Hashing

Contents

- Static Hashing
  - File Organization
  - Properties of the Hash Function
  - Bucket Overflow
  - Indices
- Dynamic Hashing
  - Underlying Data Structure
  - Querying and Updating
- Comparisons
  - Other types of hashing
  - Ordered Indexing vs. Hashing

Static Hashing

- Hashing provides a means for accessing data without the use of an index structure.
- Data is addressed on disk by computing a function on a search key instead.

Organization

- A **bucket** in a hash file is unit of storage (typically a disk block) that can hold one or more records.
- The **hash function**, h, is a function from the set of all search-keys, K, to the set of all bucket addresses, B.
- Insertion, deletion, and lookup are done in constant time.

Querying and Updates

- To insert a record into the structure compute the hash value $h(K_i)$, and place the record in the bucket address returned.
- For lookup operations, compute the hash value as above and search each record in the bucket for the specific record.
- To delete simply lookup and remove.

Properties of the Hash Function

- The distribution should be uniform.
  - An ideal hash function should assign the same number of records in each bucket.
- The distribution should be random.
  - Regardless of the actual search-keys, the each bucket has the same number of records on average
  - Hash values should not depend on any ordering or the search-keys

Bucket Overflow

- How does bucket overflow occur?
  - Not enough buckets to handle data
  - A few buckets have considerably more records then others. This is referred to as skew.
    - Multiple records have the same hash value
    - Non-uniform hash function distribution.

Solutions

- Provide more buckets then are needed.
- Overflow chaining
  - If a bucket is full, link another bucket to it. Repeat as necessary.
  - The system must then check overflow buckets for querying and updates. This is known as **closed hashing**.

Alternatives

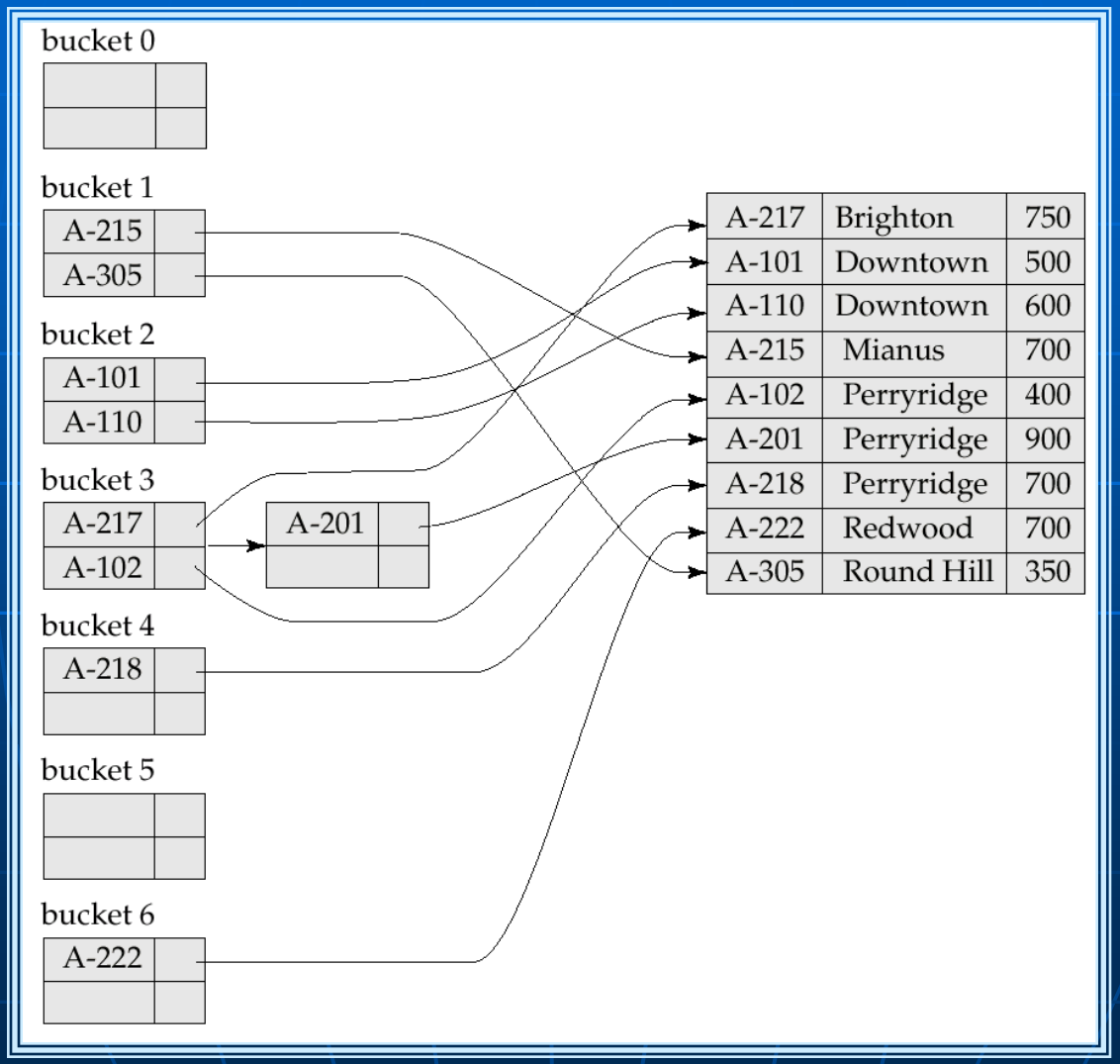
- Open hashing
  - The number of buckets is fixed
  - Overflow is handled by using the next bucket in cyclic order that has space.
    - This is known as **linear probing**.
- Compute more hash functions.

Note: Closed hashing is preferred in database systems.

Indices

- A **hash index** organizes the search keys, with their pointers, into a hash file.
- Hash indices never primary even though they provide direct access.

Example of Hash Index



Dynamic Hashing

- More effective then static hashing when the database grows or shrinks
- **Extendable hashing** splits and coalesces buckets appropriately with the database size.
  - i.e. buckets are added and deleted on demand.

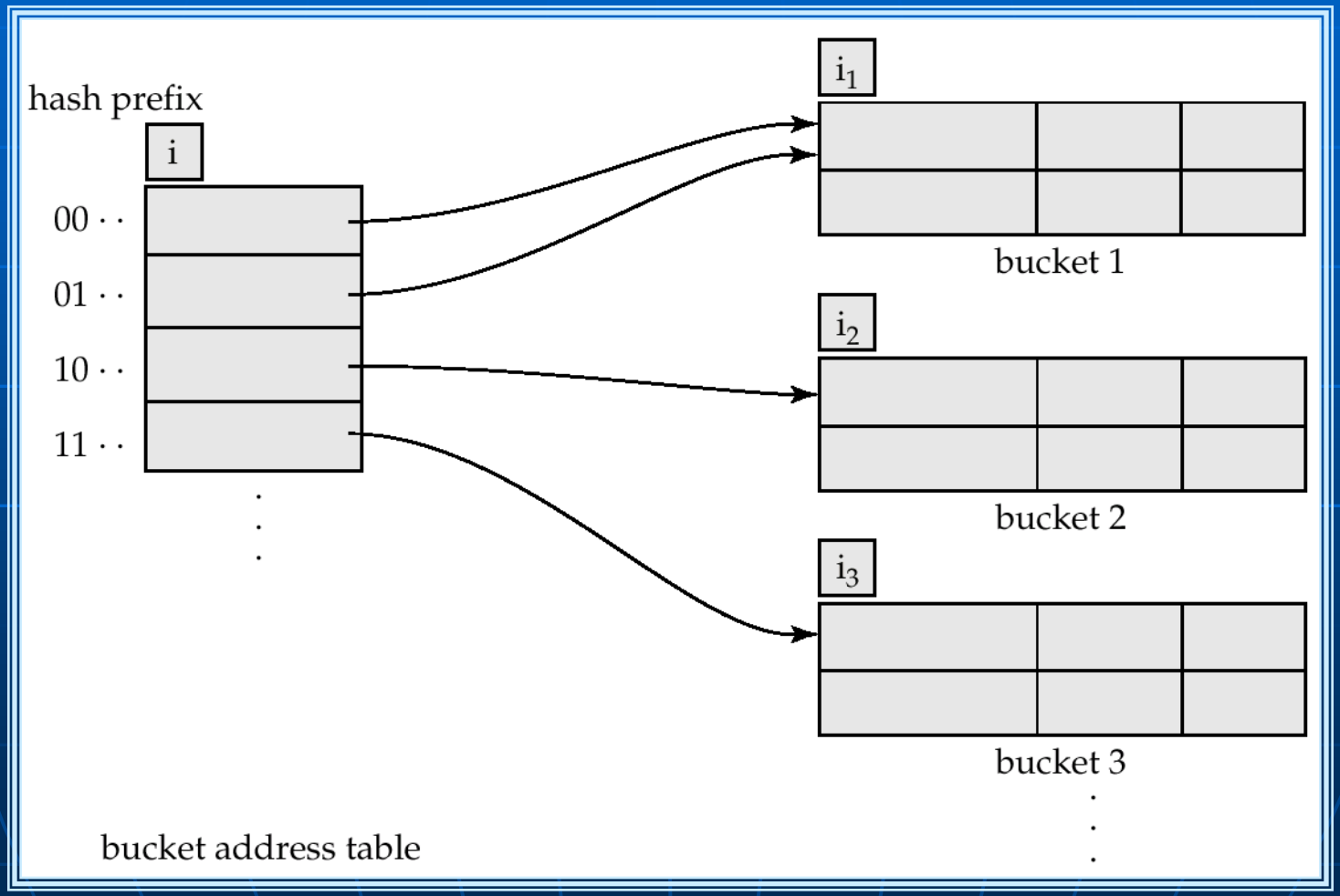
The Hash Function

- Typically produces a large number of values, uniformly and randomly.
- Only part of the value is used depending on the size of the database.

Data Structure

- Hash indices are typically a prefix of the entire hash value.
- More then one consecutive index can point to the same bucket.
  - The indices have the same hash prefix which can be shorter then the length of the index.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

Queries and Updates

- Lookup
  - Take the first i bits of the hash value.
  - Following the corresponding entry in the bucket address table.
  - Look in the bucket.

Queries and Updates (Cont'd)

- Insertion
  - Follow lookup procedure
  - If the bucket has space, add the record.
  - If not…

Insertion (Cont'd)

- Case 1: $i = i_j$
  - Use an additional bit in the hash value
    - This doubles the size of the bucket address table.
    - Makes two entries in the table point to the full bucket.
  - Allocate a new bucket, z.
    - Set i_j and i_z to i
    - Point the second entry to the new bucket
    - Rehash the old bucket
  - Repeat insertion attempt

Insertion (Cont'd)

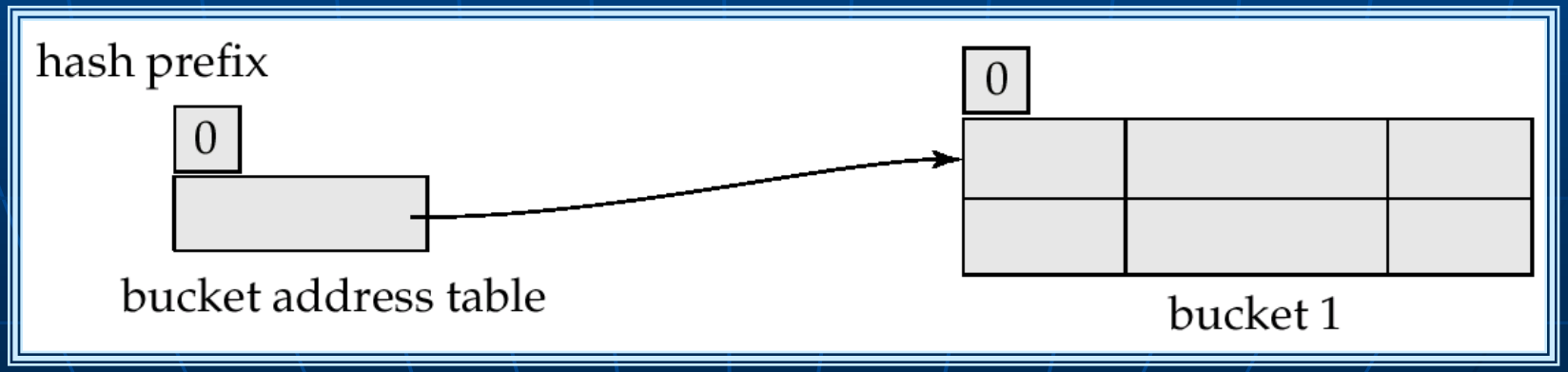
- Case 2: $i > i_j$
  - Allocate a new bucket, z
  - Add 1 to i_j, set i_j and i_z to this new value
  - Put half of the entries in the first bucket and half in the other
  - Rehash records in bucket j
  - Reattempt insertion

Insertion (Finally)

- If all the records in the bucket have the same search value, simply use overflow buckets as seen in static hashing.

Use of Extendable Hash Structure: Example

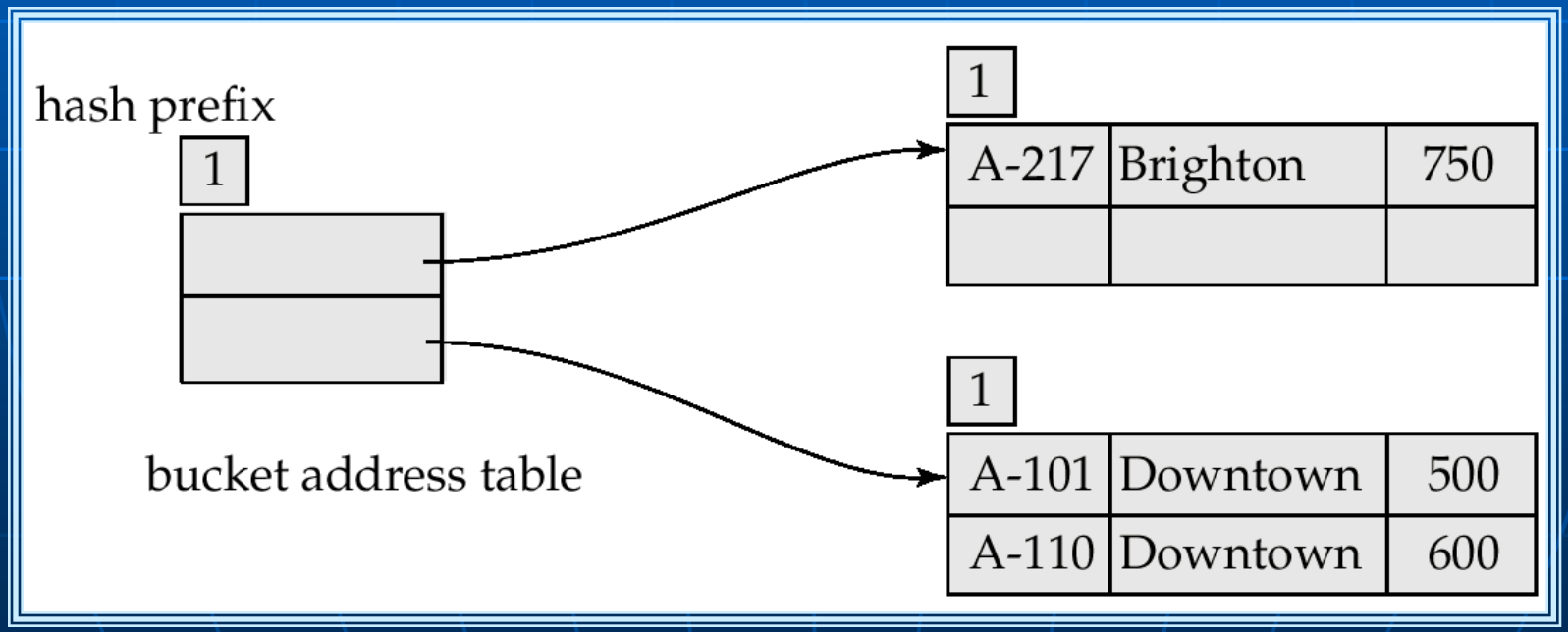
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Initial Hash structure, bucket size = 2

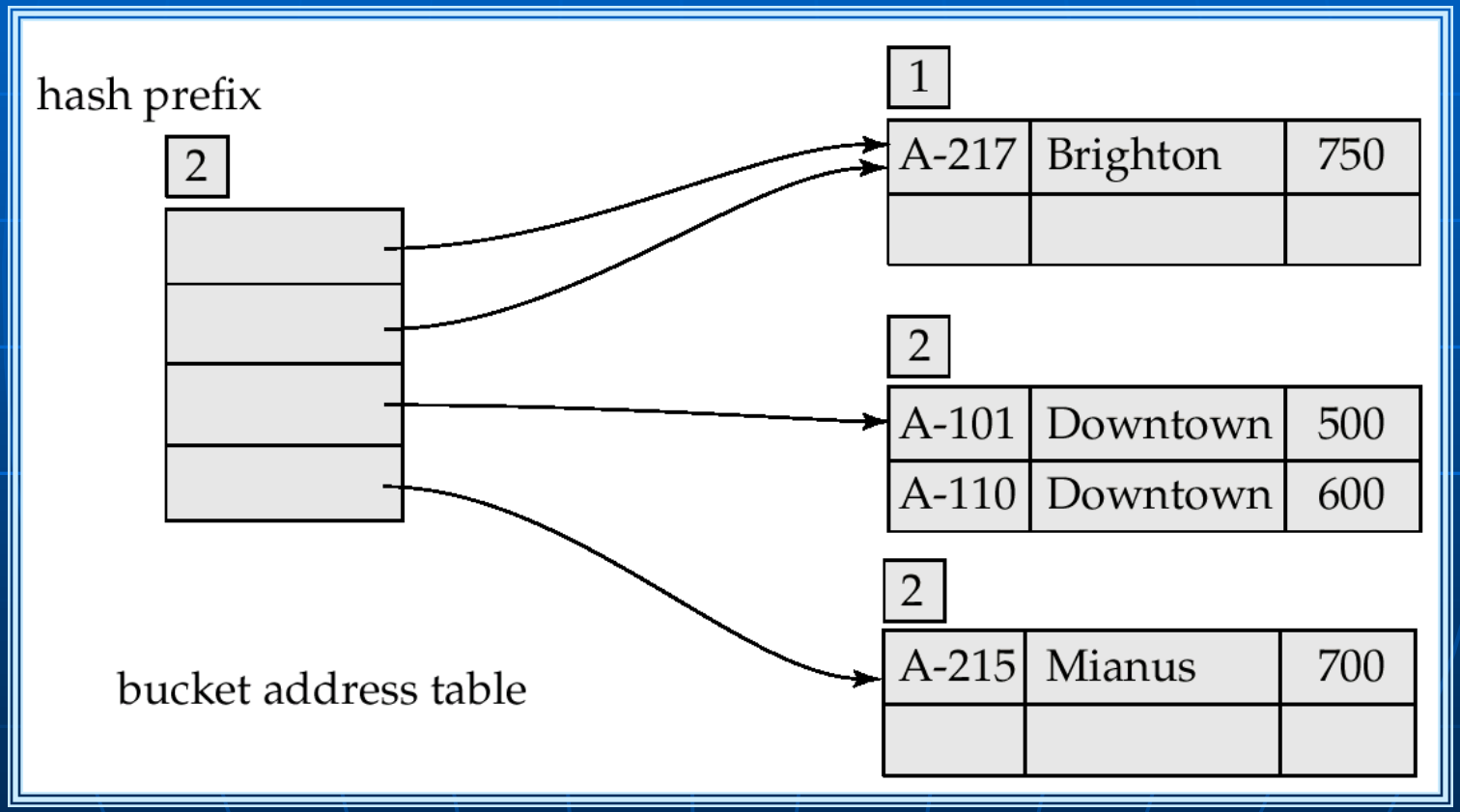
Example (Cont.)

- Hash structure after insertion of one Brighton and two Downtown records

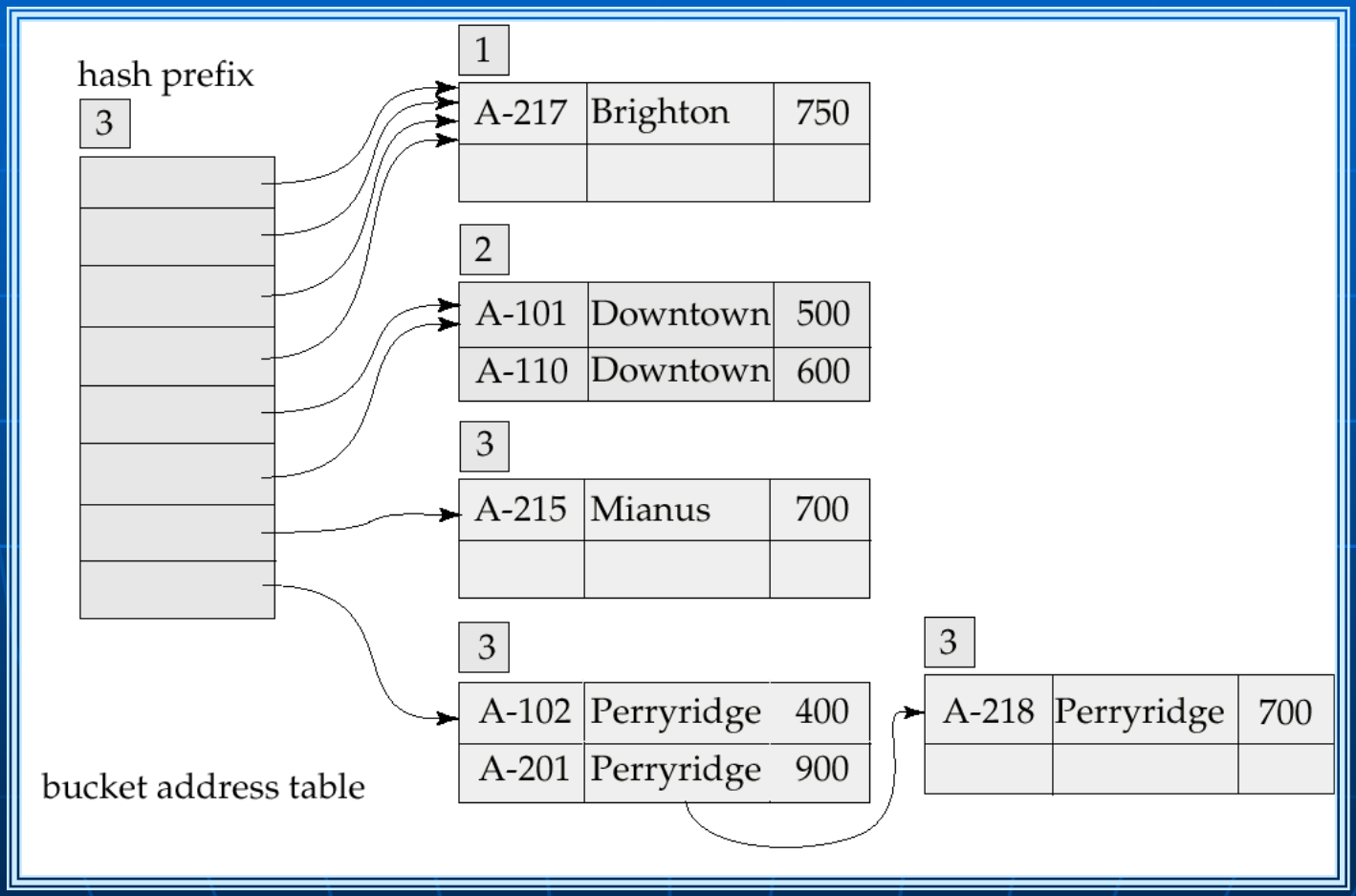


Example (Cont.)

Hash structure after insertion of Mianus record



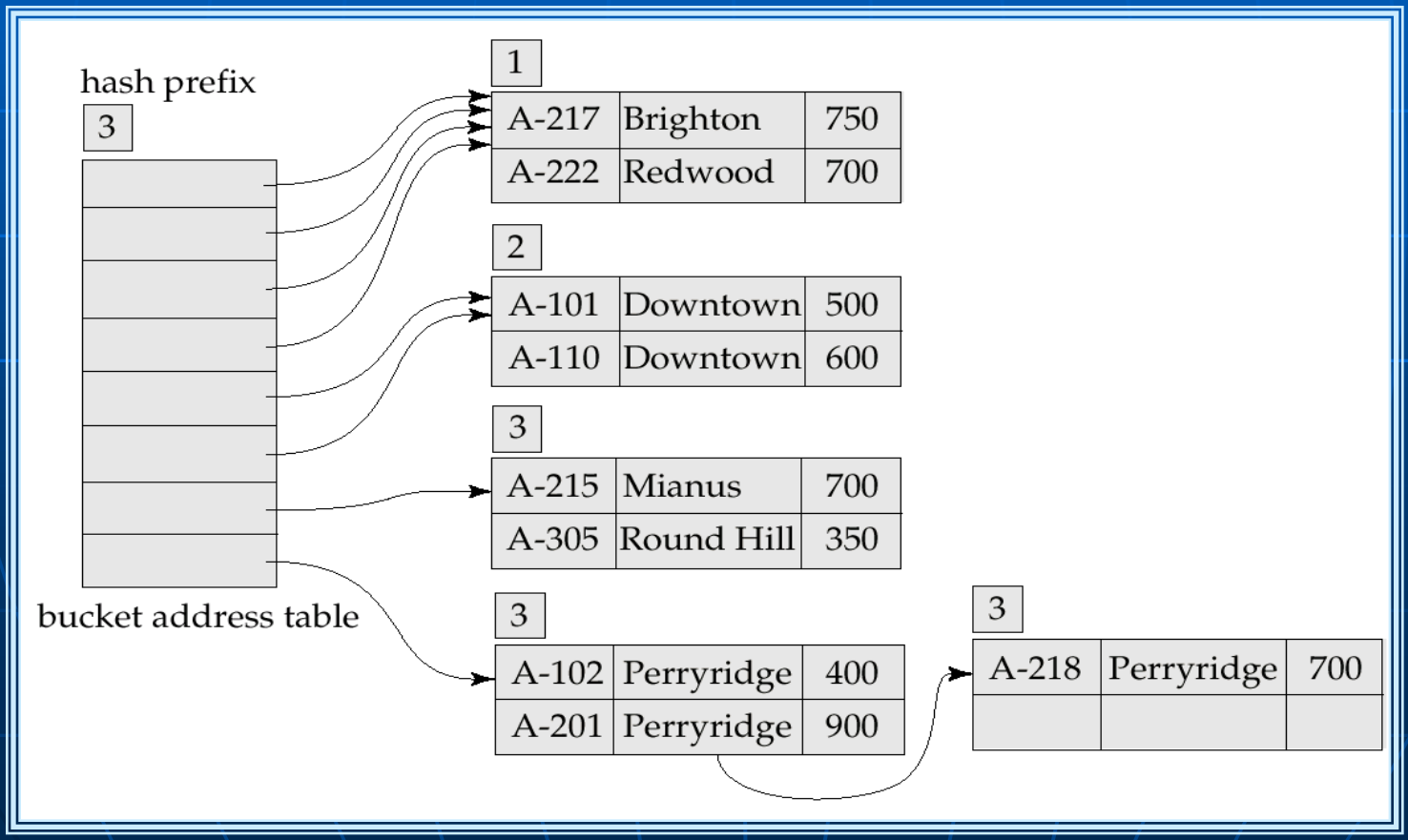
Example (Cont.)



Hash structure after insertion of three Perryridge records

Example (Cont.)

- Hash structure after insertion of Redwood and Round Hill records



Comparison to Other Hashing Methods

- Advantage: performance does not decrease as the database size increases
  - Space is conserved by adding and removing as necessary
- Disadvantage: additional level of indirection for operations
  - Complex implementation

Ordered Indexing vs. Hashing

- Hashing is less efficient if queries to the database include ranges as opposed to specific values.
- In cases where ranges are infrequent hashing provides faster insertion, deletion, and lookup then ordered indexing.