Data Structures: Static Hashing

Won Kim
(Lecture by Youngmin Oh)

Spring 2022

Static Hashing

Hashing

- Hash: "chop (meat or potatoes) into small pieces"

- Static Hashing (will learn today)
- Dynamic Hashing (will learn later)

Hash Brown Corned Beef Hash

Static Hashing

- Technique for storing and searching a key in an array (hash table) by mapping the key to an index into the array
  - Each entry in the table is called a bucket.
- O(1) average performance for search, insertion, and deletion of a key

Static Hashing: Example
(using the first letter of the key)

index	bucket
0	Korea
1	Japan
2	Qatar
3	Vietnam
4	China
5	Taiwan
6	Singapore
7	France

Static Hashing: Example (cont'd)

0	Korea	
1	Japan	
2	Qatar	
3	Vietnam	
4	China	
5	Taiwan	Thailand
6	Singapore	Spain
7	France	

collision

Static Hashing: Example (cont'd)

0	Korea		
1	Japan		
2	Qatar		
3	Vietnam		
4	China		
5	Taiwan	Thailand	overflow
6	Singapore	Spain	Sri Lanka
7	France		

Hash Table

- n indexes
- k buckets per index
  - Total n*k Keys Stored
- Total potential keys >> n
- Collision
  - > 1 key mapped to the same index
- Overflow
  - > k keys mapped to the same index

Ultimate Objectives of Hashing

- (1) minimize collision
- (2) minimize overflow
- (3) minimize hash table size
  - Keep n (indexes) and/or k (buckets) reasonably small

Two Key Elements in Hashing

- Selecting a hash function
  - Should have a uniform distribution of keys across all indexes
  - Should be easy to compute
- Collision resolution
  - What to do with the keys hashed to the same index
  - (Collision is unavoidable)

Hash Functions

- If the key is a string, convert it to a natural number.
  - e.g., “EACH” = (ASCII) 69 65 67 72
  - e.g., “EACH” = (alphabet position) 5 1 3 8
- Many possible functions.
  - e.g., Pick the first character of a string, and map it to a number. The number may be used as index into the hash table.

4 Commonly Used Hash Functions (1/4)

- Modulo (Division) Function
- Most commonly used
  - H(key) = key mod hash_table_size
    - e.g., 375 mod 101 = 72
  - All keys are guaranteed to fit in the hash table.
  - It is best to use a prime number for hash_table_size.

4 Commonly Used Hash Functions (2/4)

- Digit Folding ("hashing")
  - Add combinations of the elements of the key
  - e.g., from the key = 9010302051218, compute (9+0+1+0+3+0+2+0+5+1+2+1+8) = 32.
    - hash_table_size = 9 x 13 = 117
    - (* Why is the hash_table_size NOT 10 x 13?

      (answer: We are summing the highest number for each digit)
  - e.g., from the key = 9010302051218, compute (90+10+30+20+51+21+8) = 230
    - hash_table_size = 99 x 6 + 9 = 603

4 Commonly Used Hash Functions (3/4)

- Digital Selection
  - Select some elements of the key
    - e.g., from the key = 9010302051218, select only the even elements (000011).
  - Need to be careful to prevent heavy collision
    - e.g., (do not) select only the first two elements of the key = 9010302051218.

4 Commonly Used Hash Functions (4/4)

- Mid-Square Function
  - Square the key, convert the result to ASCII equivalent, and select k bits from the middle of the square.
    - (e.g.) If the key is 11, its square is 121. The ASCII equivalent for 121 is 111001, and some bits from the middle may be selected as index into a hash table.
    - The number of bits to select depends on the hash table size. If k bits are selected, the range of the values is 2^k.

Collision Resolution

- In general, no hash function can prevent collision.
- Colliding keys must be stored somehow.
- Two approaches
  - Closed addressing
    - Stays with the computed index
  - Open addressing
    - Invades any available bucket for any index
- Tradeoff
  - Hash table size (memory space)
  - Time to store and search the overflow keys

Closed Addressing (1/2)

- “Full” 2-D array for the hash table
  - For each index, enough buckets for all colliding keys.
  - Fast access, no overflow
  - Potential big waste of space
  - Not practical

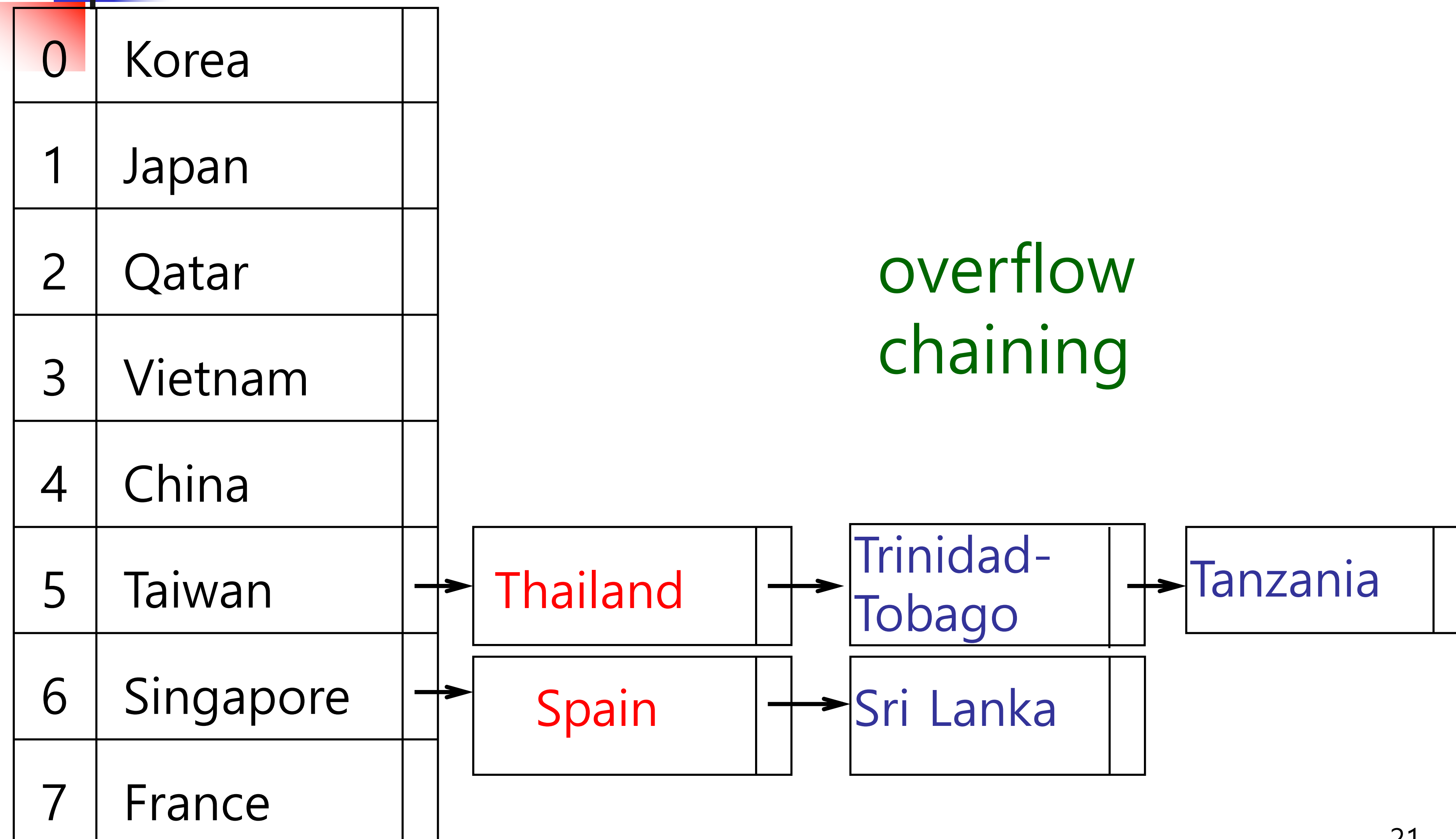
Hash Table as a "Full" 2-D Array

0	Korea		
1	Japan		
2	Qatar		
3	Vietnam		
4	China		
5	Taiwan	Thailand	
6	Singapore	Spain	Sri Lanka
7	France		

Closed Addressing (2/2)

- Overflow Chaining for each index
  - without collision buckets
  - with collision buckets for some colliding keys

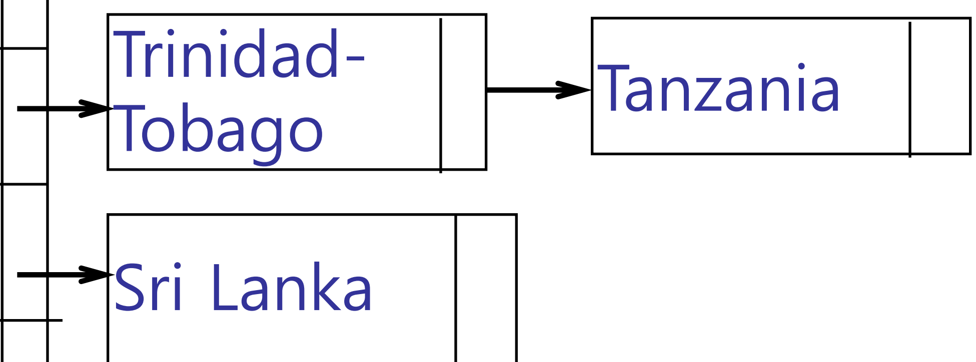
Overflow Chaining: without Collision Buckets



Overflow Chaining: with Collision Buckets

0	Korea		
1	Japan		
2	Qatar		
3	Vietnam		
4	China		
5	Taiwan	Thailand	
6	Singapore	Spain	
7	France		

overflow chaining



Search

- Use the same hash function used for insert.
- Search for the key in the indexed bucket.
- If there is no match in the bucket, continue along the overflow chain.

Search

- Use the same hash function used for insert.
- Search for the key in the indexed bucket.
- If no match, continue along the overflow chain.

Exercise: Hash Table & Overflow Chaining

hash table size: 7

no collision buckets

hash function: key mod 7

set of keys

7

9

44

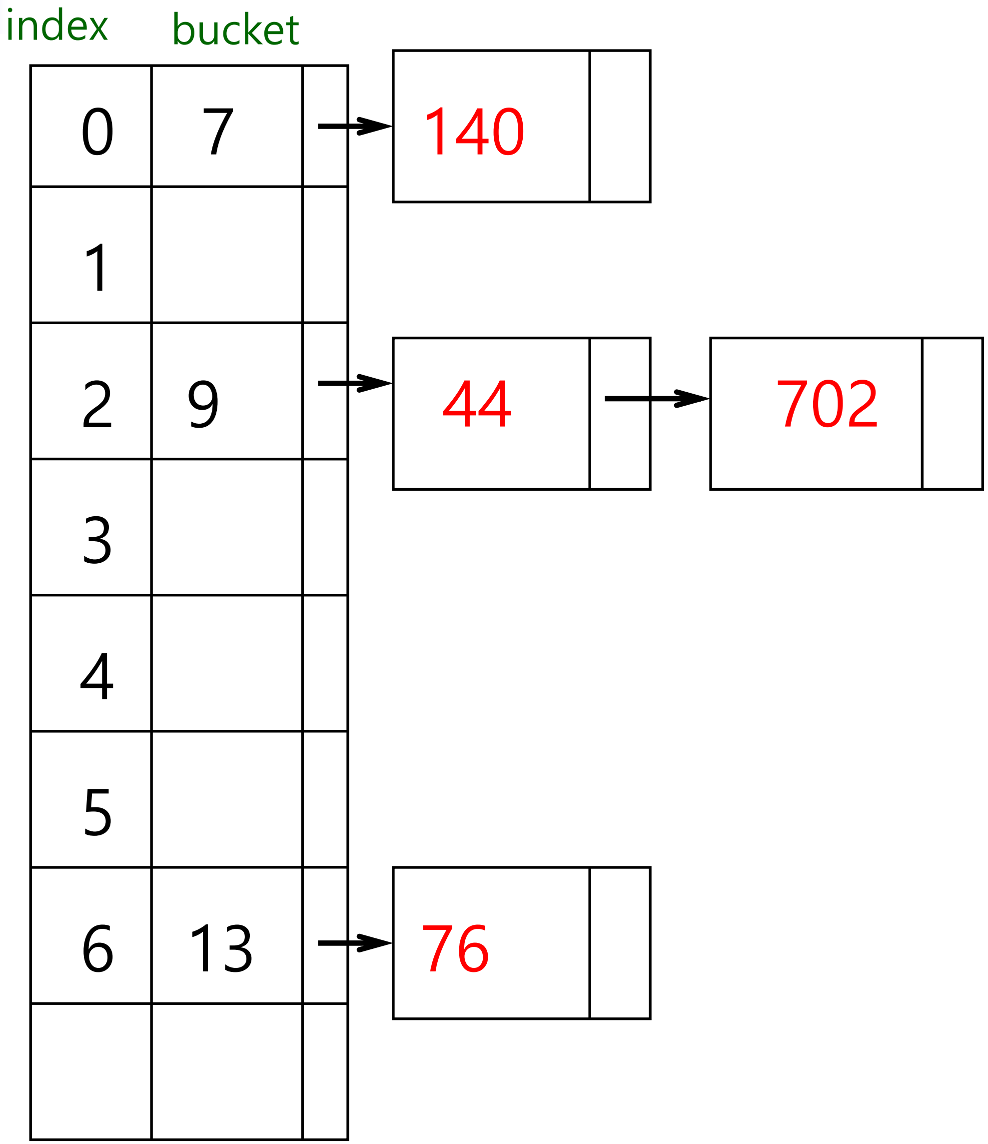
13

140

702

76

Solution



Open Addressing

- Linear Probing
- Quadratic Probing
- Double Hashing

- Difference is the way to “determine where to look, if the indexed bucket is occupied”.

Linear Probing

- If collision occurs, sequentially (one bucket at a time) search for an empty bucket and store the "homeless" key there.

Linear Probing: Example

0	Korea
1	
2	Qatar
3	
4	China
5	Taiwan
6	Singapore
7	France

insert
Spain

Spain

Linear Probing: Example (cont'd)

0	Korea
1	Spain
2	Qatar
3	
4	China
5	Taiwan
6	Singapore
7	France

insert
Spain

Linear Probing: Example (cont'd)

0	Korea
1	Spain
2	Qatar
3	
4	China
5	Taiwan
6	Singapore
7	France

insert
Sri Lanka

Sri Lanka

Linear Probing: Example (cont'd)

0	Korea
1	Spain
2	Qatar
3	Sri Lanka
4	China
5	Taiwan
6	Singapore
7	France

insert
Sri Lanka

Search

- Use the same hash function used for insert.
- If there is no match, use linear probing to find a match.

Exercise: Linear Probing

hash table size: 11

no collision buckets

hash function: key mod 11

set of keys

7

9

44

13

18

110

Solution

index	bucket
0	44
1	110
2	13
3	
4	
5	
6	
7	7
8	18
9	9
10	

Linear Probing: A Small Problem

- A “homeless” key may be deleted.
- Next search for another “homeless” key may prematurely stop there.

Linear Probing: Small Problem Example

0	Korea
1	Spain
2	Qatar
3	Sri Lanka
4	China
5	Taiwan
6	Singapore
7	France

delete
Spain

Linear Probing: Small Problem Example (cont'd)

0	Korea
1	
2	Qatar
3	Sri Lanka
4	China
5	Taiwan
6	Singapore
7	France

delete

Spain

Linear Probing: Small Problem Example (cont'd)

0	Korea
1	
2	Qatar
3	Sri Lanka
4	China
5	Taiwan
6	Singapore
7	France

search

Sri Lanka

Linear Probing: Small Problem Example (cont'd)

0	Korea
1	
2	Qatar
3	Sri Lanka
4	China
5	Taiwan
6	Singapore
7	France

search ends here

search
Sri Lanka

Linear Probing: Solution to the Small Problem

- 3 states for each index
  - Occupied, empty, deleted
- If “occupied”, continue probing.
- If “empty”, insert key.
- If “deleted”,
  - (If the operation is to insert a key), insert.
  - (If the operation is to search for a key), continue probing.

Linear Probing: Big Problem

- Formation of “Cluster”s
  - “Homeless” keys occupy somebody else’s buckets, causing somebody else to become “homeless” in a continuous chain.
  - Search time complexity can approach O(n), not O(1).

Quadratic Probing

- Linear probing modified
  - Look for an empty bucket k^2 ($k = 1, 2, 3, \ldots$) positions from the computed index.
  - Leaves room for legitimate keys.
- Partially solves the cluster formation problem of linear probing.

Quadratic Probing: Example

0	Korea
1	Qatar
2	
3	
4	China
5	Taiwan
6	Singapore
7	France

insert
Spain

Spain

Quadratic Probing: Example (cont'd)

0	Korea
1	Qatar
2	
3	
4	China
5	Taiwan
6	Singapore
7	France

insert Spain

Spain

$k^2=1$

Quadratic Probing: Example (cont'd)

0	Korea
1	Qatar
2	Spain
3	
4	China
5	Taiwan
6	Singapore
7	France

$k^2=4$

insert
Spain

Exercise: Quadratic Probing

hash table size: 11

no collision buckets

hash function: key mod 11

set of keys

7
9
44
13
18
110

Solution

index	bucket
0	44
1	110
2	13
3	
4	
5	
6	
7	7
8	18
9	9
10	

Double Hashing

- Use of two separate hash functions
  - one to find the index
  - another to find the "index interval" for inserting a "homeless" key.
    - The same index interval is successively used.
  - e.g., h1: key % 13,
    h2: 1+ key % 11
  - The second hash function should
    - not compute to zero
    - not be the same as the first hash function.
  - major difference from linear and quadratic probing
    - Bucket index depends on the key value.

Double Hashing: Example

0	Korea
1	Qatar
2	
3	China
4	
5	Taiwan
6	Singapore
7	France

insert
Spain

Spain

Double Hashing: Example (cont'd)

0	Korea
1	Qatar
2	
3	China
4	
5	Taiwan
6	Singapore
7	France

insert
Spain

Spain

h=3

Double Hashing: Example (cont'd)

0	Korea
1	Qatar
2	
3	China
4	Spain
5	Taiwan
6	Singapore
7	France

occupied

h=3

insert

Spain

Exercise: Double Hashing

hash table size: 11

no collision buckets

hash function 1: key mod 11

hash function 2: 1+ key mod 7

set of keys

7
9
44
13
18
110

Solution

index	bucket
0	44
1	18
2	13
3	
4	
5	
6	110
7	7
8	
9	9
10	

Note

- The collision scenarios shown in examples in this class are the result of using a very tiny array for a hash table just to convey the concepts to students.
- In practice, any reasonable software engineer will create a much larger array, and select hash functions carefully, and collisions do not happen too often.

Performance

- best case: O(1)
- avg case: O(1)
  - best possible search algorithm
- worst case: O(n)
  - (* as hash table gets nearly full *)

Open Addressing: Measures

- Loading density
  - α= # of buckets occupied / total # of buckets in the hash table
- Avg. # of key comparisons
  - $(2 - \alpha) / (2 - 2\alpha)$
- Examples
  - for α= 0.2 -> (2-0.2) / (2-2x0.2) = 1.125 (almost 1)
  - for α= 0.99 -> (2-0.99) / (2-2x0.99) = 50.5

Problems with Hashing

- Cannot be used for a range query
  - (e.g.) all keys < 250
  - (e.g.) all keys between 30 and 45
- Uses an array (with all the problems of an array)
  - wastes space if the hash table has lots of unused buckets
  - requires the hash table to be recreated if the hash table is (nearly) full

Bloom Filters

Bloom Filter

- An application of hashing
- Invented by Burton Bloom in 1970
- It is used for quickly determining using a small memory whether new data does not already exist in a large dataset.
- Note the 3 key phrases above.
- Reference
  - http://prakhar.me/articles/bloom-filters-for-dummies/

2 Elements of the Bloom Filter

- An array of N bits (initially all set to 0)
- A collection of *k* hash functions
  - Each hash function maps a key to an index in the bit array.

Problem Bloom Filter Solves

- Does item “x” exist in a dataset?
- How to answer this fast using a small memory?
- Dataset may be very big or expensive to access.
- Filter negative results before accessing data.
- Allow false positive (“exist”) errors, as they only cost an extra data access.
- Don’t allow false negative (“not exist”) errors, because they result in wrong answers.

Application Example

- Web crawler (for a search engine) building up a large set of URLs
  - 1. Get a new URL (webpage).
  - 2. Extract all URLs on that page.
  - 3. For each URL on that page, check if this URL has not been crawled already. (* This step uses the Bloom Filter. *)
  - 4. If it has not been crawled already, add the new URL to the URL dataset, and return to Step 1.

Algorithm (1/2)

- An array of N bits (to represent the large dataset of the URLs)
  - Choose N to be much greater than the number of expected URLs.
  - Initialize all N bits to 0 (zero).
- Inserting new input data (new URL)
  - Apply each of the *k* hash functions to the input data. The result m (m < N) is an index in the bit array. There are *k* such results.
  - Set the m^{th} bit of the N-bit array to 1, for each of the *k* hashing results. The result is the N-bit array modified with *k* bits set to 1.

Algorithm (2/2)

- Check if the key is already in the dataset
  - Apply each of the *k* hash functions to the input data. The result m (m < N) is an index in the bit array. There are *k* such results.
  - Look up the *k* positions in the current bit array.
  - (Before setting bits in the *k* positions to 1) If at least one of the *k* positions in the current N-bit array is zero, then the key is DEFINITELY NOT in the dataset
  - Otherwise, the input data MAY BE in the dataset.
- Bloom Filter does not allow deletion.

Example: Bit Array and 2 Hash Functions

- The bit array has 11 bits, all of which are initially set to zero - 00000000000.
- h(x)   (* x is the new key to be inserted *)
  - Convert x to binary equivalent *bx*.
  - Take the odd numbered bits in *bx* and generate a new decimal number *odx* corresponding to these bits.
  - Apply the function *odx modulo 11*.
- g(x)
  - the same as h(x), except that even numbered bits are taken from *bx*

Illustration: First Input Key

- Assume the first input key is 25.
- Apply the key to the first hash function, h(x).
  - x = 25. Then *bx* = 11001.
  - Taking odd bits from *bx* results in a binary number 101, which is binary for 5.
  - 5 % 11 = 5
- Apply the key to the second hash function, g(x).
  - Taking even bits results in a binary number 10, which is binary for 2.
  - 2 % 11 = 2

First 3 Input Keys; and 2 Tests

x	bx	h(x)	g(x)	Bit array
25	0000011001	5	2	00100100000
159	0010011111	0	7	10100101000
585	1001001001	7	9	10100101010
118		5	3	
162		2	0	

Test Results

- 118
  - current array: 10100101010
  - h(x)=5: array[5] = 1
  - g(x)=3: array[3] = 0
  - conclusion: 118 is NOT in the dataset (correct)
- 162
  - current array: 10100101010
  - h(x)=2: array[2] = 1
  - g(x)=0: array[0] = 1
  - conclusion: 162 IS in the dataset (false positive)

Exercise

- query data: 317 → 00100111101
  - current bit array: 10100101010

Solution

- test key: 317 → 00100111101
  - current array: 10100101010
  - h(x): *bx* = 010111 = 23; 23 % 11 = 1; array[1] = 0
  - g(x): *bx* = 00110 = 6 6 % 11 = 6; array[6] = 0
  - conclusion: 317 is NOT in the dataset (correct)

How to Reduce the False Positives

- Reducing the false positives means reducing the probability that a non-existing key will hash to 1 bits in the N-bit array
- Two Ways
  - One is to make the bit array larger
  - Another is to use additional hash functions
- In practice, false positive rate may be reduced to 15 to 1% of the tests.

End of Lecture