



COMP231

File and Index Structure



Outline

- Disks and Files
- Indexes and Databases
- Hash Files



Disks and Files

- DBMS stores information on (“hard”) disks.
 - A disk is a sequence of bytes, each has a *disk address*.
 - **READ:** transfer data from disk to main memory (RAM).
 - **WRITE:** transfer data from RAM to disk.
- Data are stored and retrieved in units called *disk blocks* or *pages*.
 - Each page has a fixed size, say 512 bytes. It contains a sequence of *records*.
 - In many (not always) cases, records in a page have the same size, say 100 bytes.

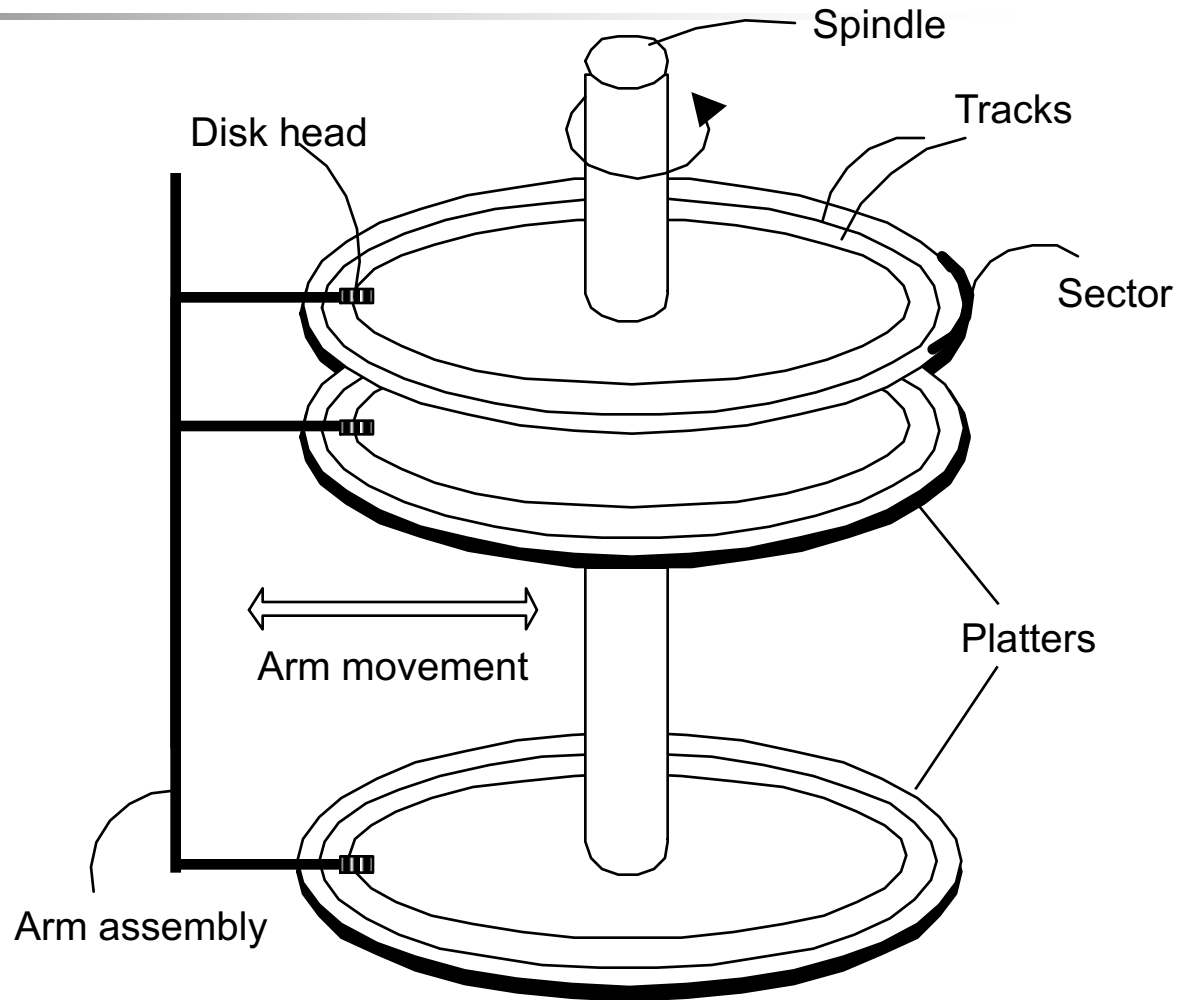


Why Not Store Everything in Main Memory?

- *Costs too much.* RAM is much more expensive than disk.
- *Main memory is volatile.* We want data to be saved between runs.
- Typical storage hierarchy:
 - Main memory (RAM) for currently used data.
 - Disk for the main database (secondary storage).
 - Tapes for archiving older versions of the data (tertiary storage).

Components of a Disk

Block size is a multiple of *sector size* (which is fixed).

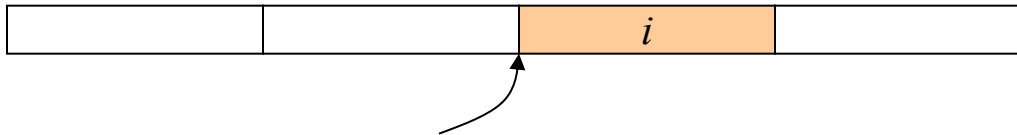




File Organization

- Database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of *fields*.
 - one file \leftrightarrow one table
 - one record \leftrightarrow one tuple
 - a record/tuple has a fixed length
- Easy to implement but limited by the file system

Fixed-Length Records

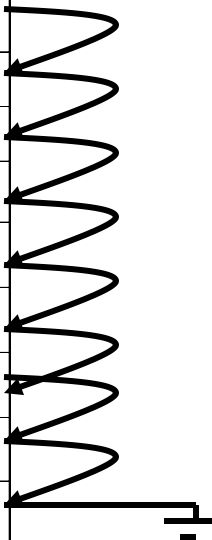


- Simple approach:
 - store record i starting from byte $n \times (i - 1)$, where n is the size of each record.
 - Record access is simple but records may cross disk blocks.
- When record i is deleted, how do you handle the released space?
 - Shifting records: move records $i+1, \dots, n$ to $i, \dots, n-1$
 - move record n to i
 - link all free records on a free list

Sequential File Organization

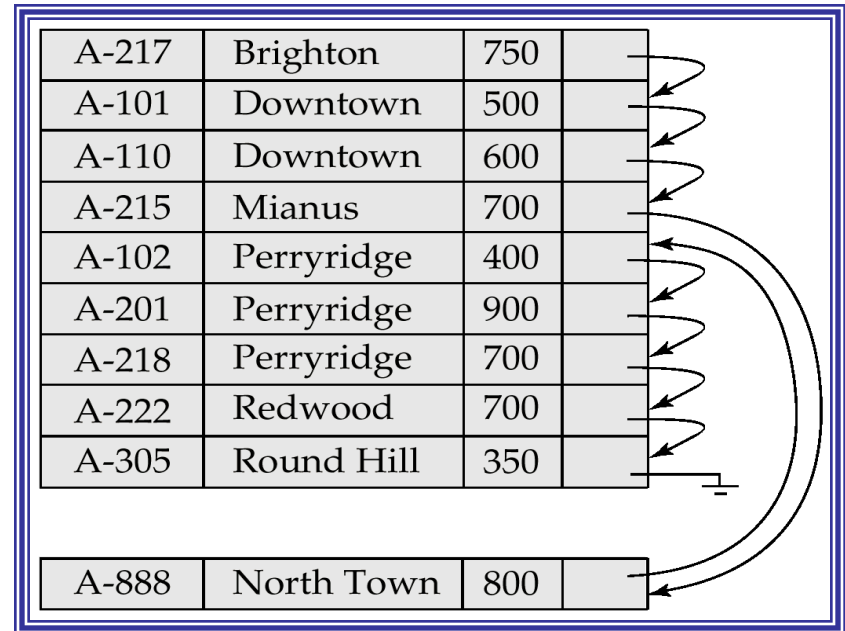
- Suitable for application that require sequential processing of the entire file
- The records in the file are **ordered by a search-key**

Brighton	A-217	750	
Downtown	A-101	500	
Downtown	A-110	600	
Mianus	A-215	700	
Perryridge	A-102	400	
Perryridge	A-201	900	
Perryridge	A-218	700	
Redwood	A-222	700	
Round hill	A-305	350	



Sequential File Organization (cont.)

- **Deletion** – use pointer chains
- **Insertion** – must locate the position in the file where the record is to be record
 - if there is free space insert there
 - if no free space, insert the record in an overflow block
 - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order



Clustering File Organization

- Simple file structure stores each relation in a separate file
- can instead store several relations in one file using a **clustering** file organization
- E.g., clustering organization of customer and depositor.

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stanford
Turner	A-305	

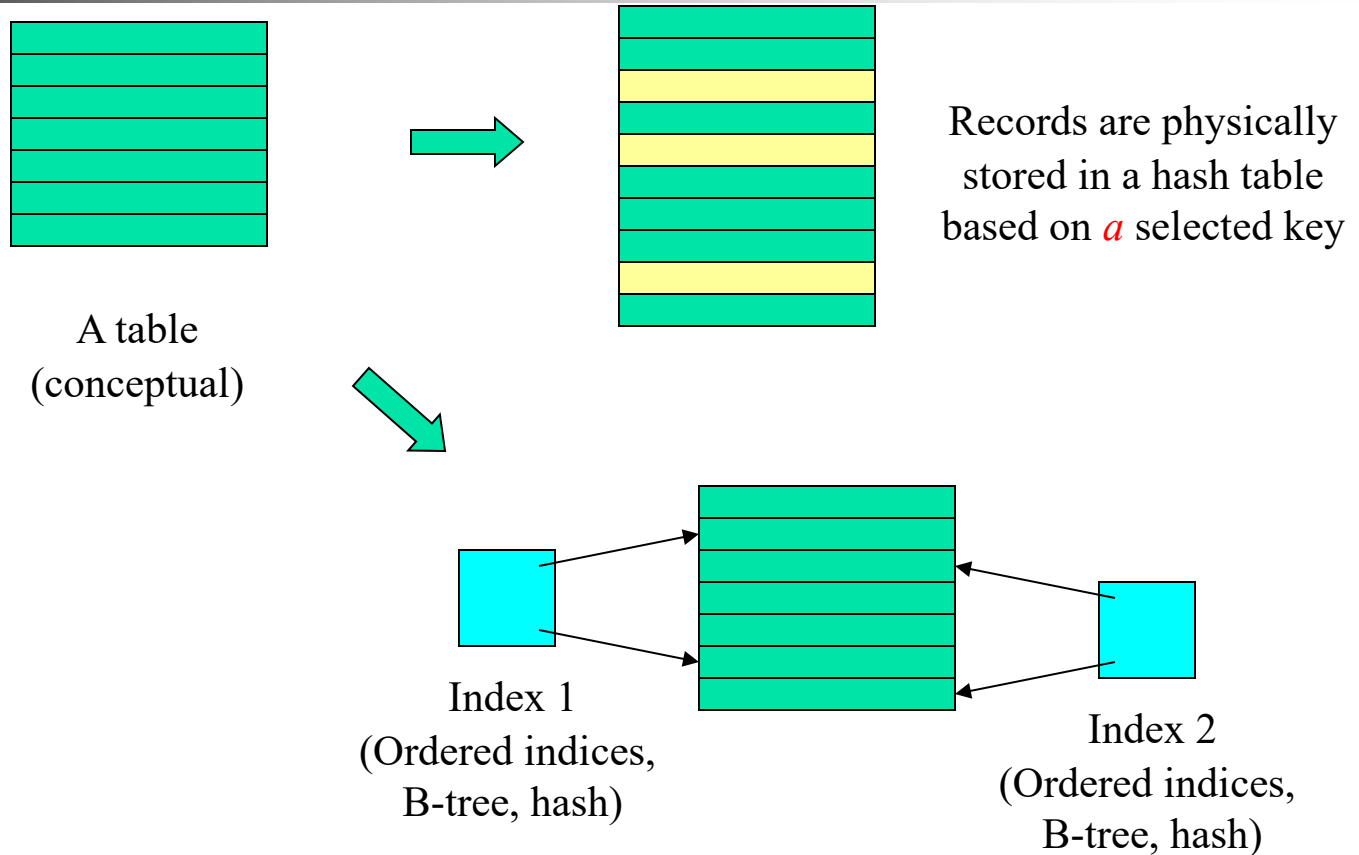
- Good for queries involving depositor \bowtie customer, and for queries involving one single customer and his accounts
- Bad for queries involving only customer
- Result in variable size records



Outline

- Disks and Files
- Indexes and Databases
- Hash Files

Indexes and Databases





Basic Concepts

- Indexing mechanisms **speed up access** to desired data
 - E.g. If you know the call number of a book, you can go directly to the book shelf in the library; otherwise you have to search through all the book shelves
- **Search key** – attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the format:

Search-key	pointer
------------	---------



Basic Concepts

Search-key	pointer
------------	---------

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - **Ordered indices**: search keys are stored in sorted order
 - **Hash indices**: search keys are distributed uniformly across “buckets” using a “hash function”.



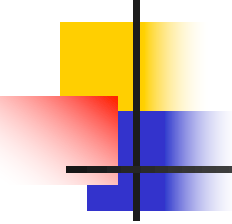
Index Evaluation Metrics

- How are indexing techniques evaluated?
 - What access operations are supported efficiently? E.g.,
 - records with a specified value in an attribute (*equality queries*)
 - or records with an attribute value falling in a specified range of values (*range queries*)
 - Access time
 - Insertion time
 - Deletion time
 - Space overhead
(size of indexes / size of data records)



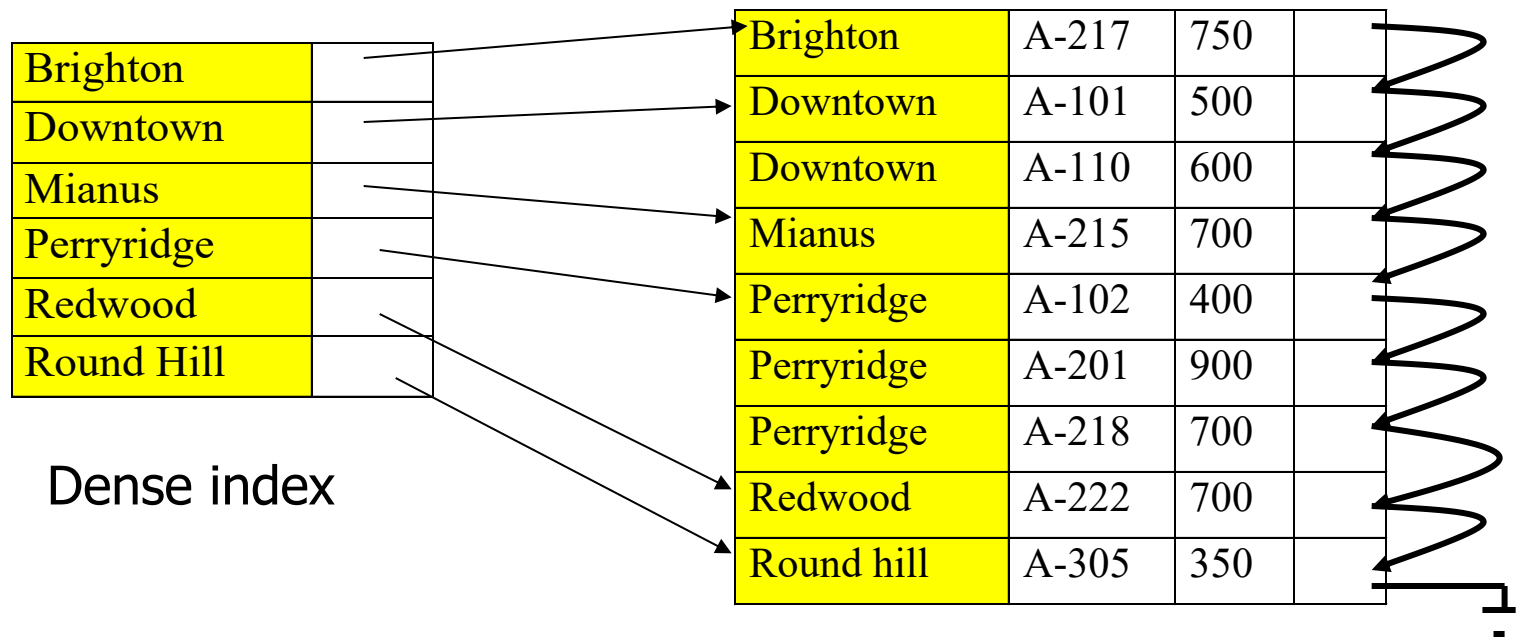
Ordered Indices

- In an **ordered index**, index entries are stored based on the search key value. E.g., author catalog in library.
 - Indexes are mostly ordered indexes except those based on hash files

- 
-
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **clustering index**
 - The search key of a primary index is usually but not necessarily the primary key.
 - **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.

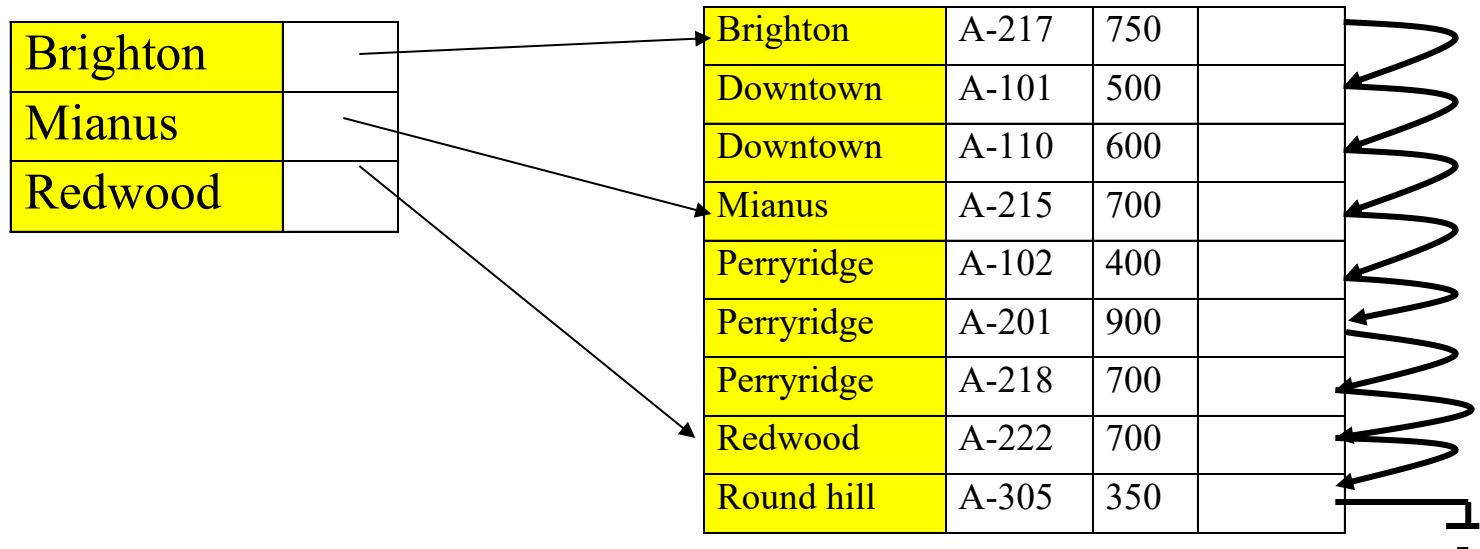
Index Structure Design: Dense Index Files

- *Every search-key value* in the data file is indexed.



Sparse Index Files

- Not all of the search-key values are indexed
 - Adv: reduce index size Disadv: slower



- Records in table are sorted by primary key values
- Search is done in two steps: (i) find the largest possible key (ii) search the record forward



Example of Sparse Index Files

To locate a record with search-key value K we:

- find index record with largest search-key value $\leq K$
- search file sequentially starting at the record to which index record points
- *Less space* and *less maintenance overhead* for insertion and deletions.
- Generally slower than dense index for locating records.
- Good Tradeoff: sparse index with an index entry for *every block* in file.

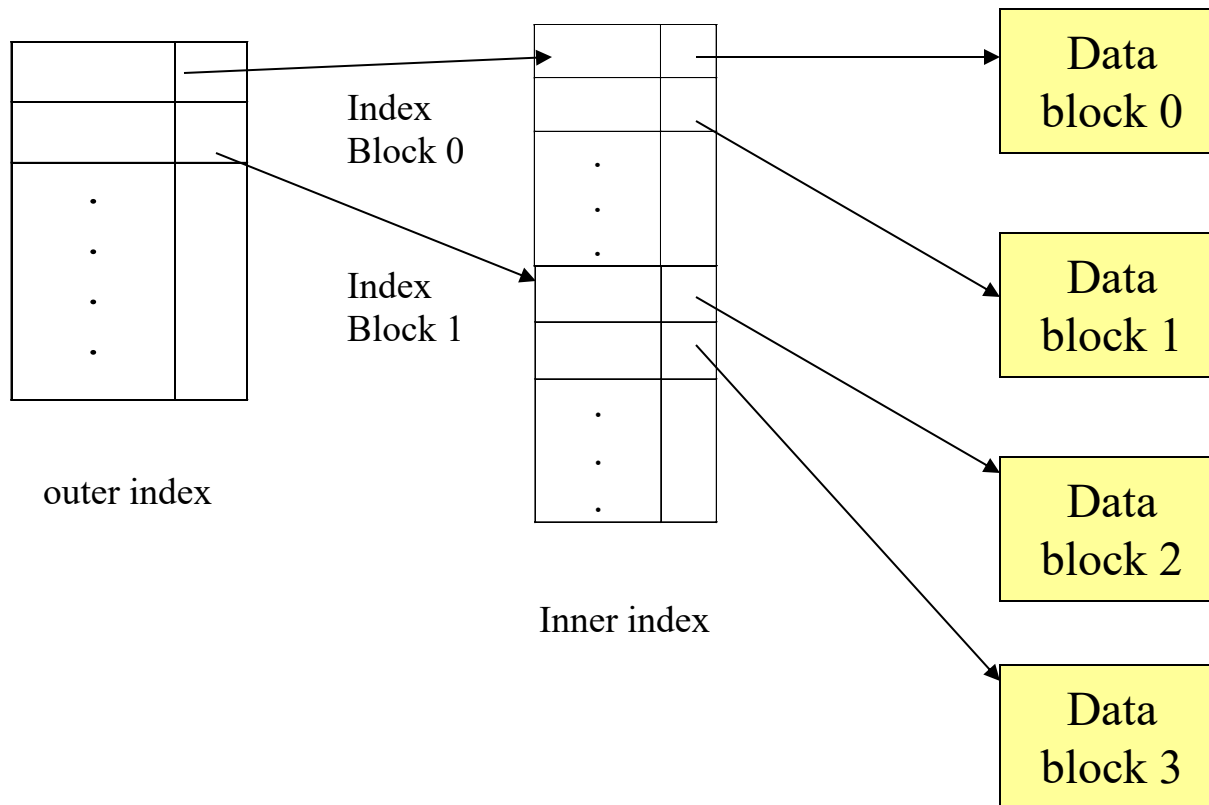
The disk must read a block of data into main memory anyway, so searching within the block cost little.



Multilevel Index

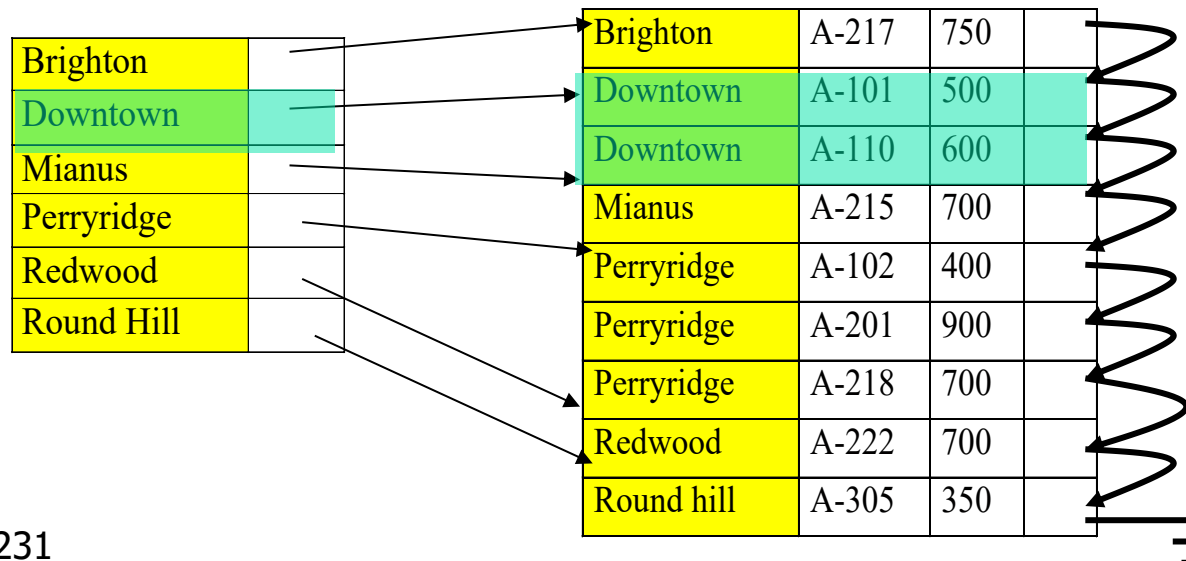
- If primary index does not fit in memory, access becomes expensive. (Why?)
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it.
 - **Outer index** – a sparse index of primary index
 - **Inner index** – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on (**multi-level indices**).
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
 - Dense indices – similar to file record deletion



Index Update: Deletion in Sparse Indices

- Sparse indices** – if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

The diagram illustrates the deletion of an index entry for 'Downtown' and its replacement with 'Brighton'. On the left, a small table shows the current index state. On the right, a larger table shows the file's data. Arrows indicate the mapping from the index to the file and the update process.

Downtown		Brighton	A-217	750
Mianus		Downtown	A-101	500
Redwood		Downtown	A-110	600
		Mianus	A-215	700
		Perryridge	A-102	400
		Perryridge	A-201	900
		Perryridge	A-218	700
		Redwood	A-222	700
		Round hill	A-305	350



Index Update: Insertion

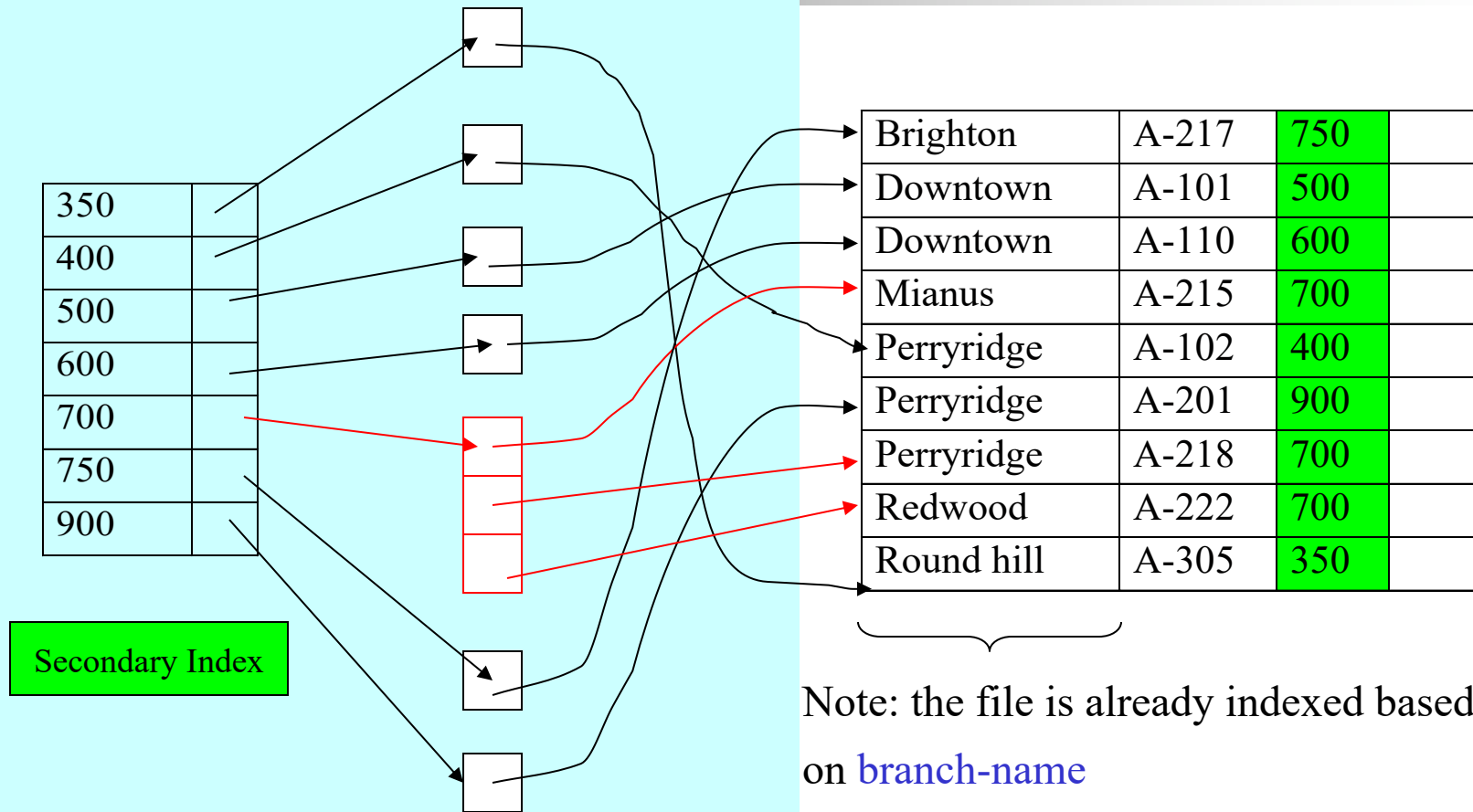
- Single-level index insertion:
 - **Perform a lookup** using the search-key value appearing in the record to be inserted.
 - **Dense indices** - if the search-key value does not appear in the index, insert it.
 - **Sparse indices** - if index stores an entry for each block of the file, no change needs to be made to the index **unless a new block** is created. In this case, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms.



Secondary Indices

- You can organize a file (say, into indexed sequential file) based on one attribute only (e.g., emp#), but it is not adequate in practice
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
 - **Example 1:** if the account database stored sequentially by **account number**, we may want to find all accounts in a particular **branch**.
 - **Example 2:** as above, but where we want to find all accounts with a specified **balance** or range of balances.
- We can have a **secondary index** with an index record for each search-key value: an index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

Secondary Index on Balance Field of Account





Primary and Secondary Indices

- **Secondary indices** have to be **dense** because records are not sorted by the secondary index values
- Indices offer substantial benefits when searching for records, always select some **attributes to index** and re-examine the selection periodically
- When a file is modified, every index on the file must be updated. **Updating indices** imposes **overhead** on database modification
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive (each record access may fetch a new block from disk.)



Outline

- Disks and Files
- Indexes and Databases
- Hash Files



Hash Files

- In previous data structure course, you may learn hash table, which is **main-memory resident**
- A hash method includes a **hash function** and a **collision handling mechanism**
- In main memory, the unit of access is based on byte or word sizes
- In disk-based hash file, the unit of access is based on disk block size (from 512 to 2048 bytes, depending on OS)

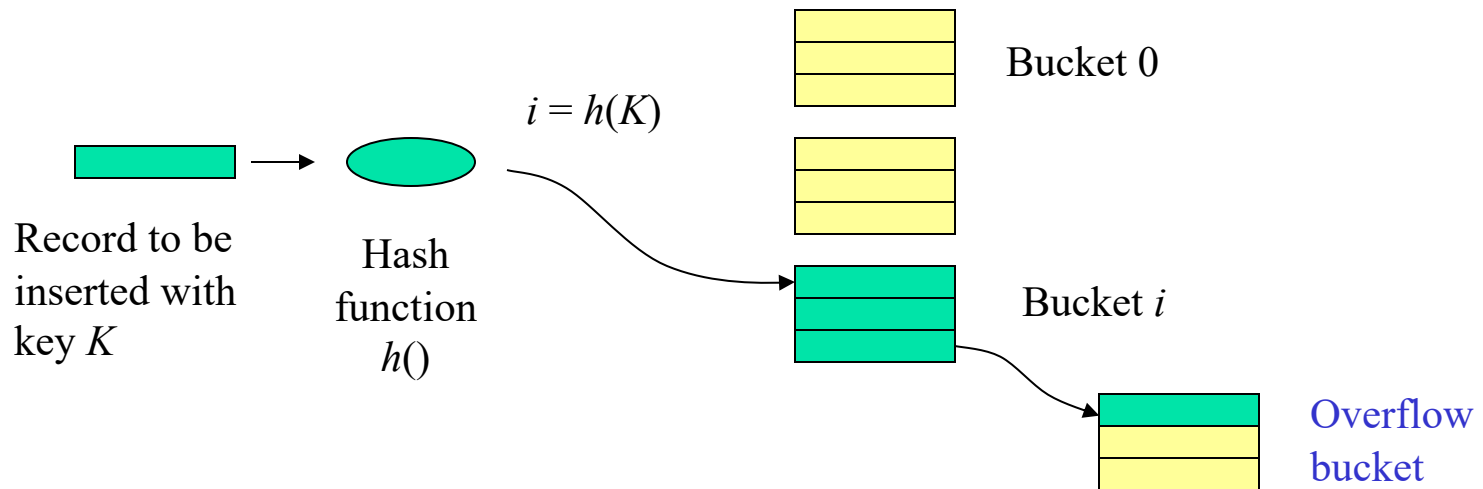


Hash Files

- Static Hashing
- Dynamic Hashing

Static External Hashing

- A hash file consists of M buckets of the same size: bucket₀, bucket₁,... bucket _{$M-1$}



- Collisions occur when a new record hashes to a bucket that is already full. A new bucket is created and chained to the overflowed bucket
 - To **reduce overflow** records, a hash file is typically kept 70-80% full

Static External Hashing – Properties

- The hash function $h()$ should distribute the records **uniformly** among the buckets; otherwise, search time will increase due to many overflow records
 - An ideal hash function will assign roughly the **same number of records to each bucket** *irrespective of the actual distribution of search-key values in the file*
- The **number** of buckets M must be **fixed** when the file is created; not good when the file size changes widely
 - As a design criteria, you need to determine the **load factor**

Static External Hashing – Example



- Hash file organization of **account** file, using **branch-name** as key.
- An example of a hash function defined on a set of characters in a file organization:
 - There are 10 buckets
 - Take the ASCII code of each character as an integer
 - **Sum** up the binary representations of the characters and then applies **modulo** 10 to the sum to get the bucket number



Example of hash File Organization

Bucket 0

Bucket 1

Bucket 2

Bucket 3

Brighton	A-217	750
Round Hill	A-305	350

Bucket 4

Redwood	A-222	700

Bucket 5

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

Bucket 6

Bucket 7

Mianus	A-215	700

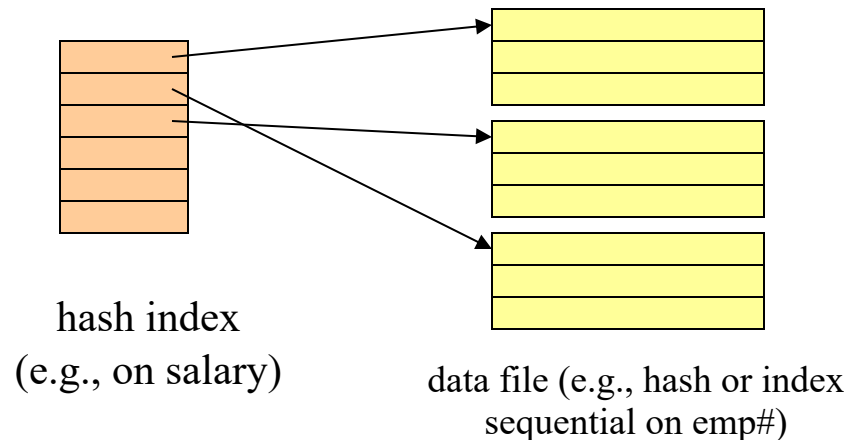
Bucket 8

Downtown	A-101	500
Downtown	A-110	600

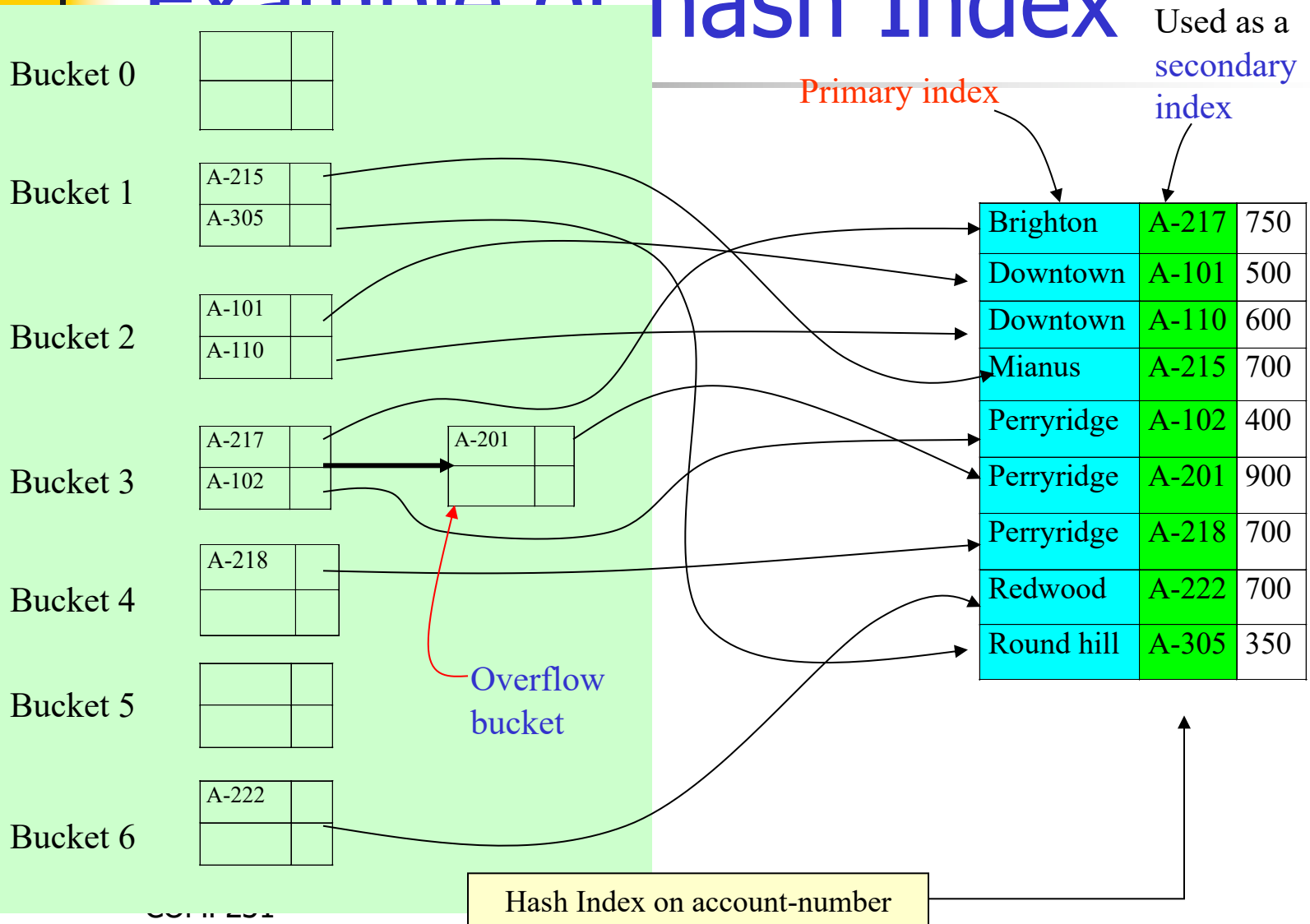
Bucket 9

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.
- Hash indices are **always secondary indices**



Example of hash Index





Hash Files

- Static Hashing
- Dynamic Hashing



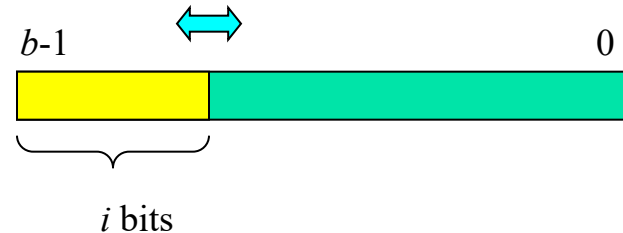
Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
 - When the hash function takes modulo 10 in the previous example, the number of buckets is fixed to 10
 - The trick is how to change the hash function so that **the number of buckets can change**
 - *And at the same time, without the need of rehashing the existing records!*
 - Imagine if you change modulo 10 to modulo 13, then every existing record has to be rehashed – not a good idea

Extendable Hashing

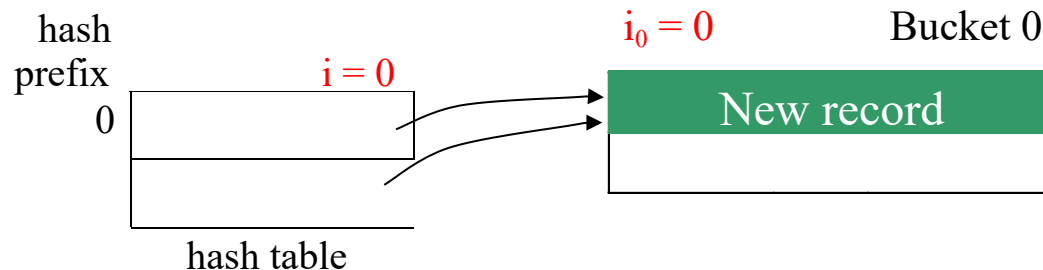
Extendable hashing - one form of dynamic hashing

- Hashing function generates values over a large range - typically b -bit integers, with $b = 32$.



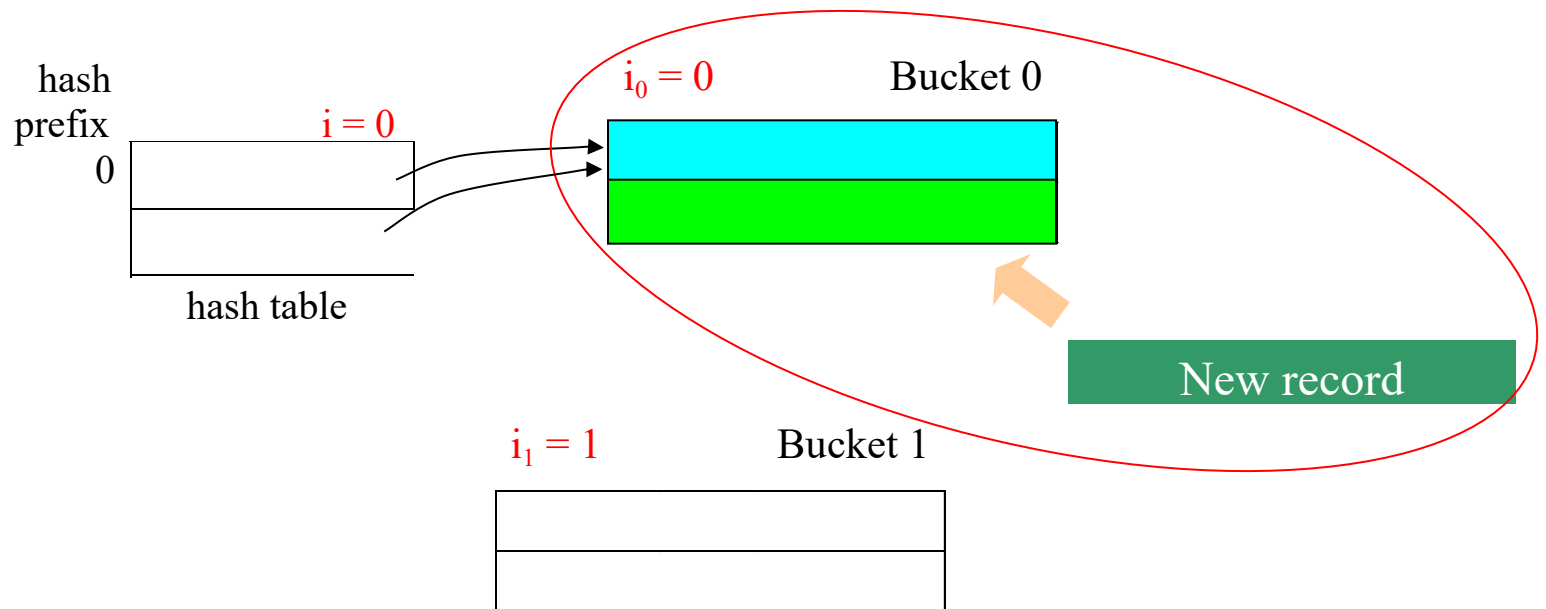
- At any time, use only a prefix of the b -bit integers to index into a table of bucket addresses. Let the length of the prefix be i bits, $0 < i < 32$
- Initially $i = 1$, meaning that it can index at most **2 buckets**
- When the 2 buckets are full, we can use 2 bits ($i = 2$), meaning that we can now index at most **4 buckets**, and so on and so forth....
- i **grows and shrinks** as the size of the database grows and shrinks.
- Actual number of buckets is $< 2^i$, which may change due to bucket merging and splitting

Extendable Hash Structure – General Ideas



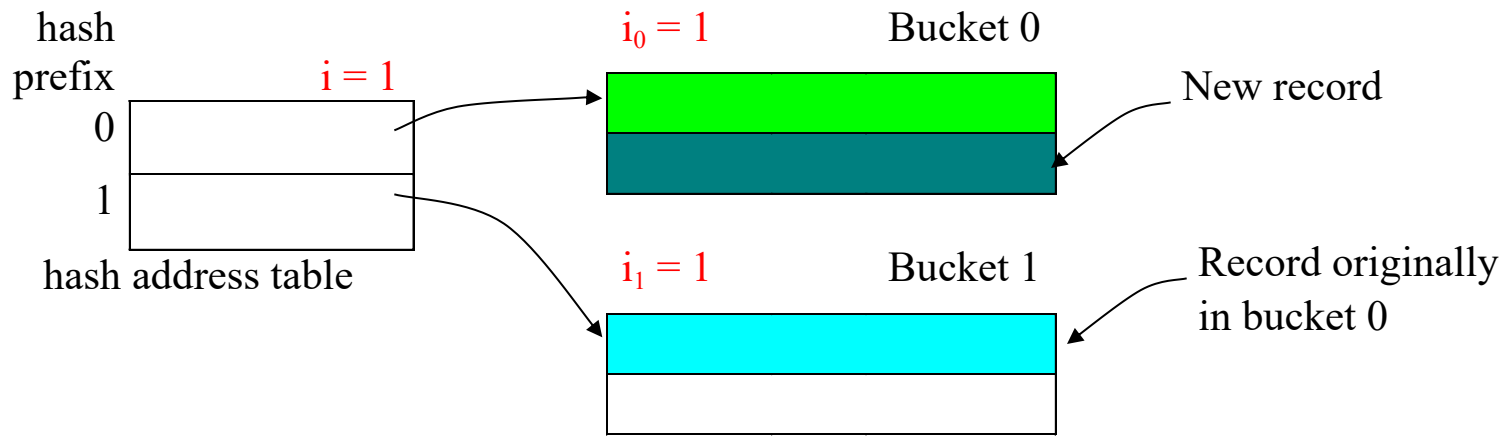
- Initially, $i = 0$, use 0 bit in the hash key, resulting in two entries in the hash address table
- Suppose we start with only 1 or 2 records, we need only 1 bucket initially
- Both entries in the hash address table point to the same bucket
- $i_0 = 0$ means no bit had been used to separate records in the bucket (I.e., records are all hashed into bucket 0 irrespective of the any bit setting in the hash key values)

Extendable Hash Structure – Bucket Expansion



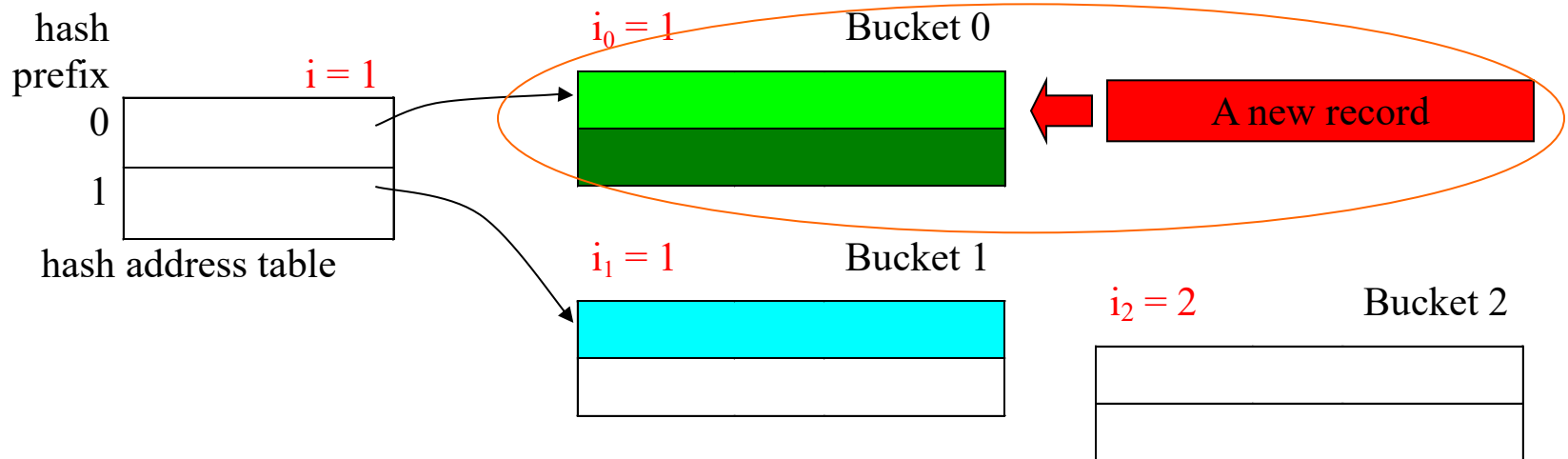
- Suppose bucket 0 is full and a new record arrives
- Create a new bucket, rehash the three records (two existing ones and the new record) into buckets 0 and 1

General Extendable Hash Structure



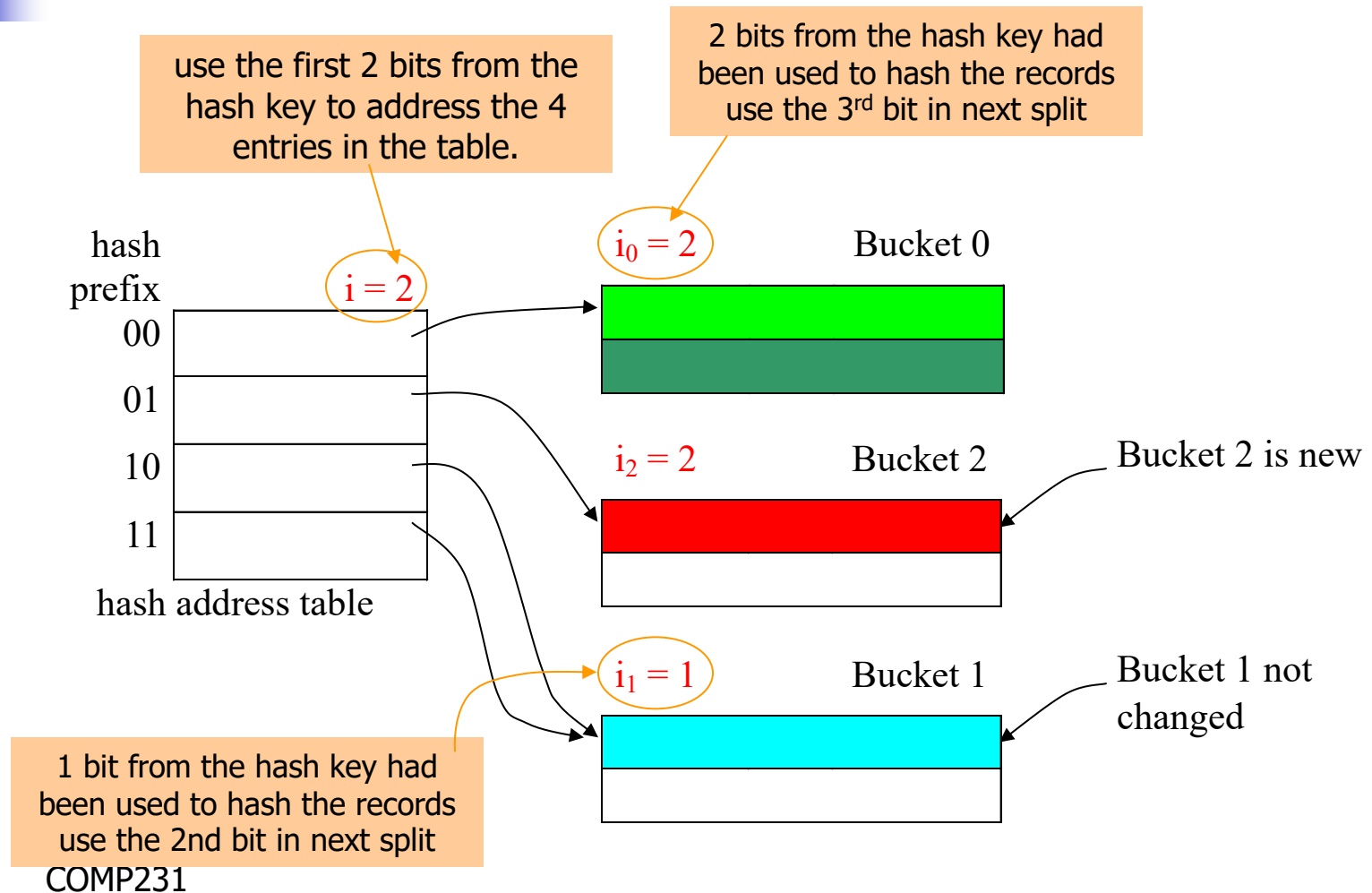
- Note: why do we need to keep i , i_0 and i_1 ?
- i is the maximum number of bits used in hashing so far; i_0 and i_1 are the number of bits used for these particular buckets

General Extendable Hash Structure



- 1) Upon inserting of a new (red) record, bucket 0 is full again
- 2) Bucket 2 is created, and the three records (two existing ones and the new one) are rehashed among buckets 0 and 2 based on the second bit

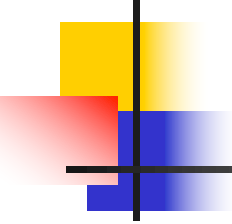
General Extendable Hash structure

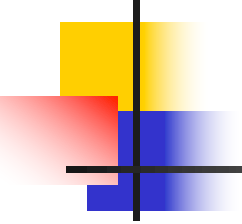




Extendable Hash Structure – Properties

- Every expansion **doubles** the number of entries in the table
- Multiple entries in the bucket address table may point to the same bucket. It means that the bucket hasn't been expanded while other buckets had been expanded multiple times
- Each bucket j stores a value i_j ; entries in the same bucket **have the same values on the first i_j bits of the hash keys**

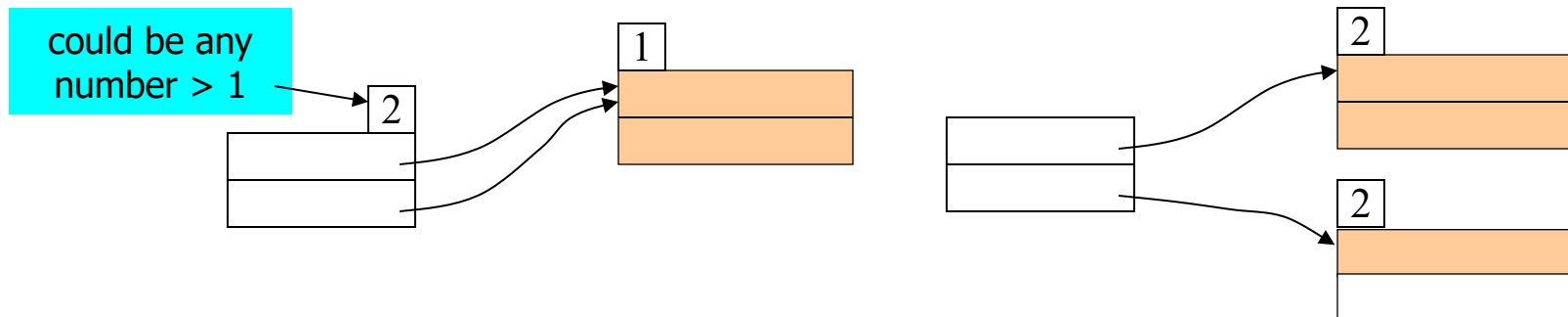
- 
-
- To locate the bucket containing search-key K_j :
 1. Compute $h(K_j) = X$
 2. Use the first i high order bits of X to look up the hash address table, and follow the pointer to appropriate bucket

- 
-
- To insert a record with search-key value K_j
 - look up the bucket where it should belong, say j .
 - If there is room in bucket j
 - insert record in the bucket,
 - else
 - the bucket must be split and insertion re-attempted.

Split in Extendable hash

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

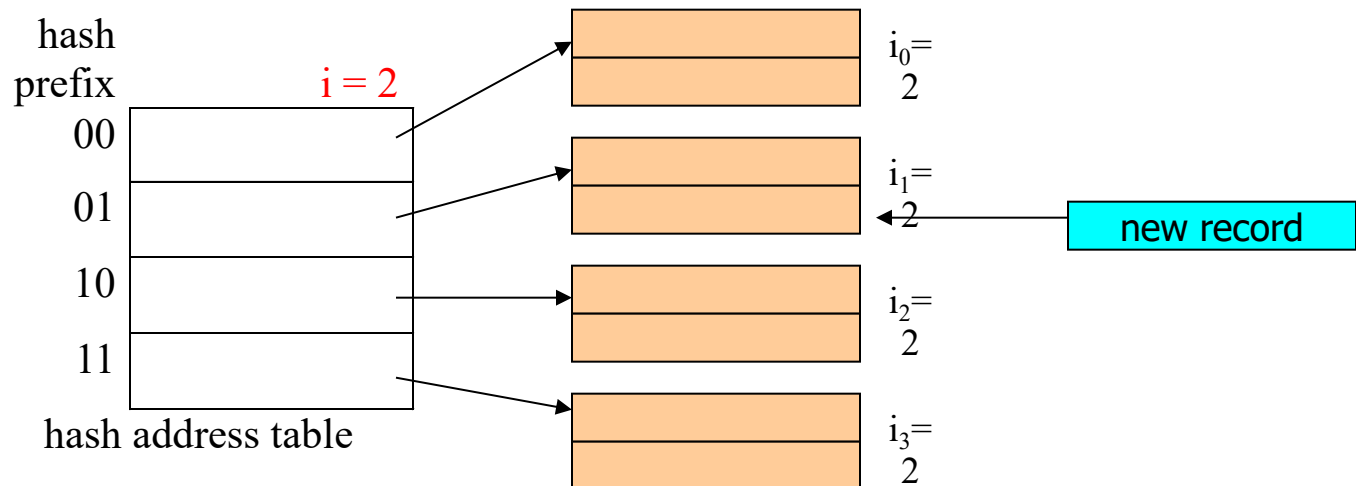
- If $i > i_j$ (*more than one pointer* to bucket j)
 - allocate a new bucket z , and set i_j and i_z to the old i_j+1 .
 - make the second half of the bucket address table entries pointing to j to point to z
 - remove and reinsert each record in bucket j .
 - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)



Split in Extendable hash Structure

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

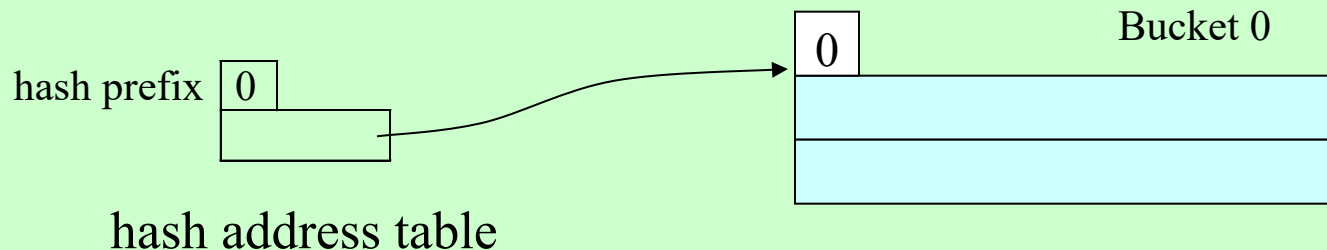
- If $i = i_j$ (*only one pointer* to bucket j)
 - **increment** i and double the size of the bucket address table.
 - **Replace** each entry in the table by two entries that point to the same bucket.
 - **Re-compute** new bucket address table entry for K_j , now $i > i_j$, so use the first case above.



Example: Use of Extendable Hash Structure

Branch-name	$h(\text{branch-name})$
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





Insert: Brighton, A-217, 750

0010 1101 1111 1011 0010 1100 0011 0000

no bit is needed from the hash value ($i=0$)

hash prefix

0

0

Brighton, A-217, 750



Insert: Downtown, A-101, 500

1010 0011 1010 0000 1100 0110 1001 1111
no bit is needed from the hash value ($i=0$)

hash prefix

0

0

Brighton, A-217, 750

Downtown, A-101, 500

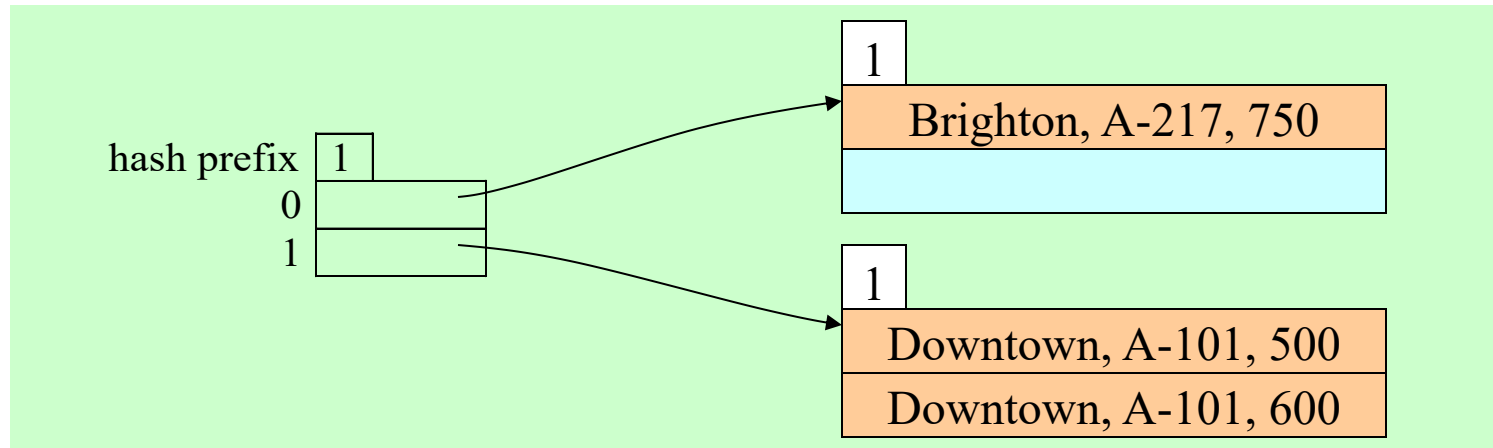
Insert: Downtown, A-101, 600

bucket full, split records according to first bit ($i=1$)



Brighton
Downtown

0010 1101 1111 1011 0010 1100 0011 0000
1010 0011 1010 0000 1100 0110 1001 1111



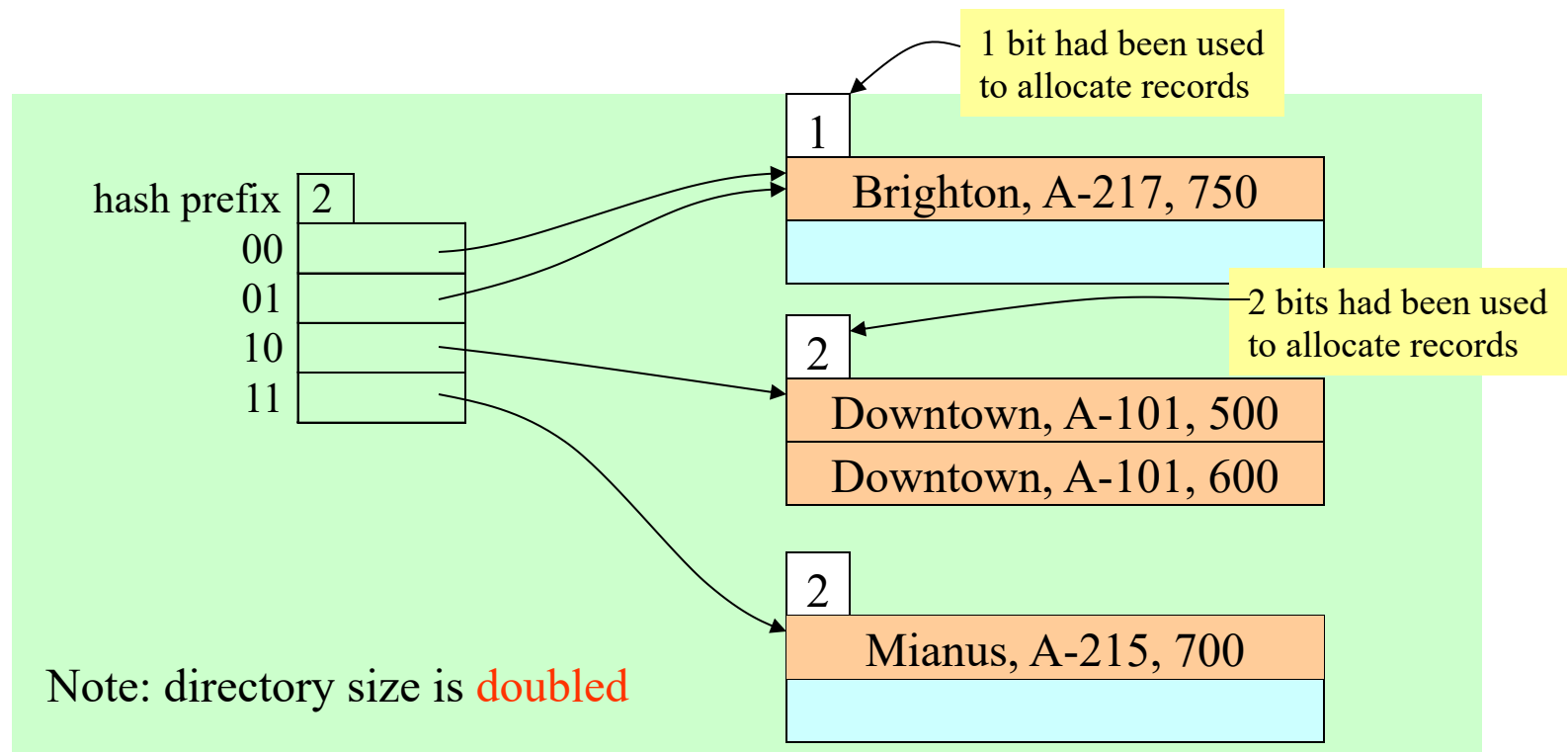
Insert: Mianus, A-215, 700

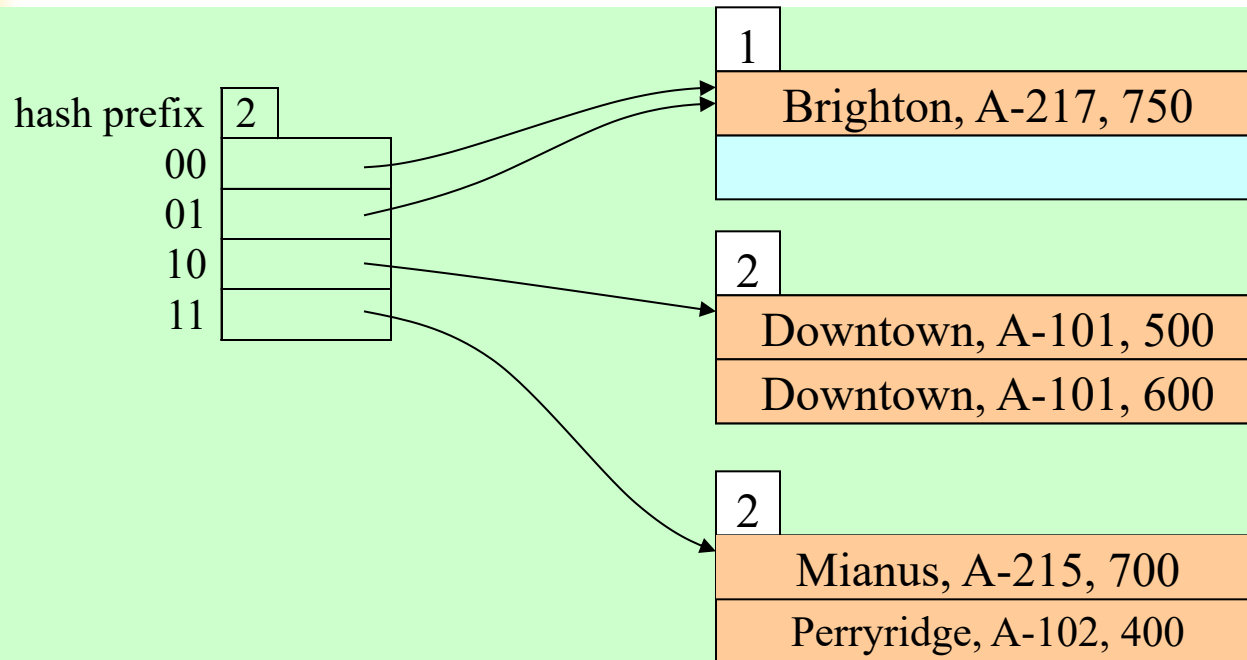
1100 0111 1110 1101 1011 1111 0011 1010

Hash into bucket 1, which is full

Mianus
Downtown

1100 0111 1110 1101 1011 1111 0011 1010
1010 0011 1010 0000 1100 0110 1001 1111





Insert:

Perryridge, A-102, 400

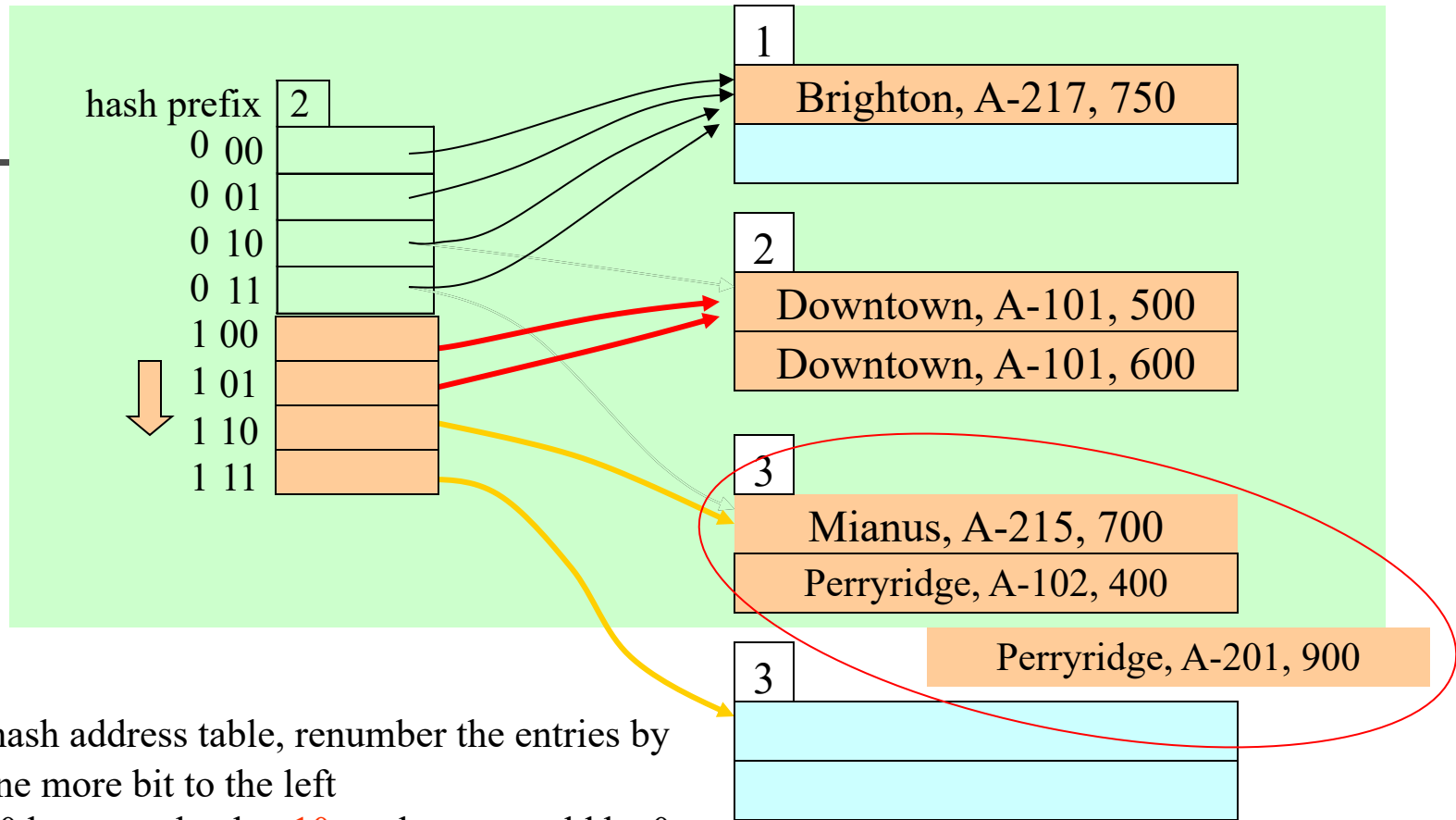
1111 0001 0010 0100 1001 0011 0110 1101

Insert:

Perryridge, A-201, 900

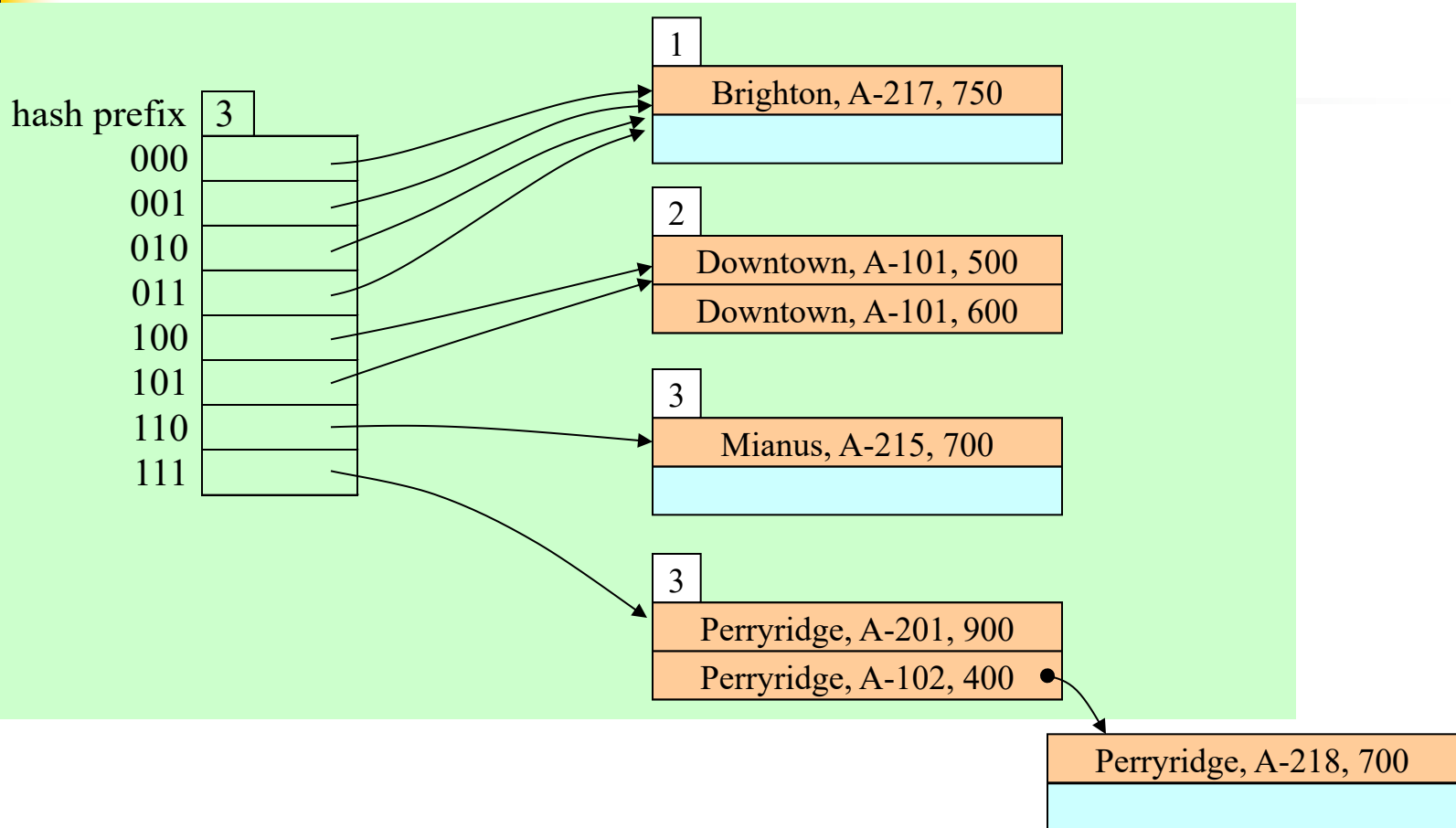
1111 0001 0010 0100 1001 0011 0110 1101

Bucket 2 overflows again!



- Expand hash address table, renumber the entries by adding one more bit to the left
- Bucket 10 becomes bucket 10x, where x could be 0 or 1
- Bucket 11 becomes bucket 110 and 111, because it is split; new bucket is added; local level is updated

Redistribute overflow and new record



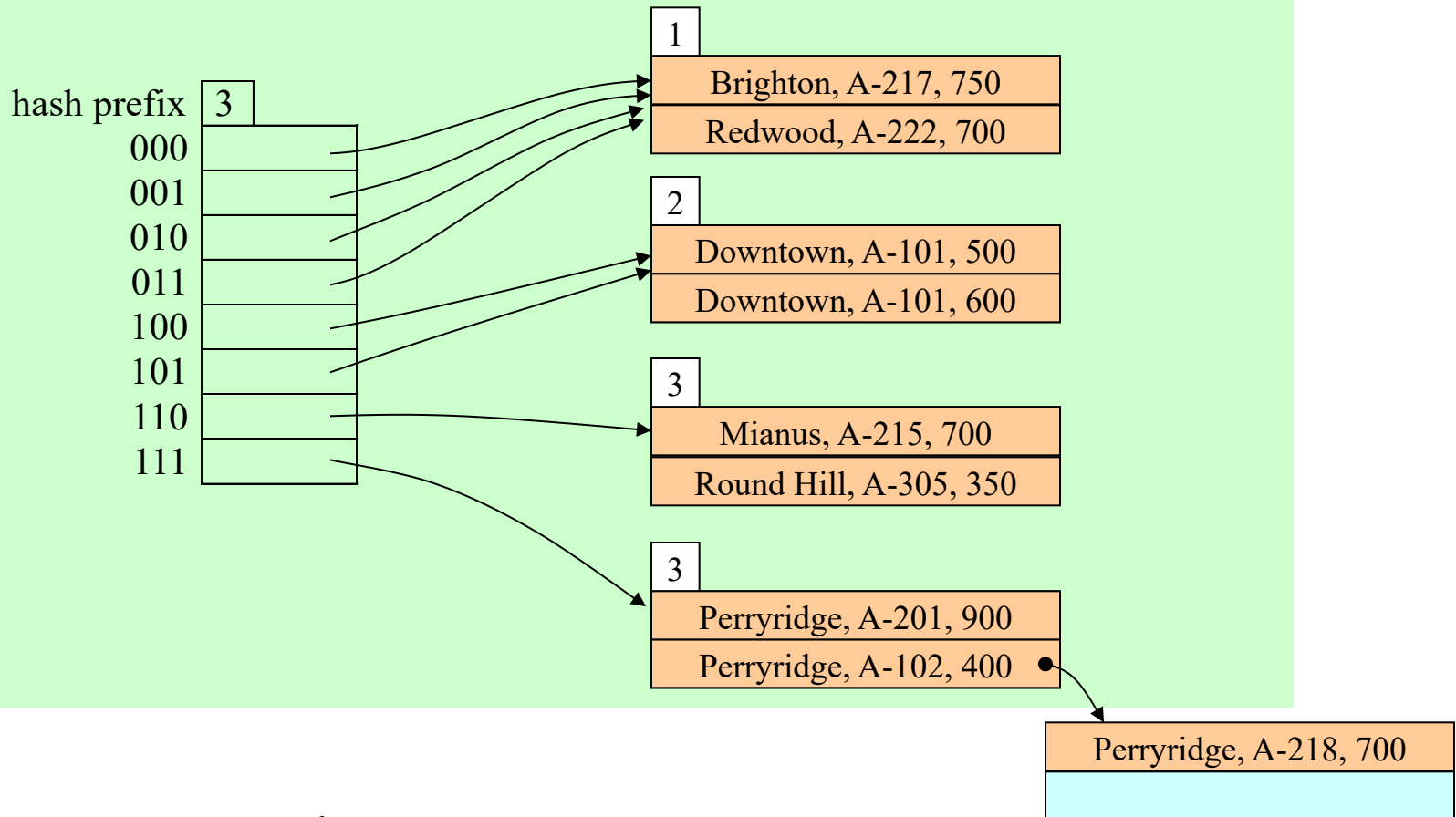
Insert:

Perryridge, A-218, 700

1111 0001 0010 0100 1001 0011 0110 1101

COMP231

Bucket 3 overflows again!



Done!
COMP231