CS 4440 A

Emerging Database Technologies

Lecture 14
03/04/26

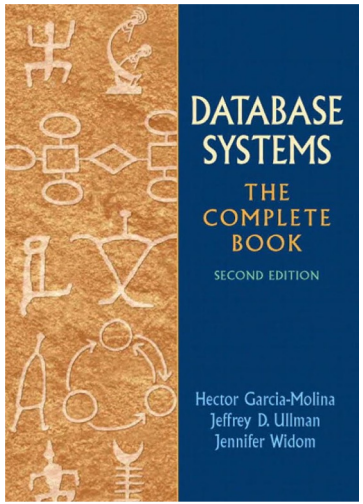
Agenda

1. Static Hash Table
    - Linear Probing Hashing
    - Cuckoo Hashing
2. Dynamic Hash Table
    - Chained Hashing
    - Extensible Hashing
    - Linear Hashing

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 14.3: Hash Tables



Acknowledgement: The following slides have been adapted from EE477 (Database and Big Data Systems) taught by Steven Whang and CS145 (Intro to Big Data Systems) taught by Peter Bailis.

Indexing vs hashing

- Indexing (including B+ trees) is good for range lookups
- Hashing is good for equality-based point lookups

```
SELECT *
FROM Movies
WHERE year >= 2000;
```

```
SELECT *
FROM Movies
WHERE title = ‘Ponyo’;
```

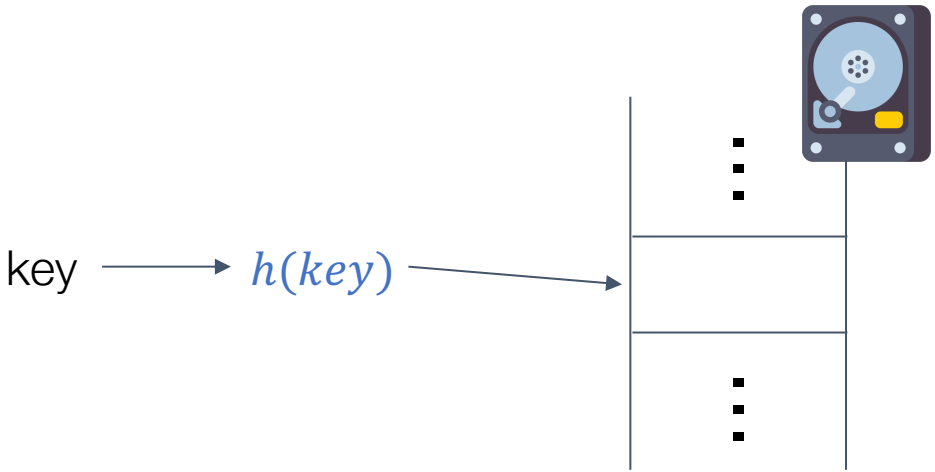
Hash table basics

- A hash function *h* takes a key and returns a block number from 0 to B - 1
- Blocks contain records and are stored in **secondary storage**

Space complexity: O(n)

Time Complexity:

- Average: O(1)
- Worst: O(n)



Hash table: Design Decisions

Hash Function

- How to map a large key space into a smaller domain of array offsets
- Trade-off between fast execution vs. collision rate

Hashing Scheme

- How to handle key collisions after hashing
- Trade-off between allocating a large hash table vs. extra steps to location/insert keys
- Static vs dynamic schemes

Hash function

For any input key, return an integer representation of that key

- Output is deterministic
- Ideally the function distributes keys to all buckets evenly

Example:

- Given a key that is a string, return the sum of the characters x_i modulo B (i.e., Σx_i % B)

We do NOT want to use a cryptographic hash function (e.g., SHA-256) for DBMS hash tables

- Commonly DBMS hash functions (fast and simple): MurmurHash, xxHash

In general, we only care about the hash function's speed and collision rate.

1. Static Hash Table

Static hash table

- The number of buckets is fixed
- Often used during query execution because they are faster than dynamic hashing schemes.
- If the DBMS runs out of storage space in the hash table, it has to rebuild a larger hash table (usually 2x) from scratch, which is very expensive!

Examples

- Linear Probing Hashing
- Robinhood Hashing (not covered)
- Cuckoo Hashing

Linear Probing Hashing

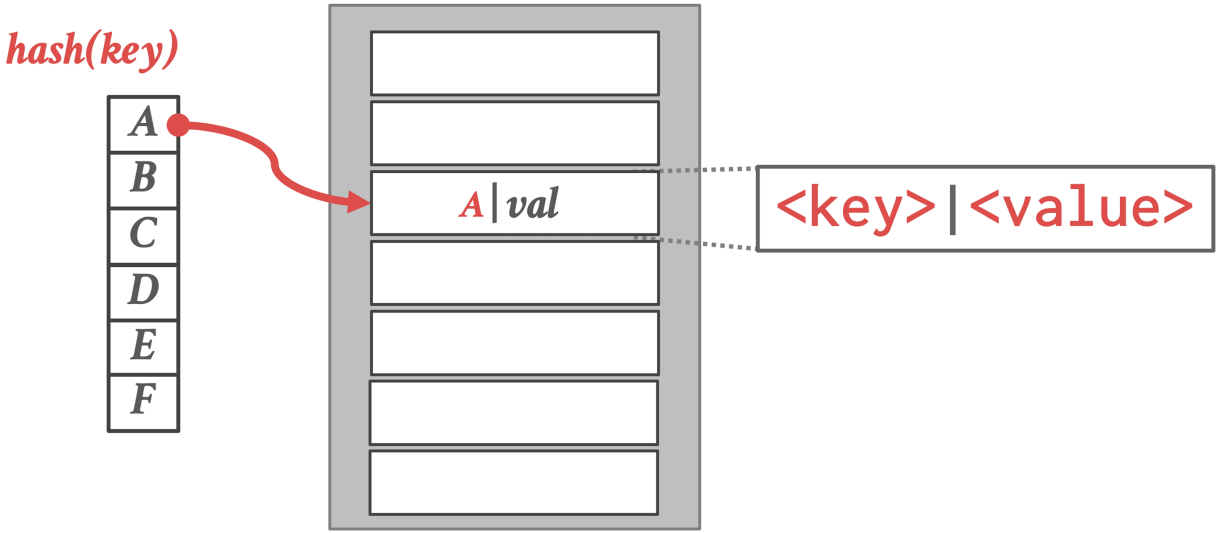
Single giant table of slots

Resolve collisions by linearly searching for the next free slot in the table.

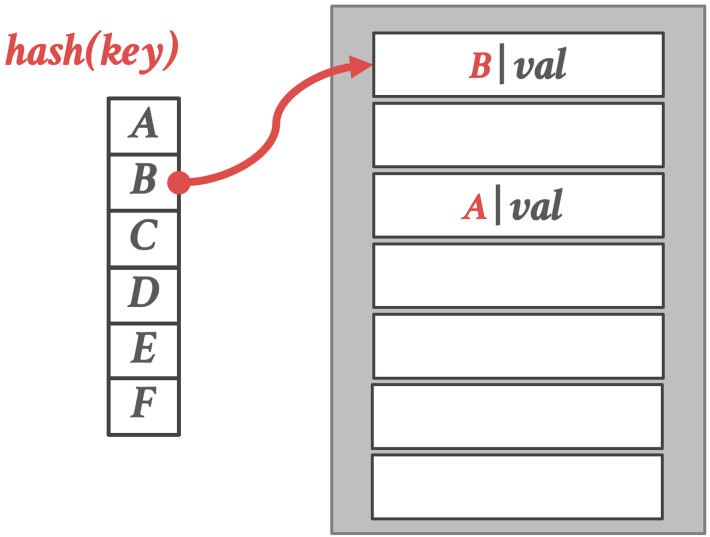
- To determine whether an element is present, hash to a location in the index and scan for it.
- Has to store the key in the index to know when to stop scanning
- Insertions and deletions are generalizations of lookups

Example: Google's absl::flat_hash_map

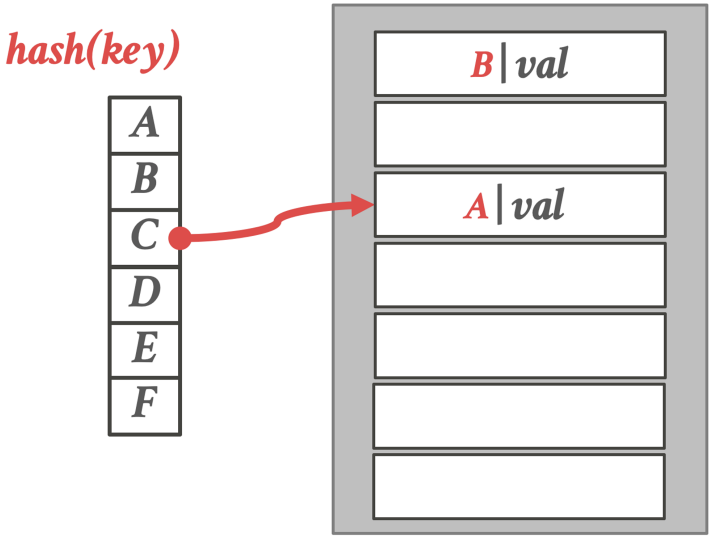
Linear Probing Hashing



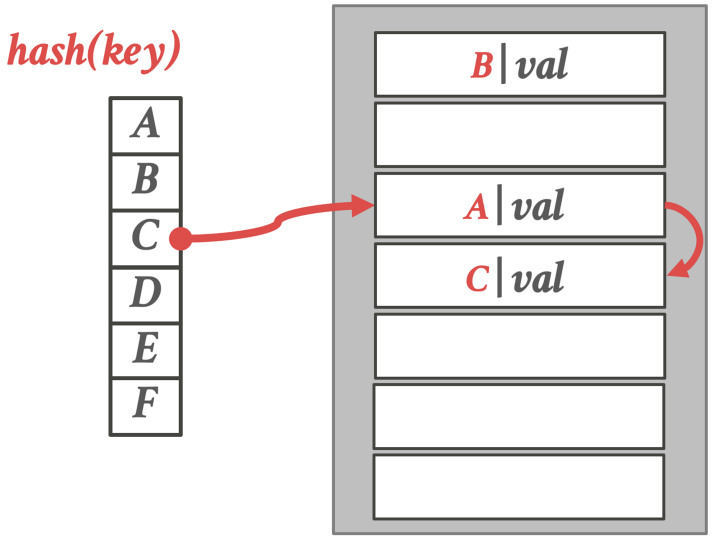
Linear Probing Hashing



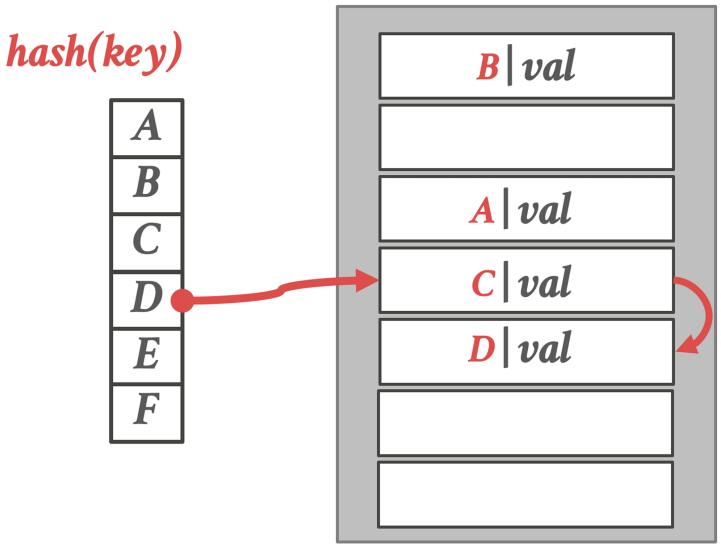
Linear Probing Hashing



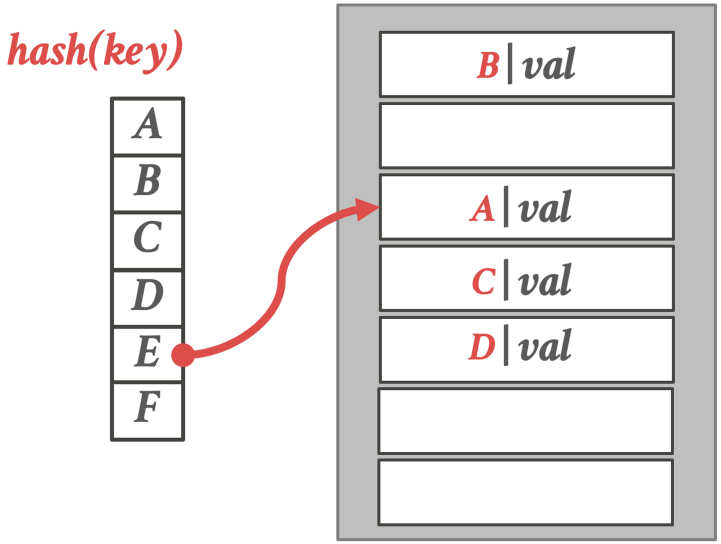
Linear Probing Hashing



Linear Probing Hashing

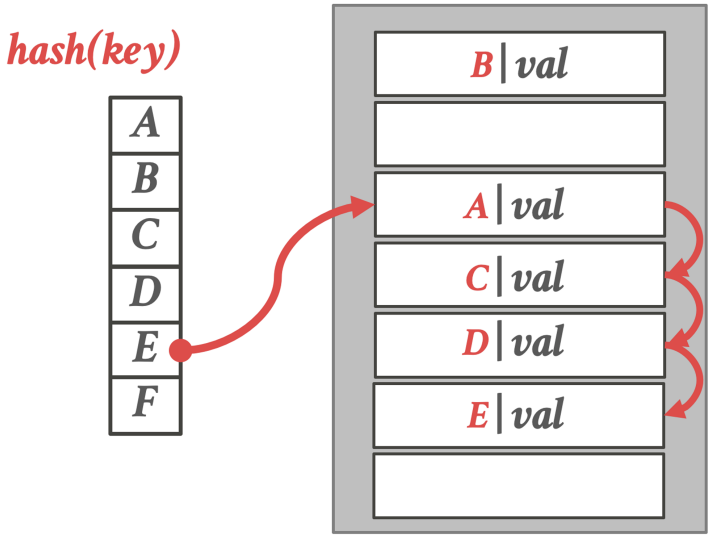


Linear Probing Hashing

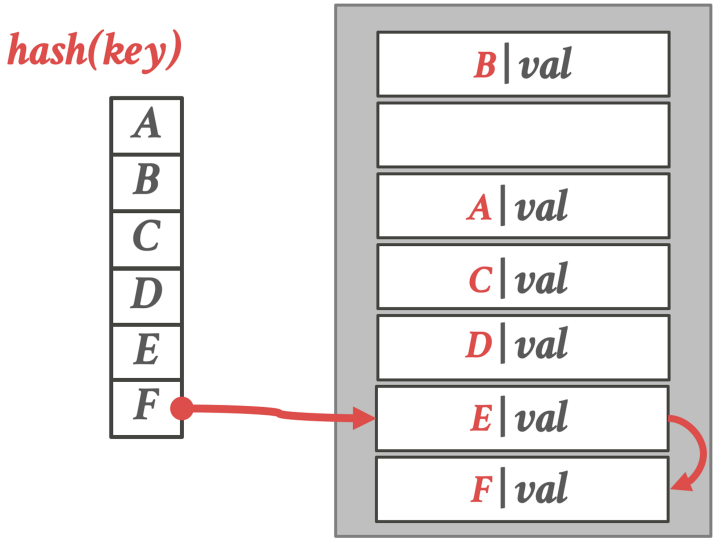


Q: What would happen in this case?

Linear Probing Hashing



Linear Probing Hashing



Linear Probing Hashing - Delete

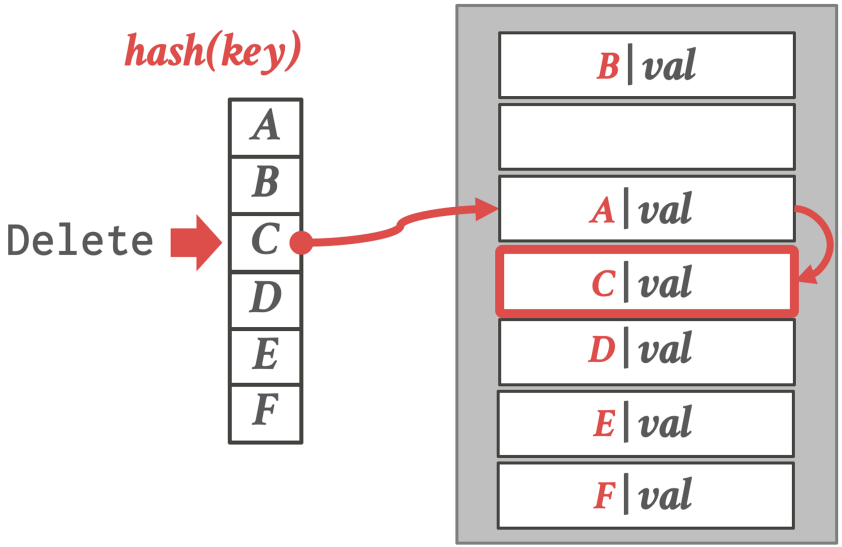
It is not sufficient to simply delete the key

This would affect searches for keys that have a hash value earlier than the emptied cell, but are stored in a position later than the emptied cell.

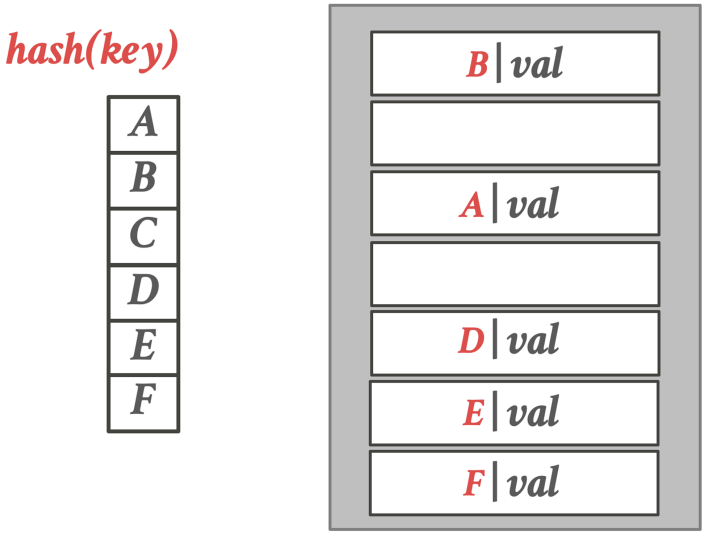
Two solutions:

- Tombstone
- Movement (less common)

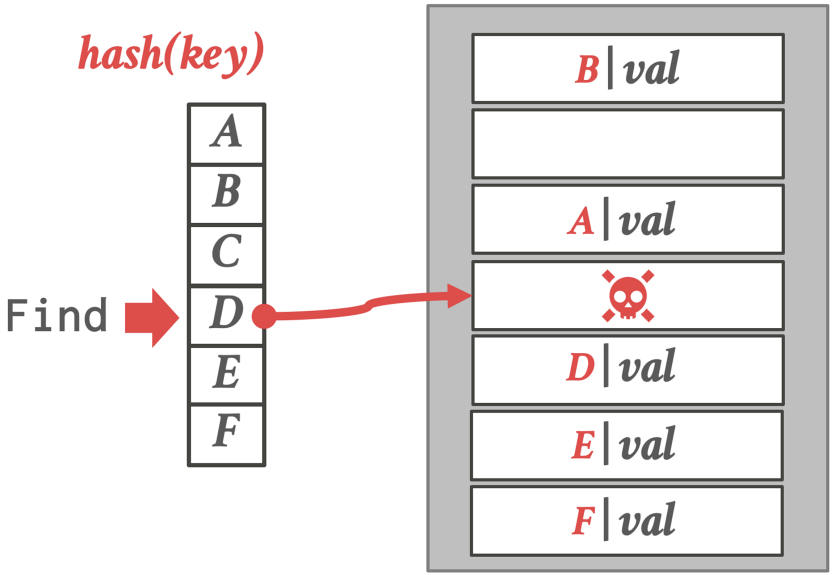
Linear Probing Hashing



Linear Probing Hashing



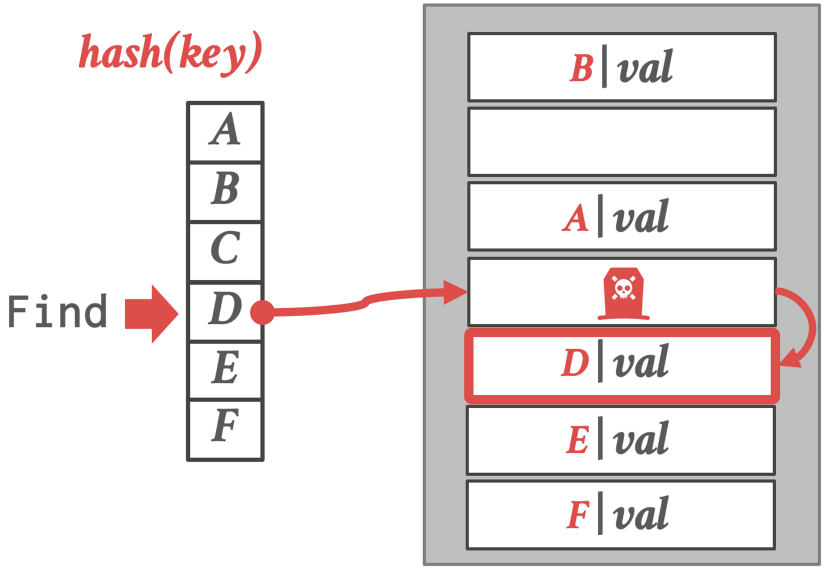
Linear Probing Hashing



Problem: look up for D is affected by the deletion

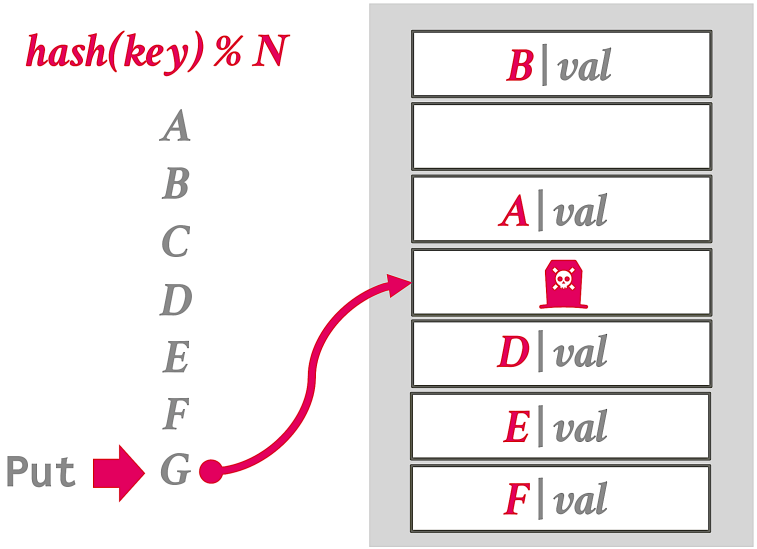
- Set a marker to indicate that the entry in the slot is logically deleted.

Linear Probing Hashing



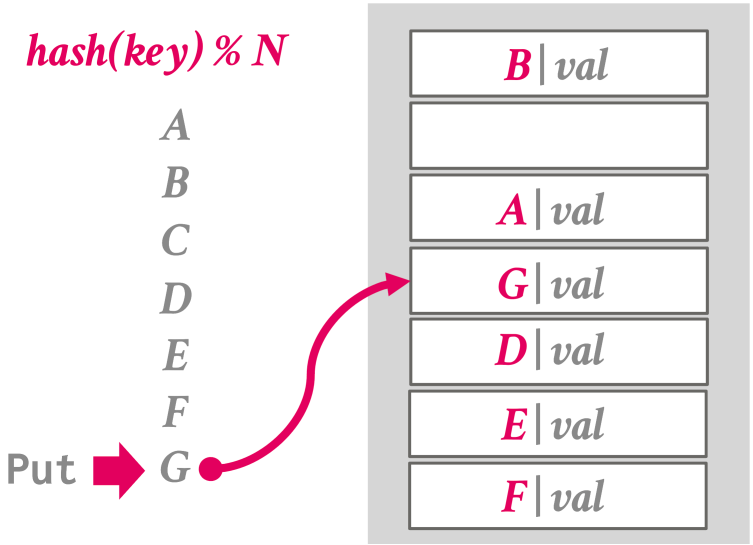
- Set a marker to indicate that the entry in the slot is logically deleted.

Linear Probing Hashing



- Set a marker to indicate that the entry in the slot is logically deleted.
- Can reuse the slot for new keys

Linear Probing Hashing



- Set a marker to indicate that the entry in the slot is logically deleted.
- Can reuse the slot for new keys

Cuckoo Hashing

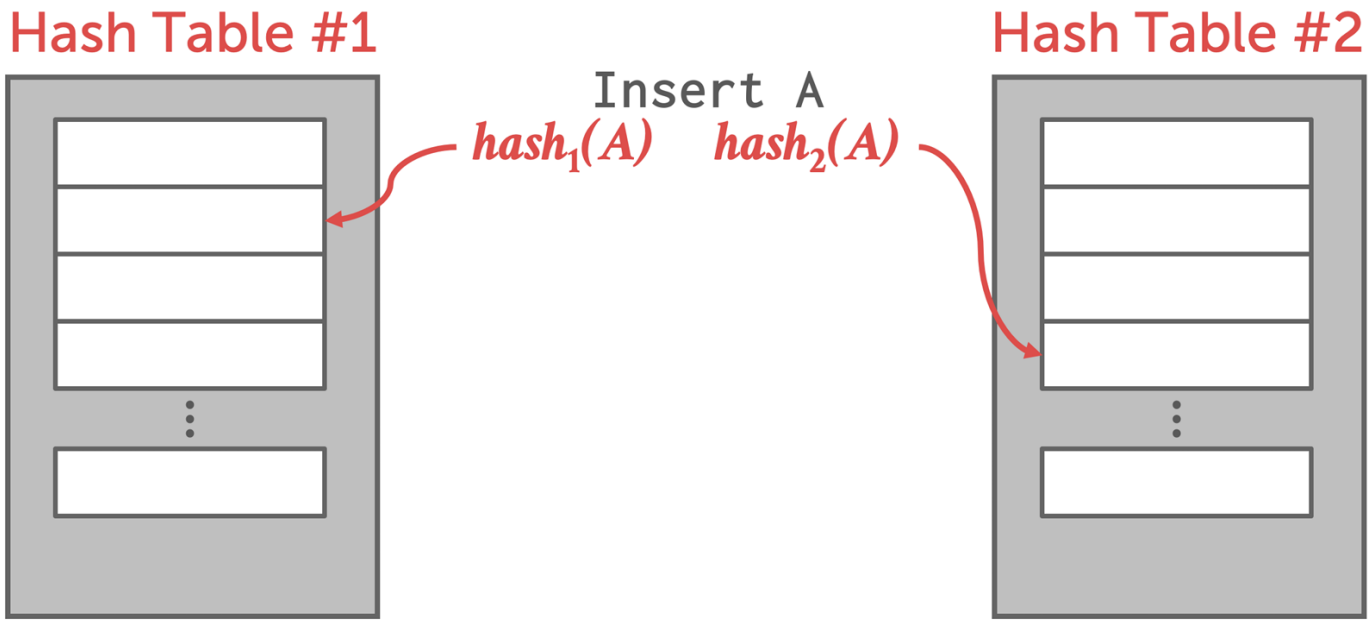
Power of 2 choices: Use multiple hash tables with different seeds

- On insert, check every table and pick one with a free slot
- If no table has a free slot, evict the element from one of then and then re-hash it to find a new location
- In rare cases, we may end up in a cycle. If this happens, we can rebuild using larger hash tables

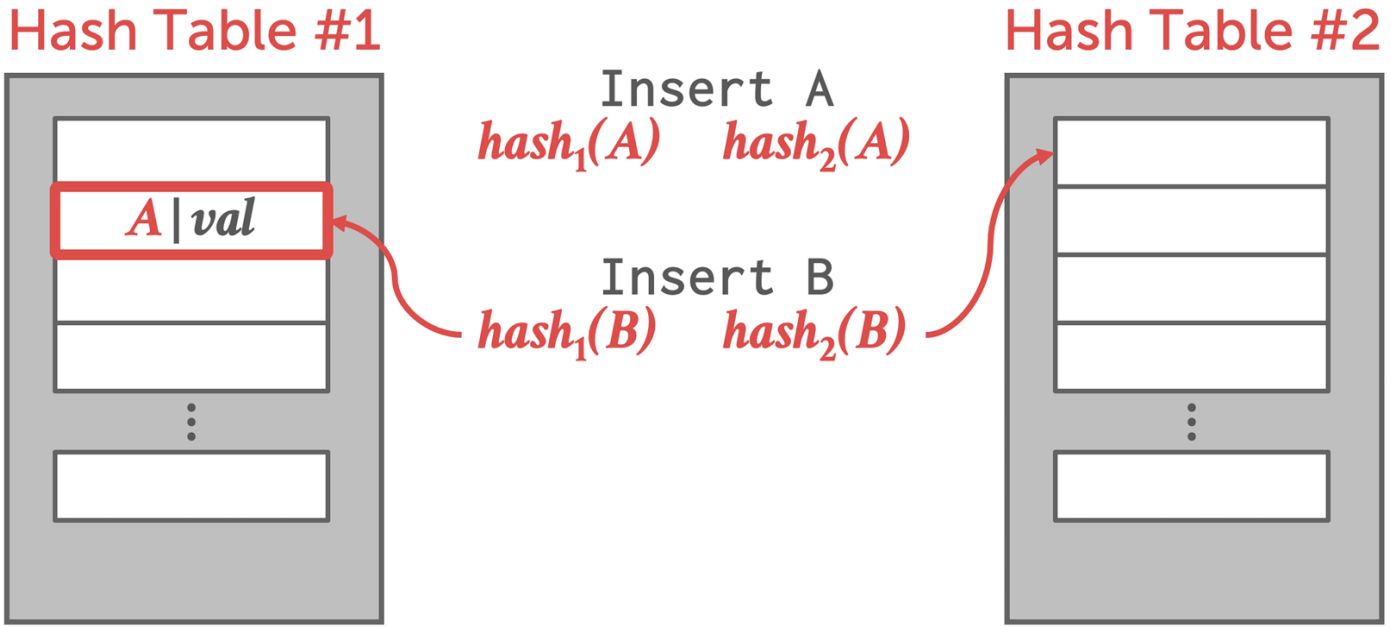
Look-ups and deletions are ~O(1) because only one location per hash table is checked.

Image source: https://theconversation.com/egg-colours-make-cuckoos-masters-of-disguise-34217

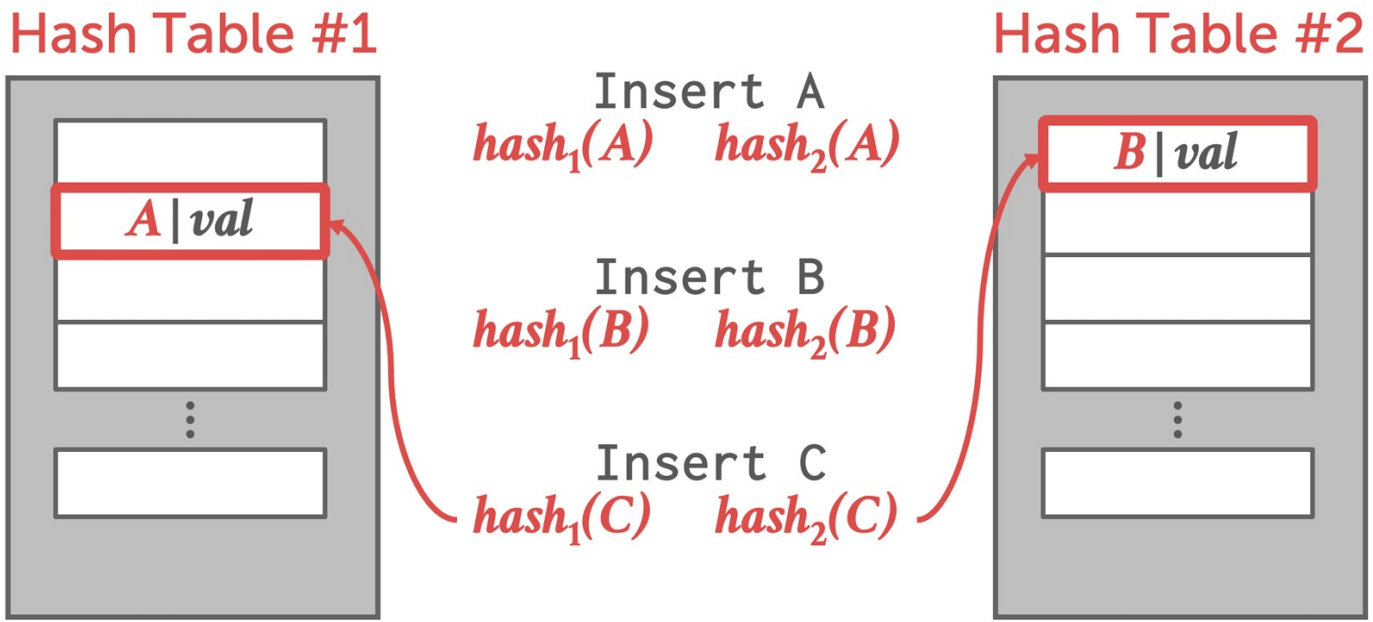
Cuckoo Hashing



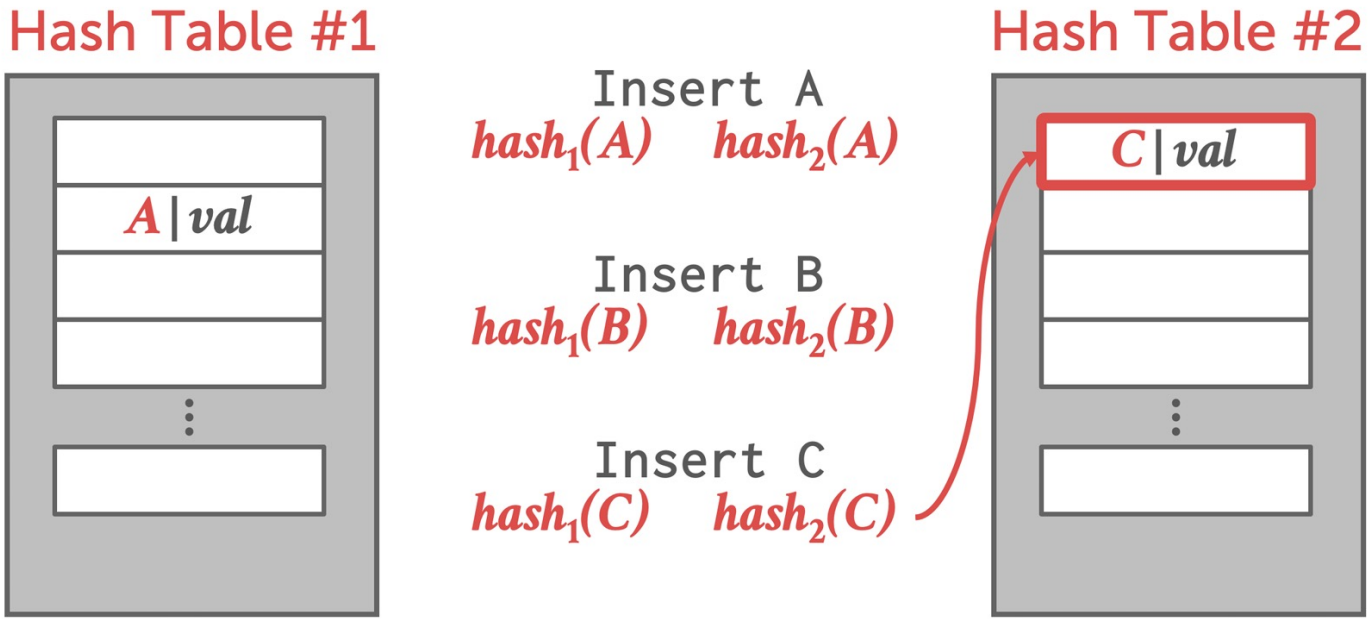
Cuckoo Hashing



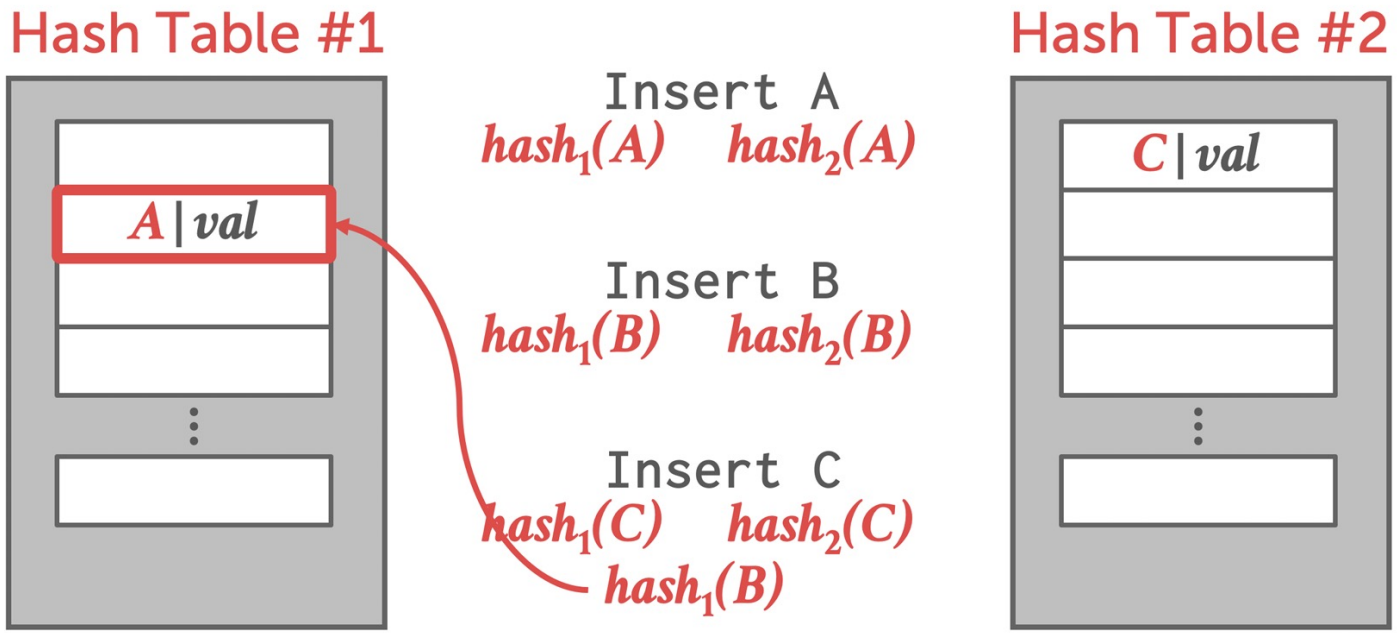
Cuckoo Hashing



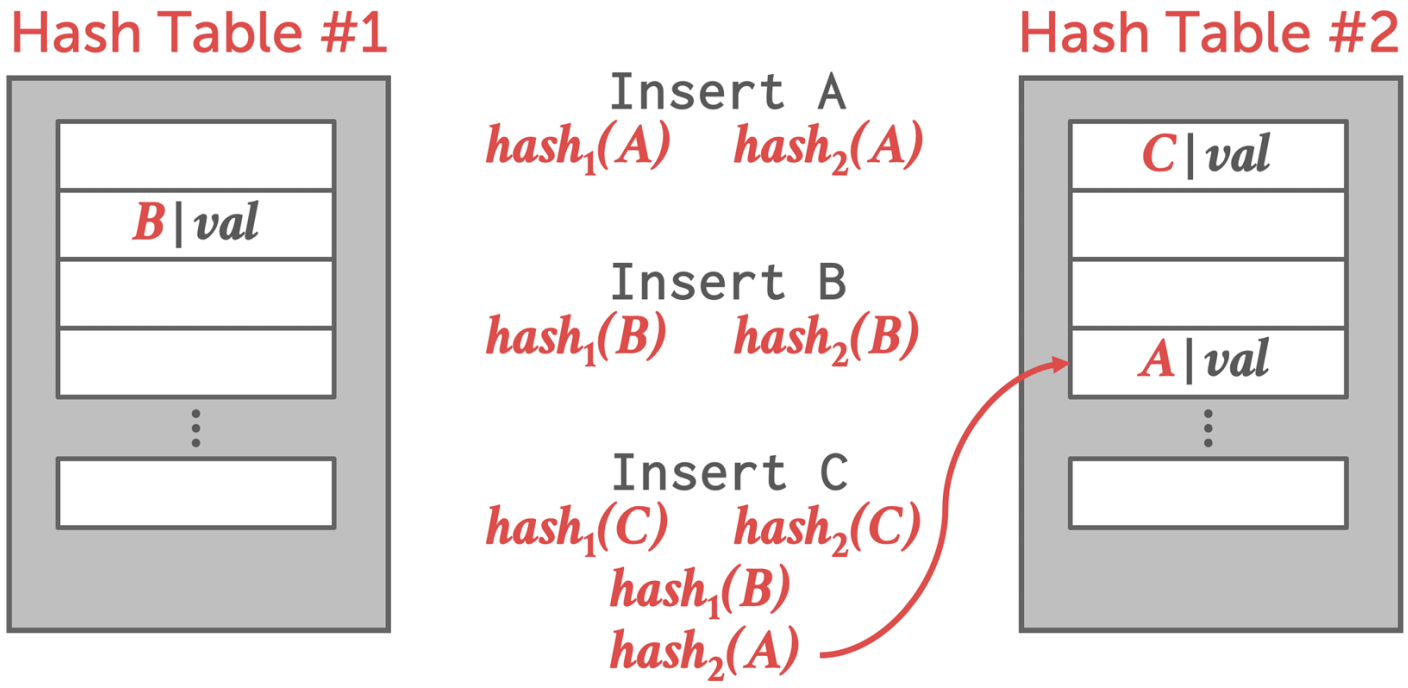
Cuckoo Hashing



Cuckoo Hashing



Cuckoo Hashing



2. Dynamic Hash Table

Dynamic hash table

The previous hash tables require the DBMS to know the number of elements it wants to store; otherwise it needs to rebuild the table to resize

Dynamic hash tables incrementally resize the hash table on demand without needing to rebuild the entire table at once.

- **Key Trade-off:** Eliminates massive rebuild costs in exchange for more complex maintenance overhead during normal operations

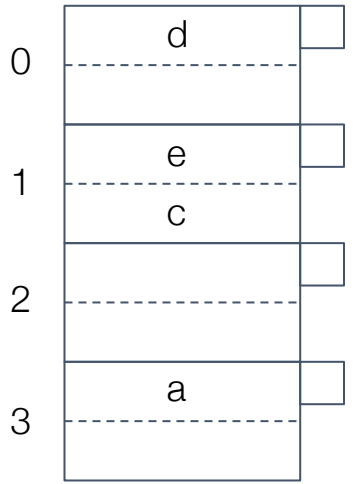
Examples:

- Chained Hashing
- Extensible Hashing
- Linear Hashing

Chained Hashing

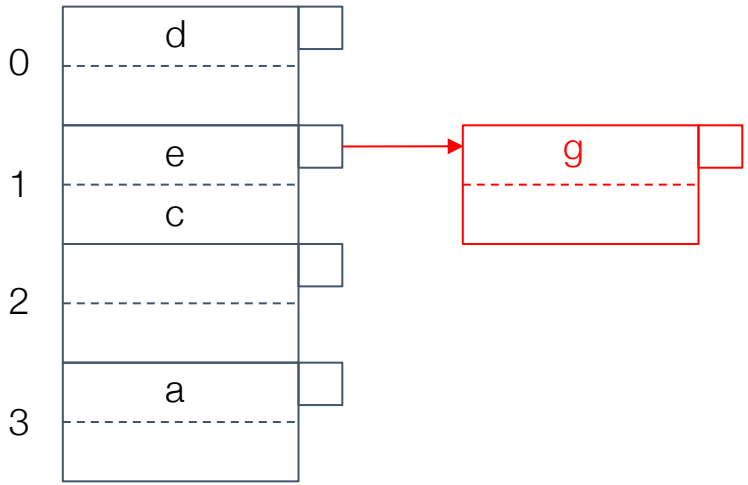
- Maintain a linked list of buckets for each slot in the hash table.

- Resolve collisions by placing all elements with the same hash key into the same bucket.
  - To determine whether an element is present, hash to its bucket and scan for it.
  - Insertions and deletions are generalizations of lookups.



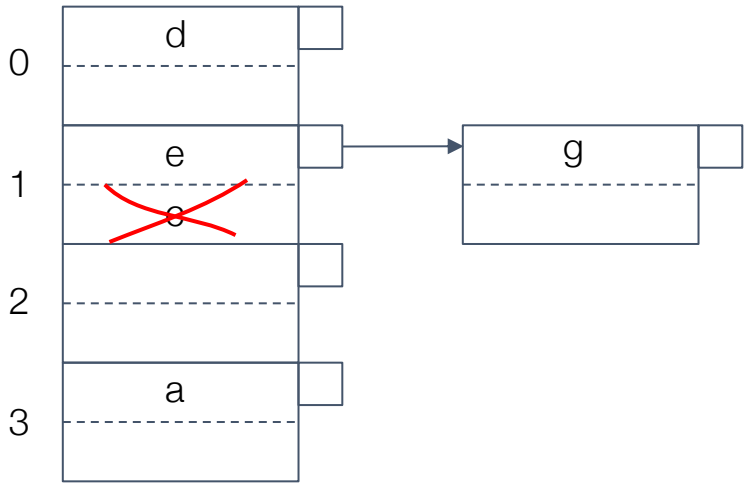
Chained Hashing

- Add g where h(g) = 1



Chained Hashing

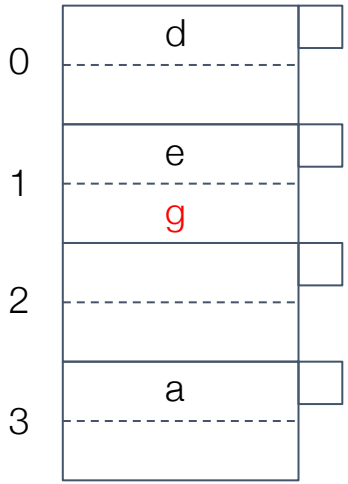
- Remove c where h(c) = 1



Chained Hashing

- Remove c where h(c) = 1

Q: What's the worst-case scenario for chained hashing?



Extensible Hashing

Chained-hashing approach that splits buckets incrementally instead of letting the linked list grow forever.

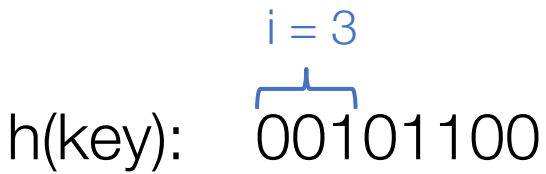
How it works at a high level:

- Uses a *global directory* (an array of pointers) that points to data pages.
- The directory doubles in size when any bucket overflows.
- Only the overflowing bucket is split, not the entire table.
- Uses a *global depth* (for the directory) and *local depths* (for data pages).

Extensible hash table

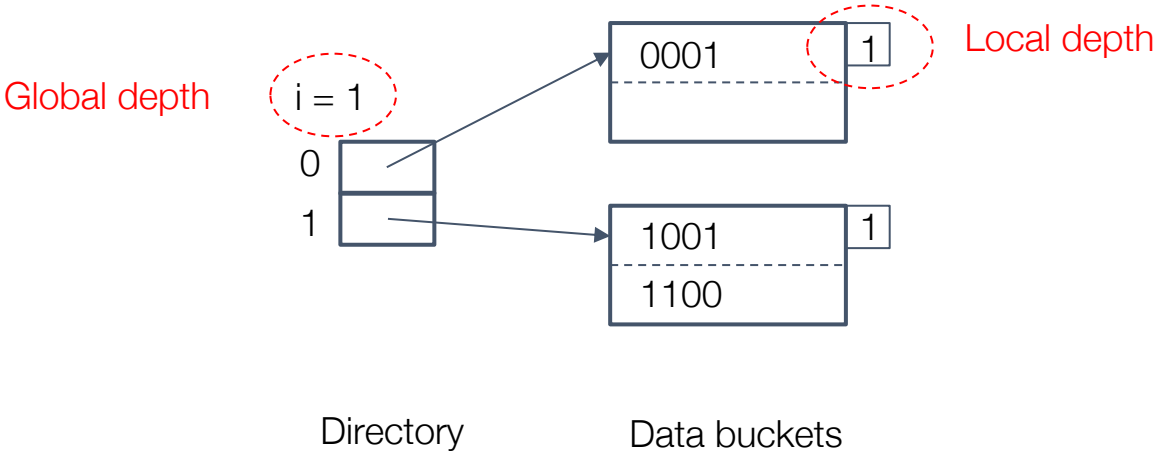
Use first i bits of hash value to locate block

- i grows over time



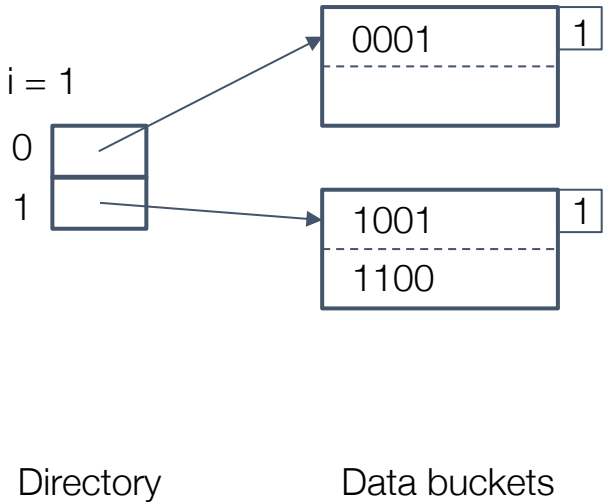
Extensible hash table

Use level of indirection where buckets are pointers to blocks



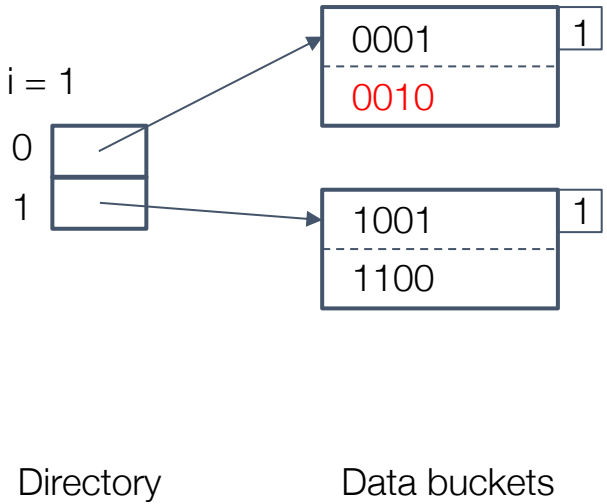
Extensible hash table

- Add 0010



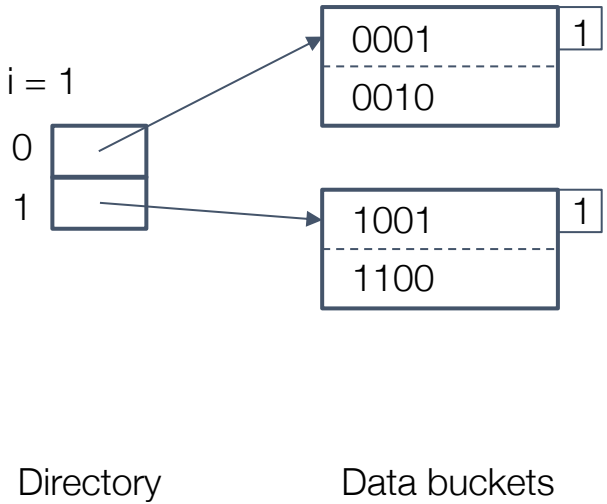
Extensible hash table

- Add 0010



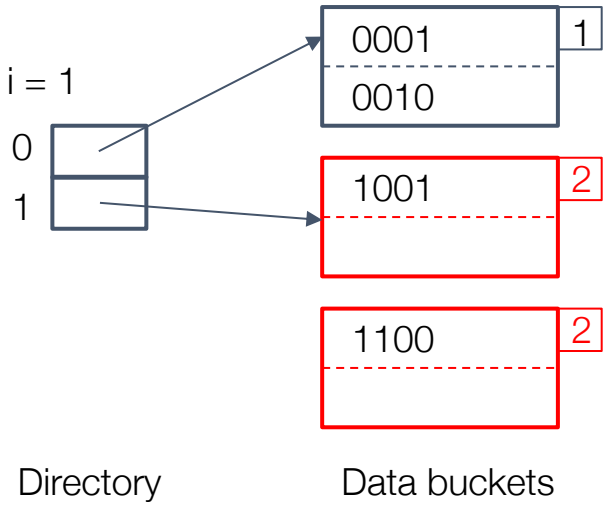
Extensible hash table

- Add 1010



Extensible hash table

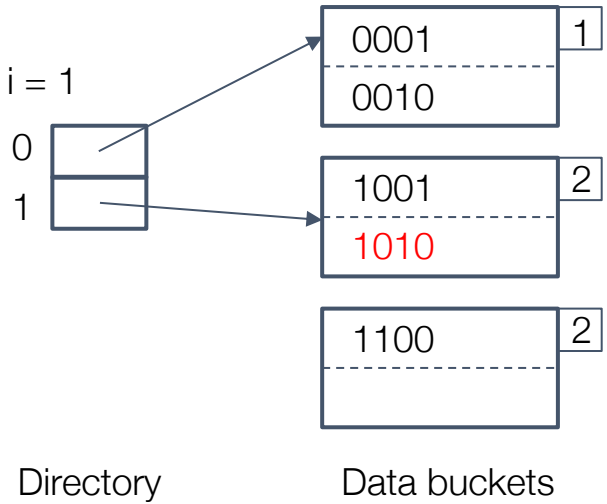
- Add 1010



May need to repeat splitting until there is space

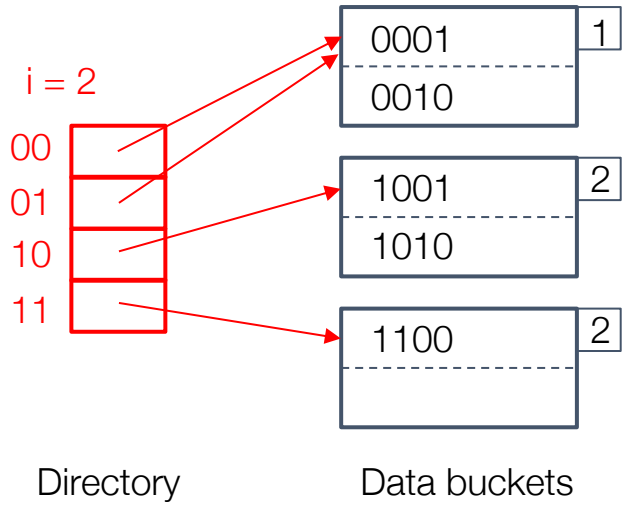
Extensible hash table

- Add 1010



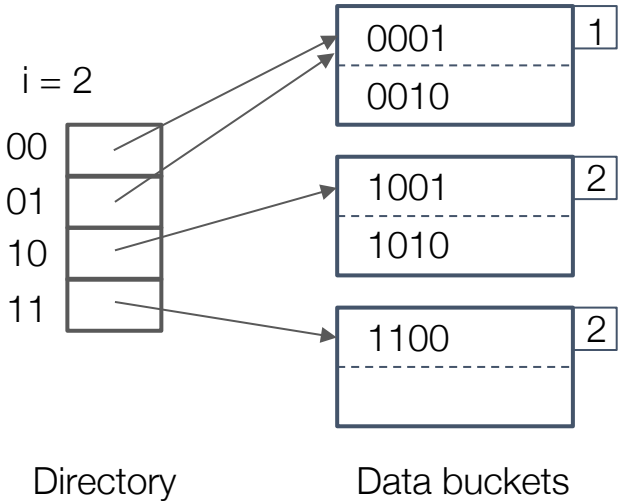
Extensible hash table

- Add 1010



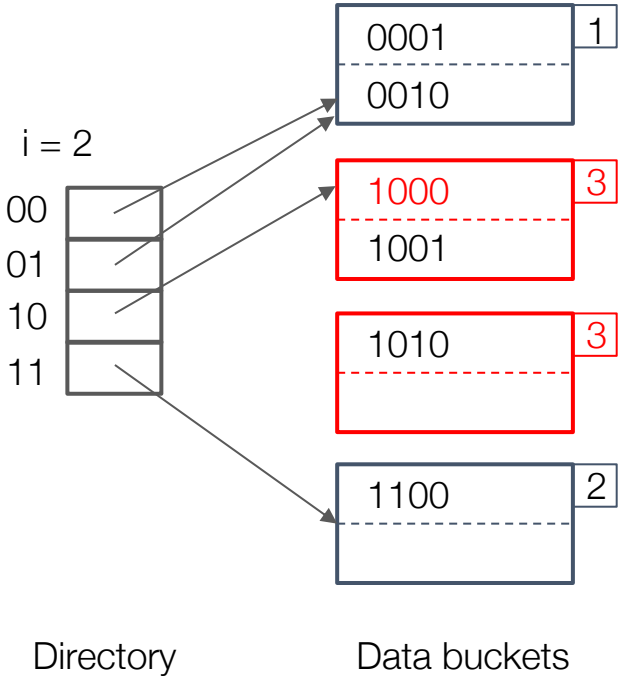
In-class Exercise

- Add 1000
- What happens in this case?



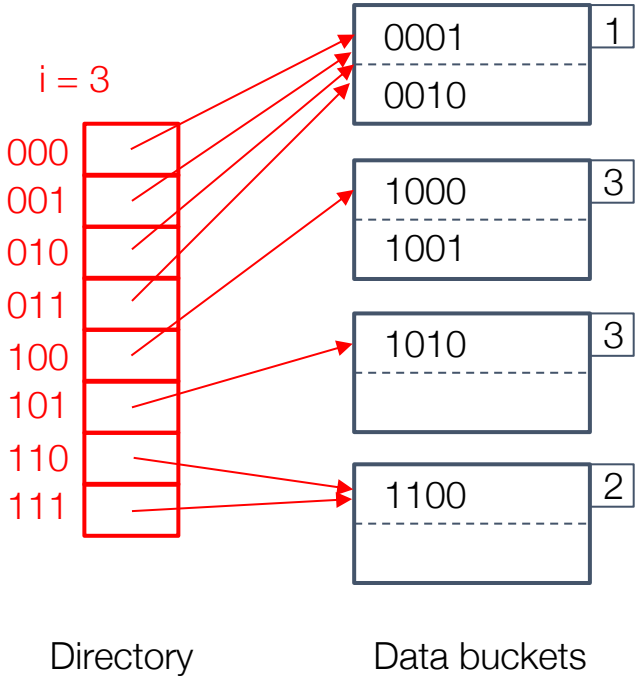
In-class Exercise

- Add 1000



In-class Exercise

- Add 1000



Extensible hashing summary

If bucket array fits in memory, lookup is always 1 disk I/O

Can grow table with little wasted space and avoiding full reorganizations

However, doubling the bucket array is expensive

- Splitting can occur frequently if the number of records per block is small
- At some point, the bucket array may not fit in memory

Linear hashing (covered next) grows the number of buckets more slowly

Linear hashing

How it works at a high level:

- No directory - uses the bucket array directly.
- Splits buckets in round-robin order regardless of which bucket overflowed.
- Maintains a pointer to the "next bucket to split”

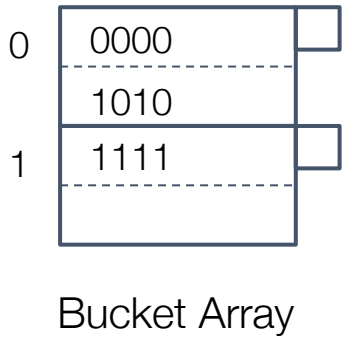
Can use different overflow criterion:

- Space Utilization
- Average Length of Overflow Chains

Linear hash tables

- Use last i bits of hash value to locate block
- Hash table grows linearly

split pointer	p = 0
# bits used	i = 1
# buckets	n = 2
# records	r = 3



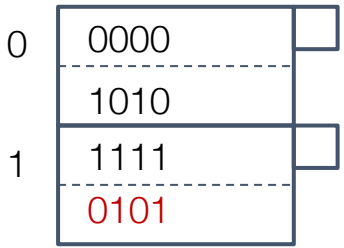
Bucket Array

Policy: $r \leq 1.7n$

Linear hash tables

- Add 0101

split pointer	p = 0
# bits used	i = 1
# buckets	n = 2
# records	r = 4



Policy: r ≤ 1.7n

Violation!

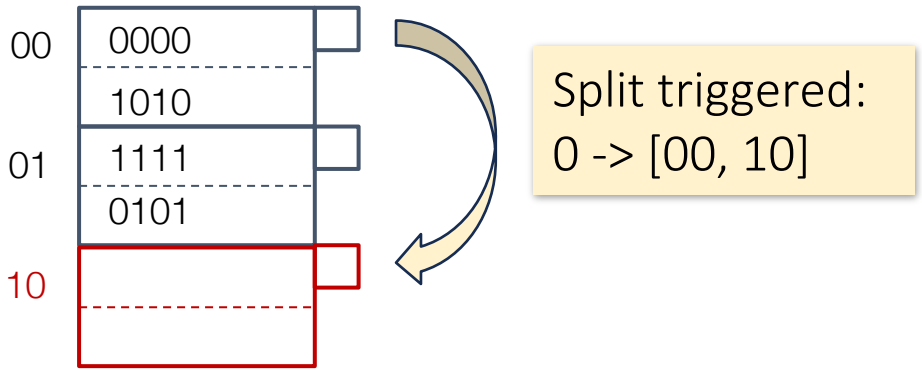
Linear hash tables

- Add 0101

split pointer	p = 0
# bits used	i = 2
# buckets	n = 3
# records	r = 4

Policy: r ≤ 1.7n

Violation!

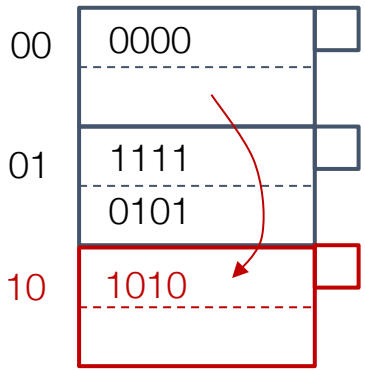


Linear hash tables

- Add 0101

split pointer	p = 0
# bits used	i = 2
# buckets	n = 3
# records	r = 4

Policy: r ≤ 1.7n



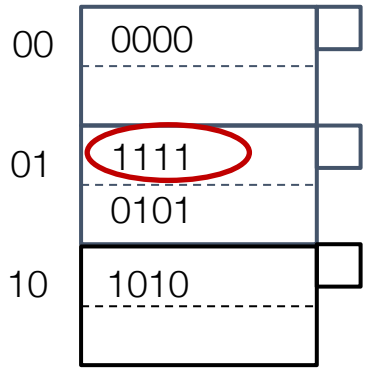
Linear hash tables

- Add 0101

split pointer	p = 1
# bits used	i = 2
# buckets	n = 3
# records	r = 4

Policy: r ≤ 1.7n

Split pointer moves to the next bucket



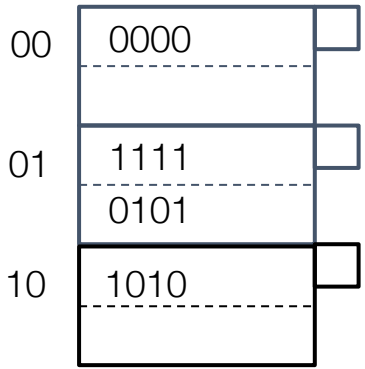
1111 stays here because there is no 11 bucket yet

Linear hash tables

- Add 0001

split pointer	p = 1
# bits used	i = 2
# buckets	n = 3
# records	r = 4

Policy: r ≤ 1.7n



Linear hash tables

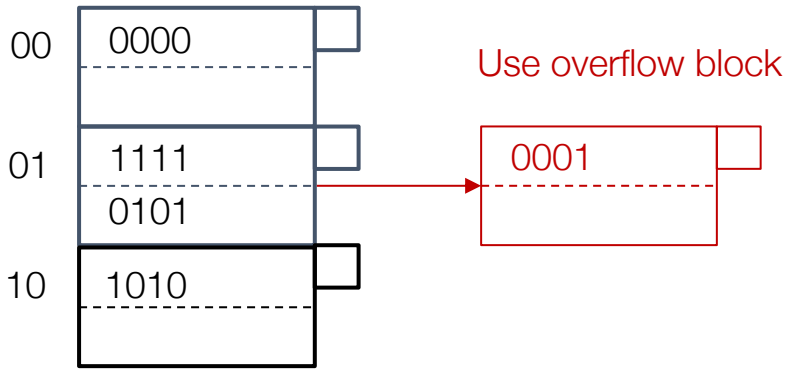
- Add 0001

split pointer	p = 1
# bits used	i = 2
# buckets	n = 3
# records	r = 5

Policy: r ≤ 1.7n

No violation!

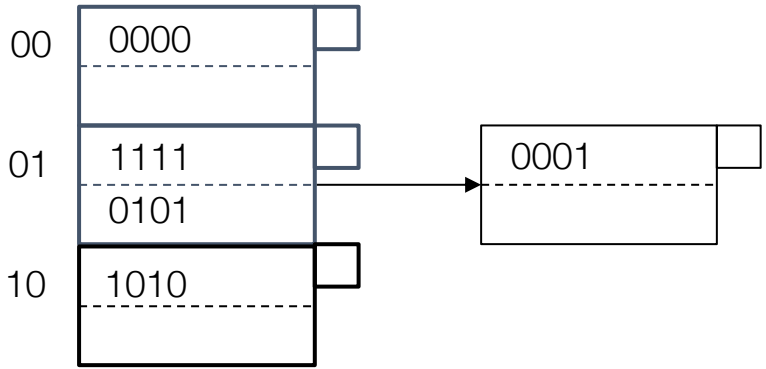
Only add new bucket when policy is violated



In-class Exercise

- Continuing with example, add 0111.
  What happens here?

split pointer	p = 1
# bits used	i = 2
# buckets	n = 3
# records	r = 5



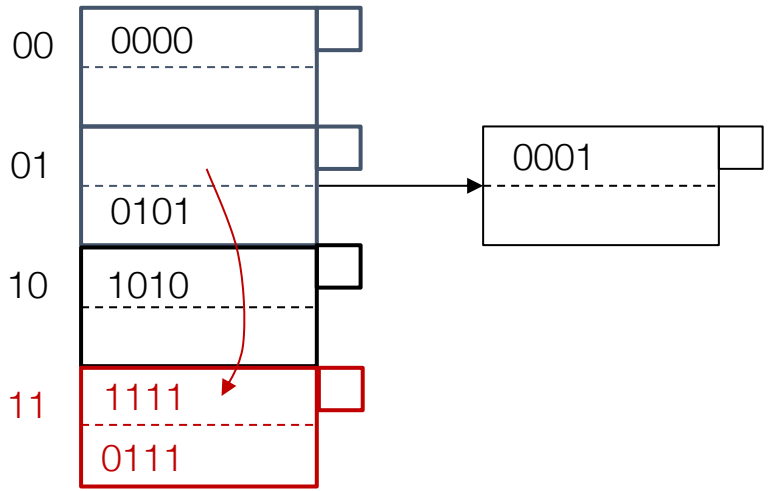
Policy: r ≤ 1.7n

In-class Exercise

- Continuing with example, add 0111.
  What happens here?

split pointer	p = 2
# bits used	i = 2
# buckets	n = 4
# records	r = 6

Policy: r ≤ 1.7n



Linear hashing summary

- Can grow table with little wasted space and avoiding full reorganizations
- Compared to extensible hashing, there is no array of buckets
- However, there can be a long chain of overflow blocks