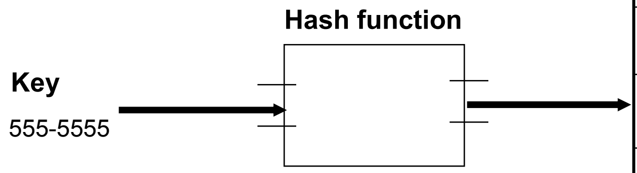


Report an emergency from 555-5555

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



Key

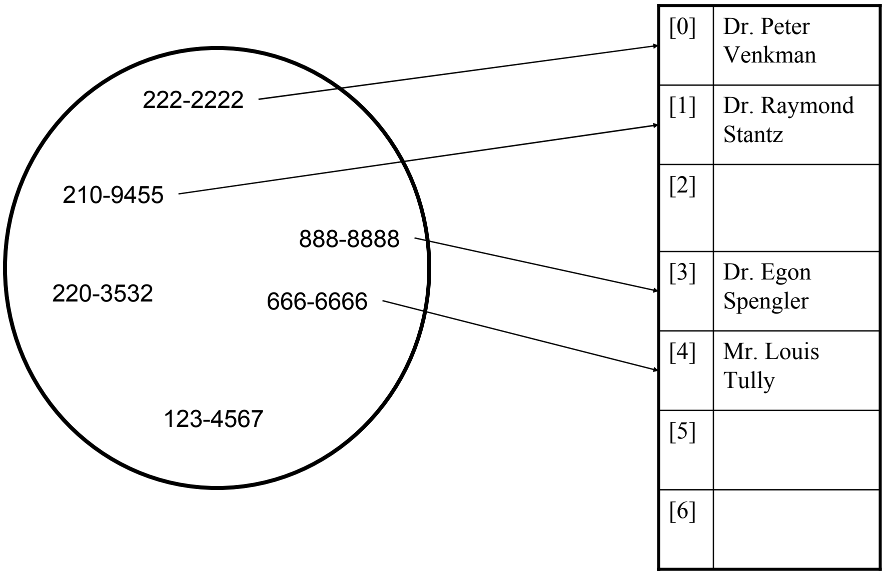
555-5555

Hash function

Hash table

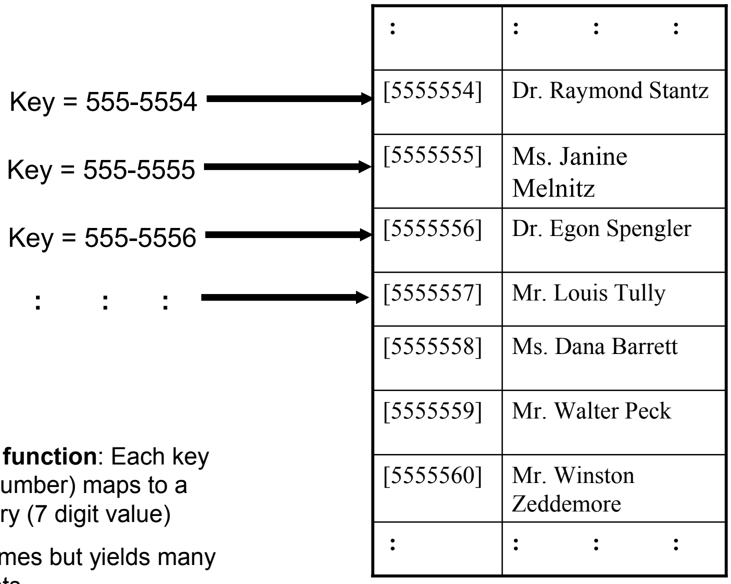
[0]	Dr. Peter Venkman
[1]	Dr. Raymond Stantz
[2]	Ms. Janine Melnitz
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully
[5]	Ms. Dana Barrett
[6]	Mr. Walter Peck

Hashing Is The Mapping Of Keys To Positions In A Table: Storing Or Searching



[0]	Dr. Peter Venkman
[1]	Dr. Raymond Stantz
[2]	
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully
[5]	
[6]	

An Example Of How Hashing Can Be Done: Take One

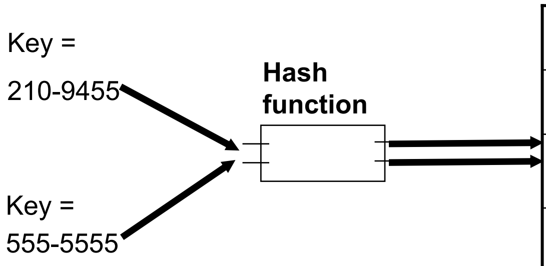


:	: : :
[5555554]	Dr. Raymond Stantz
[5555555]	Ms. Janine Melnitz
[5555556]	Dr. Egon Spengler
[5555557]	Mr. Louis Tully
[5555558]	Ms. Dana Barrett
[5555559]	Mr. Walter Peck
[5555560]	Mr. Winston Zeddemore
:	: : :

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

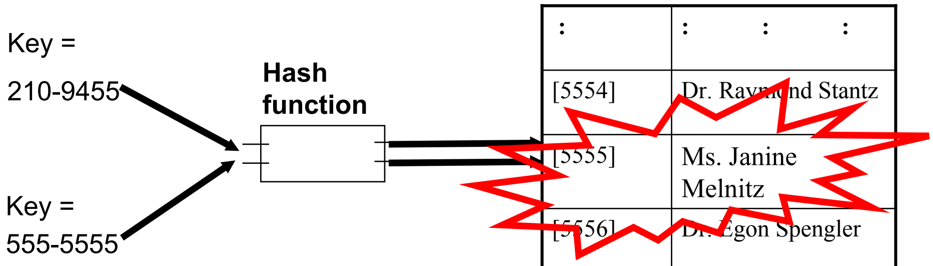
An Example Of How Hashing Can Be Done: Take Two



:	: : :
[5554]	Dr. Raymond Stantz
[5555]	Ms. Janine Melnitz
[5556]	Dr. Egon Spengler
[5557]	Mr. Louis Tully
[5558]	Ms. Dana Barrett
[5559]	Mr. Walter Peck
[5560]	Mr. Winston Zeddemore
:	: : :

In reality, even if the number of keys = number of array elements, multiple keys may be mapped to the same list element

Collision At 5555



:	: : :
[5554]	Dr. Raymond Stantz
[5555]	Ms. Janine Melnitz
[5556]	Dr. Egon Spengler
[5557]	Mr. Louis Tully
[5558]	Ms. Dana Barrett
[5559]	Mr. Walter Peck
[5560]	Mr. Winston Zeddemore
:	: : :

In reality, even if the number of keys = number of array elements, multiple keys may be mapped to the same list element

Perfect Hash Functions

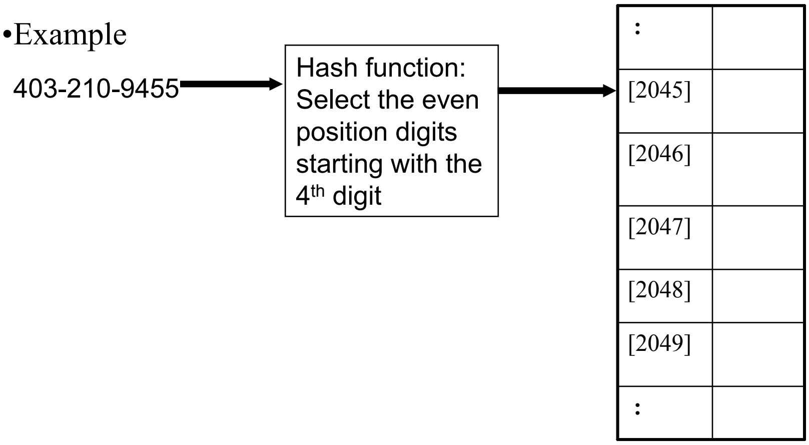
- Ideal case: Every key maps to a **unique** location in the hash table.
- Typically not possible because:
    1. All of the keys are not known in advance
        - e.g., flight numbers mapping to actual flights
    2. Only a small percentage of the possible key combinations are used.
        - e.g., a company with 500 employees would not create a hash table mapped to the 1 billion combinations of SIN numbers

Example Hash Functions

1. Selecting digits
2. Folding
3. Modular arithmetic
4. Converting characters to integers

1. 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.
- Example



403-210-9455

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

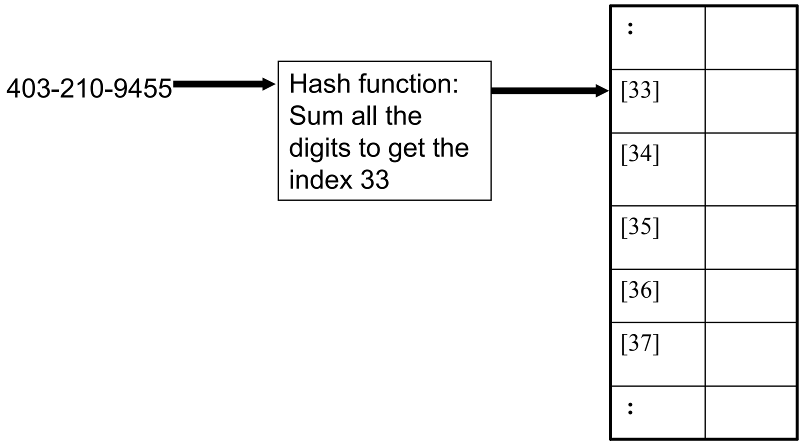
:	
[2045]	
[2046]	
[2047]	
[2048]	
[2049]	
:	

1. Example Hash Function: Selecting Digits (2)

- The mapping of a key to an index is quick.
- It usually does not evenly distribute items through the hash table (may lead to many collisions or clustering around certain parts of the has table).

2. Example Hash Function: Folding

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



403-210-9455

Hash function: Sum all the digits to get the index 33

:

[33]

[34]

[35]

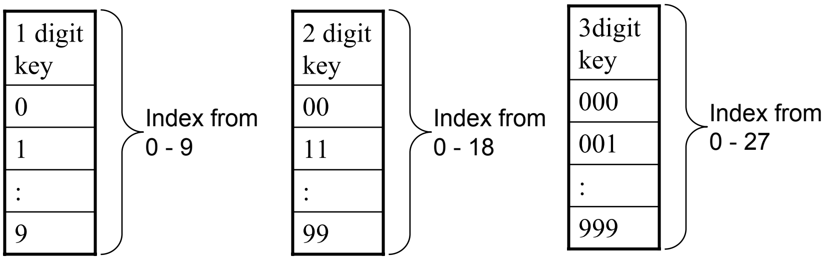
[36]

[37]

:

2. Example Hash Function: Folding (2)

- Analysis of the example:
  - Range of possible keys is limited:



1 digit key

0

1

:

9

Index from 0 - 9

2 digit key

00

11

:

99

Index from 0 - 18

3digit key

000

001

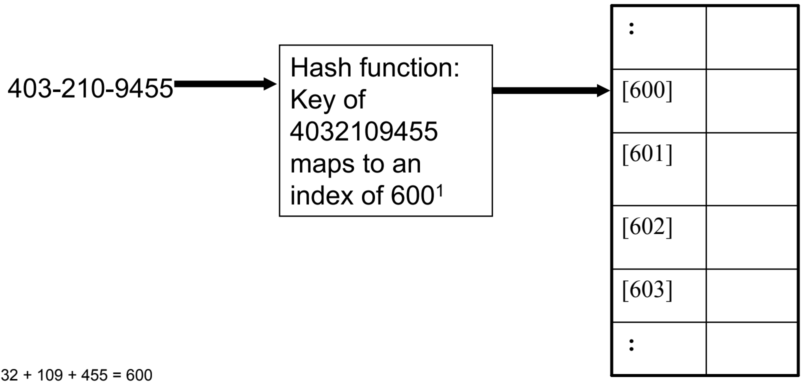
:

999

Index from 0 - 27

2. Example Hash Function: Folding (3)

- 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 two:



403-210-9455

Hash function: Key of 4032109455 maps to an index of 600[1]

:

[600]

[601]

[602]

[603]

:

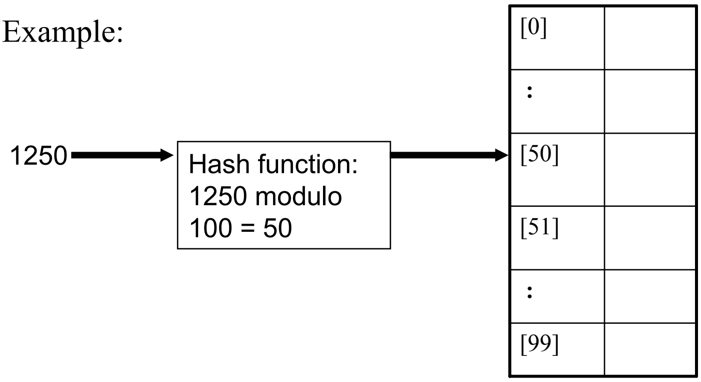
1: 4 + 032 + 109 + 455 = 600

2. Example Hash Function: Folding (4)

- 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.

3. Example Hash Function: Modular arithmetic

- The index = (key) modulo (table size)
- Example:



1250

Hash function:
1250 modulo
100 = 50

[0]

:

[50]

[51]

:

[99]

3. Example Hash Function: Modular arithmetic (2)

- 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)

4. 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:
  1. Convert the characters to an integer value e.g., Unicode

Character key:	'T'	'O'	'N'	'E'
Integer value:	84	79	78	69

  2. 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

Characteristics Of A Good Hash Function

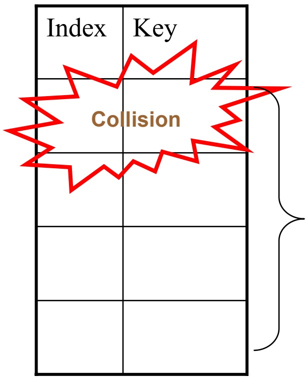
1. It should be as uncomplicated as possible and fast to compute e.g., a single mathematical or bitwise operation.
2. It should scatter the data evenly throughout the hash table – collisions are unavoidable except for the case of perfect hashing but they should be minimized.
   a) The calculation of the index should involve the entire search key.
   b) If the hash function uses modulo arithmetic then the base should be prime
      - e.g., index = key MODULO <base>

Collision Resolution Techniques

1. Closed hashing/Open addressing
   a) Linear probing
   b) Quadratic probing
   c) Double hashing using key dependent increments
   d) Increasing the size of the hash table: rehashing
2. Restructuring the hash table: Open hashing/closed addressing
   a) Buckets
   b) Separate (external) chaining

Closed Hashing / Open Addressing

- When collisions occur, find a new table entry to make the insertion.
- Each table entry can only store one key.

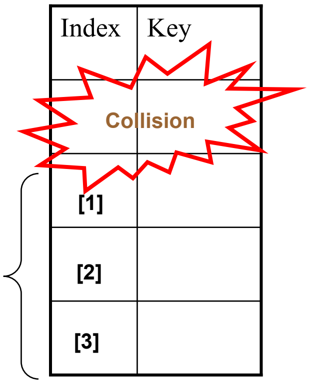


Closed hashing: Can't look outside the table for new places to insert hashed keys.

Closed Hashing / Open Addressing

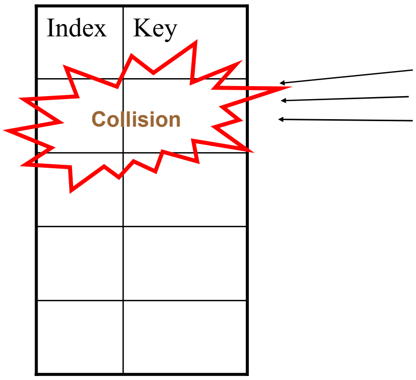
- When collisions occur, find a new table entry to make the insertion.
- Each table entry can only store one key.

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



Open Hashing / Closed Addressing

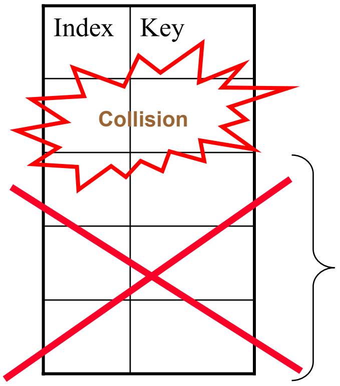
- When a collision occurs, accommodate the addition key by adding additional keys at the same location in the table.
- Each table entry can only store multiple keys.



Open up a table element when hashing: Multiple keys can be hashed and stored here

Open Hashing / Closed Addressing

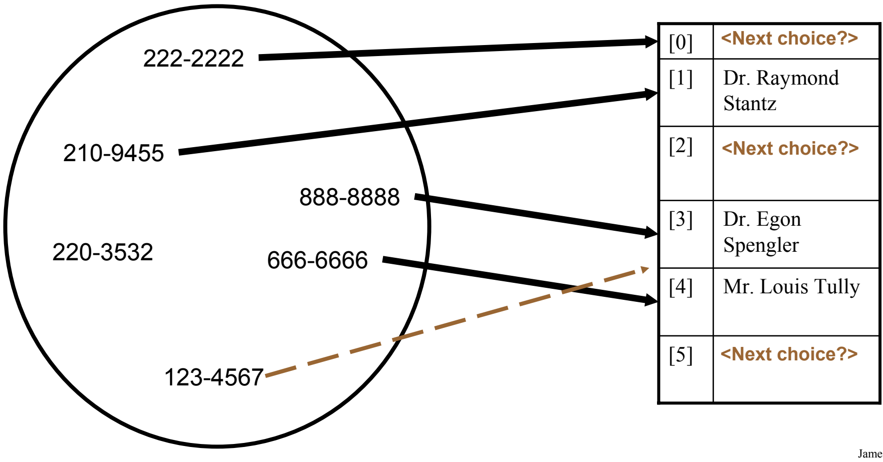
- When a collision occurs, accommodate the addition key by adding additional keys at the same location in the table.
- Each table entry can only store multiple keys.



Closed addressing: The remaining addresses are "closed" off when collisions occur.

1. Closed Hashing / Opening Addressing

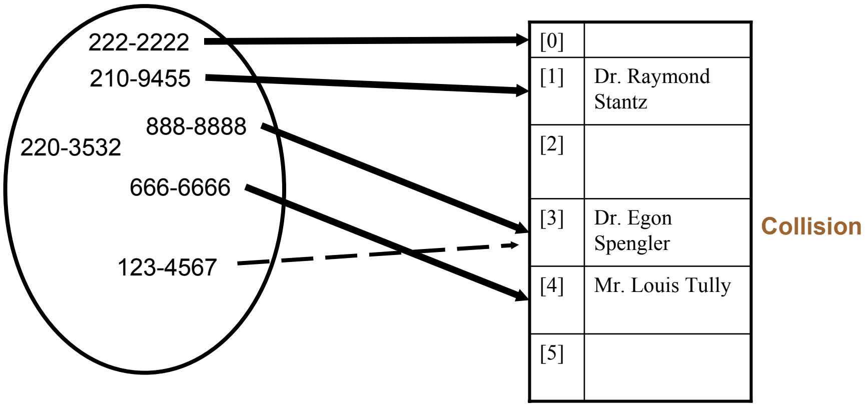
- When an attempt to insert a new element into hash table hashes to an element that has already been occupied (collision), find a new location *within the table* to insert the element.



[0]	<Next choice?>
[1]	Dr. Raymond Stantz
[2]	<Next choice?>
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully
[5]	<Next choice?>

1A) Linear Probing

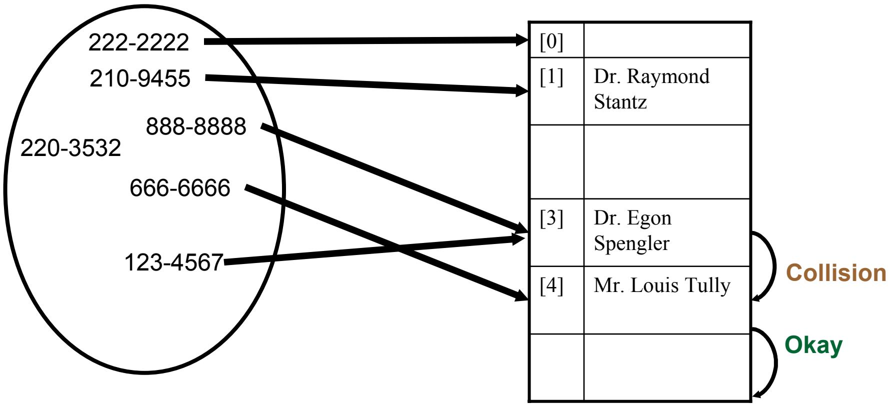
- When a collision occurs, sequentially search the table until an empty location is found.



[0]	
[1]	Dr. Raymond Stantz
[2]	
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully
[5]	

1A) Linear Probing

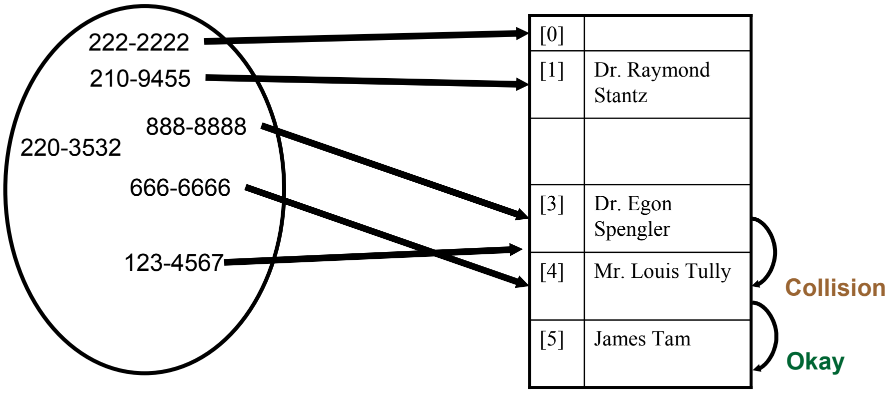
- When a collision occurs, sequentially search the table until an empty location is found.



[0]	
[1]	Dr. Raymond Stantz
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully

1A) Linear Probing

- When a collision occurs, sequentially search the table until an empty location is found.

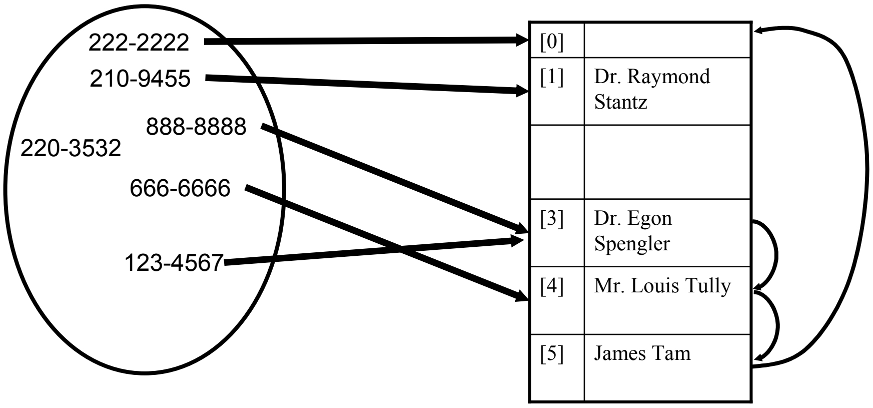


222-2222
210-9455
888-8888
220-3532
666-6666
123-4567

[0]	
[1]	Dr. Raymond Stantz
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully
[5]	James Tam

1A) Linear Probing: End Of The Table

- The table is treated as circular: When the end of the table has been probed, begin probing at the beginning.
- New position = (current position + 1) modulo <table size>



222-2222
210-9455
888-8888
220-3532
666-6666
123-4567

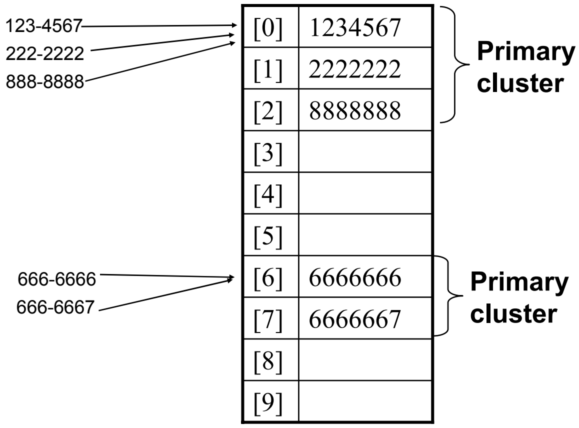
[0]	
[1]	Dr. Raymond Stantz
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully
[5]	James Tam

1A) Linear Probing

- Strength: As long as there is an unused location in the table, this approach will always be able to find it (eventually).

1A) Linear Probing

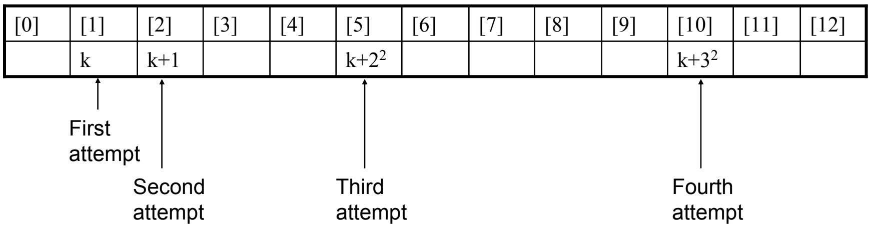
- Weakness: Table entries tend to cluster around parts of the table leaving some continuous sections that are occupied and others that are empty (uneven distribution)



[0]	1234567
[1]	2222222
[2]	8888888
[3]	
[4]	
[5]	
[6]	6666666
[7]	6666667
[8]	
[9]	

1B) Quadratic Probing

- Uses a collision resolution technique to avoid the problem of primary clustering found with linear probes.
  - General formula for the index = hashed key + n^2
  - First attempt: index = hashed key
  - Second attempt: index = (hashed key + $1^{2)}$ modulo <table size> = H.K. + 1
  - Third attempt: index = (hashed key + $2^{2)}$ modulo <table size> = H.K. + 4
  - Fourth attempt: index = (hashed key + $3^{2)}$ modulo <table size> = H.K. + 9
  - : : : : :

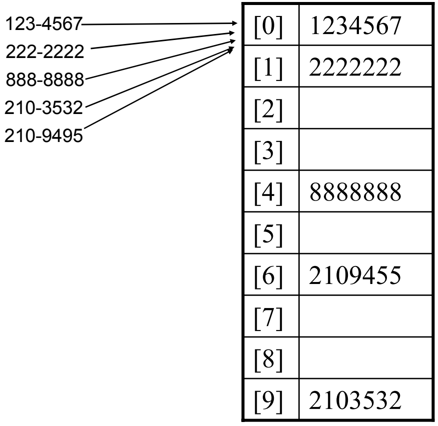


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
	k	k+1			$k+2^2$					$k+3^2$		

Where n is the number of attempts to find a new unoccupied entry in the hash table and it is assumed that the table indexes from 0 – (size – 1)

1B) Quadratic Probing (2)

- Secondary clustering may occur: The initial collision requires a recalculation of a new table entry. (Severity of the problem has not been fully explored – worse case is overflow – no entry found).



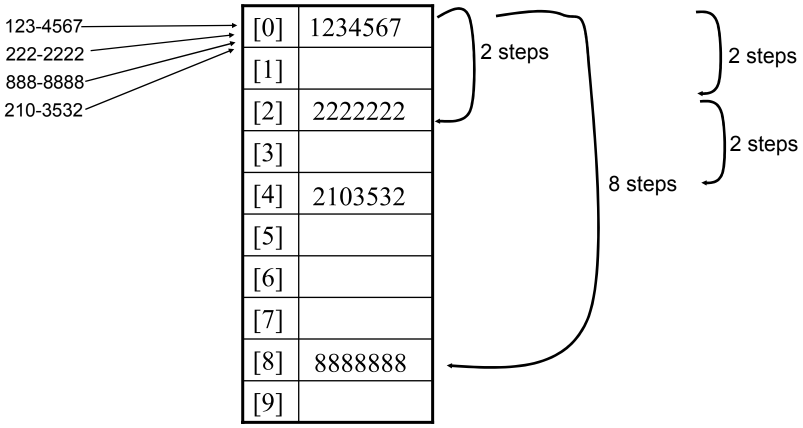
[0]	1234567
[1]	2222222
[2]	
[3]	
[4]	8888888
[5]	
[6]	2109455
[7]	
[8]	
[9]	2103532

1C) Double Hashing Using Key Dependent Increments

- The previous two approaches for collision resolution are key-independent: finding a new table entry is not effected by the value of the key.
- Double hashing: When the result of the hash function results in a collision, a second hash function is used.
- Key dependent: The value of the key is used to determine the increment for the second hash function.
- With double hashing:
  - Make sure that that increment for the second hash function never yields zero.
  - Do not use the same hash function the second time around <gee no kidding...>

1C) Double Hashing Using Key Dependent Increments

- Example: Last digit of key is the increment (key modulo 10)



[0]	1234567
[1]	
[2]	2222222
[3]	
[4]	2103532
[5]	
[6]	
[7]	
[8]	8888888
[9]	

1C) Double Hashing Using Key Dependent Increments

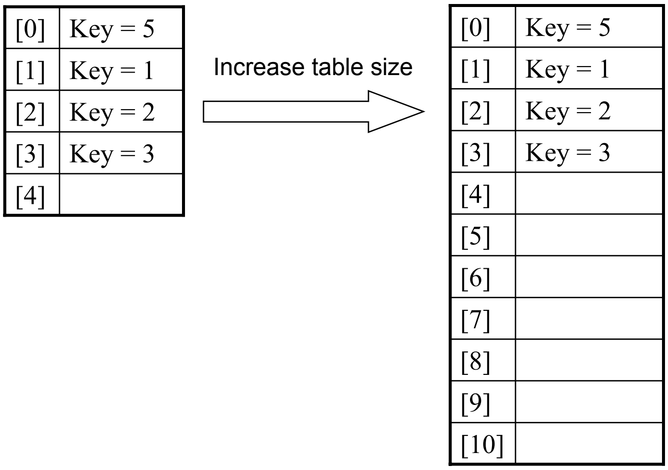
- Having a different increment may reduce the overflow problems found with quadratic probing and usually results in less clustering than with linear probing.

1D) Increasing The Size Of Table: Rehashing

- As the hash table gets fuller, the probability of a collision increases.
- Whenever the table reaches some predetermined percentage utilization the size of the table can be increased.
- The array should not be doubled (because the size of the table should be a prime number).
- Also existing elements cannot simply be copied to the new table.

1D) Increasing The Size Of The Table (2)

- Example: index = key modulo <table size>



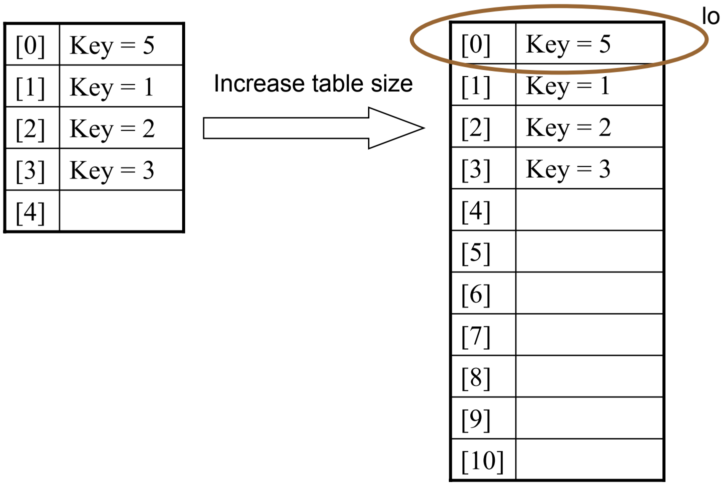
[0]	Key = 5
[1]	Key = 1
[2]	Key = 2
[3]	Key = 3
[4]	

Increase table size

[0]	Key = 5
[1]	Key = 1
[2]	Key = 2
[3]	Key = 3
[4]	
[5]	
[6]	
[7]	
[8]	
[9]	
[10]	

1D) Increasing The Size Of The Table (2)

- Example: index = key modulo <table size>



[0]	Key = 5
[1]	Key = 1
[2]	Key = 2
[3]	Key = 3
[4]	

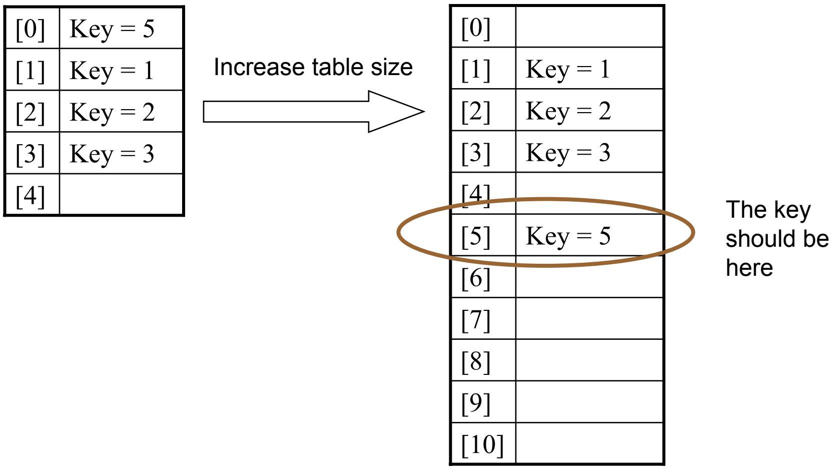
Increase table size

[0]	Key = 5
[1]	Key = 1
[2]	Key = 2
[3]	Key = 3
[4]	
[5]	
[6]	
[7]	
[8]	
[9]	
[10]	

This key is in the wrong location

1D) Increasing The Size Of The Table (2)

•Example: index = key modulo <table size>



[0]	Key = 5
[1]	Key = 1
[2]	Key = 2
[3]	Key = 3
[4]	

Increase table size

[0]	
[1]	Key = 1
[2]	Key = 2
[3]	Key = 3
[4]	
[5]	Key = 5
[6]	
[7]	
[8]	
[9]	
[10]	

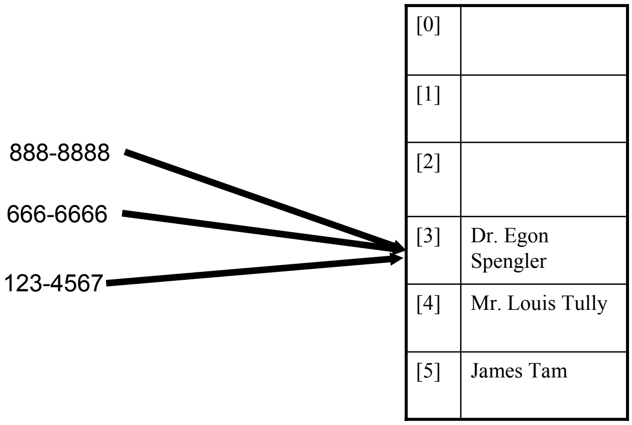
The key should be here

1D) Increasing The Size Of The Table (3)

- When the table size is increased take care that:
  1. The size of the table is still prime
  2. That all previous entries are rehashed to new locations based on the new table size – slow but necessary.

1) Closed Hashing: Deletions

- Deletion of entry [4] can be problematic



888-8888

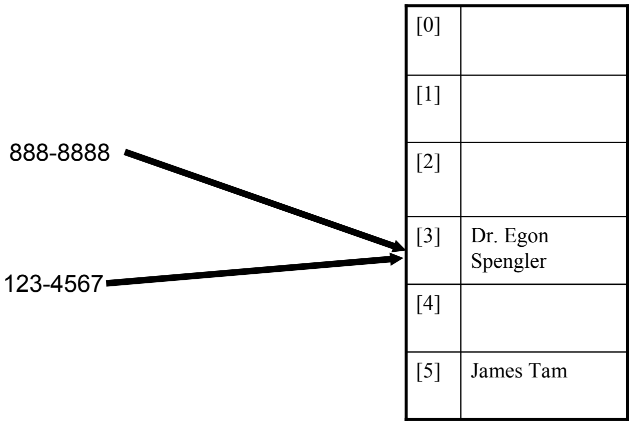
666-6666

123-4567

[0]	
[1]	
[2]	
[3]	Dr. Egon Spengler
[4]	Mr. Louis Tully
[5]	James Tam

1) Closed Hashing: Deletions (2)

- Searching the table for 123-4567 will hash to index [3]



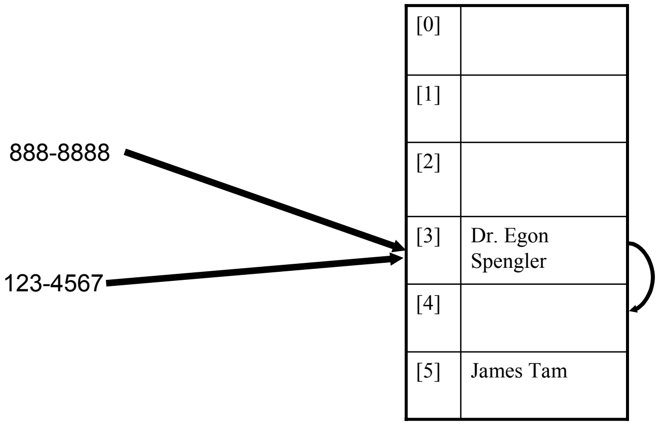
888-8888

123-4567

[0]	
[1]	
[2]	
[3]	Dr. Egon Spengler
[4]	
[5]	James Tam

1) Closed Hashing: Deletions (3)

- Linear probing means that the next entry will be searched



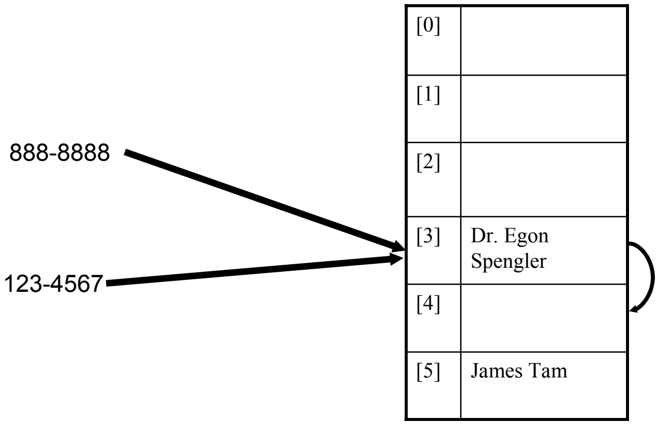
888-8888

123-4567

[0]	
[1]	
[2]	
[3]	Dr. Egon Spengler
[4]	
[5]	James Tam

1) Closed Hashing: Deletions (4)

- If entry [4] is treated as empty then the search will show as unsuccessful.



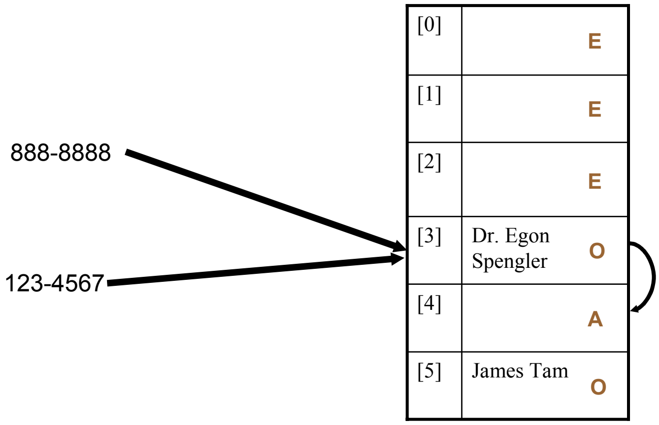
888-8888

123-4567

[0]	
[1]	
[2]	
[3]	Dr. Egon Spengler
[4]	
[5]	James Tam

1) Closed Hashing: Deletions (5)

- Consequence: There must be three kinds of locations in the hash table: Occupied, empty, available (entry was deleted from the table)



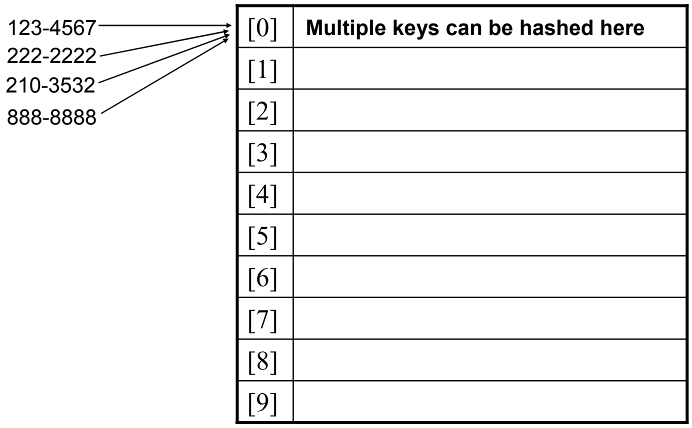
[0]		E
[1]		E
[2]		E
[3]	Dr. Egon Spengler	O
[4]		A
[5]	James Tam	O

1) Closed Hashing: Deletions (6)

- Searches:
  - The search algorithm will continue if an available table entry is encountered and will only stop if:
    - A successful match is found
    - An empty location is found
- Insertions:
  - The insertion will occur if the table entry is either empty or available (when a new entry is placed at an available location then the status changes from available to occupied).
- Because of the additional complexity arising from deletions from a hash table when closed hashing is used, table entries will not be deleted.

2. Restructuring the hash table

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

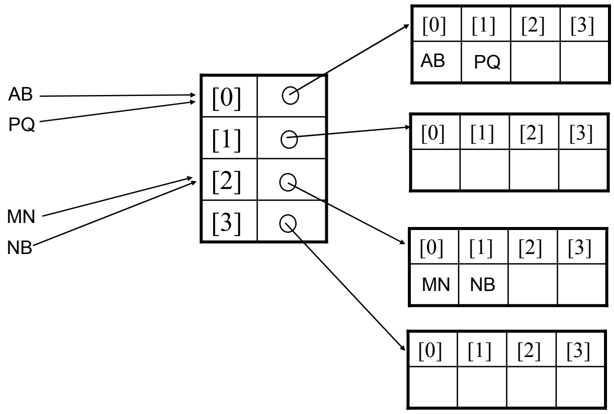


123-4567
222-2222
210-3532
888-8888

[0]	**Multiple keys can be hashed here**
[1]	
[2]	
[3]	
[4]	
[5]	
[6]	
[7]	
[8]	
[9]	

2A) Implementing Each Table Entry As A "Bucket"

- Each hash table entry is a 1D array



AB
PQ
MN
NB

[0]	○
[1]	○
[2]	○
[3]	○

[0]	[1]	[2]	[3]
AB	PQ		

[0]	[1]	[2]	[3]

[0]	[1]	[2]	[3]
MN	NB		

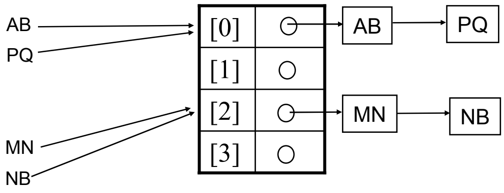
[0]	[1]	[2]	[3]

2A) Implementing Each Table Entry As A Bucket (2)

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

2B) Separate (External) Chaining

•Each table entry is a reference to a linked list.



Contrasting Closed Vs. Open Hashing

	Closed hashing (open addressing)	Open hashing: separate chaining)
Description	•Resolve collisions by finding another place in the hash table	•Resolve collisions by inserting additional elements at the same location in the hash table
Strengths	•May be faster in practice because the table doesn’t change in size	•May require less memory (smaller hash table) •Fewer compares
Weaknesses	•Requires a larger hash table (percentage utilization of the table should be lower than with open hashing)	•May be slower in practice because of the dynamic memory allocations •More complex: needs another data structure

Load Factor

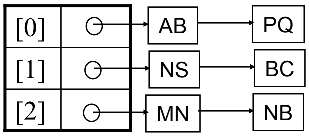
- Describes the utilization of the hash table.
- ά = (Current no. of occupied table elements) / (Table size)
  - Closed hashing: 0.0 <= ά <= 1.0

ά = .75

[0]	Key = 5
[1]	Key = 1
[2]	Key = 2
[3]	

- With open addressing, with external/separate chaining the load factor can be greater than one):

ά = 2



Guidelines For Load Factors

- Closed hashing:
  - Resolve collisions by finding another place in the table to insert to e.g., linear probing, quadratic probing.
  - Generally the load factor should be kept below 0.5 – 0.67 (depending upon the hashing algorithm)
- Open hashing:
  - Resolve collisions by allowing more than one element to be inserted at a particular location in the hash table e.g., making table elements buckets, making table elements linked lists.
  - Generally the load factor should be kept around 1.0.

The Time Efficiency Of A Has Function Is Dependent On The Load Factor

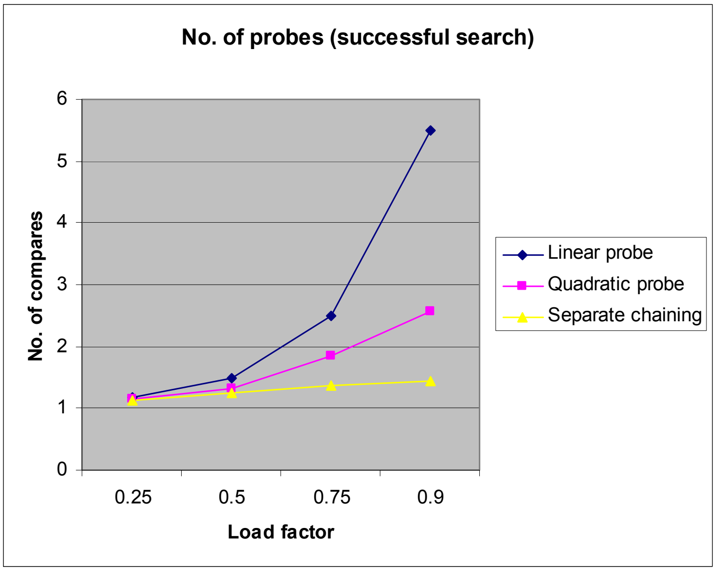
- As the load factor increases, the number of comparisons/probes increases:

	Number of probes/compares	
	Successful search	**Unsuccessful search**
Linear probing	$\frac{1}{2} * \left(1 + \frac{1}{(1- \acute{\alpha})}\right)$	$\frac{1}{2} * \left(1 + \frac{1}{(1- \acute{\alpha})^2}\right)$
Quadratic probing/double hashing	$\frac{-\log_e(1 - \acute{\alpha})}{\acute{\alpha}}$	$\frac{1}{(1- \acute{\alpha})}$
Separate/external chaining	$1 + \left(\frac{\acute{\alpha}}{2}\right)$	$\acute{\alpha}$

Number Of Comparisons/Probes: Successful Search

	Utilization of hash table (ά)						
	0.25	**0.5**	**0.75**	**0.9**	**1.0**	**2.0**	**4.0**
Linear probe	1.17	1.5	2.5	5.5	NA	NA	NA
Quadratic probe	1.15	1.33	1.85	2.56	NA	NA	NA
Separate chaining	1.13	1.25	1.37	1.45	1.5	2	3

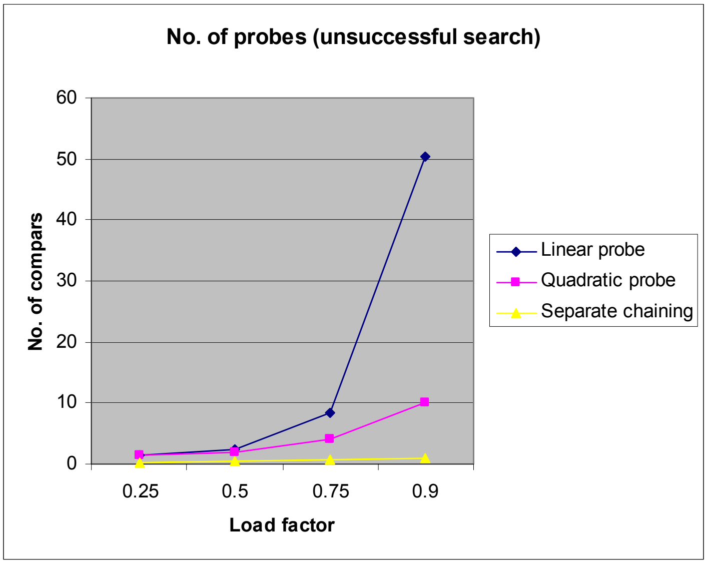
Number Of Comparisons/Probes: Successful Search (2)



Number Of Comparisons/Probes: Unsuccessful Search

	Utilization of hash table (ά)						
	0.25	**0.5**	**0.75**	**0.9**	**1.0**	**2.0**	**4.0**
Linear probe	1.39	2.5	8.5	50.5	NA	NA	NA
Quadratic probe	1.33	2	4	10	NA	NA	NA
Separate chaining	0.25	0.5	0.75	0.9	1.0	2.0	4.0

Number Of Comparisons/Probes: Unsuccessful Search (2)



You Should Now Know

- Basic hashing terminology.
- What are some common types of hash functions as well methods for determining the strengths/weaknesses of a particular function?
- What are common approaches for collision resolution:
    - Closed hashing/open addressing techniques: linear and quadratic probing, double hashing with key dependent increments, increasing the table size and rehashing keys.
    - Open hashing/closed addressing.
- What are the strengths/weaknesses of each approach to collision resolution?

Sources Of Lecture Material

- *"Data Abstraction and Problem Solving With Java: Walls and Mirrors"* updated edition by Frank M. Carrano and Janet J. Prichard
- *"Data Structures and Abstractions With Java"* by Frank M. Carrano and Walter Savitch
- "*Introduction to Algorithms*" by Thomas M. Cormen, Charles E. Leiserson and Ronald L. Rivest
- CPSC 331 course notes by Marina L. Gavrilova http://pages.cpsc.ucalgary.ca/~marina/331/