COP 4710: Database Systems
Spring 2006

CHAPTER 10 – INDEXING – Part 2

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CSB 242, 823-2790
http://www.cs.ucf.edu/courses/cop4710/spr2006

School of Electrical Engineering and Computer Science
University of Central Florida

Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

Hash file organization of *account* file, using *branch-name* as key (See figure in next slide.)

- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i*.
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3

Example of Hash File Organization

Hash file organization of *account* file, using *branch-name* as key

(see previous slide for details).

bucket 0

bucket 1

bucket 2

bucket 3

A-217	Brighton	750
A-305	Round Hill	350

bucket 4

A-222	Redwood	700

bucket 5

A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700

bucket 6

bucket 7

A-215	Mianus	700

bucket 8

A-101	Downtown	500
A-110	Downtown	600

bucket 9

Hash Functions

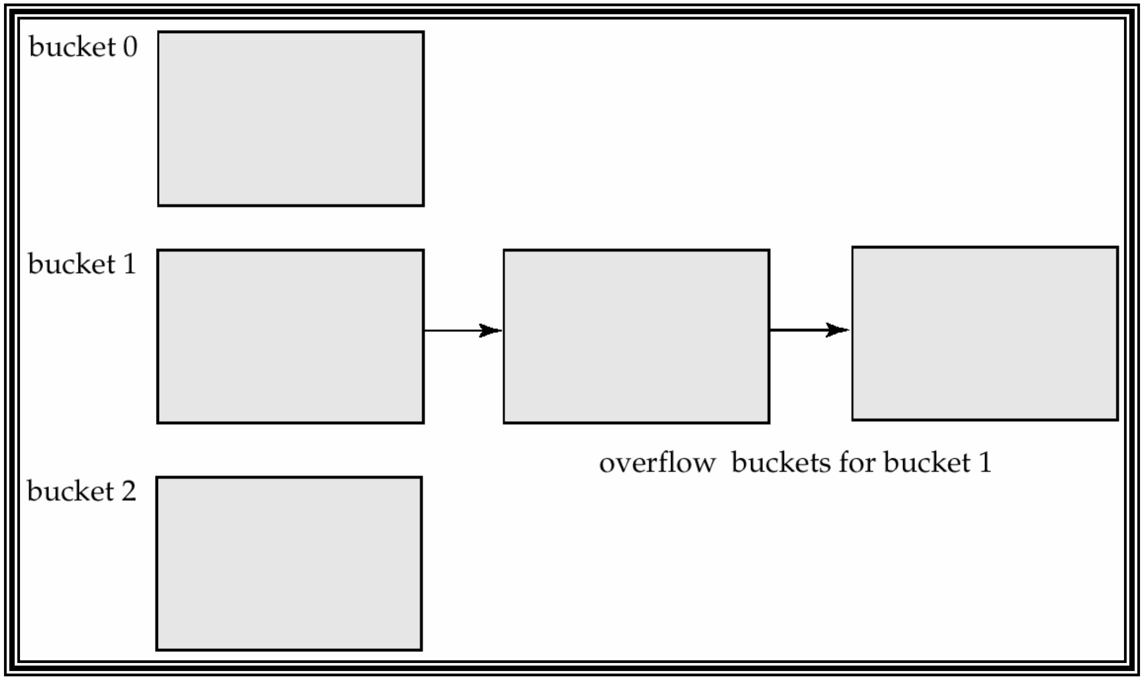
- Worst has function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*.

Handling of Bucket Overflows (cont.)

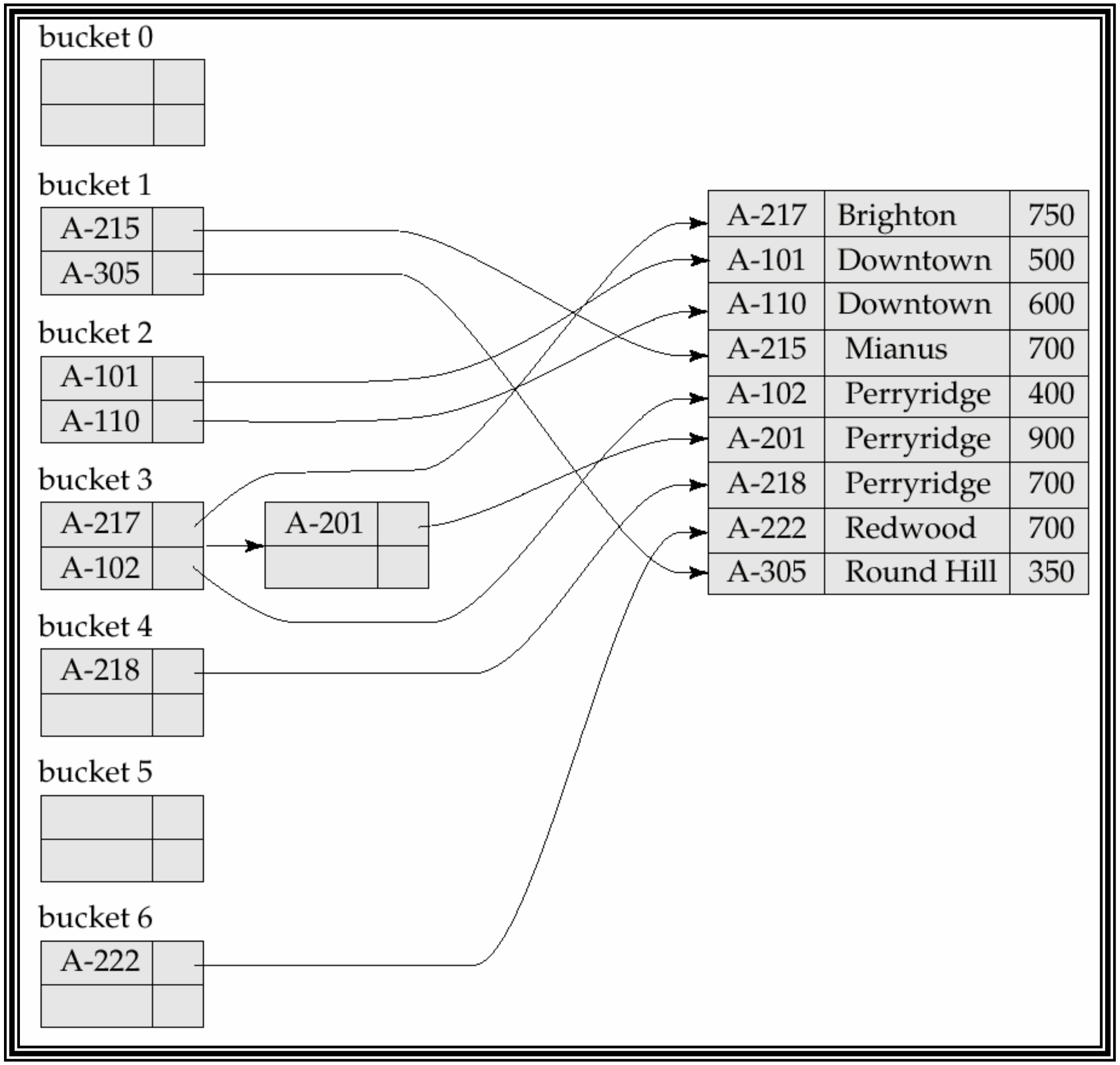
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called closed hashing**.**
  – An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index



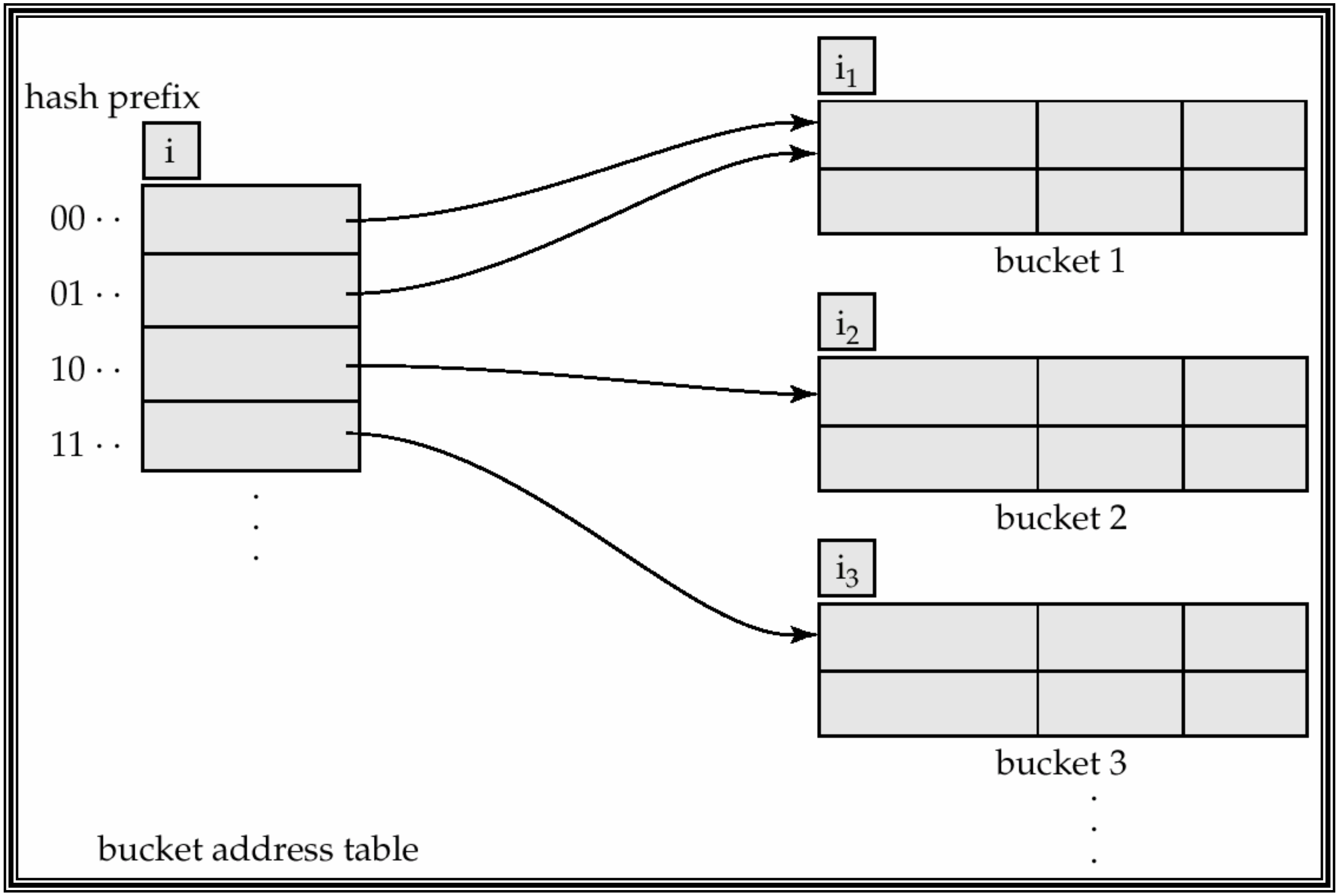
Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically b-bit integers, with $b = 32$.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be i bits, $0 \leq i \leq 32$.
  - Bucket address table size = $2^{i.}$ Initially $i = 0$
  - Value of i grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket.
  - Thus, actual number of buckets is $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Use of Extendable Hash Structure

- Each bucket j stores a value i_j; all the entries that point to the same bucket have the same values on the first i_j bits.
- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say j.
  - If there is room in the bucket j insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)
    - Overflow buckets used instead in some cases

Updates in Extendable Hash Structure

To split a bucket *j* when inserting record with search-key value K_j:

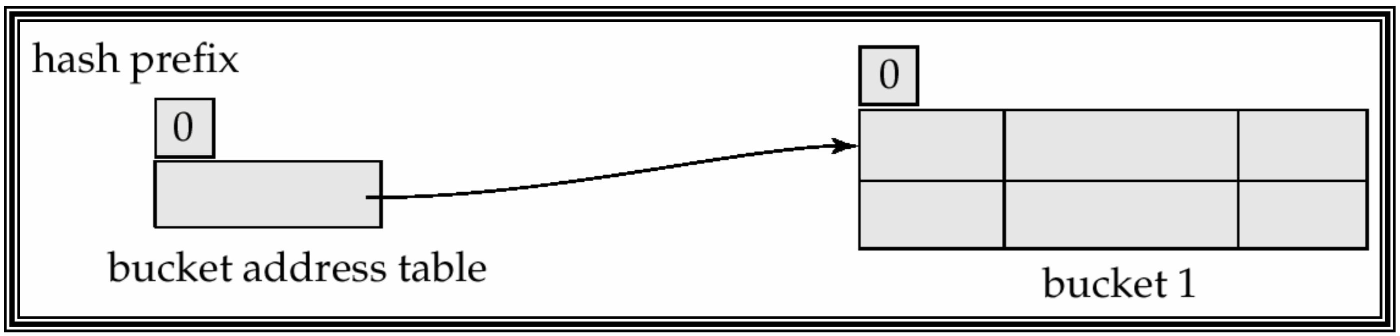
- If $i > i_j$ (more than one pointer to bucket *j*)
  - allocate a new bucket *z*, and set i_j and i_z to the old i_j -+ 1.
  - make the second half of the bucket address table entries pointing to *j* to point to *z*
  - remove and reinsert each record in bucket *j*.
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket *j*)
  - increment *i* and double the size of the bucket address table.
  - replace each entry in the table by two entries that point to the same bucket.
  - recompute new bucket address table entry for K_j
    Now $i > i_j$ so use the first case above.

Updates in Extendable Hash Structure
(cont.)

- When inserting a value, if the bucket is full after several splits (that is, *i* reaches some limit *b*) create an overflow bucket instead of splitting bucket entry table further.
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Use of Extendable Hash Structure: Example

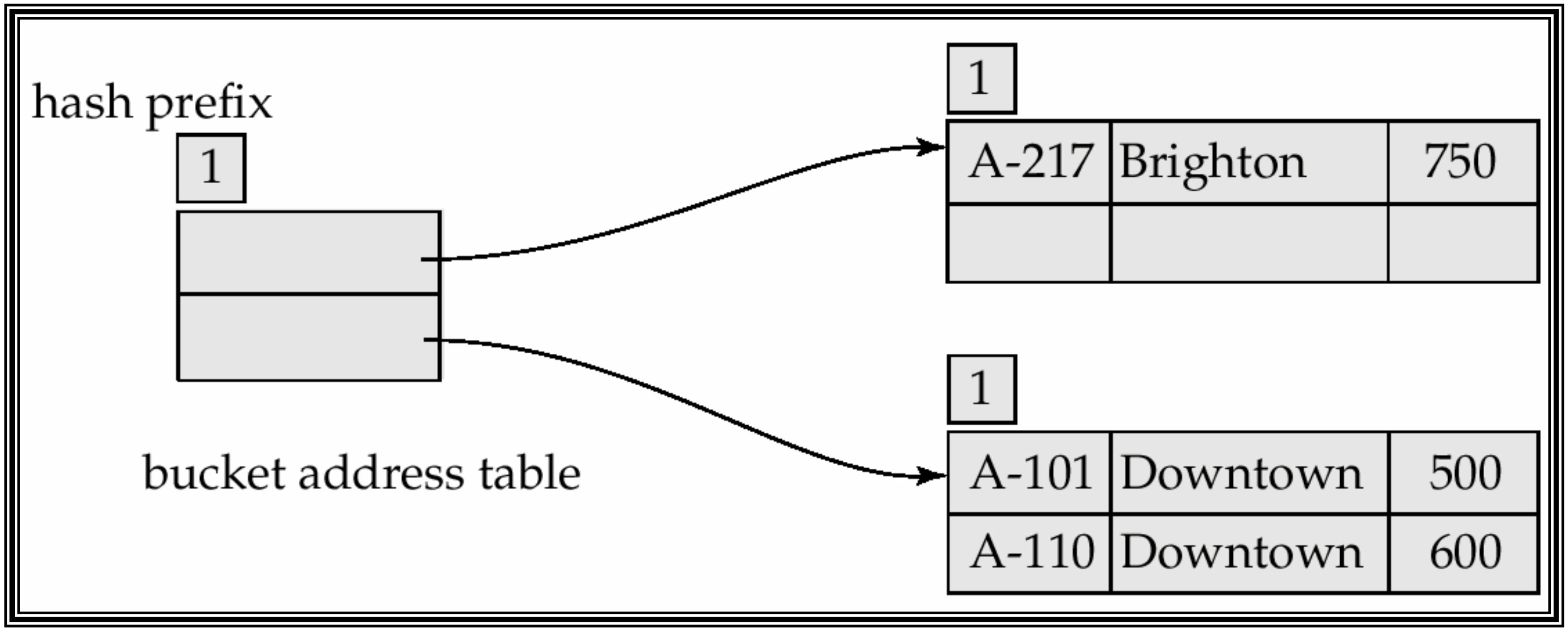
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



Initial Hash structure, bucket size = 2

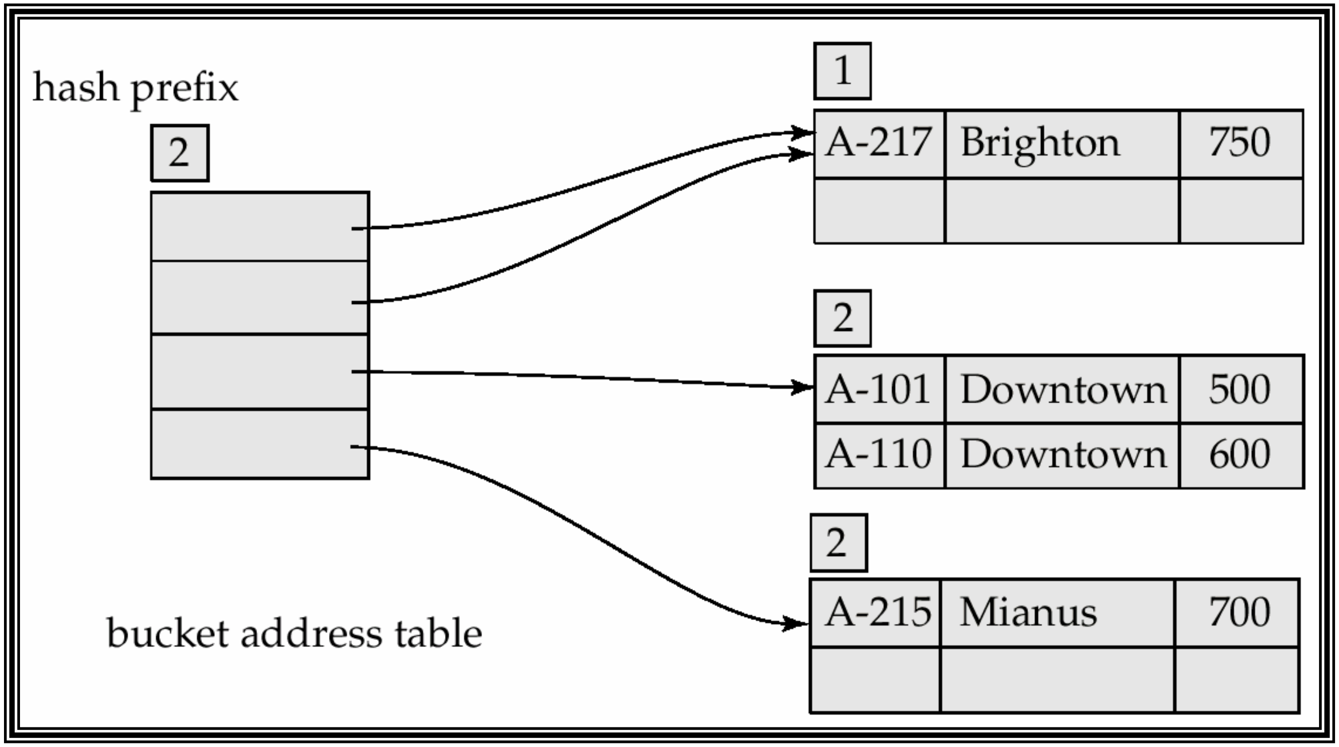
Example (cont.)

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

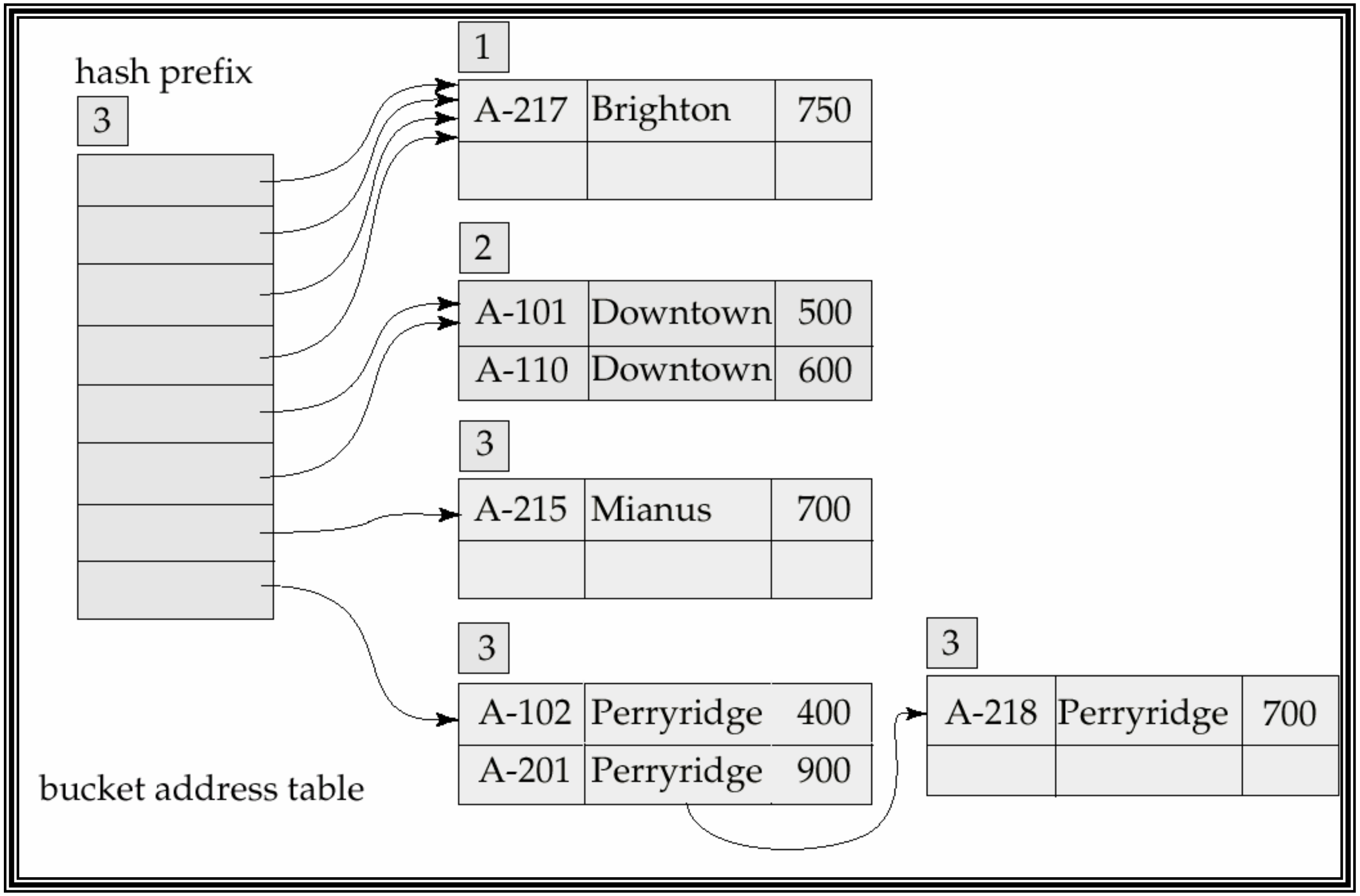


Example (cont.)

Hash structure after insertion of Mianus record



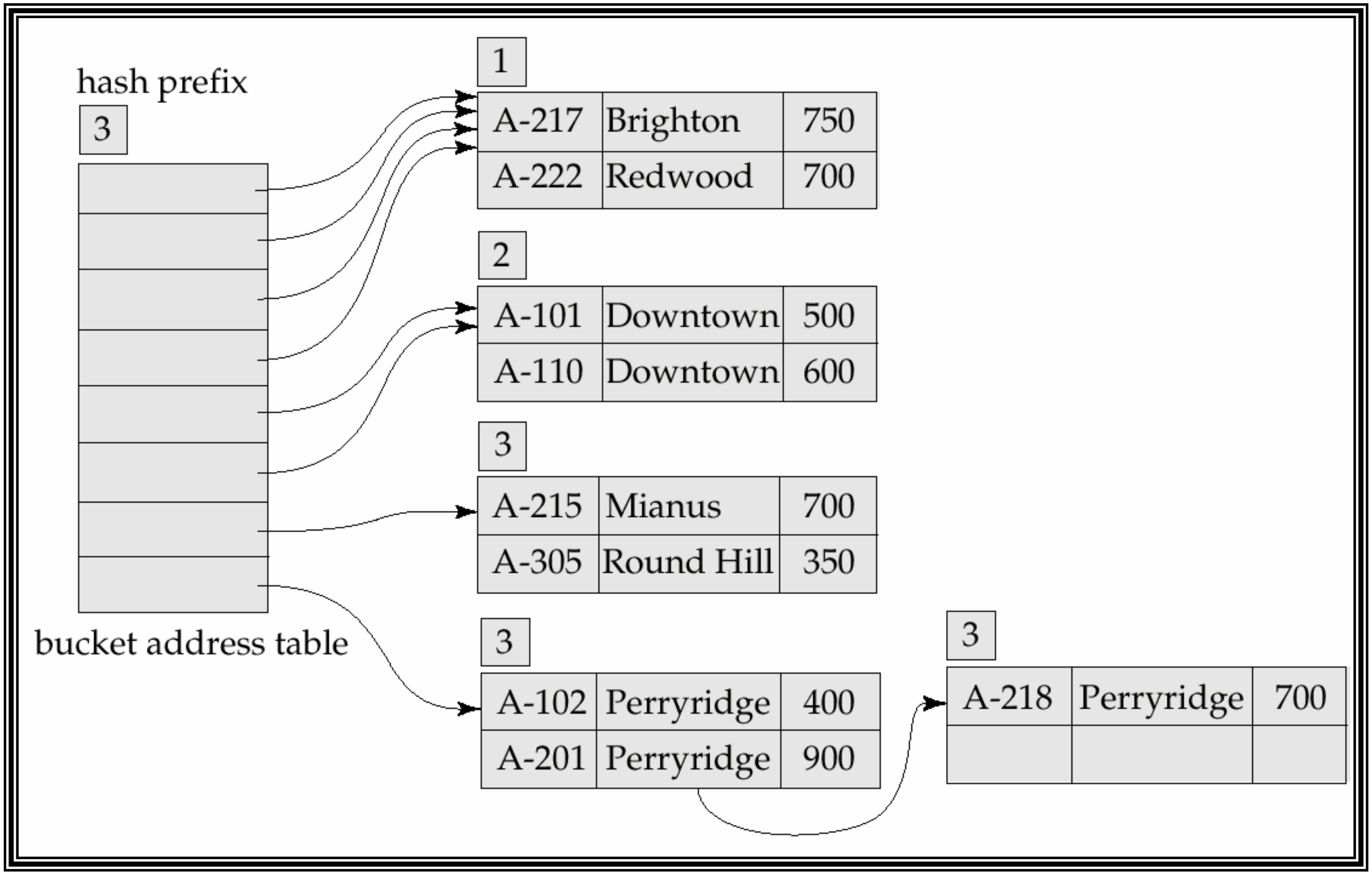
Example (cont.)



Hash structure after insertion of three Perryridge records

Example (cont.)

- Hash structure after insertion of Redwood and Round Hill records



Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Need a tree structure to locate desired record in the structure!
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism which avoids these disadvantages at the possible cost of more bucket overflows

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred

Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:

  select *account-number*
  from *account*
  where *branch-name* = "Perryridge" **and** *balance* = 1000

- Possible strategies for processing query using indices on single attributes:

  1. Use index on *branch-name* to find accounts with balances of $1000; test *branch-name* = "Perryridge".
  2. Use index on *balance* to find accounts with balances of $1000; test *branch-name* = "Perryridge".
  3. Use *branch-name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained.

Indices on Multiple Attributes

- Suppose we have an index on combined search-key (*branch-name*, *balance*).
- With the **where** clause
  where *branch-name* = "Perryridge" **and** *balance* = 1000
  the index on the combined search-key will fetch only records that satisfy both conditions.
  Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  where *branch-name* - "Perryridge" **and** *balance* < 1000
- But cannot efficiently handle
  where *branch-name* < "Perryridge" **and** *balance* = 1000
  May fetch many records that satisfy the first but not the second condition.