

CS 7270: Advanced Database Systems Fall 2025

Lecture 03

Hash tables

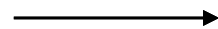
Yuvaraj Chesetti
(chesetti.y@northeastern.edu)

Hash tables

Data structure for the dictionary problem

Set of key-value pairs

Key	Value
X	2020
D	102
A	8321
Q	234
E	9819



Query(D): 102
Query(Y): \emptyset
Query(X): 2020

Hash tables

Data structure for the dictionary problem

Set of key-value pairs

Key	Value
X	2020
D	102
A	8321
Q	234
E	9819

Query(D): 102

Query(Y): \emptyset

Query(X): 2020

Insert(W, 2034)

Update(D, 212)

Delete(X)

Hash tables

Data structure for the dictionary problem

Set of key-value pairs

Key	Value
X	2020
D	102
A	8321
Q	234
E	9819

Query(D): 102
Query(Y): \emptyset
Query(X): 2020

Insert(W, 2034)
Update(D, 212)
Delete(X)

Key	Value
X	2020
W	2034
D	212
A	8321
Q	234
E	9819

Hash tables

Data structure for the dictionary problem

Set of key-value pairs

Key	Value
X	2020
D	102
A	8321
Q	234
E	9819

Query(D): 102
Query(Y): \emptyset
Query(X): 2020

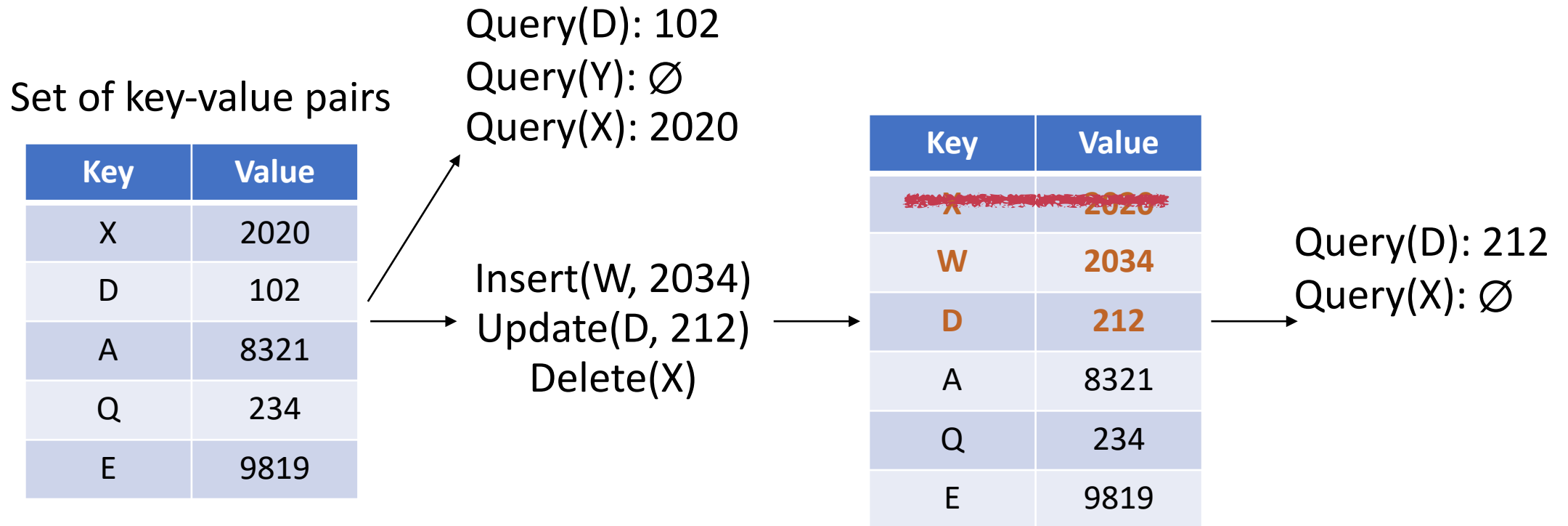
Insert(W, 2034)
Update(D, 212)
Delete(X)

Key	Value
X	2020
W	2034
D	212
A	8321
Q	234
E	9819

Query(D): 212
Query(X): \emptyset

Hash tables

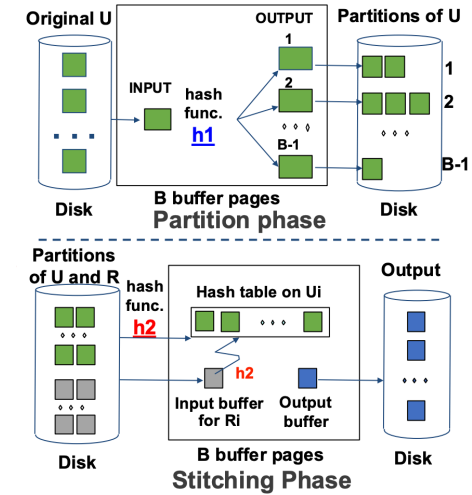
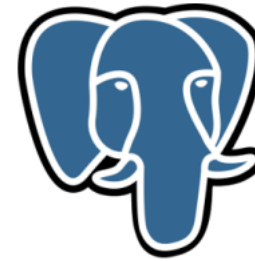
Data structure for the dictionary problem



Operations: $O(1)$ (expected)

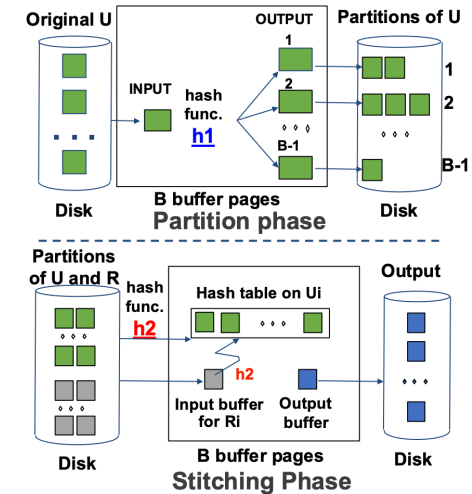
Core part of databases (and Comp. Science)

- Indexing/Key-value stores
- Caches
- Hash-Joins
- Data Partitioning/Load balancing

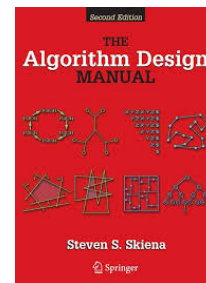


Core part of databases (and Comp. Science)

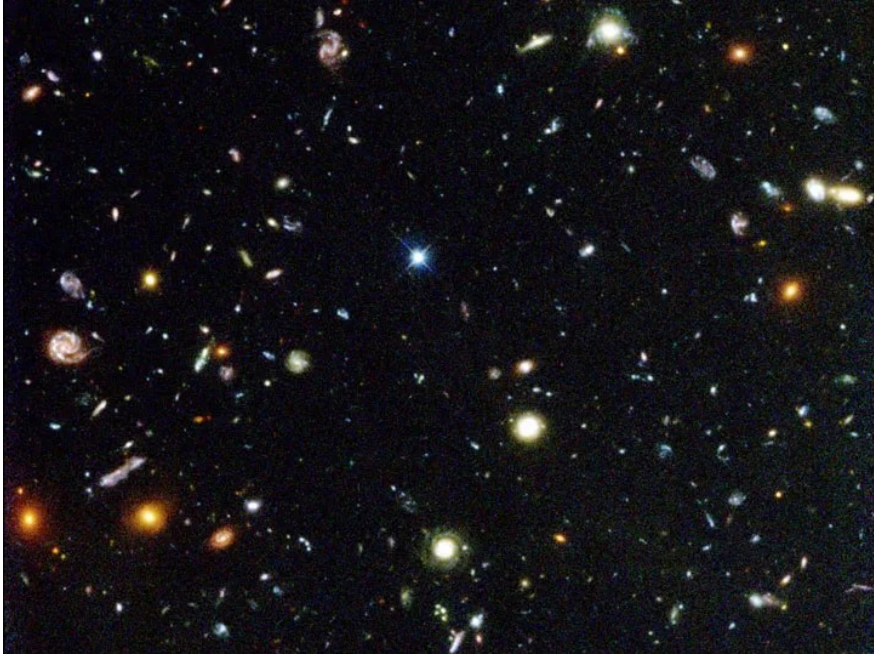
- Indexing/Key-value stores
- Caches
- Hash-Joins
- Data Partitioning
- Load-Balancing



Hashing has a variety of clever applications beyond just speeding up search. I once heard Udi Manber—then Chief Scientist at Yahoo—talk about the algorithms employed at his company. The three most important algorithms at Yahoo, he said, were hashing, hashing, and hashing.



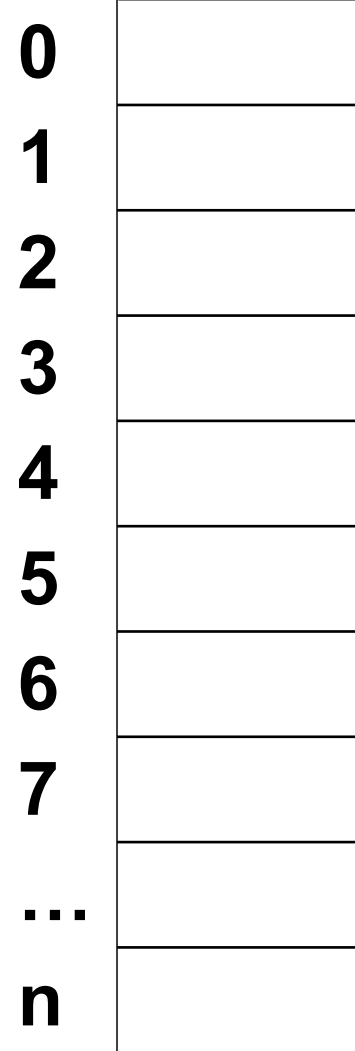
Hash-table idea



key space (e.g., integers, strings)

Hash function

$h(k)$



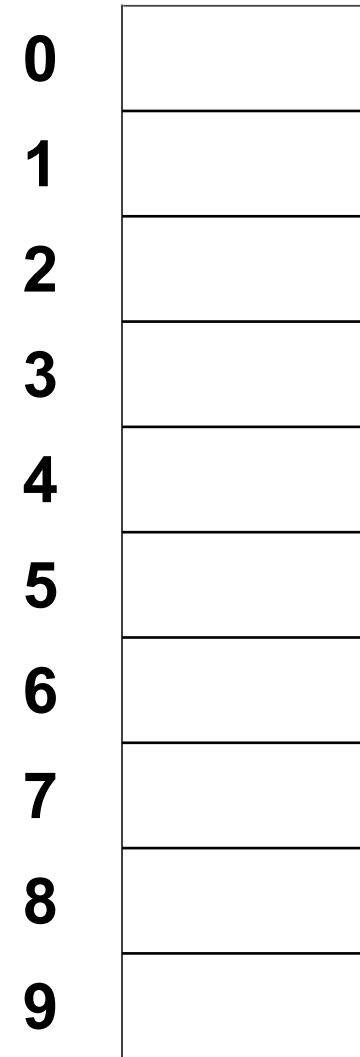
Hash table slots

Example

Key-space: Integers $[0, 2^{64}]$

Set:
107
122318
41201
942
1928

Hash function
 $h(k) = k \bmod 10$



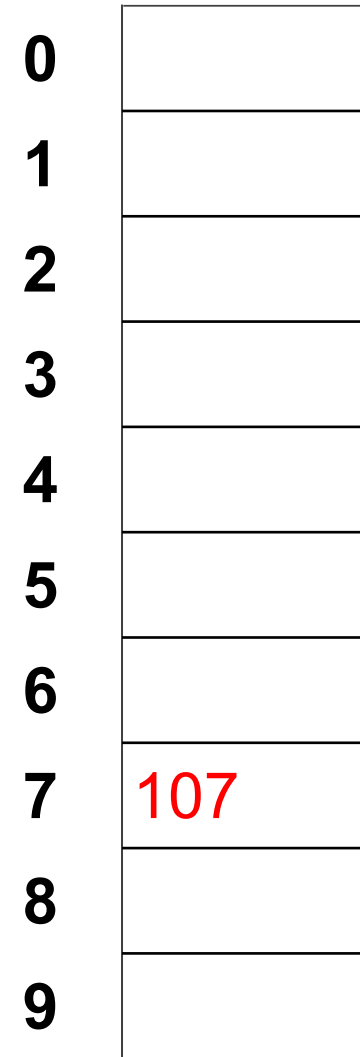
10 Slots

Example

Key-space: Integers $[0, 2^{64}]$

Set:
107
122318
41201
942
1928

Hash function
 $h(k) = k \bmod 10$



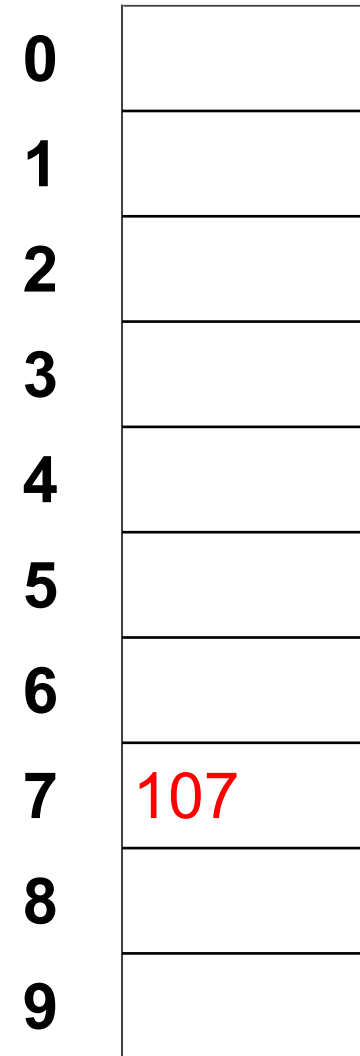
10 Slots

Example

Key-space: Integers $[0, 2^{64}]$

Set:
107
122318
41201
942
1928

Hash function
 $h(k) = k \bmod 10$



10 Slots

Example

Key-space: Integers $[0, 2^{64}]$

Set:
107
122318
41201
942
1928

Hash function
 $h(k) = k \bmod 10$



0	
1	
2	
3	
4	
5	
6	
7	107
8	123318
9	

10 Slots

Example

Key-space: Integers $[0, 2^{64}]$

Set:
107
122318
41201
942
1928

Hash function
 $h(k) = k \bmod 10$



0	
1	
2	
3	
4	
5	
6	
7	107
8	123318
9	

10 Slots

Example

Key-space: Integers $[0, 2^{64}]$

Set:
107
122318
41201
942
1928

Hash function
 $h(k) = k \bmod 10$



0	
1	41201
2	942
3	
4	
5	
6	
7	107
8	123318
9	

10 Slots

Collisions in hash functions

Key-space: Integers $[0, 2^{64}]$

Set:
107
122318
41201
942
1928

Hash function
 $h(k) = k \bmod 10$



Collision!! Slot is already occupied!

0	
1	41201
2	942
3	
4	
5	
6	
7	107
8	122318
9	

10 Slots

Hash table design

- **Hash function**
 - Simple/fast to compute
 - Avoids collisions
 - Evenly distributes keys among cells
- **Collision resolution**
 - Chaining
 - Open Addressing

Hash functions

- Simple/fast to compute
- **Theoretical:** Universal hashing
 - Choose from a family of hash functions
 - Probability that $h(x) = h(y)$ when $x \neq y$ is $\leq 1/m$
 - m : number of slots in the hash table
 - $O(1)$ to compute
- **Practice:** MurmurHash, XXHash, FarmHash
 - Don't guarantee universal hashing, but work well enough

Collision resolution schemes

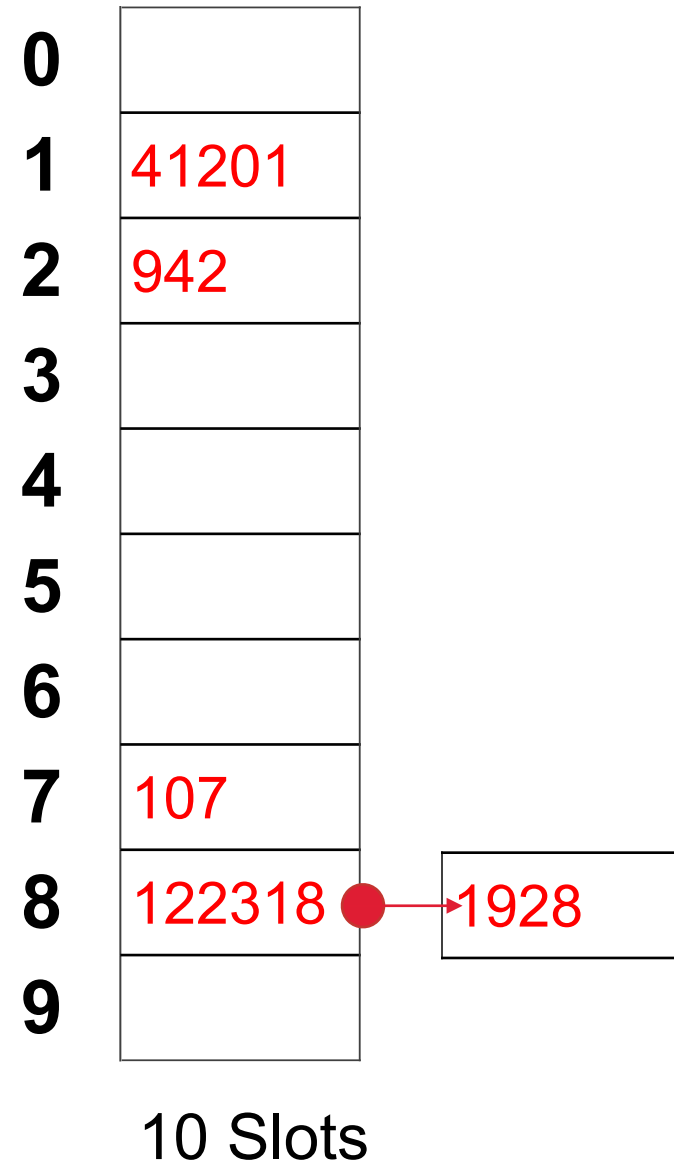
- Chaining
- Open-Addressing (Linear Probing)
- Cuckoo Hashing

Chaining

All keys that map to the same hash value are kept in a list (or “bucket”).

Insert:
1928

Hash function
 $h(k) = k \bmod 10$



Chaining

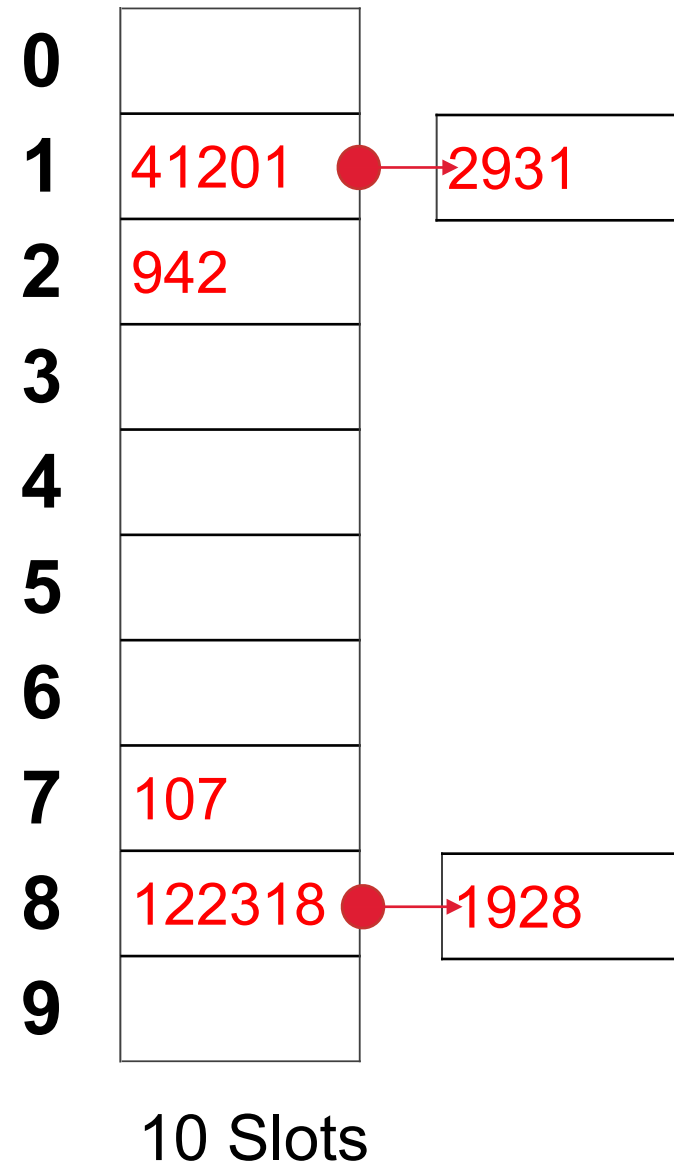
All keys that map to the same hash value are kept in a list (or “bucket”).

Insert:

1928

2391

Hash function
 $h(k) = k \bmod 10$



Chaining

All keys that map to the same hash value are kept in a list (or “bucket”).

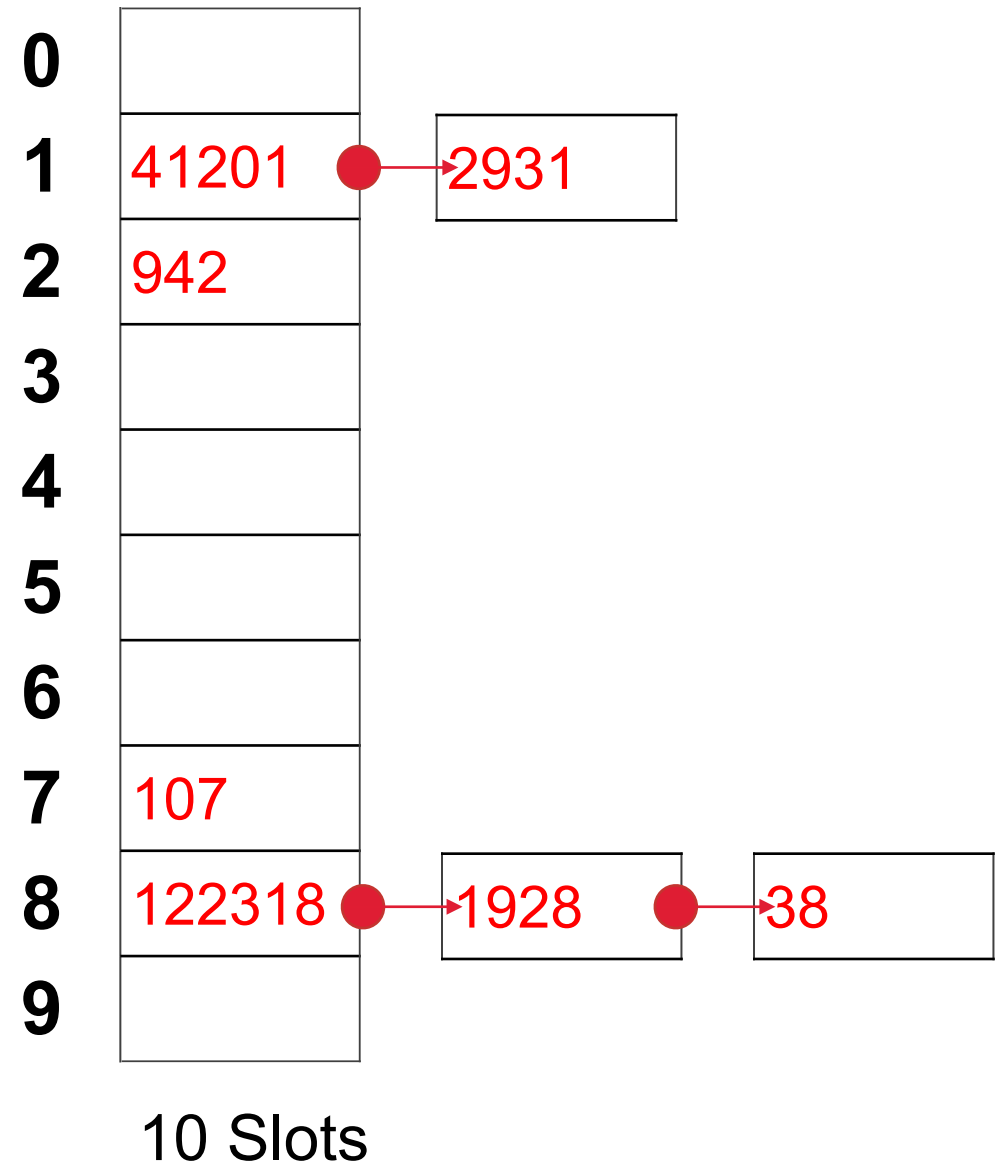
Insert:

1928

2391

38

Hash function
 $h(k) = k \bmod 10$



Performance of chaining hash tables

- Pointers
 - Space overhead
 - Random access patterns (low cache utilization)
- Pre-allocate buckets
 - Low space efficiency
 - Fast queries

Open addressing

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Insert:
1928

Hash function
 $h(k) = k \bmod 10$



0	
1	41201
2	942
3	
4	
5	
6	
7	107
8	122318
9	

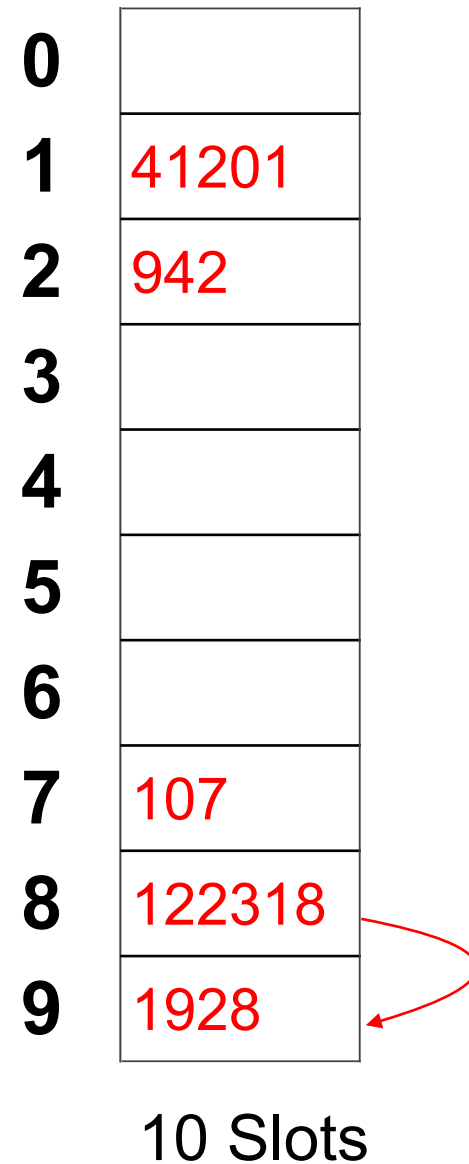
10 Slots

Open addressing

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Insert:
1928

Hash function
 $h(k) = k \bmod 10$



Open addressing

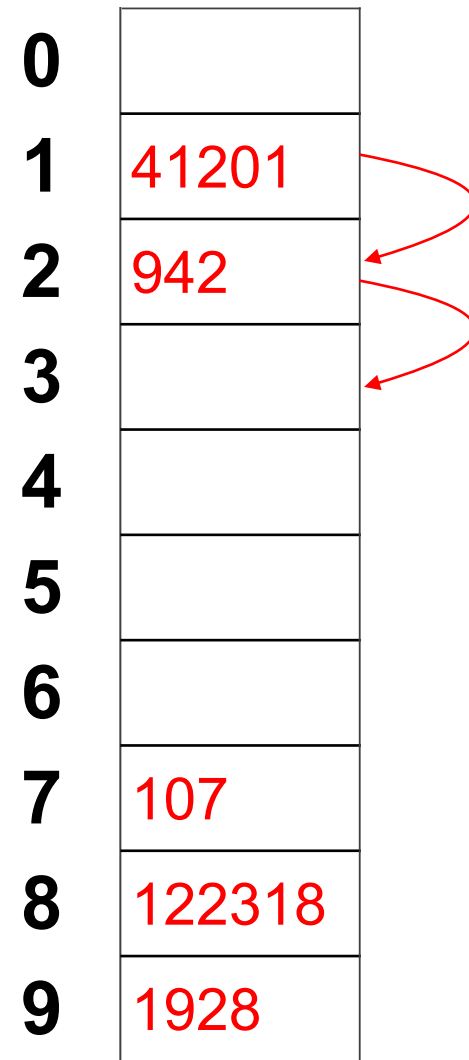
Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Insert:

1928

2391

Hash function
 $h(k) = k \bmod 10$



10 Slots

Open addressing

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Insert:

1928

2391

Hash function
 $h(k) = k \bmod 10$



0	
1	41201
2	942
3	2931
4	
5	
6	
7	107
8	122318
9	1928

10 Slots

Open addressing

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Insert:

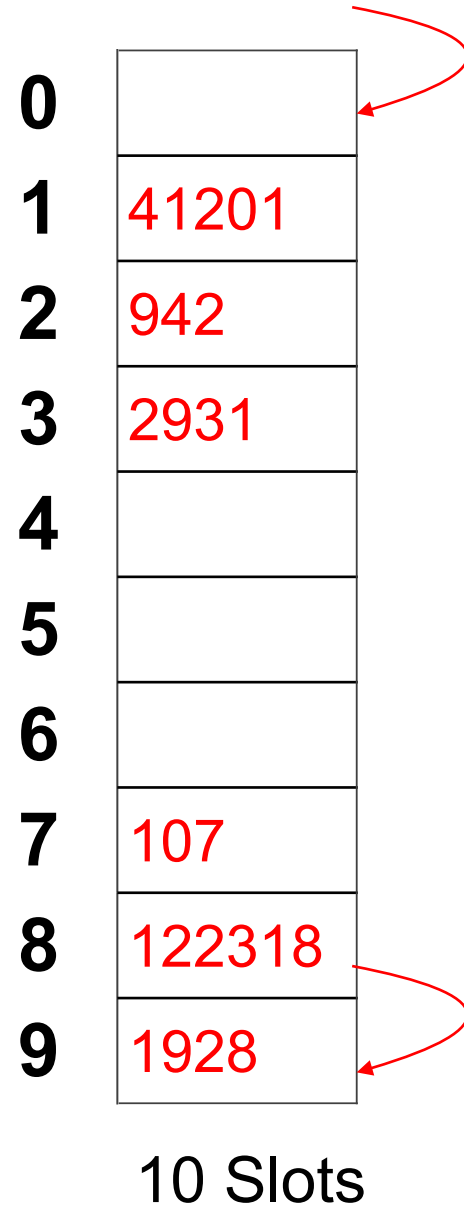
1928

2391

38

Hash function

$$h(k) = k \bmod 10$$



Open addressing

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Insert:

1928

2391

38

Hash function
 $h(k) = k \bmod 10$



0	38
1	41201
2	942
3	2931
4	
5	
6	
7	107
8	122318
9	1928

10 Slots

Open addressing

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Lookup: Check $h(k)$, then $h(k)+1$, $h(k)+2$ until item found or empty slot

Lookup:
38

Hash function
 $h(k) = k \bmod 10$



0	38
1	41201
2	942
3	2931
4	
5	
6	
7	107
8	122318
9	1928

10 Slots

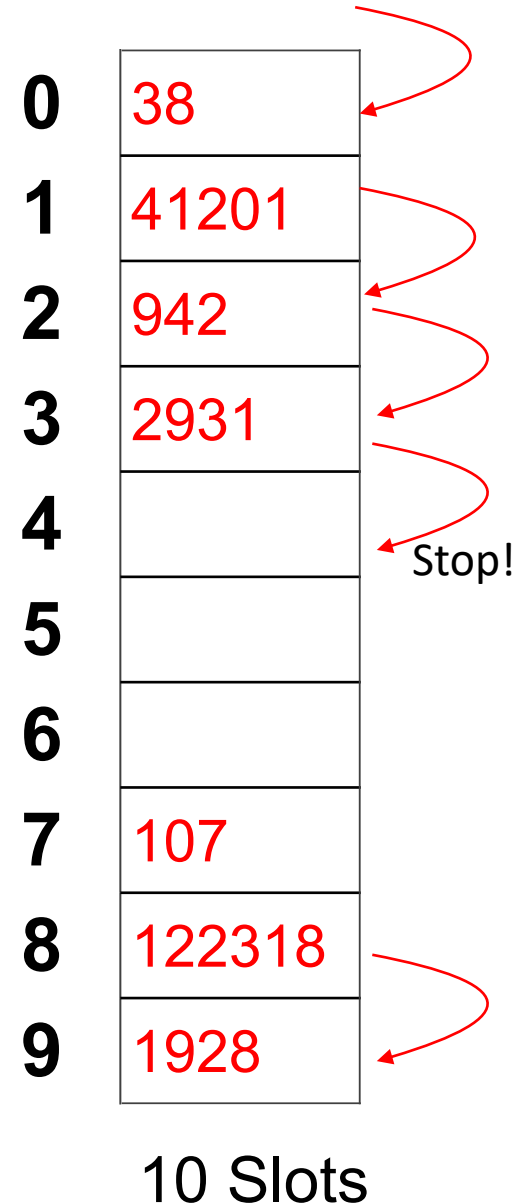
Open addressing

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Lookup: Check $h(k)$, then $h(k)+1$, $h(k)+2$ until item found or empty slot

Lookup:
1468

Hash function
 $h(k) = k \bmod 10$



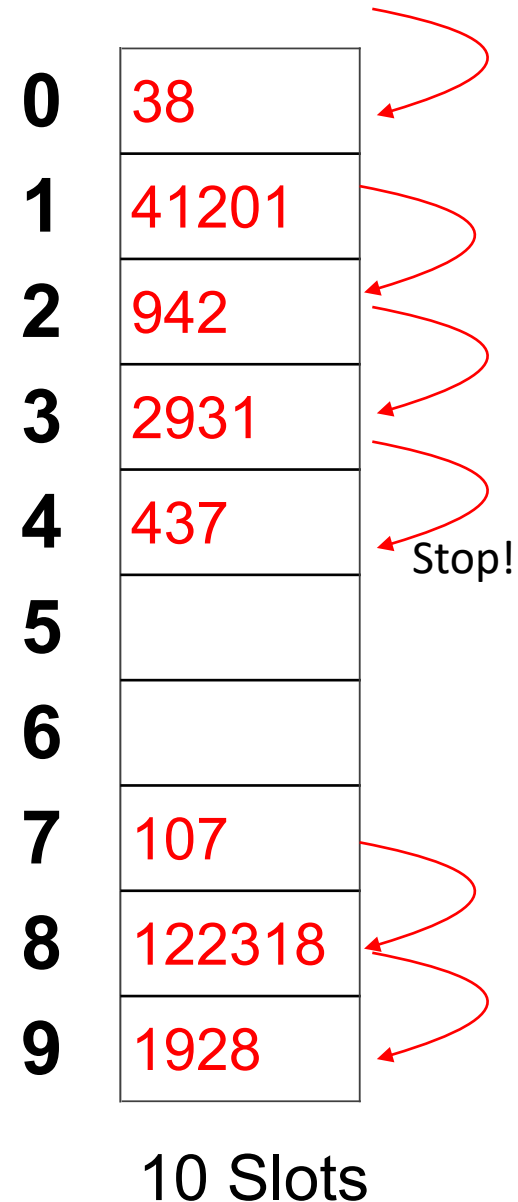
Primary clustering

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Lookup: Check $h(k)$, then $h(k)+1$, $h(k)+2$ until item found or empty slot

Insert:
437

Hash function
 $h(k) = k \bmod 10$



Primary clustering

Linear Probing: after checking spot $h(k)$, try spot $h(k)+1$, if that is full, try $h(k)+2$, then $h(k)+3$, etc.

Lookup: Check $h(k)$, then $h(k)+1$, $h(k)+2$ until item found or empty slot

Insert:
437

Hash function
 $h(k) = k \bmod 10$



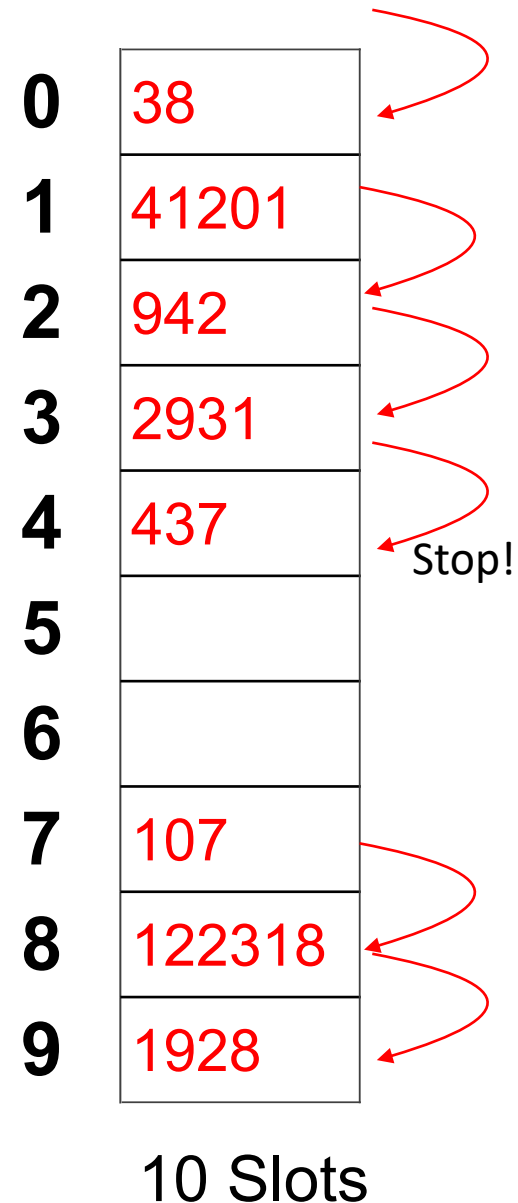
Primary clustering: more inserts, longer chains

m : number of items, n : number of slots

Load factor: m/n

Chain length: $\frac{\log N}{(1 - \alpha)^2}$, where α is the load factor

Choose between space efficiency or fast queries



Primary clustering

m : number of items, n : number of slots

Load factor: $\alpha = m/n$ **Chain length:** $\frac{\log n}{(1 - \alpha)^2}$

Choose: Space efficiency or Fast queries

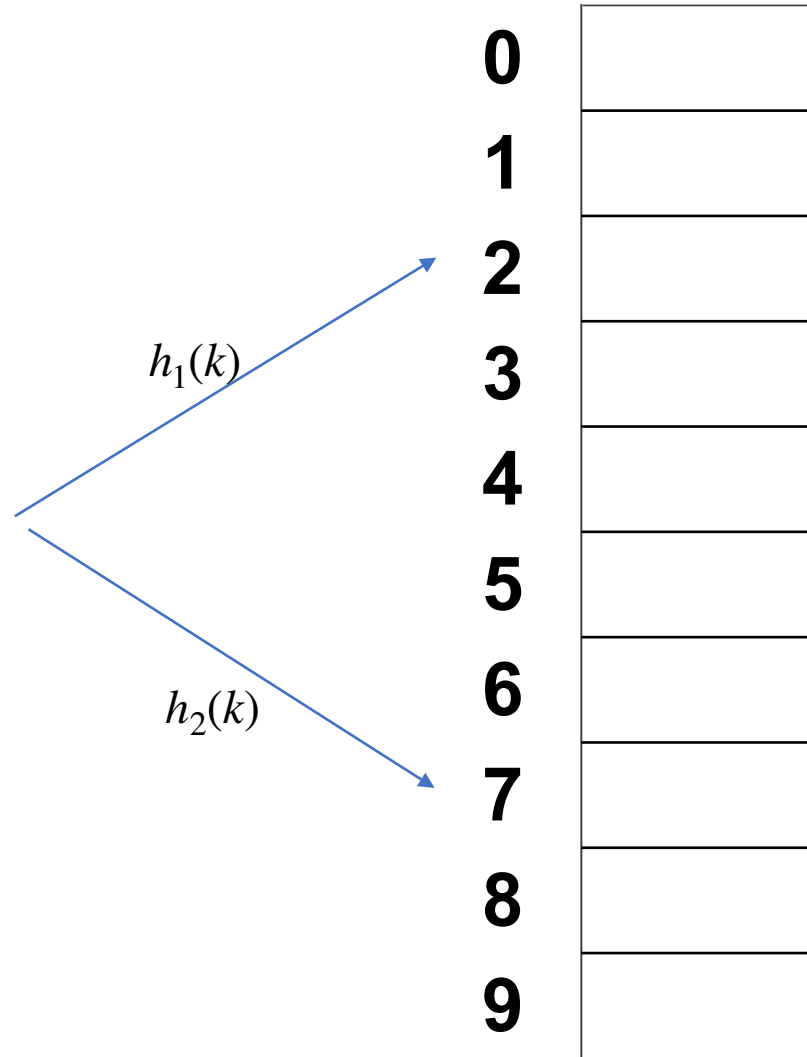
Or alternate probing:

Quadratic Probing (after checking spot $h(k)$, try spot $h(k)+1^2$, if that is full, try $h(k)+2^2$, then $h(k)+3^2$, etc.

Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

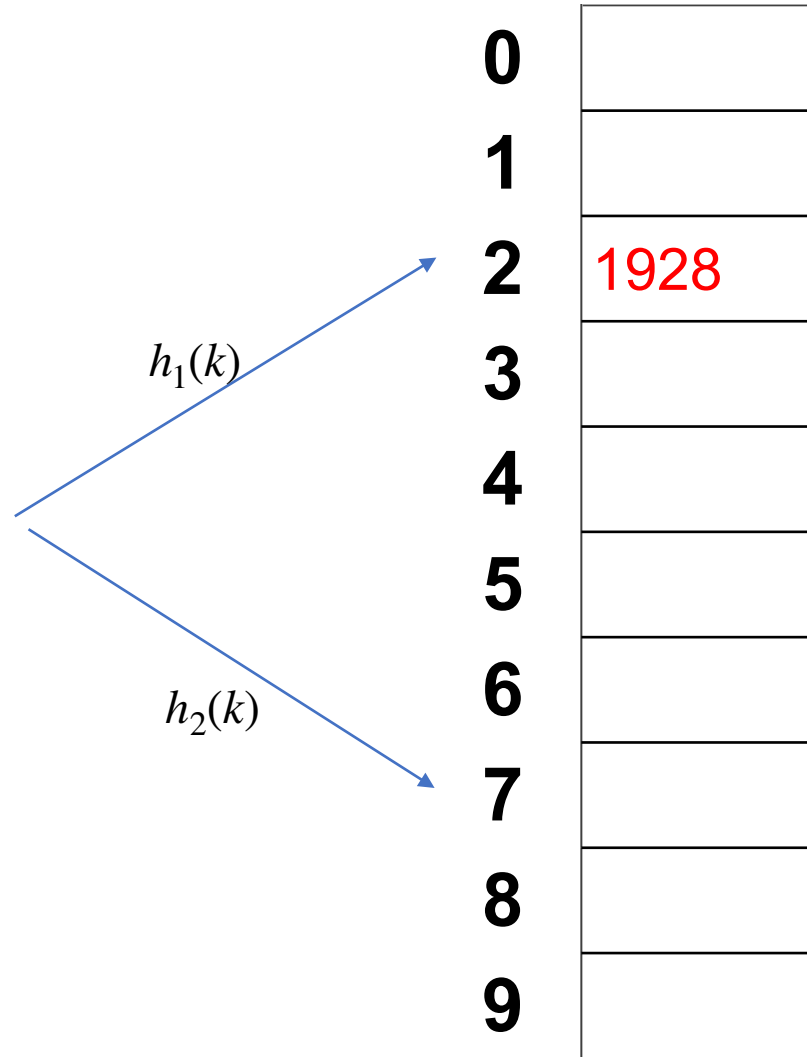
Insert: 1928



Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

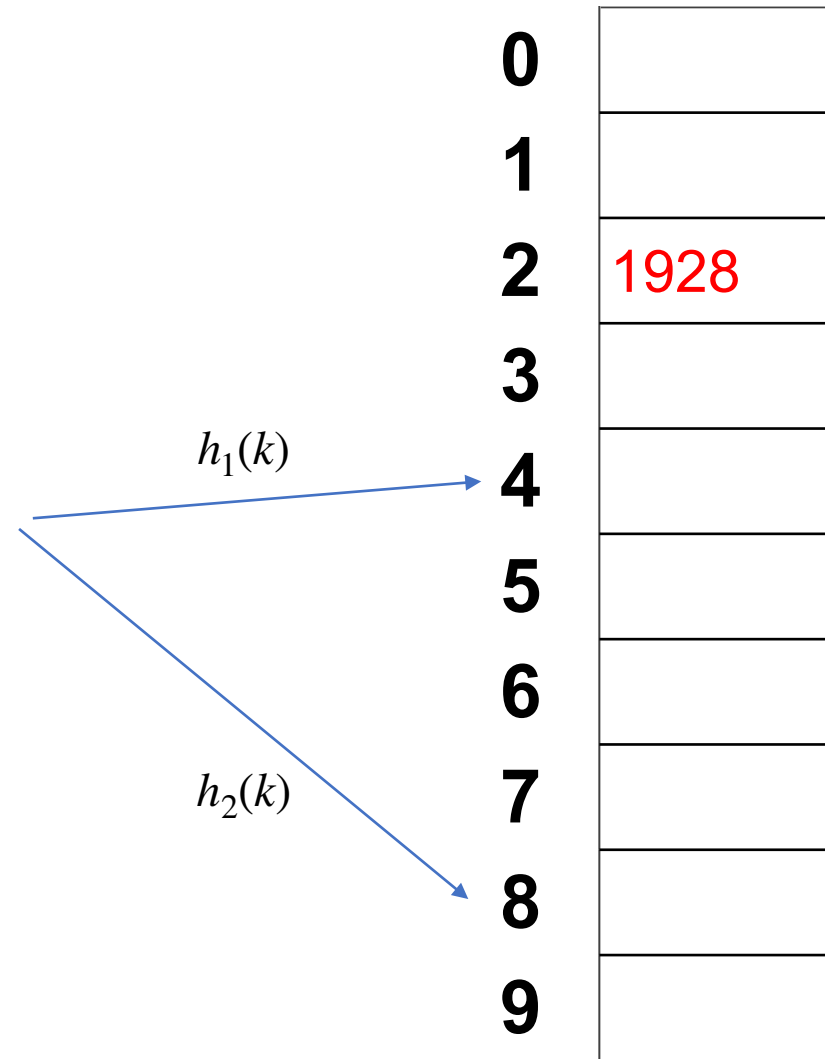
Insert: 1928



Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

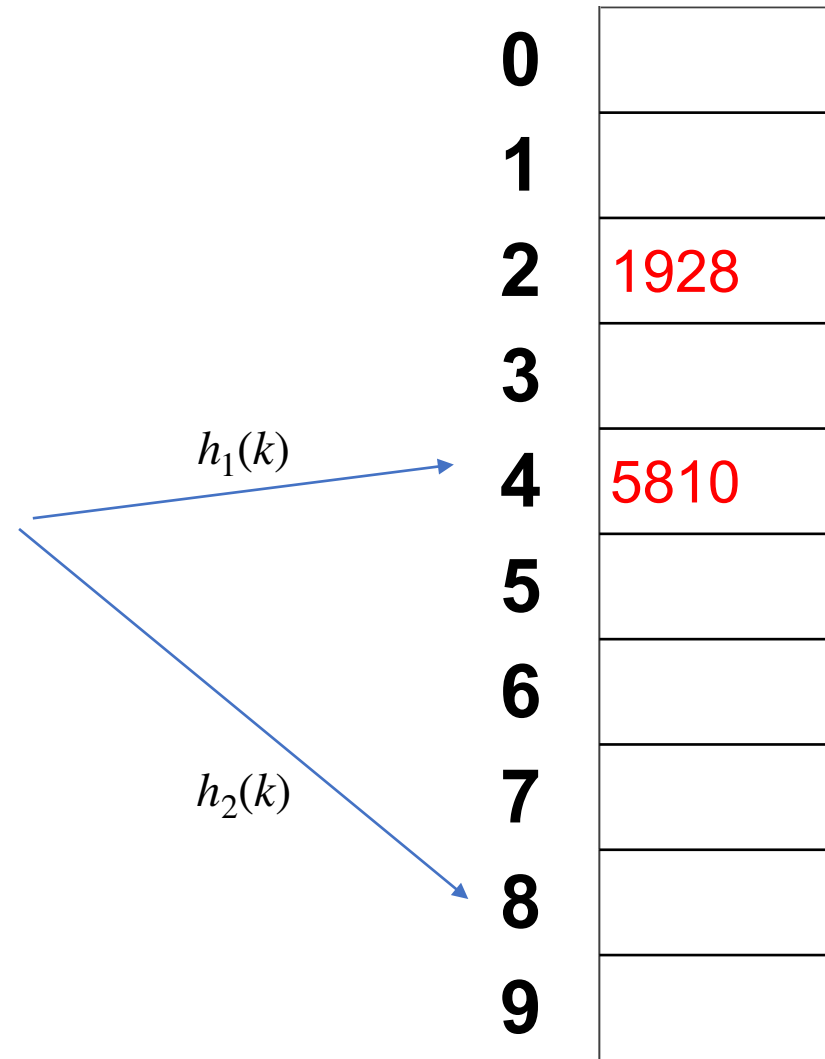
Insert: 5810



Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

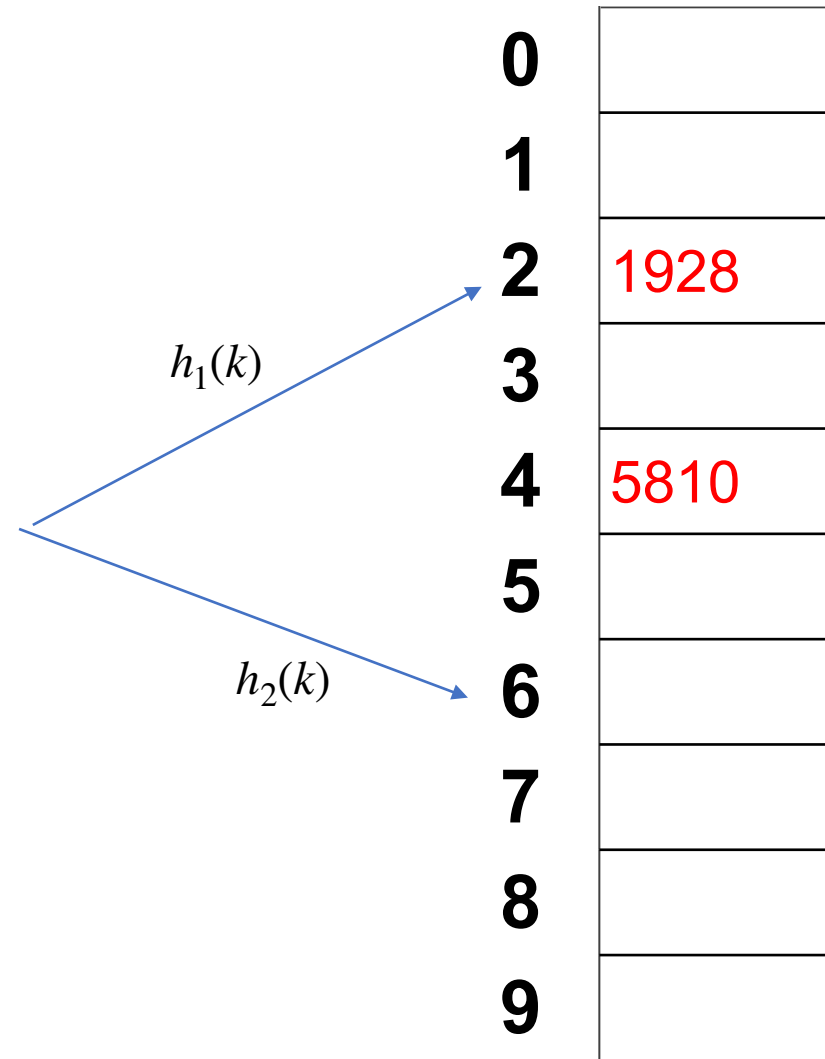
Insert: 5810



Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

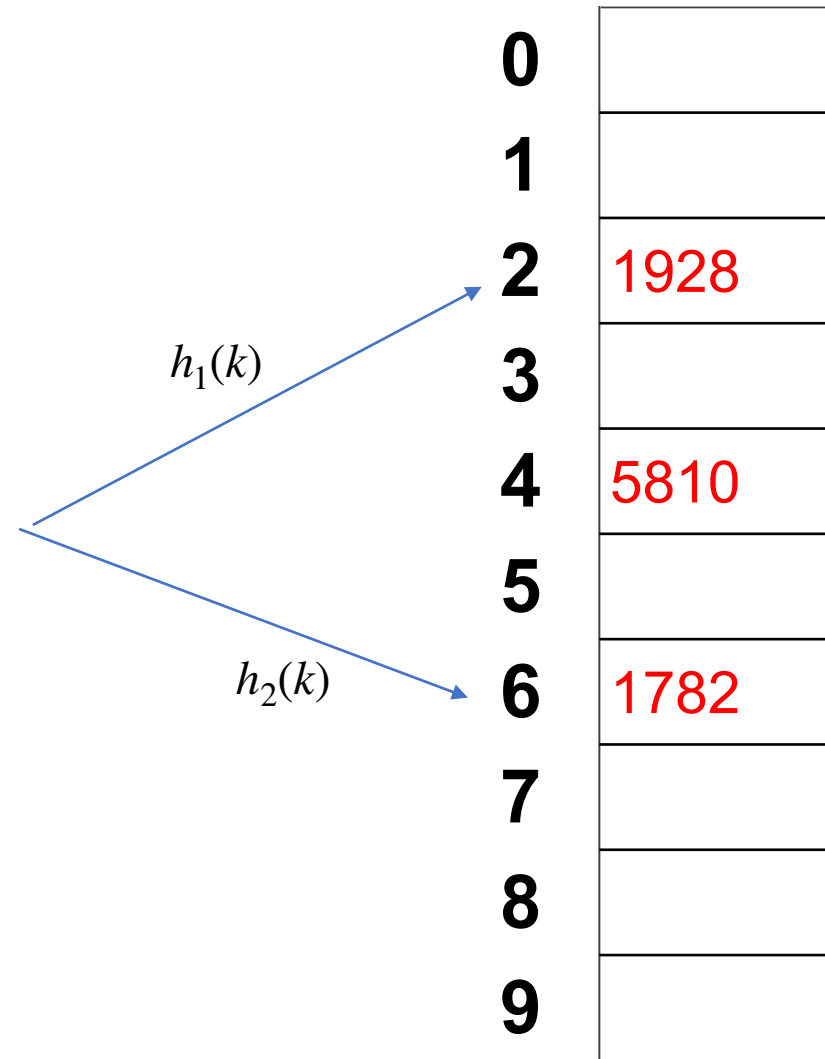
Insert: 1782



Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

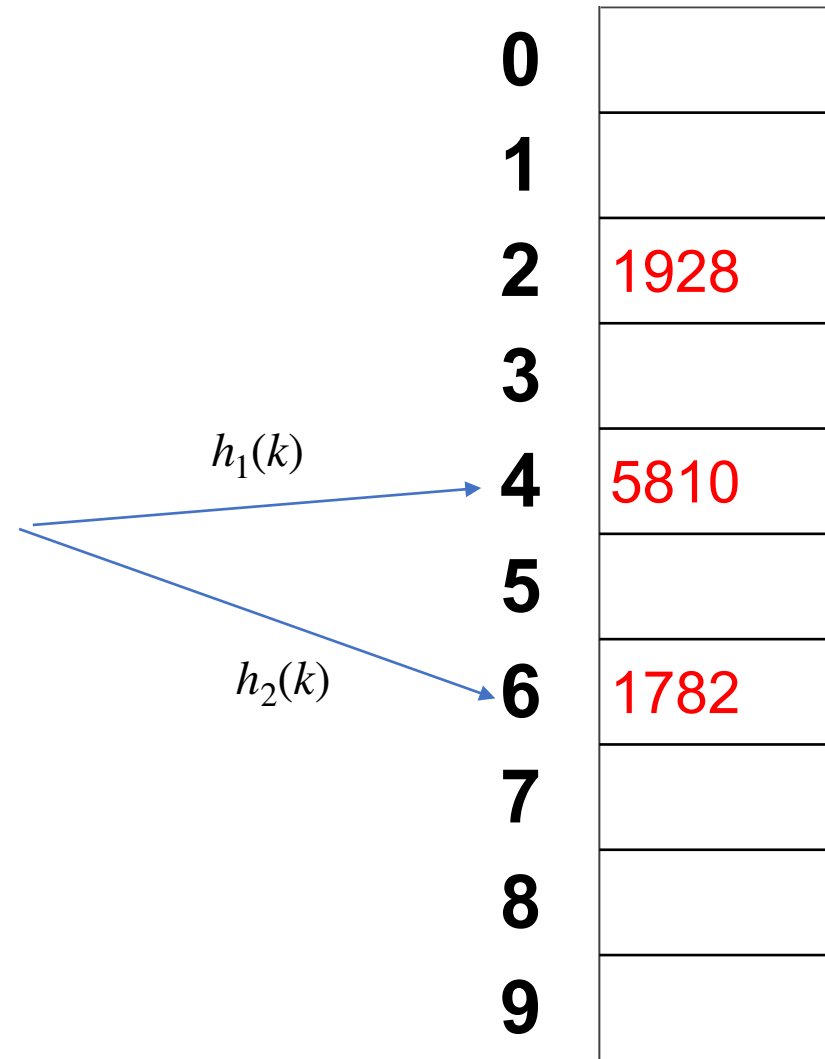
Insert: 1782



Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

Insert: 6891

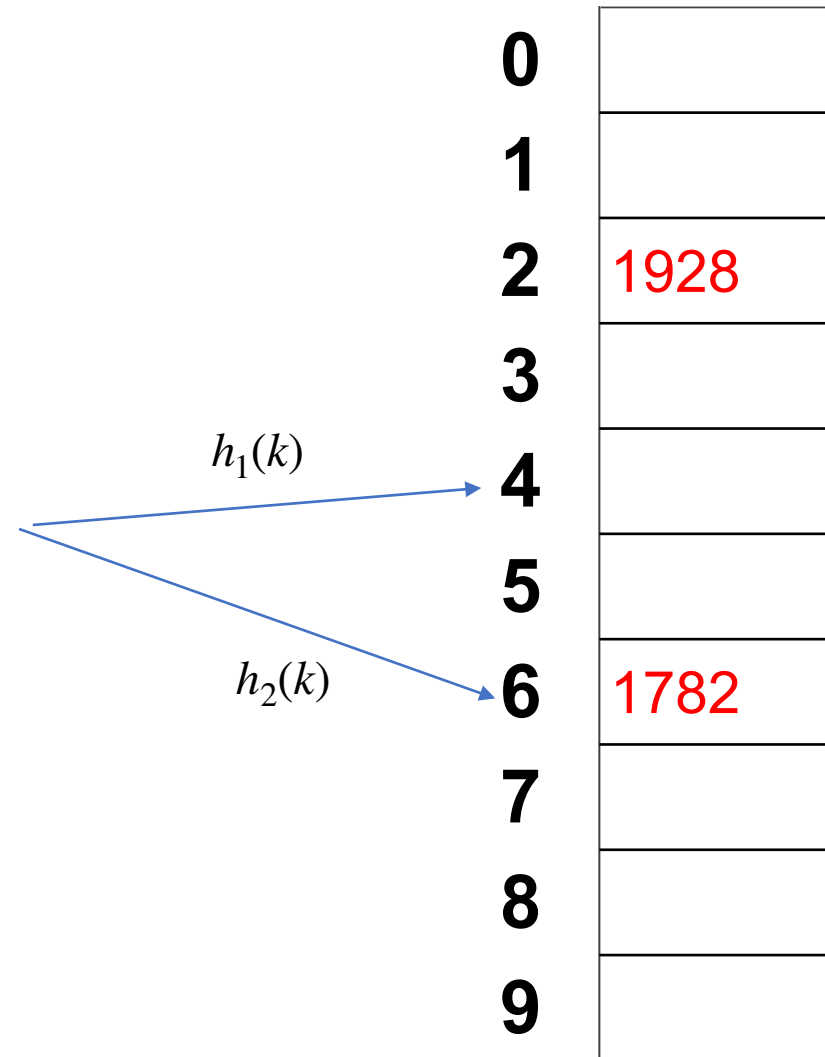


Both occupied,
kick one out!

Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

Insert: 6891



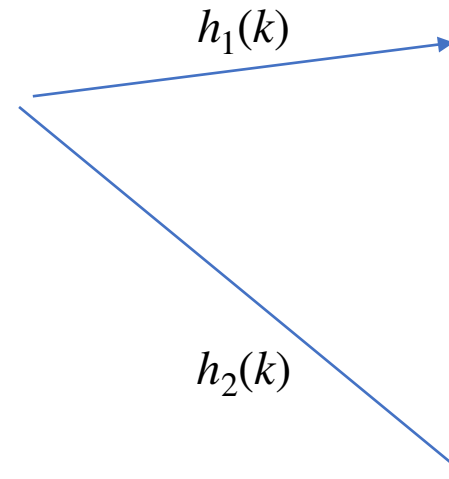
5810

Both occupied,
kick one out!

Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

Insert: 5810



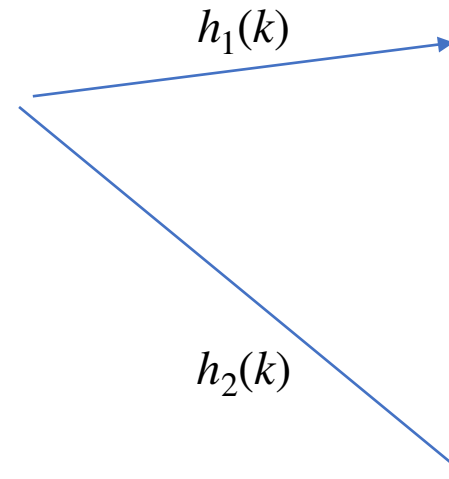
0	
1	
2	1928
3	
4	6891
5	
6	1782
7	
8	
9	

Insert the kicked element again

Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

Insert: 5810



0	
1	
2	1928
3	
4	6891
5	
6	1782
7	
8	5810
9	

Insert the kicked element again

Cuckoo hashing

Use 2 hash functions, insert in empty slot
If not empty, kick one item

Insert: 5810

$h_1(k)$

$h_2(k)$

0

1

2

3

4

5

6

7

8

9

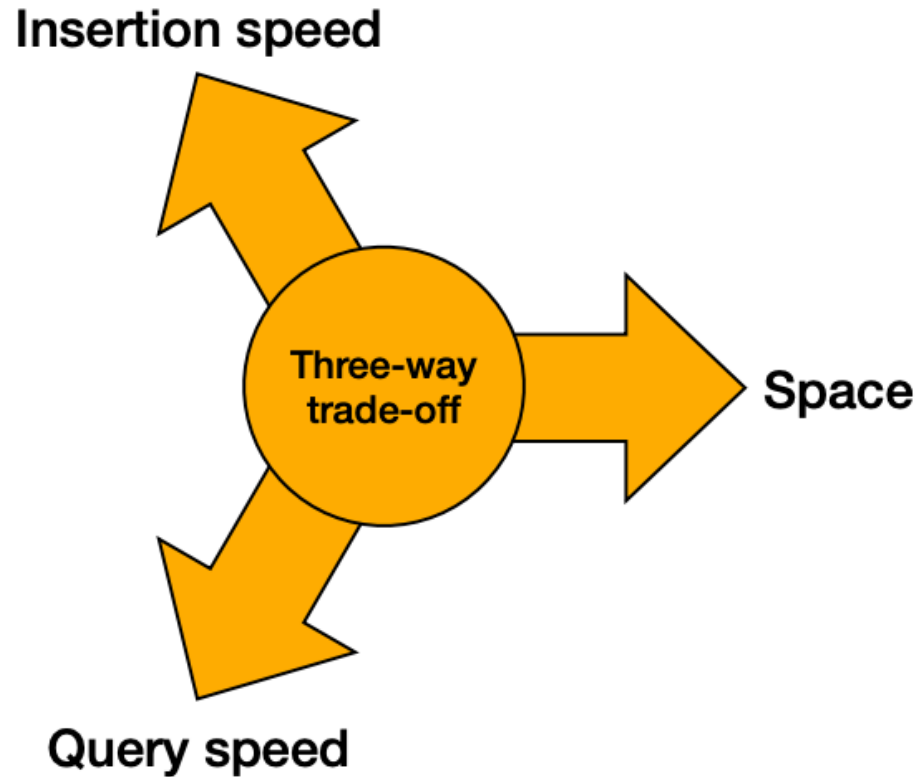
1928
6891
1782
5810

Insert the kicked element again

Lookups: $O(1)$ worst case (2 Probes)

Inserts: Can trigger cascading kicks (kick-out chains)

Hash table performance criteria



Hash table design goals

- **Stability**

- Items don't move after its inserted → Fast inserts

- **Low associativity**

- Items map to a small number of locations → Fast queries

- **Space efficiency**

- Ratio of data size to hash table size → Low space usage

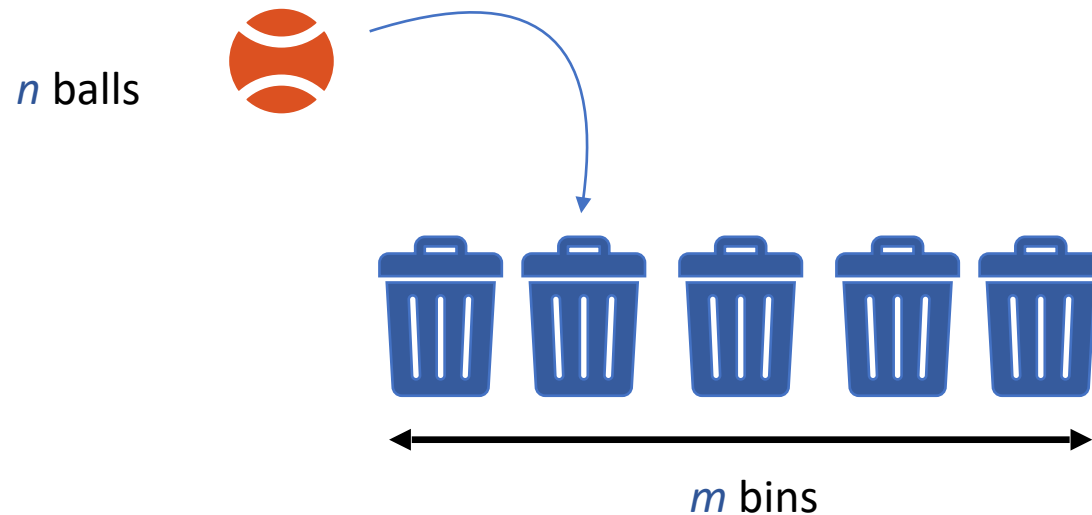
Current hash table space

	Type	Stability	Low associativity	Space efficiency
Cuckoo HT	Cuckoo	✗	✓	✓
Intel TBB	Chaining	✓	✗	✗
CLHT	Chaining	✓	✗	✗
Folklore	Linear probing	✗	✗	✓

Inherent tension between stability, low associativity, and space efficiency.

The balls and bin model

- Throw n balls into m bins:
 - Pick a bin uniformly at random
 - Insert a ball into the bin
 - Repeat n times.



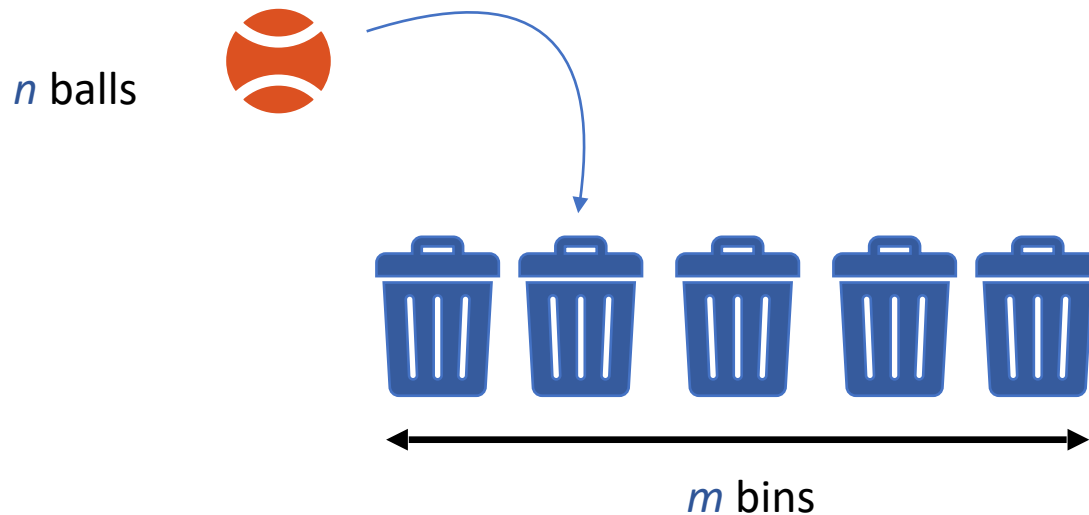
The balls and bin model

- Throw n balls into m bins:
 - Pick a bin uniformly at random
 - Insert a ball into the bin
 - Repeat n times.

Theorem: If you throw n balls into m bins, fullest bin will have $\frac{n}{m} + \sqrt{\frac{n \log m}{m}}$ w.h.p

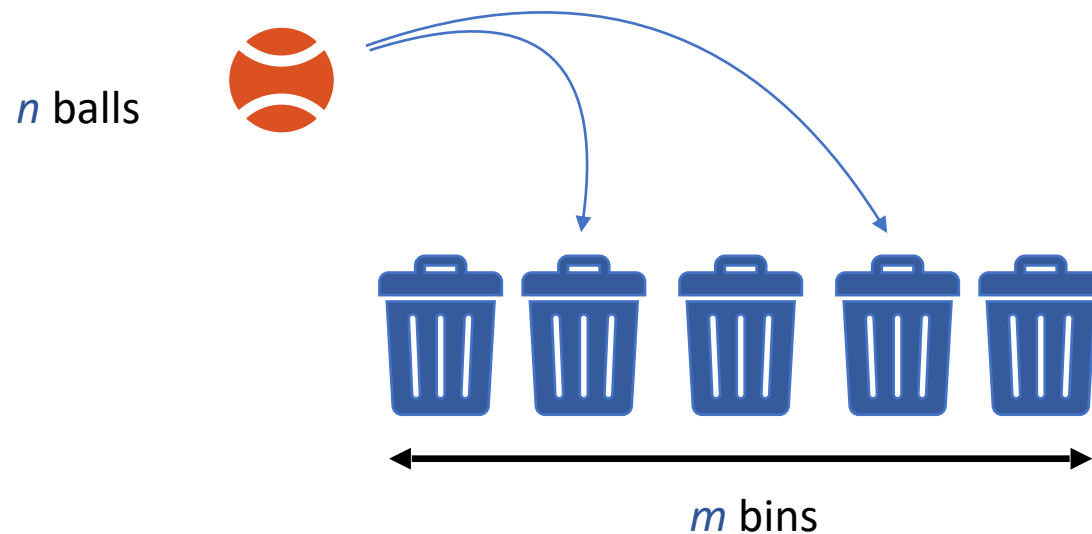
w.h.p = with high probability

$$1 - \frac{1}{n^c}, c \geq 1$$



The multiple choice paradigm

- Throw n balls into m bins:
 - Pick $d \geq 2$ bins uniformly at random
 - Insert a ball into less loaded bin
 - Repeat n times.

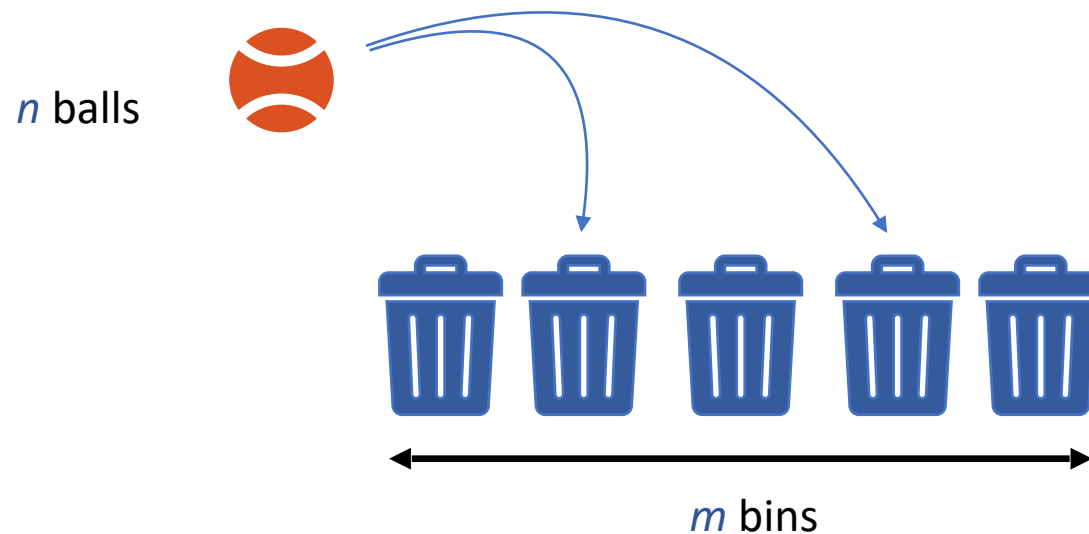


The multiple choice paradigm

- Throw n balls into m bins:
 - Pick $d \geq 2$ bins uniformly at random
 - Insert a ball into less loaded bin
 - Repeat n times.

Theorem: If you throw n balls into m bins, fullest bin will have $\frac{n}{m} + \frac{\log \log m}{\log d}$ w.h.p

- Berenbrink, Czumaj, Steger, Vöcking 2000



The multiple choice paradigm

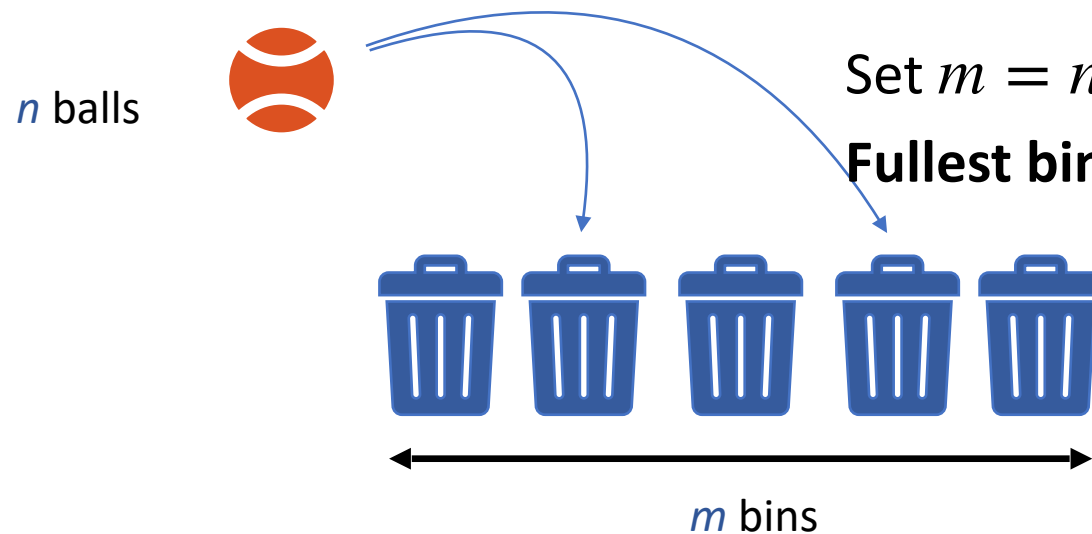
- Throw n balls into m bins:
 - Pick $d \geq 2$ bins uniformly at random
 - Insert a ball into less loaded bin
 - Repeat n times.

Theorem: If you throw n balls into m bins, fullest bin will have $\frac{n}{m} + \frac{\log \log m}{\log d}$ w.h.p

- Berenbrink, Czumaj, Steger, Vöcking 2000

Set $m = n / \log n$

Fullest bin: $\log n + \log \log n + O(1)$ w.h.p

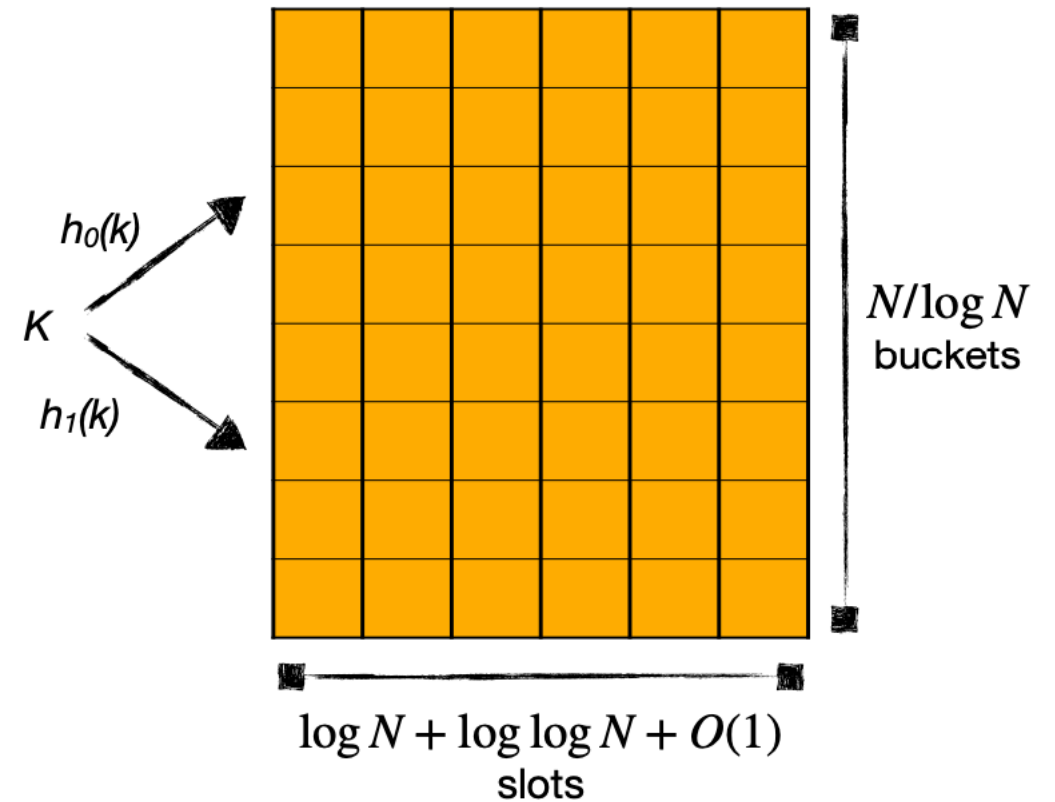


2-choice hashing

- **Stable:** no kicking
- **Low associativity:** $O(\log n)$
- **Space efficient:** $1 - o(1)$

Set $m = n/\log n$

Fullest bin: $\log n + \log \log n + O(1)$ w.h.p



2-choice hashing

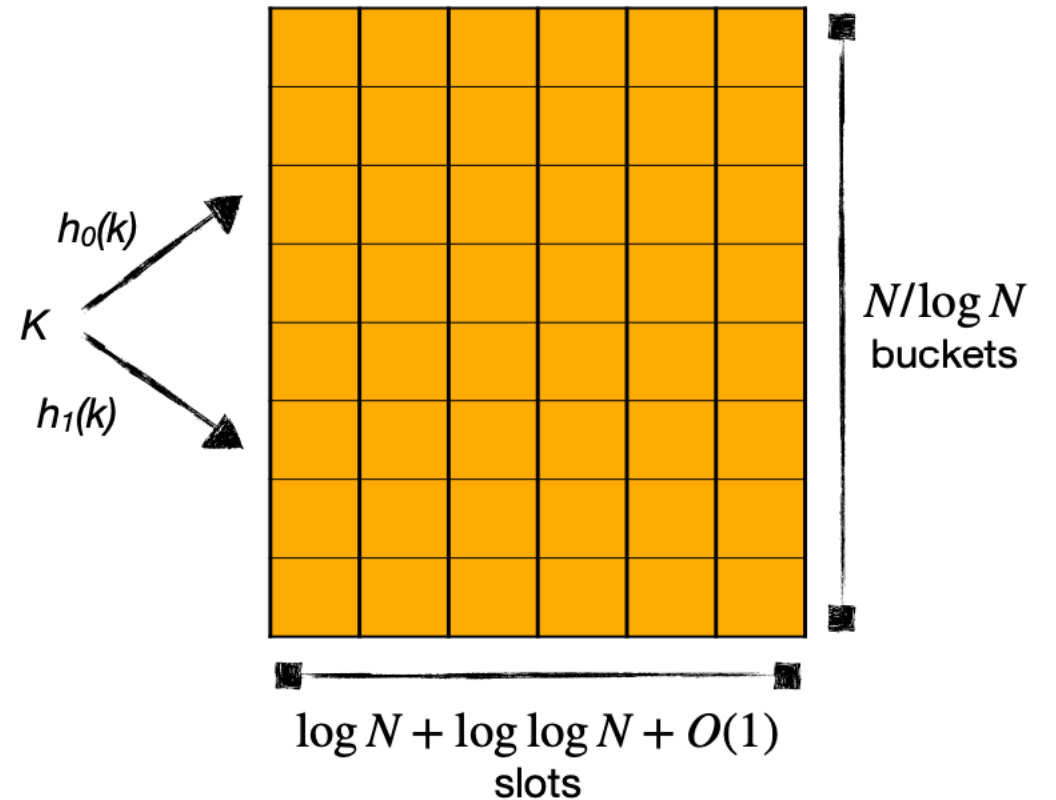
- **Stable:** no kicking
- **Low associativity:** $O(\log n)$
- **Space efficient:** $1 - o(1)$

Problem: Theorem does not hold if we delete items

Opportunity: Theorem holds with deletions if average bucket occupancy is $O(1)$

Set $m = n/\log n$

Fullest bin: $\log n + \log \log n + O(1)$ w.h.p

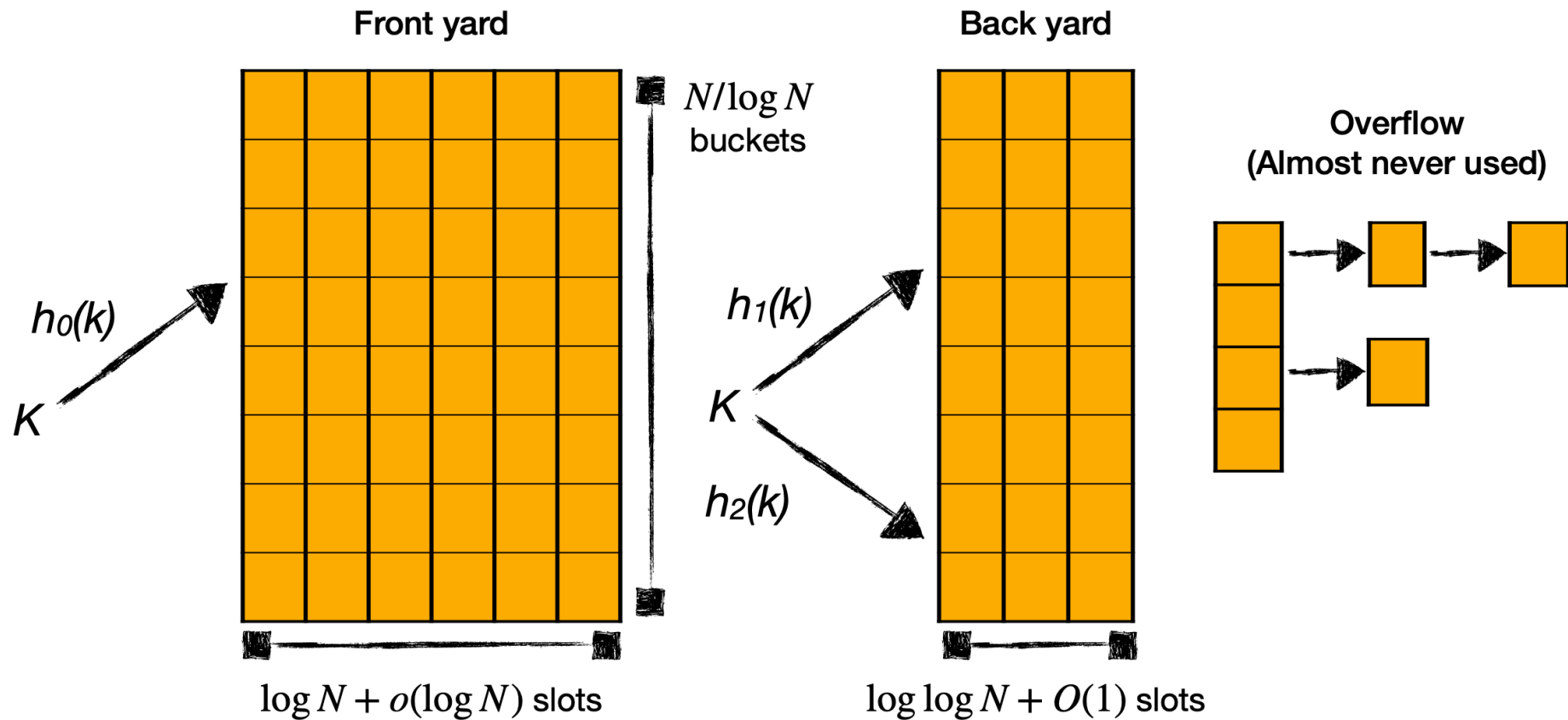


Iceberg hashing

Iceberg theorem: If you throw N balls into bins of size $\log N + o(\log N)$, the number of overflow balls will be $O(N/\log N)$

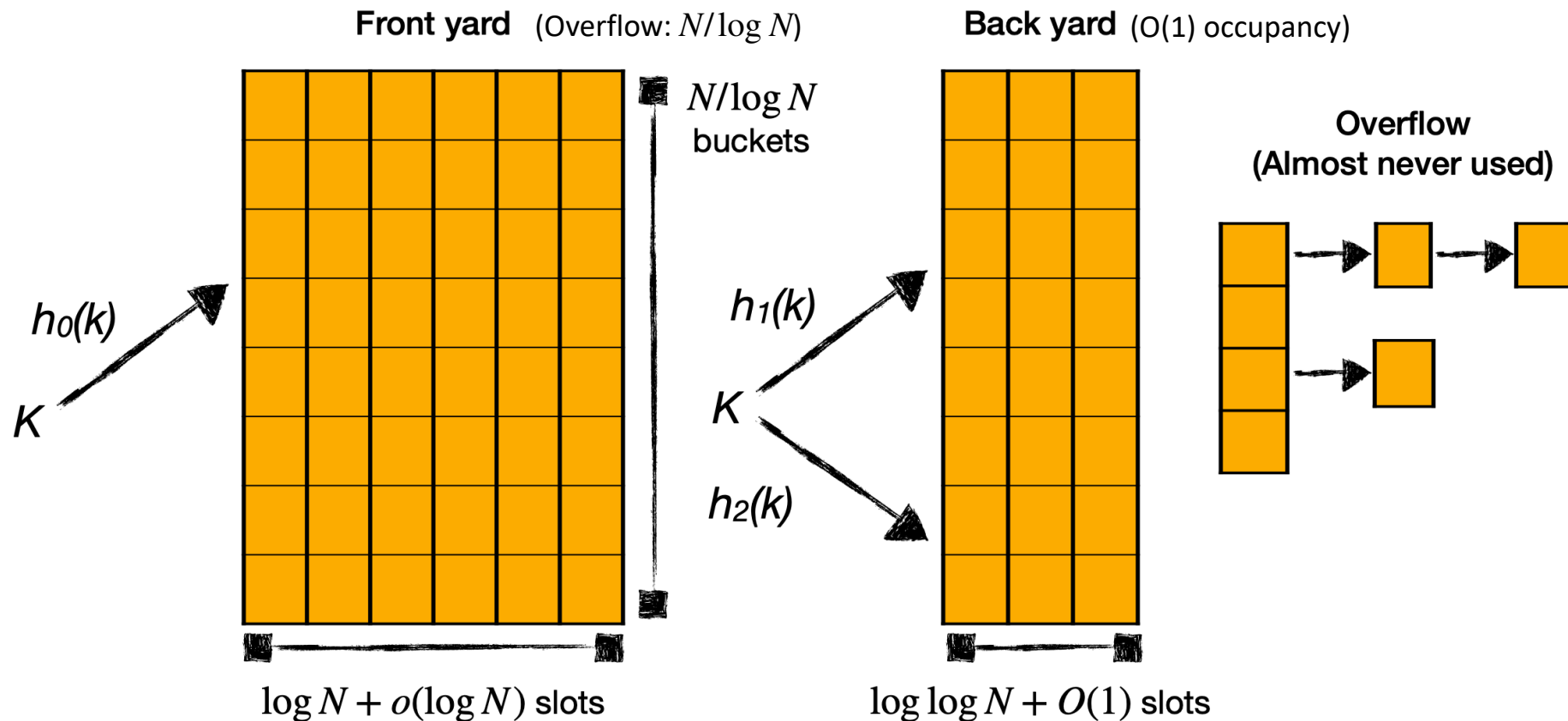
Iceberg hashing

Iceberg theorem: If you throw N balls into bins of size $\log N + o(\log N)$, the number of overflow balls will be $O(N/\log N)$



Iceberg hashing

Iceberg theorem: If you throw N balls into bins of size $\log N + o(\log N)$, the number of overflow balls will be $O(N/\log N)$

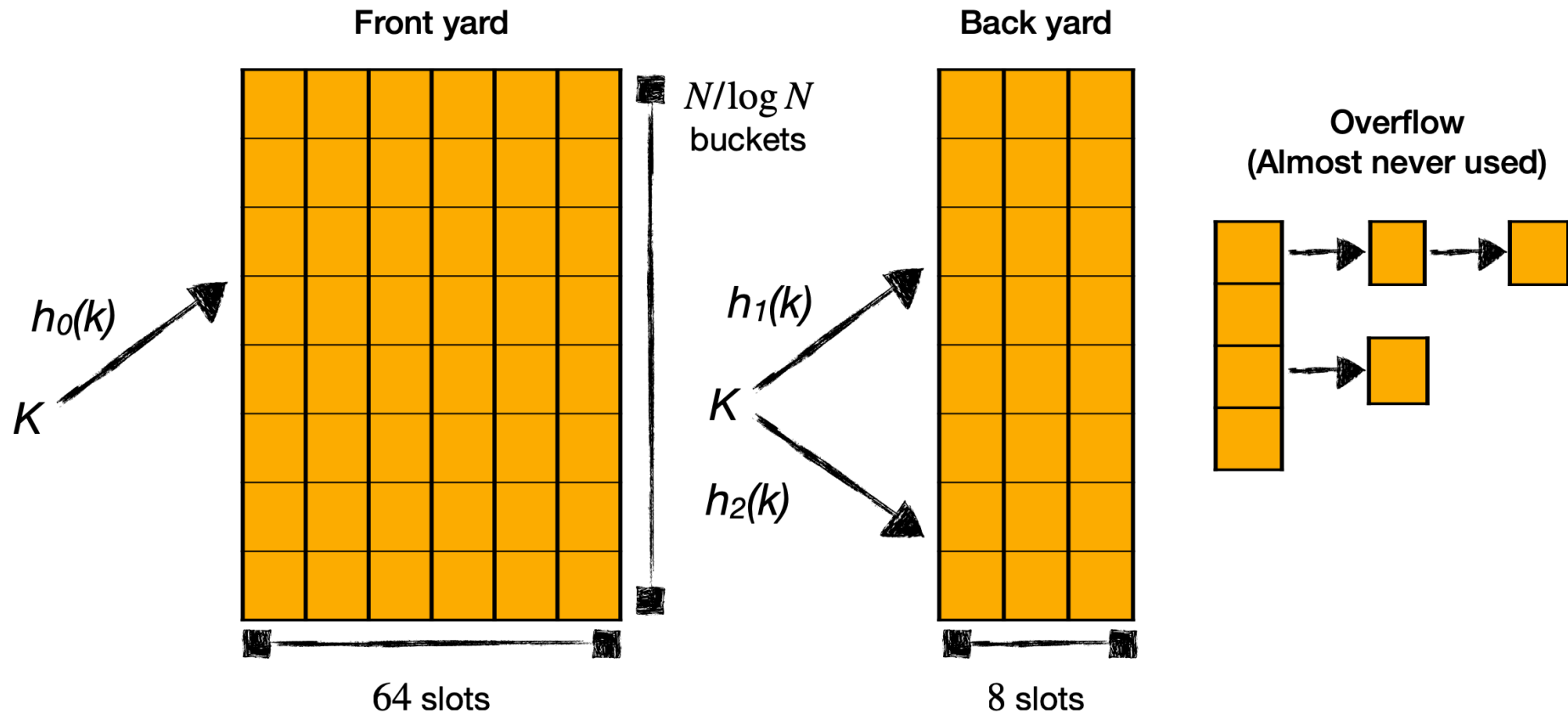


Iceberg hashing

- **Low Associativity:** keys are are in one of 4 locations
 - L1: Frontyard (1)
 - L2: Backyard (2)
 - L3: Overflow (1)
- **Stability:** Items do not move after insertion
- **Space Efficient:** Most items will be L1 or L2

Iceberg hashing

Problem: Buckets in front yard may span multiple cache lines



Block sizes

Block size in level 1

- 64 K-V pairs in each block
- E.g., Key = 8B, Value = 8B
- Block size = 1024B
- Cache line = 64B
- **1 block → 16 cache lines**

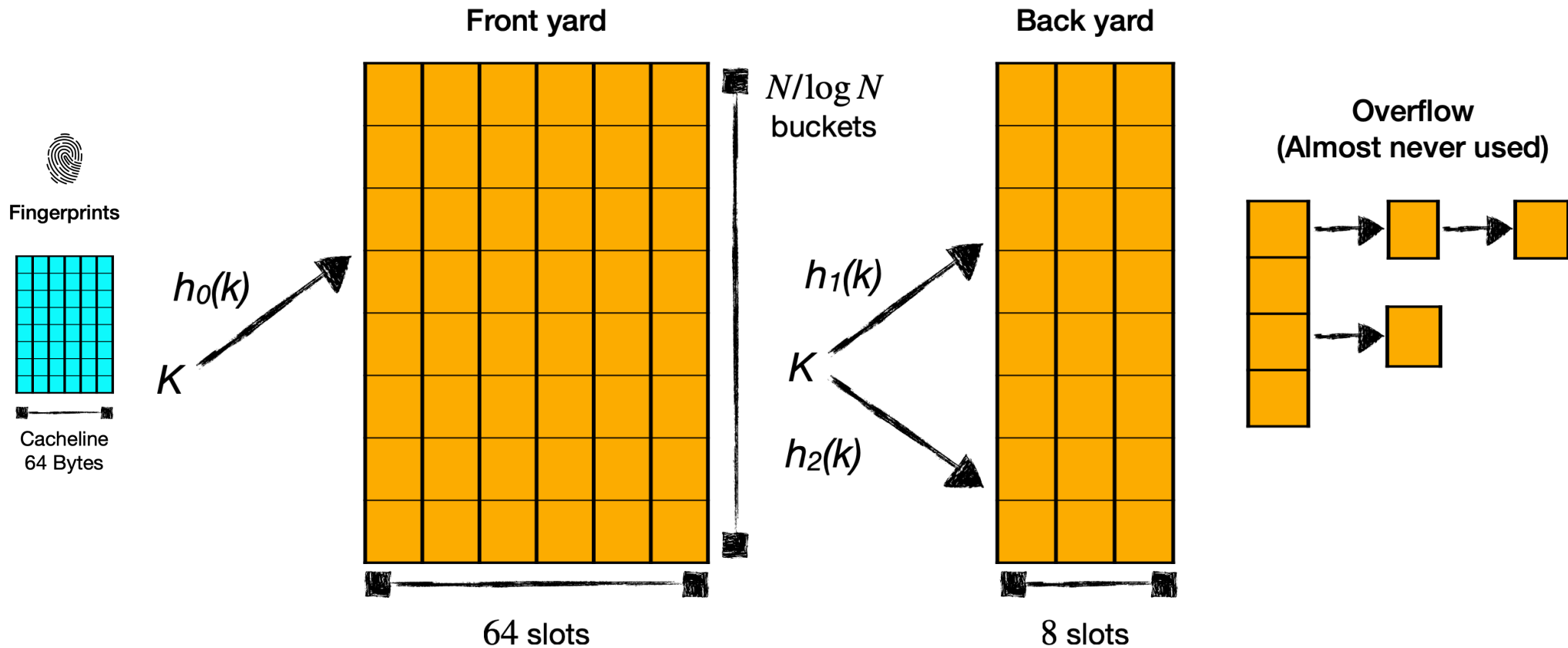
Block size in level 2

- 8 K-V pairs in each block
- E.g., Key = 8B, Value = 8B
- Block size = 128B
- Cache line = 64B
- **1 block → 2 cache lines**
- **2-choice hashing → 4 cache lines**

Iceberg hashing

Problem: Buckets in front yard may span multiple cache lines

Solution: Use a fingerprint table to filter out entries



Iceberg HT performance (DRAM)

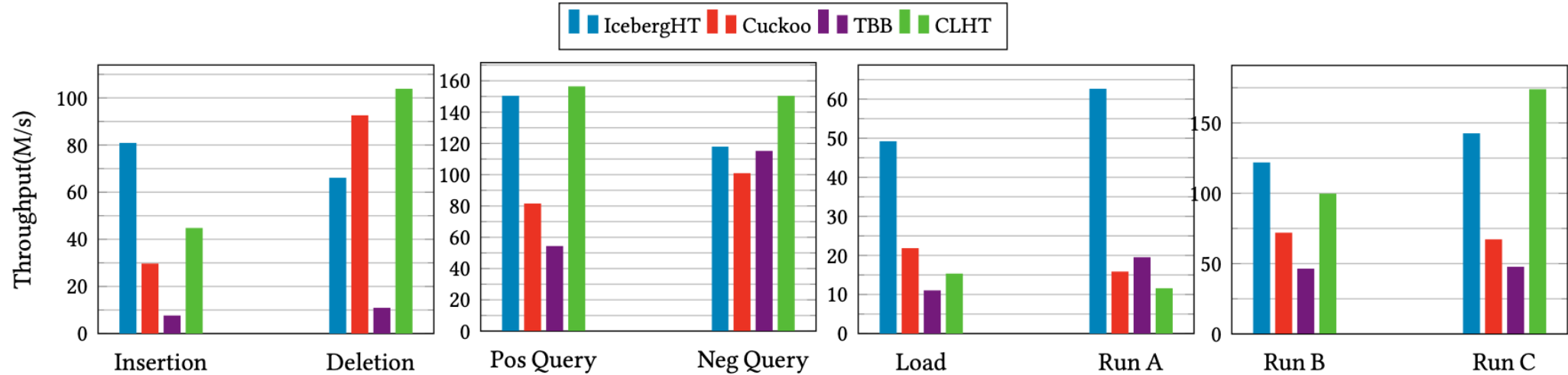


Figure 6: Throughput for insertions, deletions, and queries (positive and negative) using 16 threads for DRAM hash tables. The throughput is computed by inserting $0.95N$ keys-value pairs where N is the initial capacity of the hash table. (Throughput is Million ops/second)

Iceberg HT performance (DRAM)

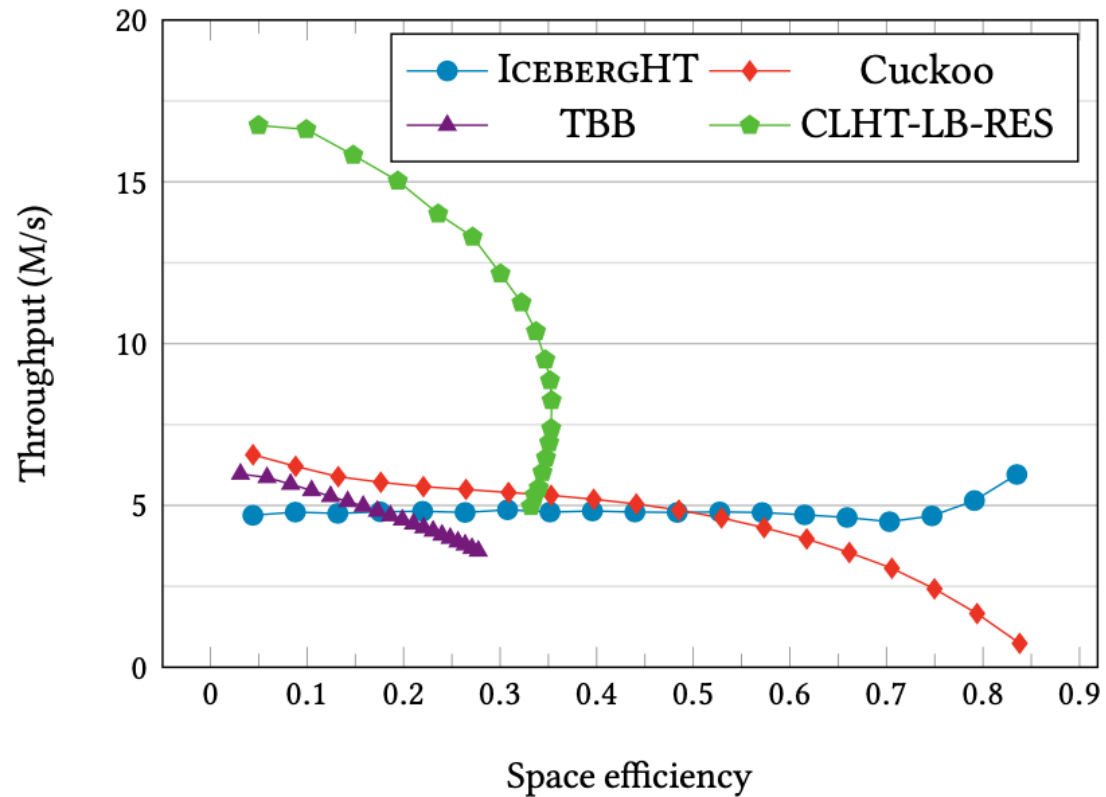


Figure 7: Insertion throughput and space efficiency performance of hash tables in DRAM. (Throughput is Million ops/second)