Chapter 12: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Multiple-Key Access

Basic Concepts

- Many queries references small number of records in file. Eg “Find all accounts at Perryridge branch”.
- Inefficient to read each record and check value of branch-name.
- Indexing mechanisms used to speed up access to desired data.
  - E.g., index in book, author catalog in library
- Database system will look index to find on which disk block corresponding record resides and fetch the disk block to get the record

- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form **(search key, pointer).** Pointer has identifier of disk block and an offset within disk block to identify record within block.
- Index files are typically much smaller than the original file

Index Evaluation Metrics

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using hash function
- Factors on which techniques will be evaluated:
  - **Access types** supported efficiently. E.g.,

    finding particular attribute values or in range values
  - **Access time :** The time it takes to find a particular data item
  - **Insertion time :** Time taken for insert data
  - **Deletion time :** Time taken for delete data
  - **Space overhead :** : The additional space occupied by an index structure.

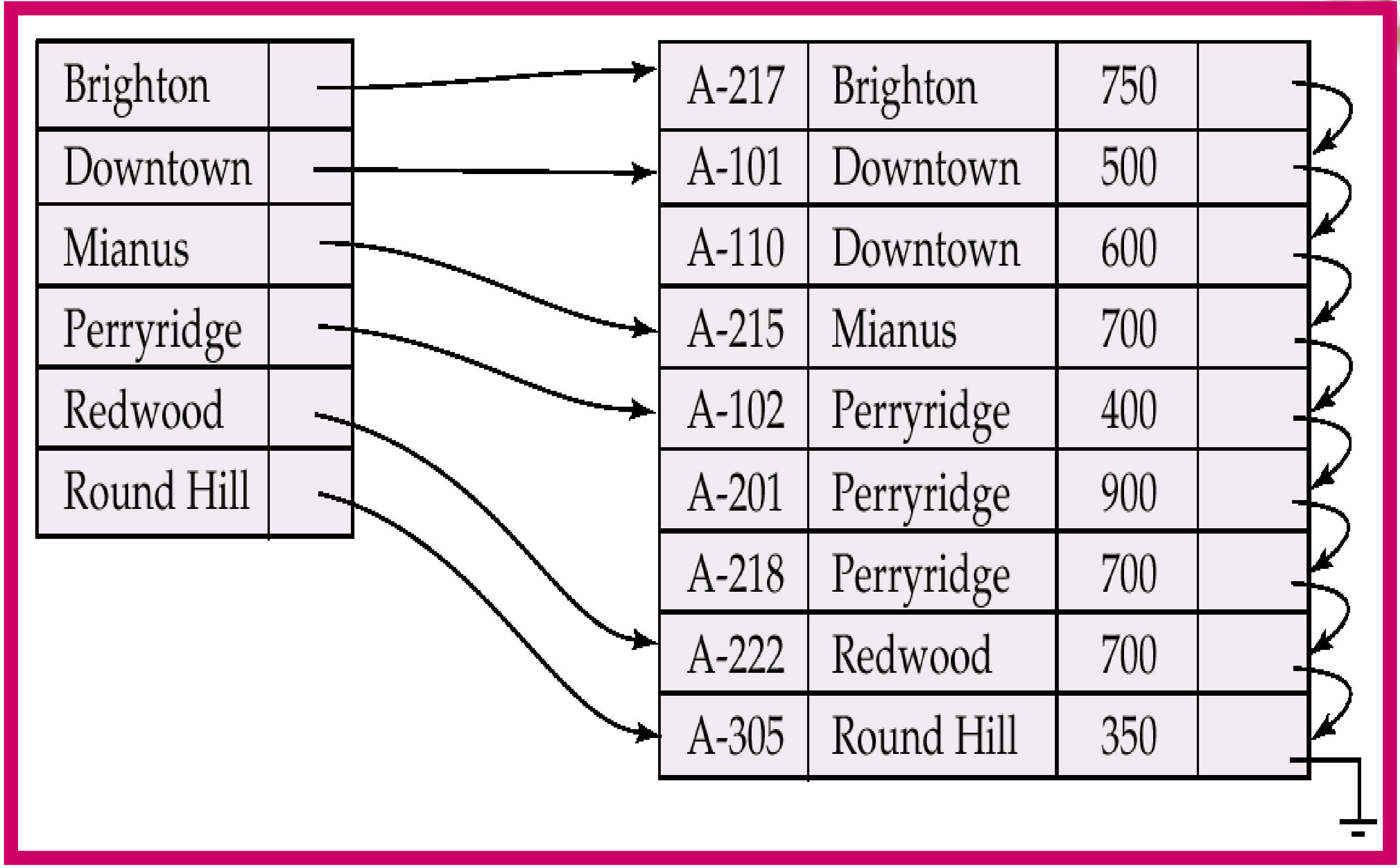
- We often want to have more than one index for a file.
- For example, libraries maintained several card catalogs: for author, for subject, and for title.
- An attribute or set of attributes used to look up records in a file is called a search key.

Ordered Indices

– To gain fast random access to records in a file, we can use an index structure
– Each index entry is associated with particular search key
– An **ordered index**, stores value of search key in sorted order and associates with each search key records that contain it.
– **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Primary indix is also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
– Index-sequential file: ordered sequential file with a primary index on search key.

- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**

Ordered Indices (Contd…)



Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

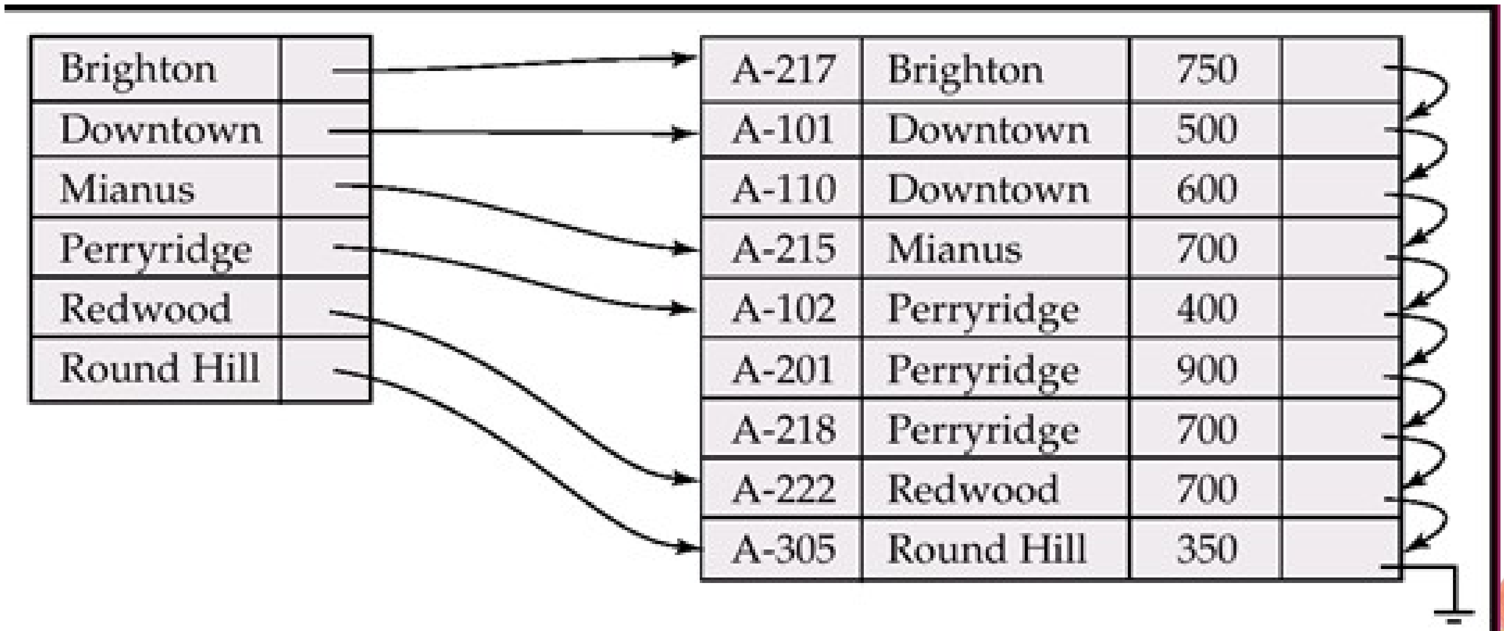
A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Primary index : , the records are stored in search-key order, with branch-name used as the search key.

Dense Index Files

- Dense index — Index record appears for every search-key value in the file.
- In dense primary index, records are sorted on same search key.
- Index record contains search key value and pointer to first data record with search key value
- Rest of records are stored sequentially
- .

- Suppose that we are looking up records for the Perryridge branch. Using the dense index of Figure we follow the pointer directly to the first Perryridge record.
- continue processing records until we encounter a record for a branch other than Perryridge



Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

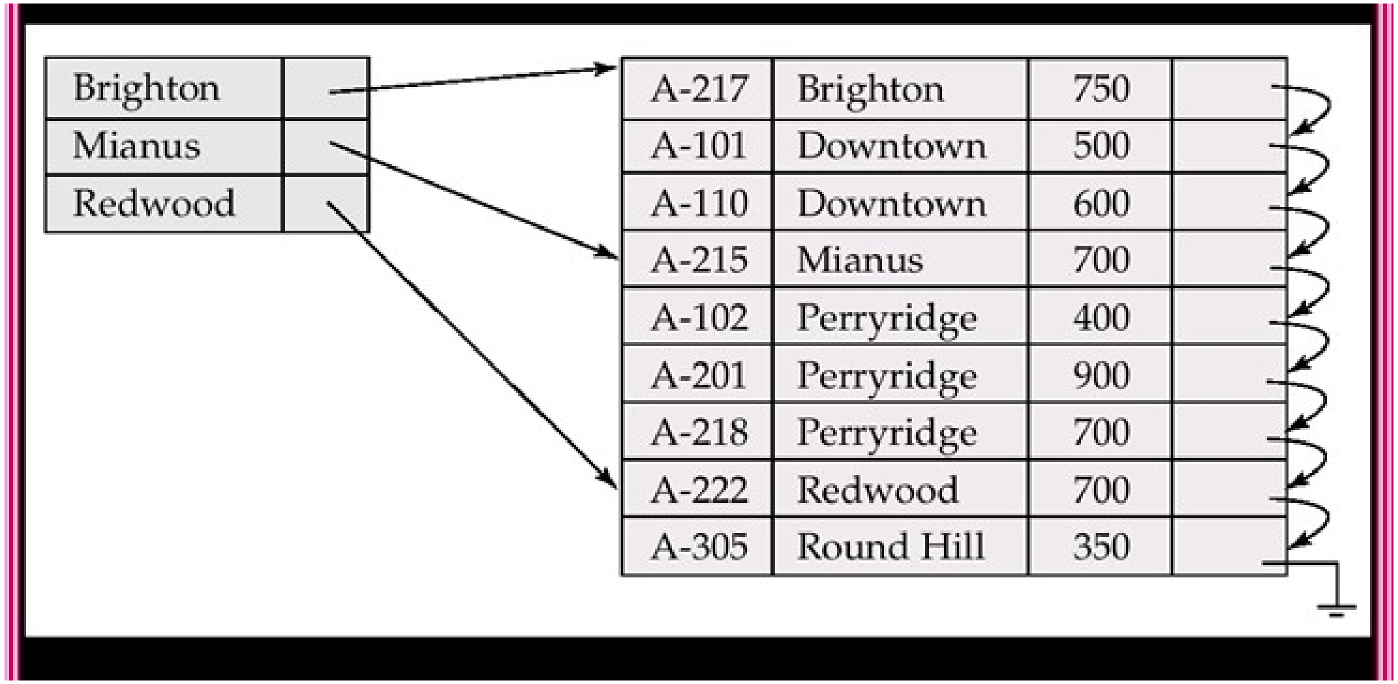
A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Sparse Index File

- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate record, find index entry with largest search key value that is less than or equal to search key value we are looking and then search file sequentially.

- For e.g finding a entry for Perryridge branch in
- Given sparse index file.entry for perrryridge branch is not available in sparse index file.
- Since the last entry (in alphabetic order) before “Perryridge” is “Mianus,” , follow that pointer.
- Then read the account file in sequential order until we find the first Perryridge record

- Sparse index file for account table/file



Brighton	
Mianus	
Redwood	

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

Dense V/S Sparse Index

- Dense Index: Faster to locate records but occupy more space Sparse Index: Less space required but not faster
- System designer must make trade off between space and access time
- Good tradeoff: sparse index with an index entry for every block in file. We can locate block containing record that we are looking
- Cost to process a database request is time to bring block from disk to memory.
- Once block is in memory, time required to scan entire block is negligible.

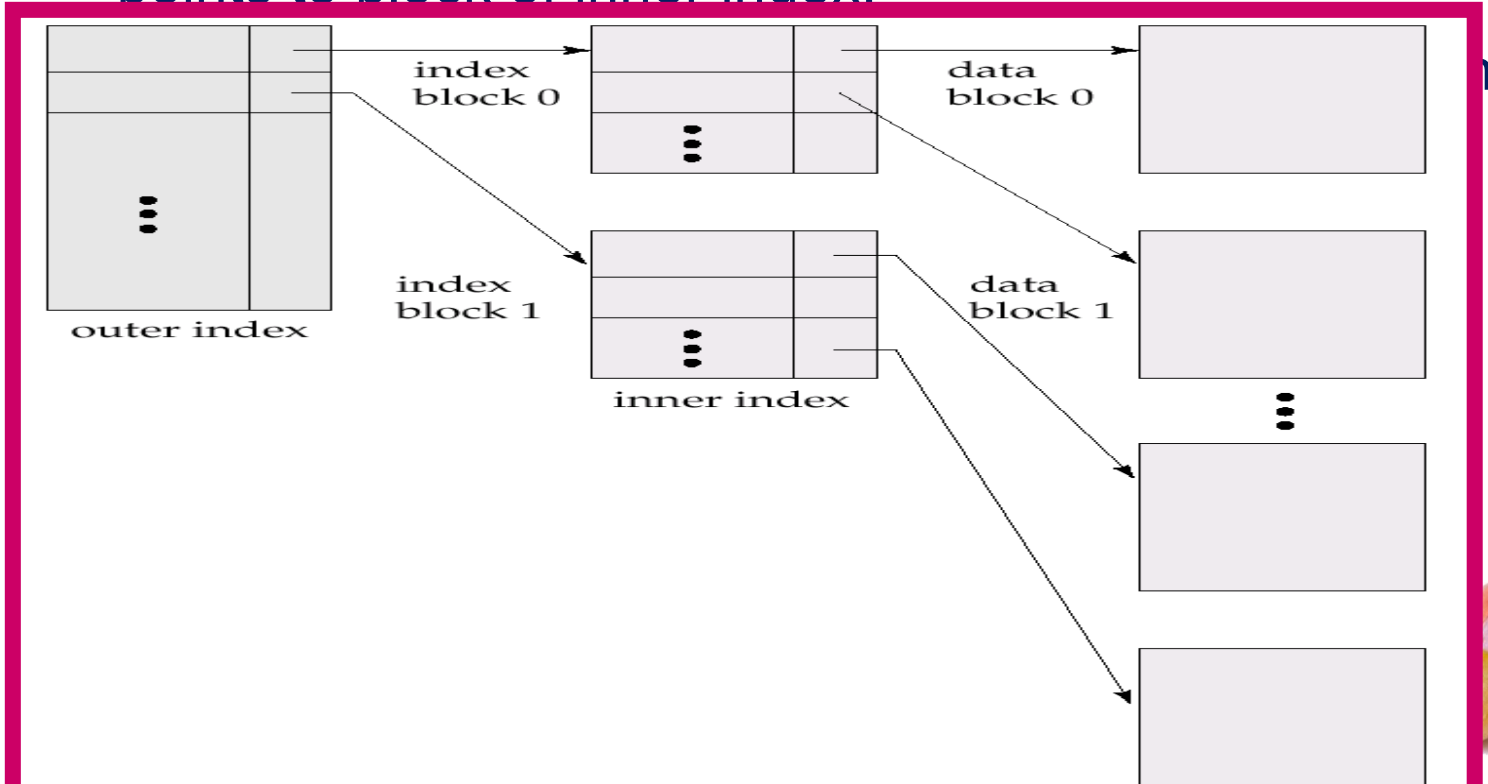
Multilevel Index

- Suppose file has 100,000 records and 10 records in each block ie 10,000 blocks.
- If one index record per block then index will have 10,000 records.
- Assume that 100 index records fit in 1 block then index file itself will need 100 blocks. Such large index files must be stored on disk.
- To locate a record, binary search is used on index file to locate entry but still it has large cost.
  - If index occupies b blocks, binary search requires $\log_2(b)$ blocks to read
  - E.g if 100 index blocks, binary search will require 7 blocks read will be required to read single record, which is very time consuming

Multilevel Index

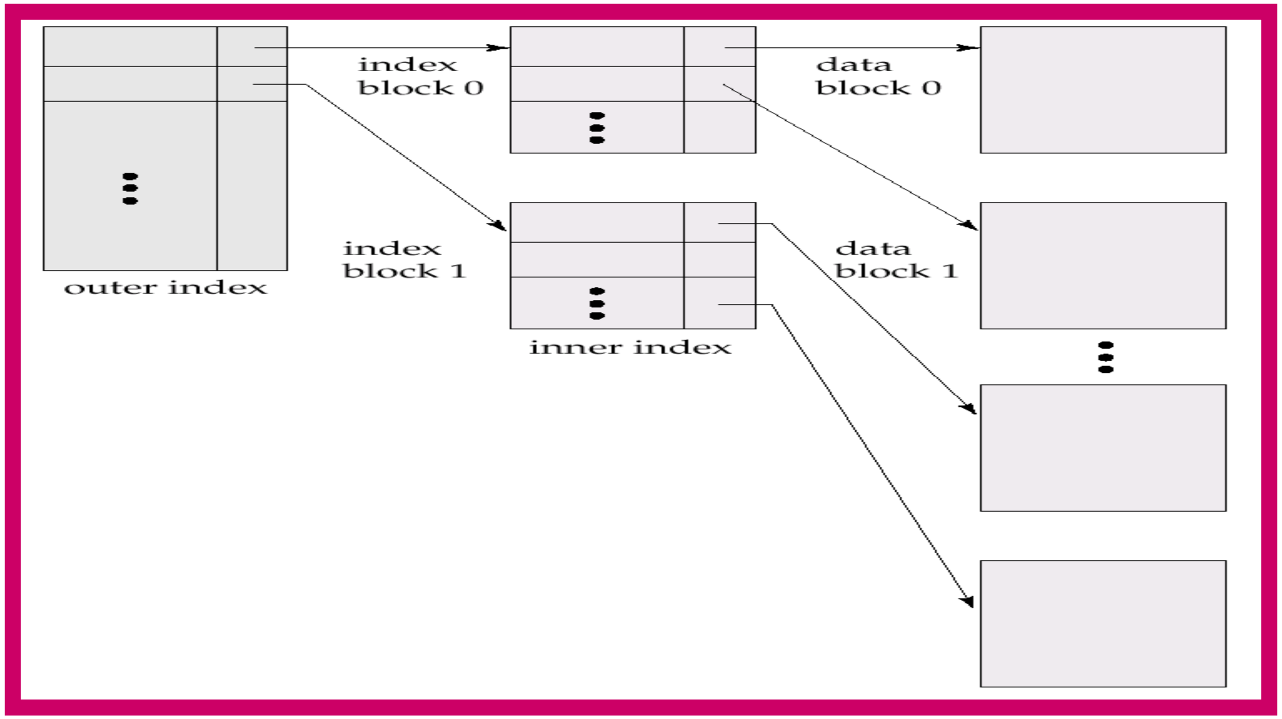
- To deal with this problem,
- Treat index file as any other sequential file and construct sparse index on primary index.
- To locate record, first use binary search on outer index to find record for largest search key value. Pointer points to block of inner index.

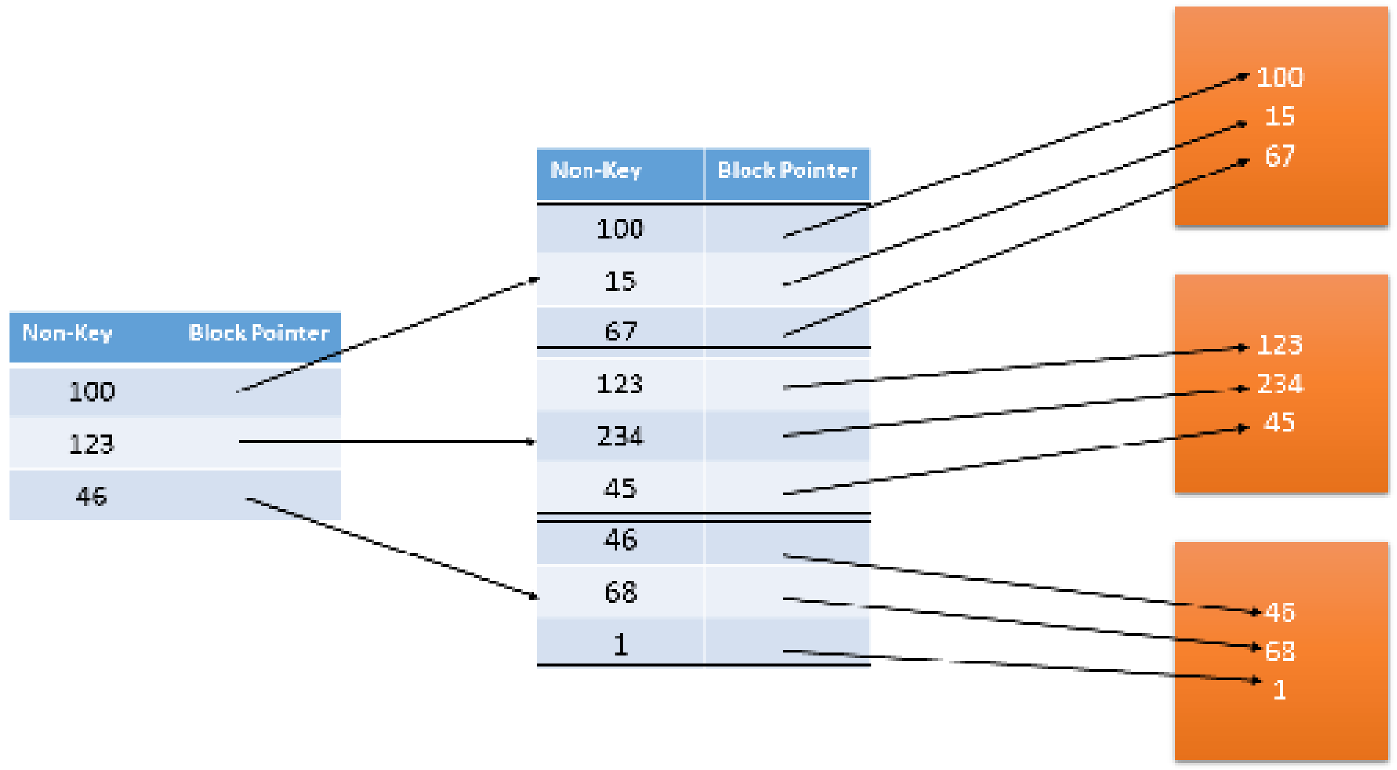
ng



#

- Sfds


index
block 0
data
block 0
index
block 1
data
block 1
outer index
inner index



Non-Key
Block Pointer
100
123
46
Non-Key
Block Pointer
100
15
67
123
234
45
46
68
1
100
15
67
123
234
45
46
68
1

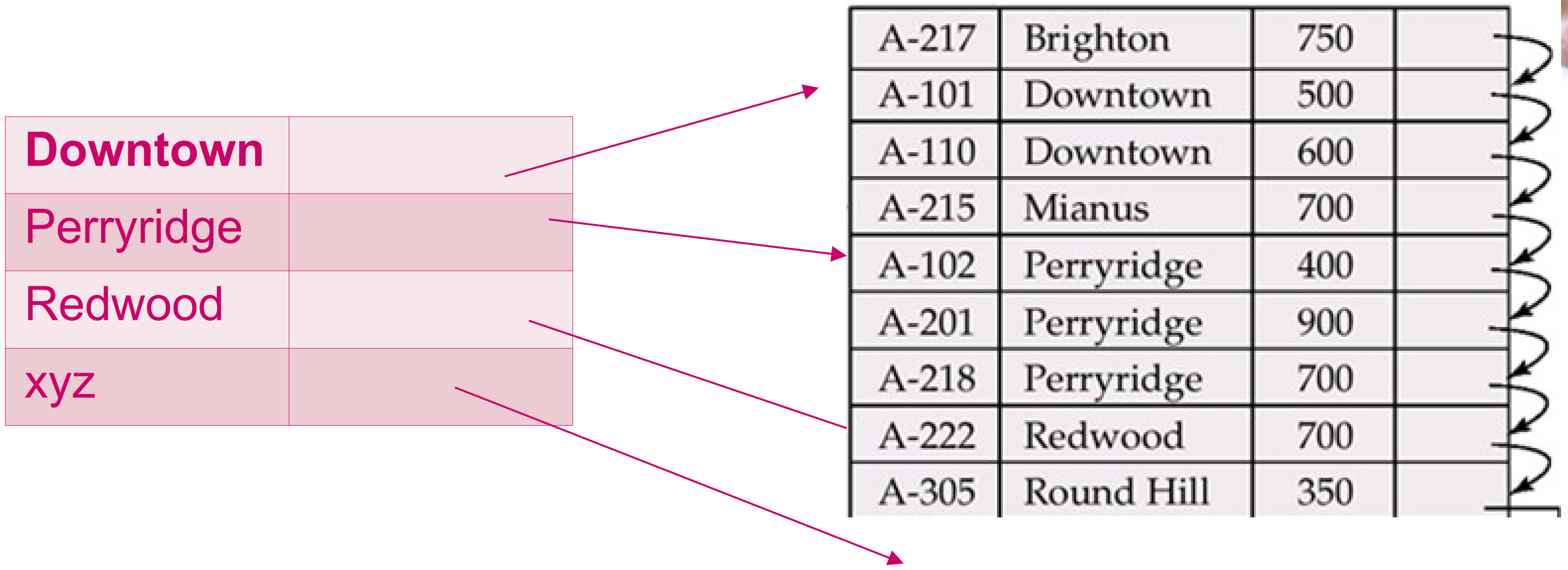
Multilevel Index

- If our file is extremely large even though outer index may grow too large to fit in memory we can create another level of index.
- Indices with two or more levels are called as multi level indices.
- Searching of record with multilevel index requires significantly fewer IO operations.

Index Update: Deletion

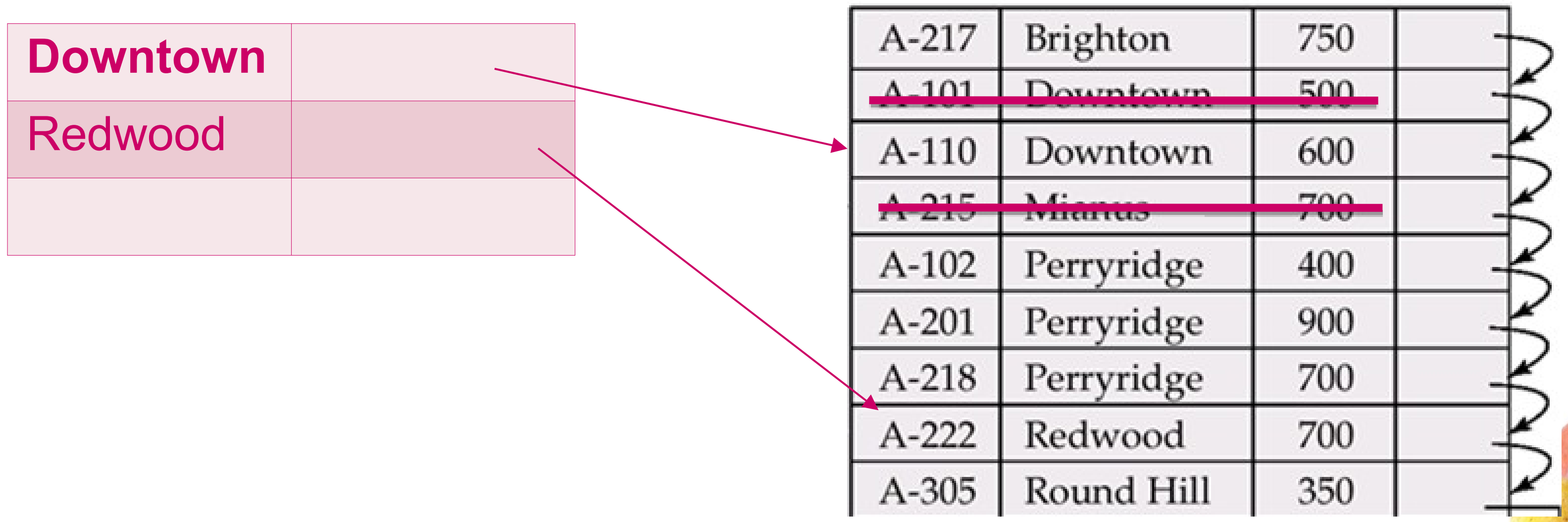
- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
- Dense indices – deletion of search-key is similar to file record deletion.
- Sparse indices – if an entry for the search key exists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

Deletion on sparse index



Downtown	
Perryridge	
Redwood	
xyz	

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



Downtown	
Redwood	

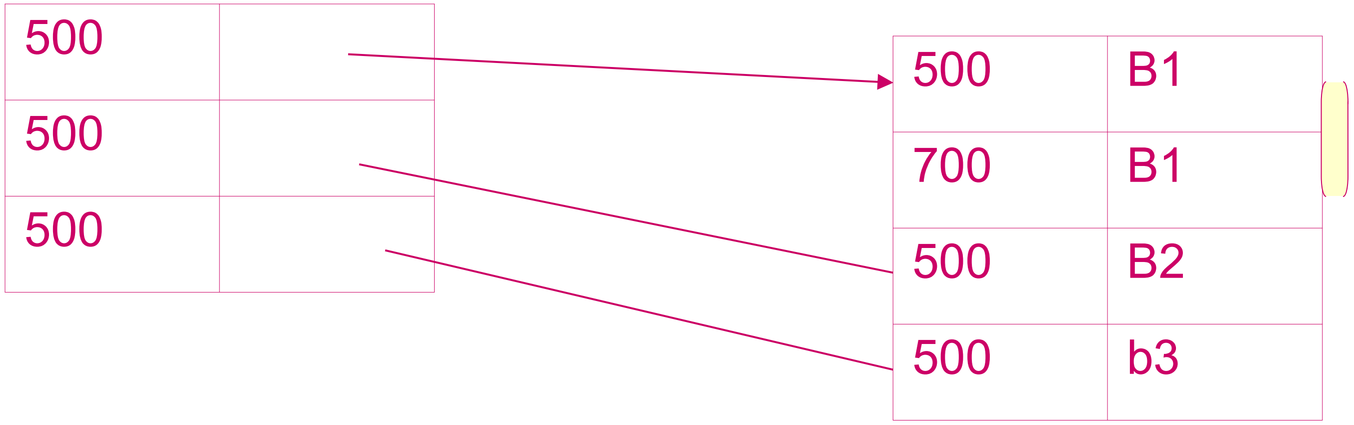
A-217	Brighton	750	
~~A-101~~	~~Downtown~~	~~500~~	
A-110	Downtown	600	
~~A-215~~	~~Mianus~~	~~700~~	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

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

– Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file.
– if the search key of a secondary index is not a candidate key, it is
– not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file
– since the records are ordered by the search key of the primary index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records
– can use an extra level of indirection to implement secondary indices on search keys that are not candidate key



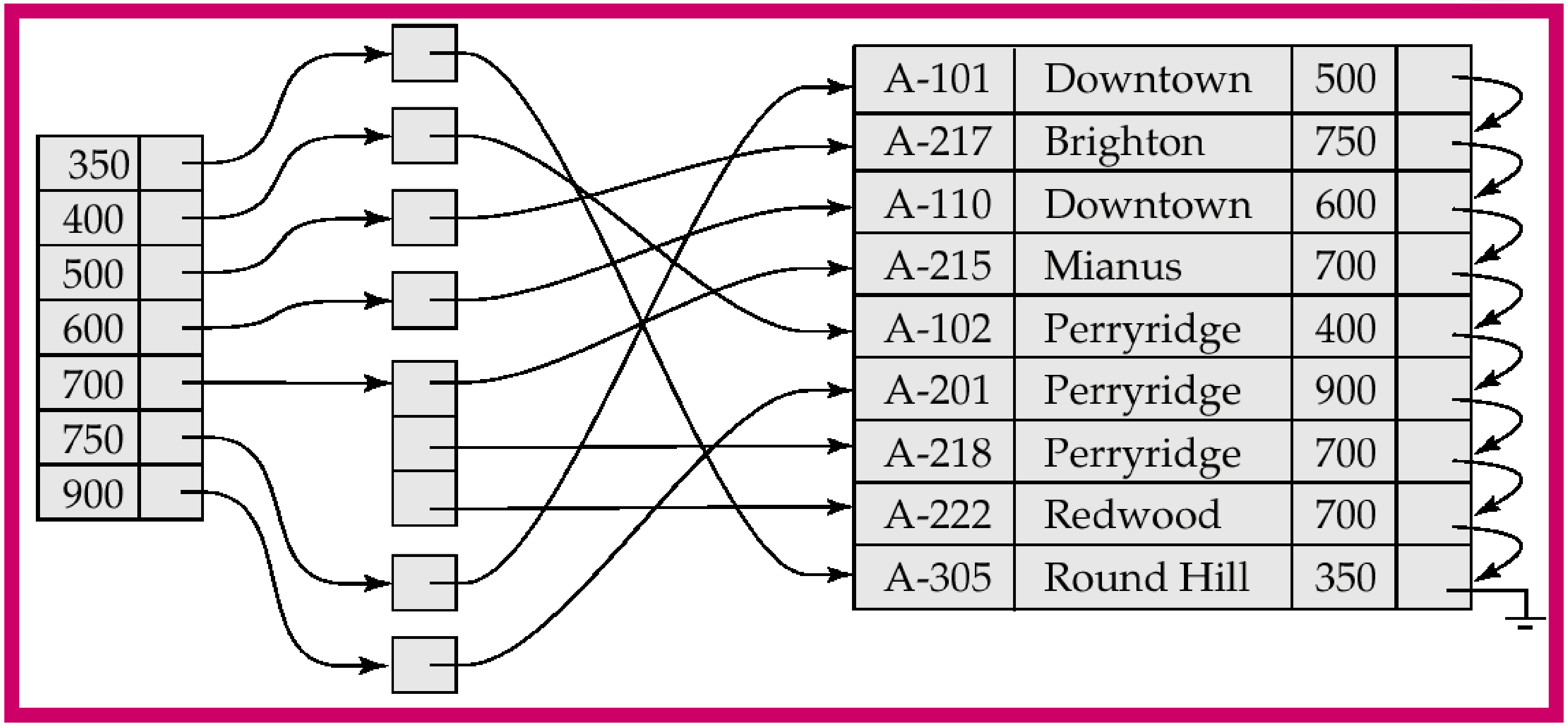
500	
500	
500	

500	B1
700	B1
500	B2
500	b3

Secondary Indices

- 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: In 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; 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*



Static Hashing

- In a **hash file organization** we obtain address of disk block containing record directly by computing **hash function** on search key value of record.
- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- To insert a record with search key Ki compute h(Ki) then search bucket with that address hash(production) = 4- > 4th disk block /bucket
- Similarly for access 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.

Hash Functions

- Worst hash function maps all search-key values to the same bucket
- Ideal hash function distributes keys uniform across all buckets so that every bucket has same number of records.
- Distribution is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible search-key values.
- Distribution is **random**, i.e. each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.

Hash Functions (contd…)

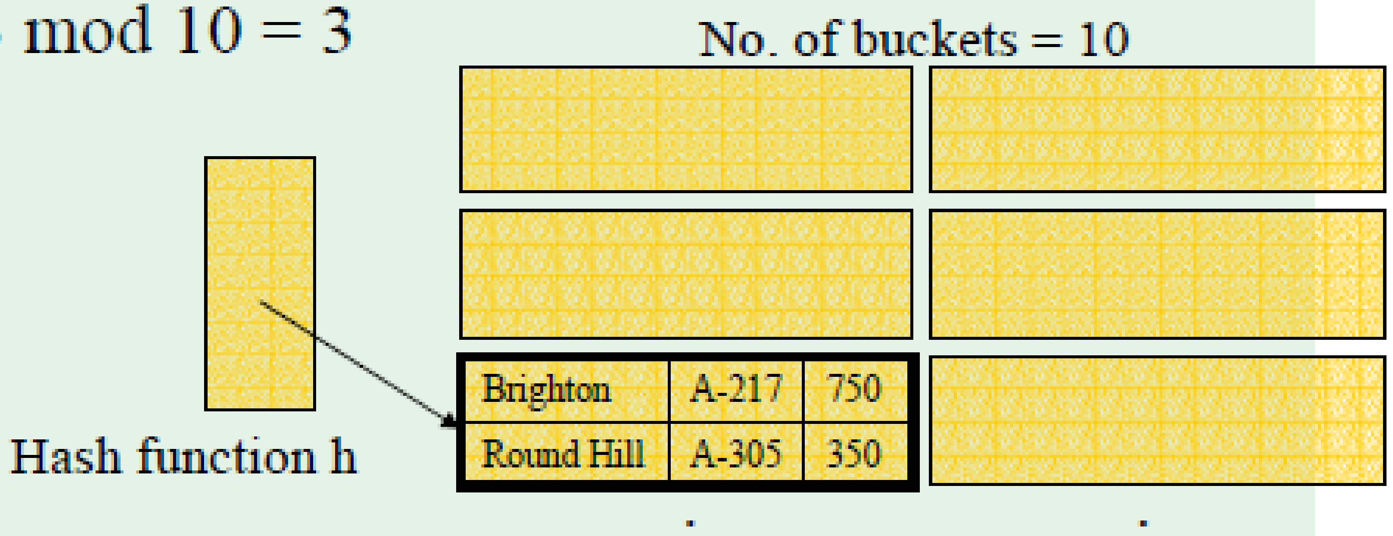
- The hash function will not be correlated to any externally visible ordering e.g. alphabetical ordering .
- Hash function maps search key beginning with ith letter to ith bucket, assuming that there are 26 buckets. It is not uniform or random distribution as there will be more records beginning with letter B or R
- Typical hash functions perform computation on the internal binary representation of the search-key.

Example of Hash File Organization (Cont.)

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

- 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

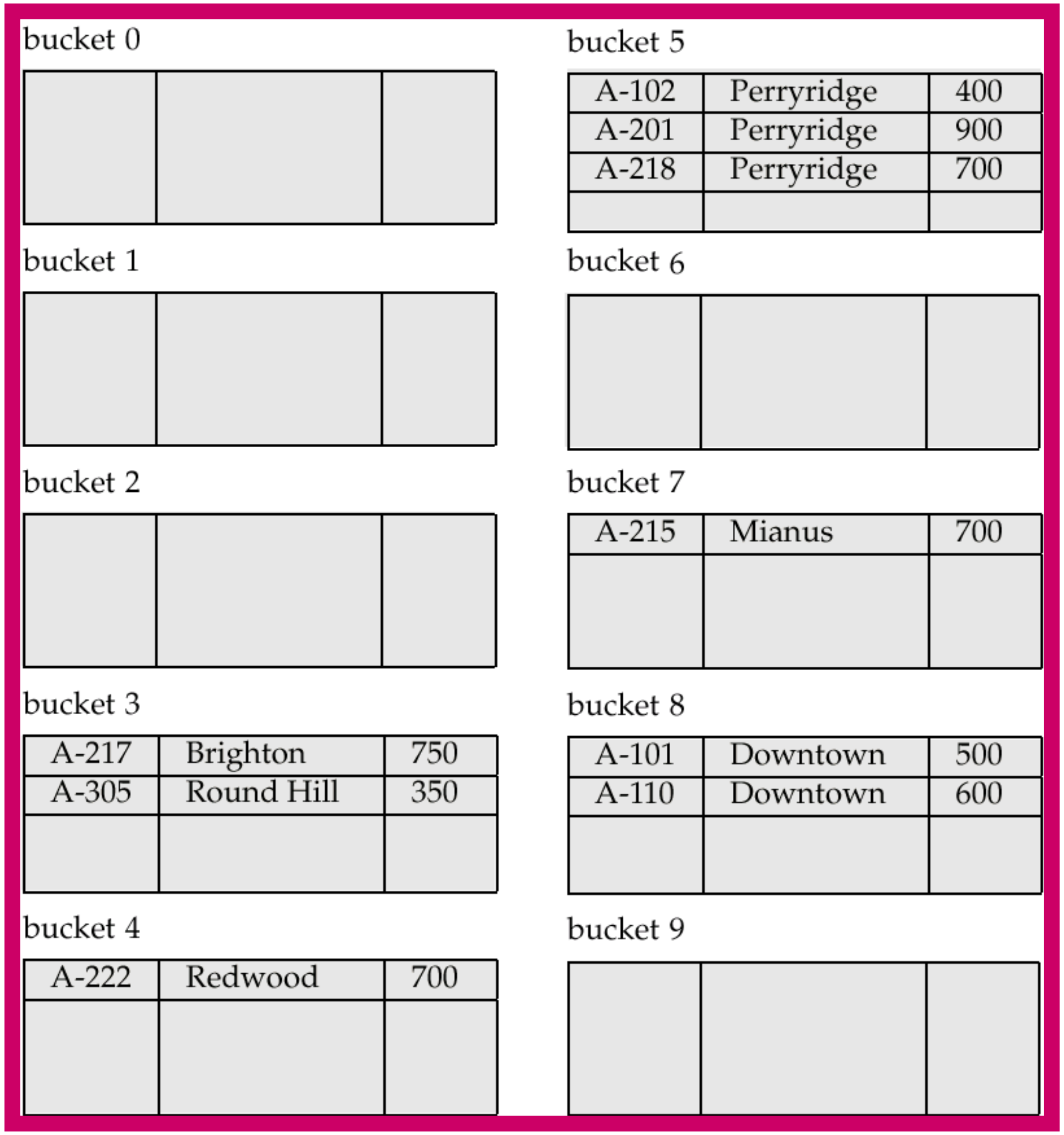
- A hash function h maps a search-key value K to an address of a bucket
- Commonly used hash function *hash value mod* n_B where n_B is the no. of buckets
- E.g. h(Brighton) = (2+18+9+7+8+20+15+14) mod 10 = 93 mod 10 = 3



Brighton	A-217	750
Round Hill	A-305	350

Example of Hash File Organization

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



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

Handling of Bucket Overflows

- When record is to be inserted and no space in bucket then it is said to be bucket overflow
- Bucket overflow can occur because of
  - Insufficient buckets
    - n_B=buckets, f_r=no. of records fit in bucket, n_r= total no. of records Then $n_b > n_r/f_r$
    - 1000 records = nr , fr=5 records in one backet . 1000/5 = 201 buckets,
    - But we assume that total no. of records are known when hash function is chosen.
  - Skew
    - Some buckets are assigned more records then others, so bucket may get overflow even when other buckets still have space. This situation is called as skew.

- 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.)

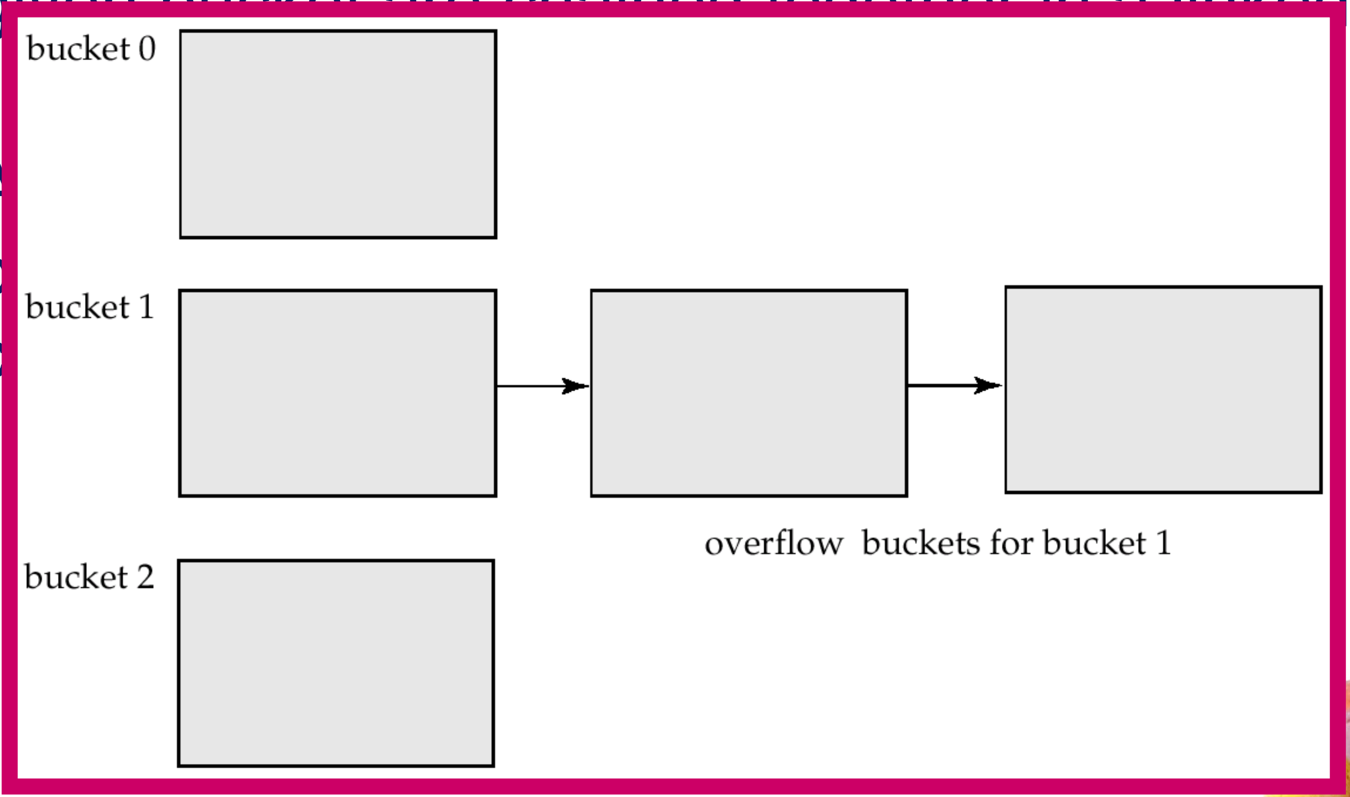
– If bucket is full overflow bucket is given. If overflow bucket is full another overflow bucket is given and so on

– Overflow chaining – the overflow buckets of a given bucket are chained together in a linked li

– A

b

c



Handling of Bucket Overflows (Cont.)

– An alternative is called open hashing in which there are no overflow buckets.
– If bucket is full system inserts records in some other bucket.
– One policy is to use next bucket (cyclic order) that has space. It is called as Linear Probing
– Lookup and insertion is easy but deletion is difficult to handle.
– Open hashing is used in compilers and assemblers but not suitable for database

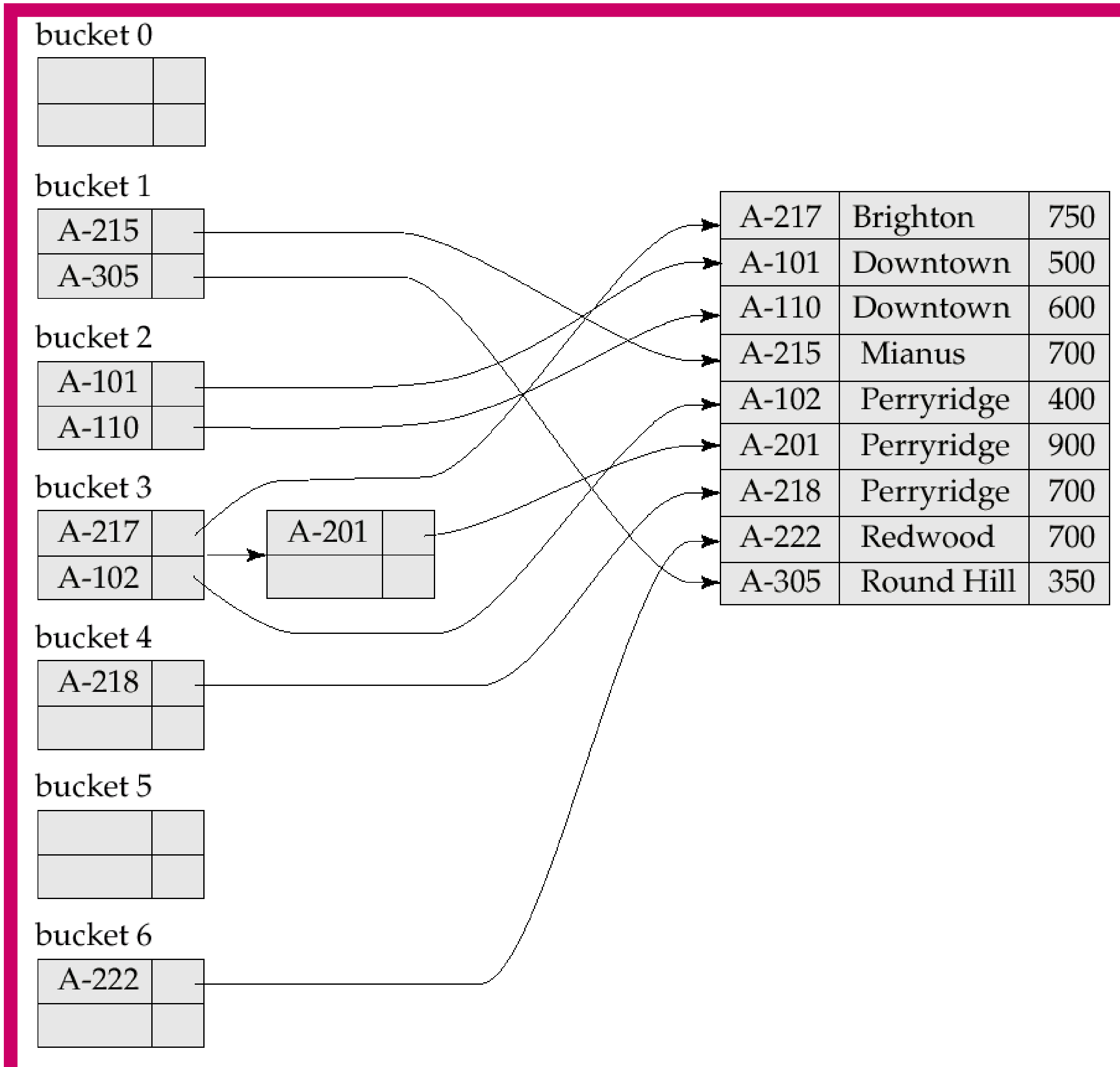
Hash Indices

- 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.
- Apply hash function on search key to identify bucket and store key and its associated pointers in bucket.
- Hash indices are always secondary index.

- Because as file is already sorted according to primary index.

- Hash index on account file for search key account number. Hash function is sum of digits of account number modulo 7. Assume that each bucket can store 2 records

Example of Hash Index

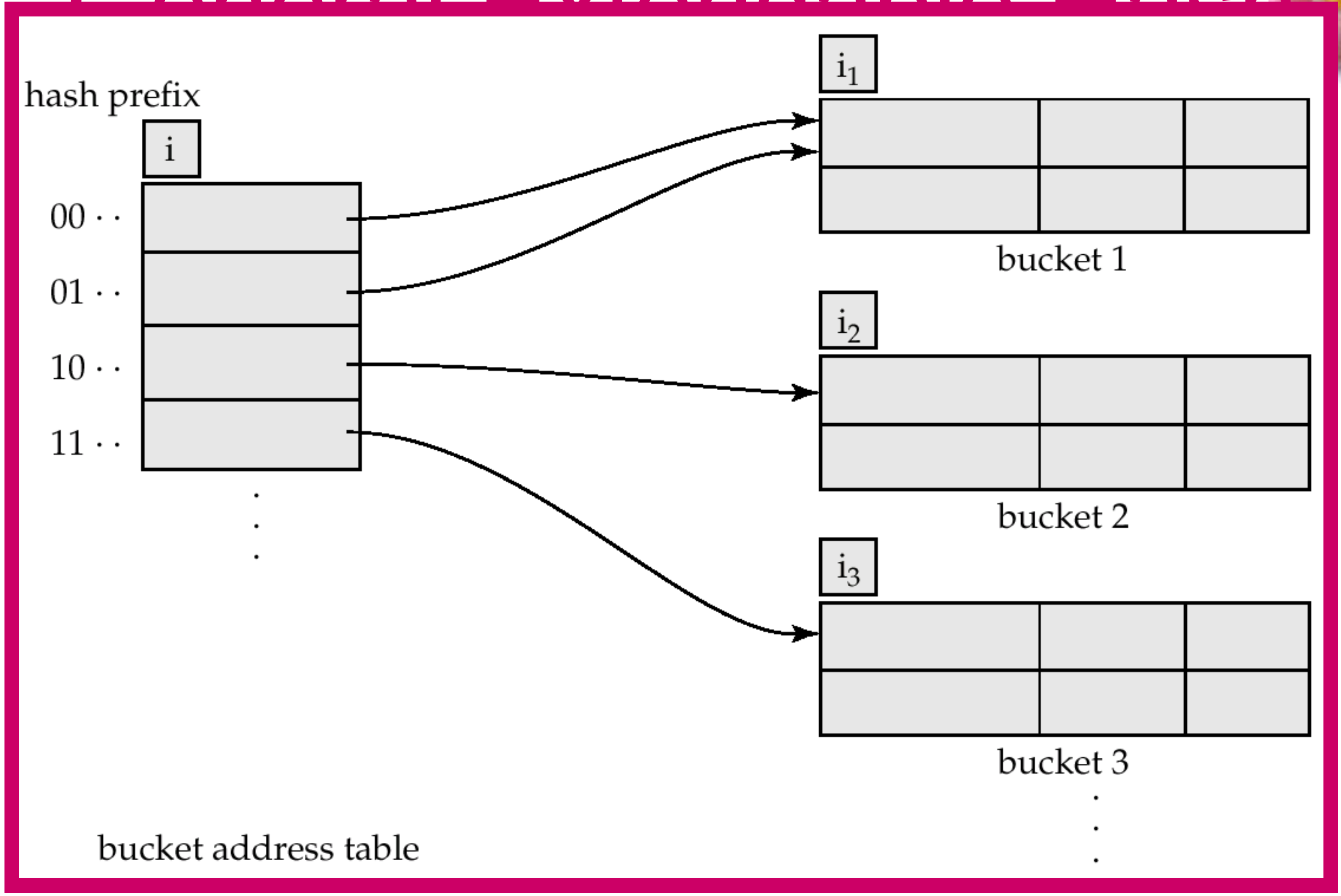


A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Dynamic Hashing

- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
- Buckets grows and shrinks as database size increases or decreases
- Hash function generates values over *b*-bit integers, with *b* = 32.
- Buckets are created as and when records are inserted into file
- At any time use only i bits 0<=i<=b.
- Bucket address table size = 2^{i}. Initially *i* = 0
- Value of *i* grows and shrinks as the size of the database grows and shrinks.
- i_j indicates length of common hash prefix for bucket j

General Extendable Hash



Use of Extendable Hash Structure

- 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 and follow 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.
  - If bucket is full , split bucket and redistribute current records + new to be inserted

Updates in Extendable Hash Structure

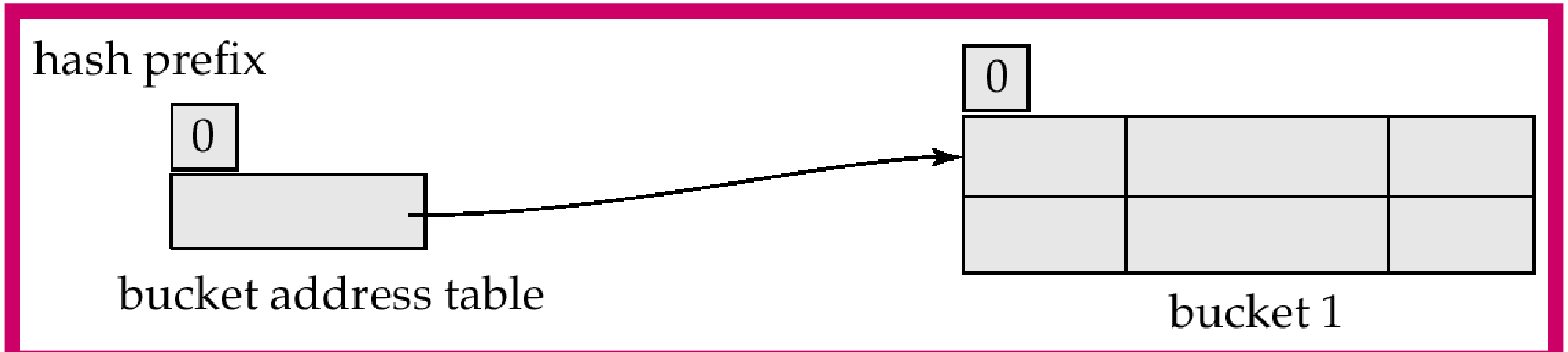
To split a bucket *j* when inserting record with search-key value K_j:

- If $i = i_j$ (*j*)
  - increment *i* aonly one pointer to bucket nd double the size of the bucket address table.
  - replace each entry in the table by two entries that point to the same bucket.
  - Allocate new bucket z and set second entry to z.
  - Set i_j and i_z to i
  - Rehash each record in bucket j depending on first i bits and keep record either in j or z.

- If $i > i_j$ (more than one pointer to bucket j)
  - Split bucket j without increasing size of bucket address table. All entries that points to j has same value for leftmost i_j bits
  - allocate a new bucket z, and set i_j and i_z to the old i_j + 1.
  - Readjust pointers: Leave first half of entries pointing to bucket j and set all remaining entries point to z
  - Rehash records in bucket j and allocate to j and z.
  - further splitting is required if the bucket is still full

Use of Extendable Hash

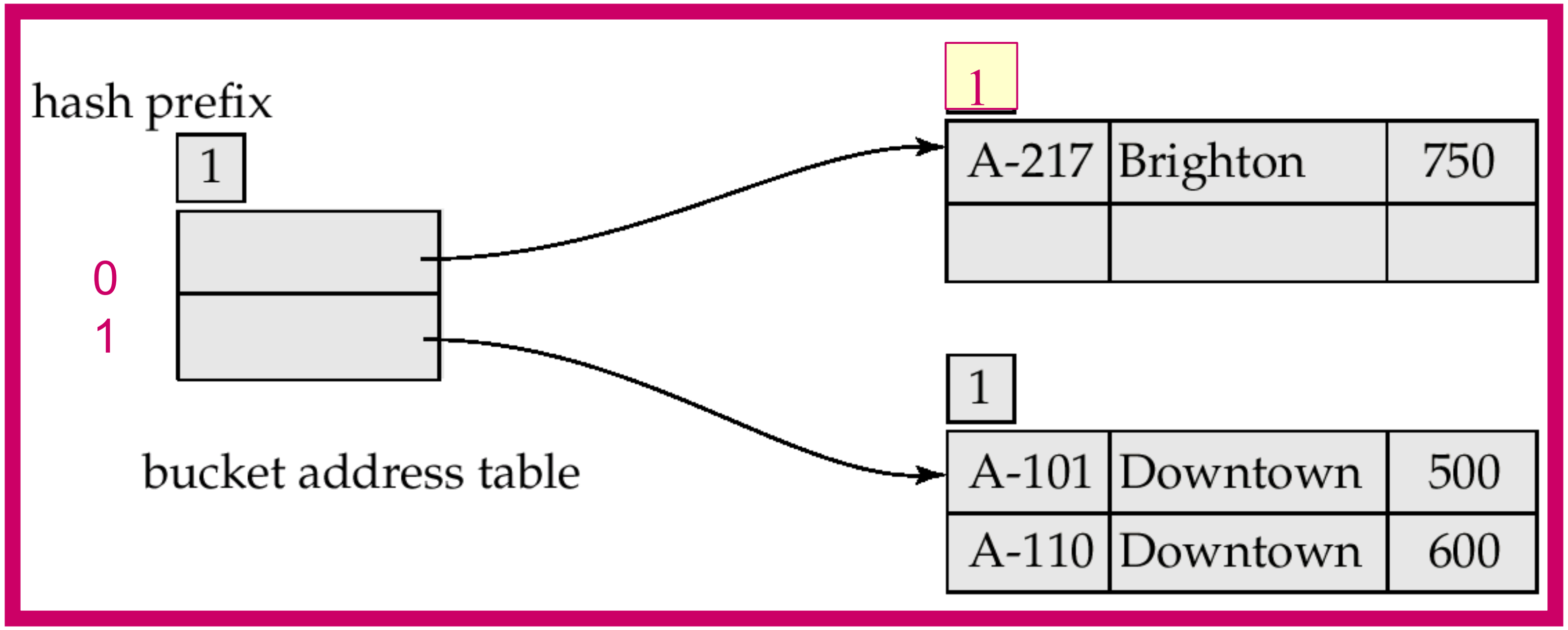
branch-name	h(*branch-name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111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



Initial Hash structure, bucket size = 2

Example (Cont.)

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



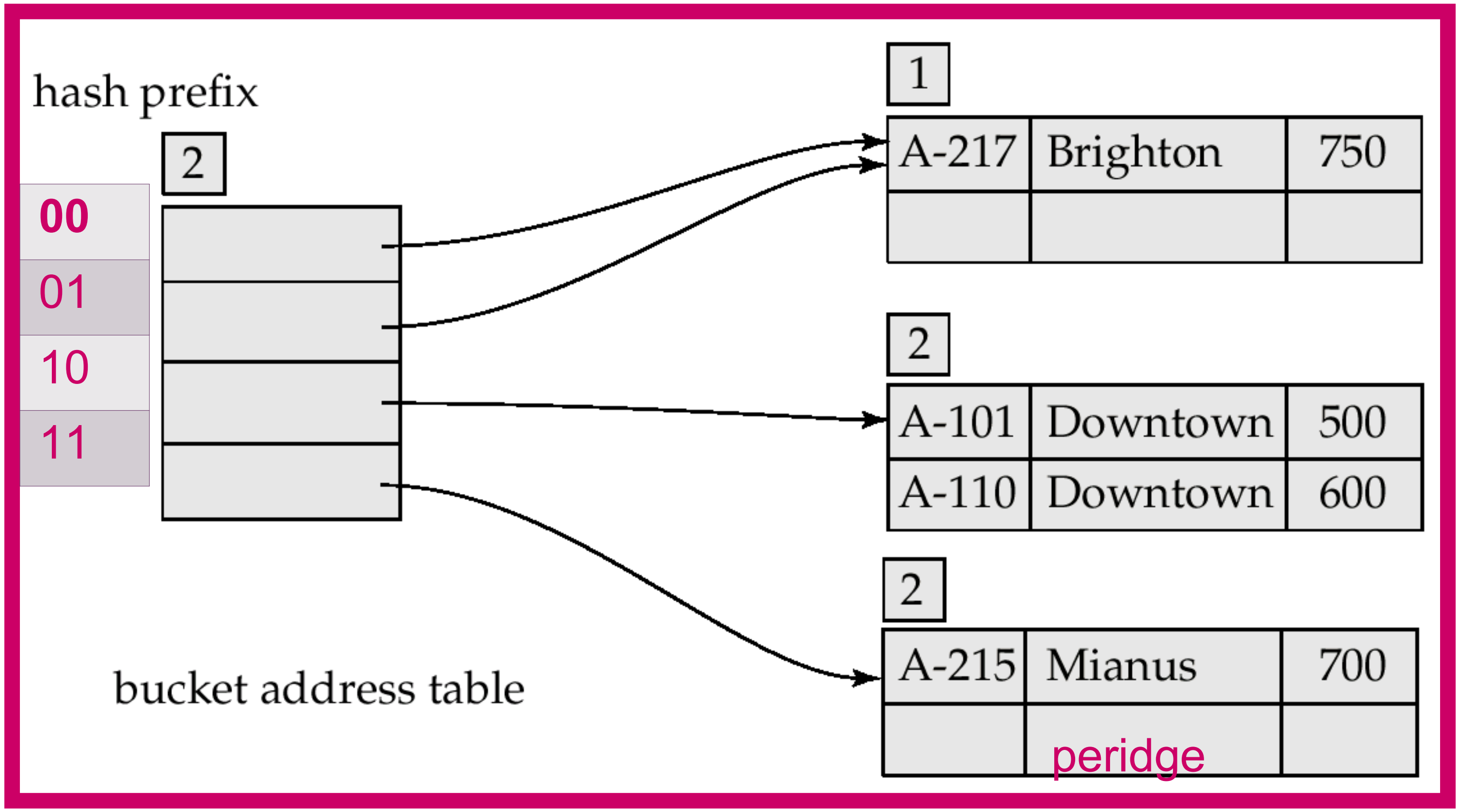
A-217	Brighton	750

A-101	Downtown	500
A-110	Downtown	600

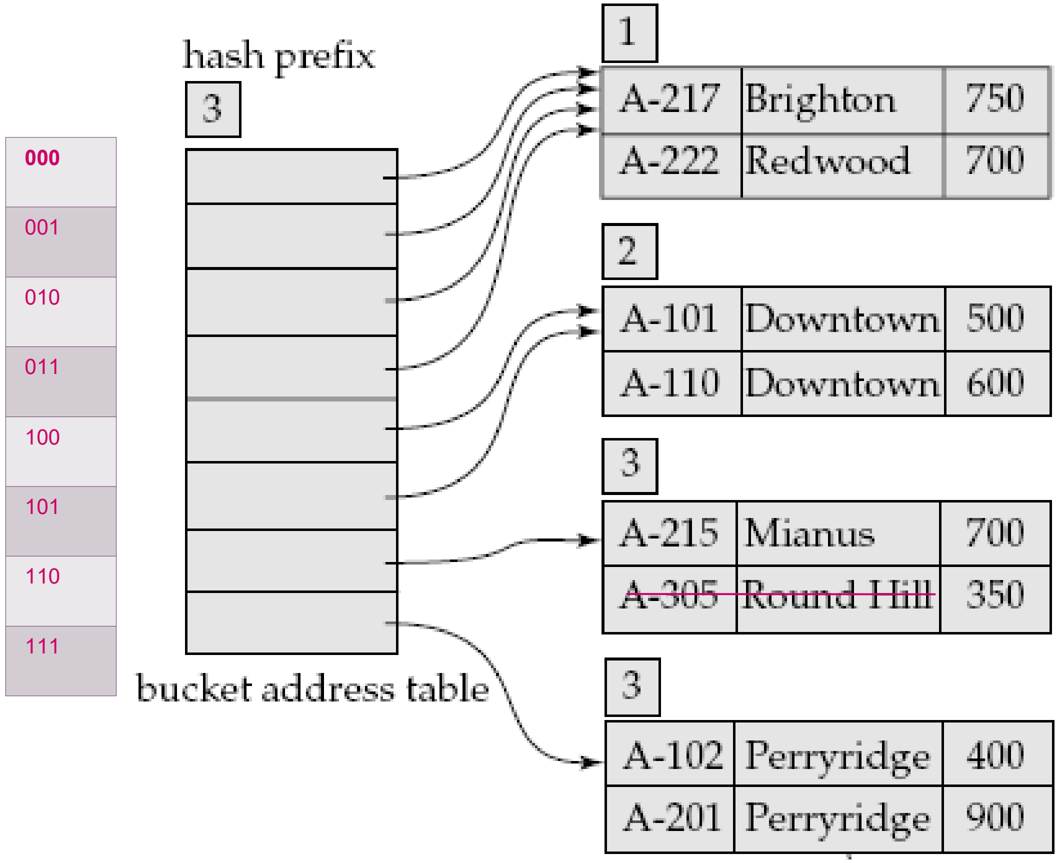
- Next, we insert (A-215, Mianus, 700). Since the first bit of h(Mianus) is 1, we must
- insert this record into the bucket pointed to by the “1” entry in the bucket address
- table. Once again, we find the bucket full and i = i1. We increase the number of bits that we use from the hash to 2
- This increase in the number of bits necessitates
- doubling the size of the bucket address table to four entries

Example (Cont.)

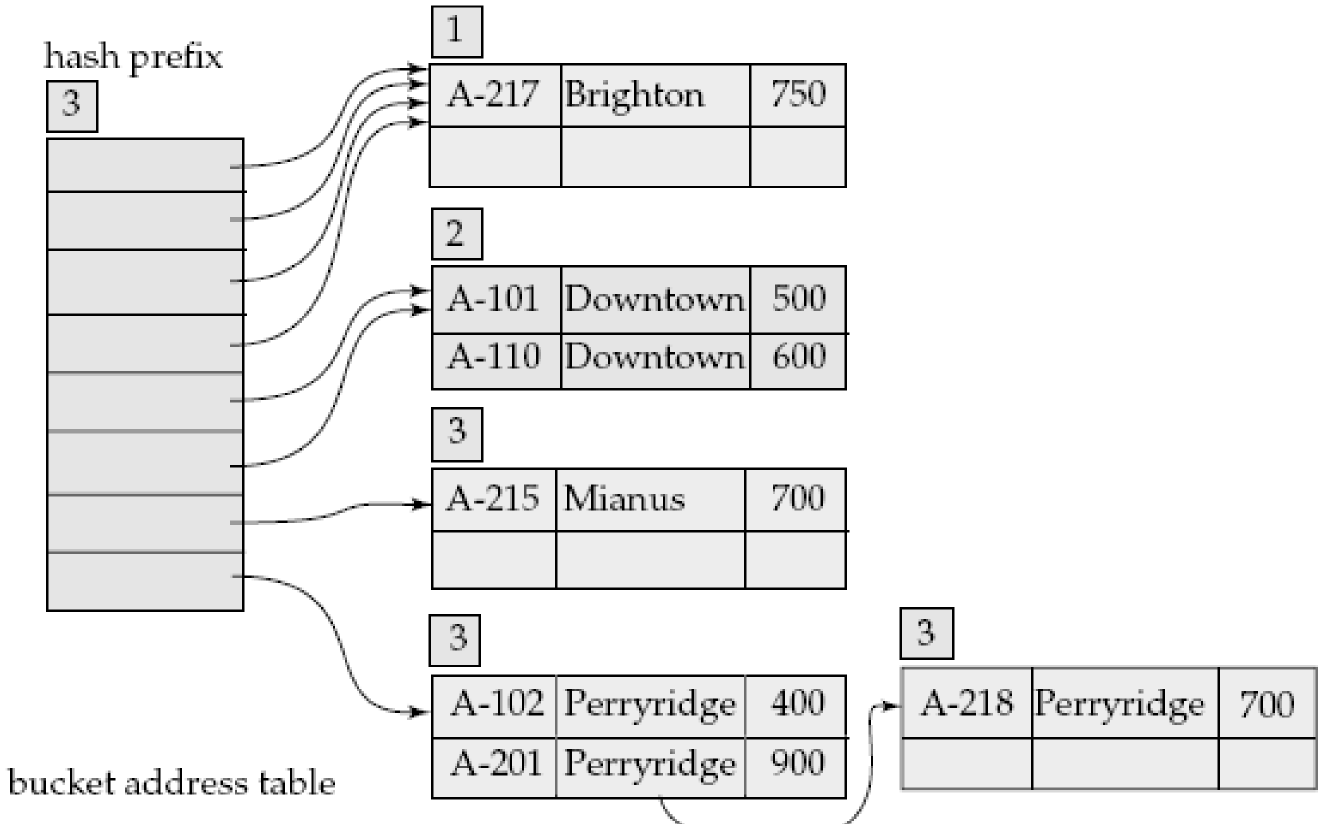
Hash structure after insertion of Mianus record



- Next,we insert (A-102, Perryridge, 400), which goes in the same bucket as Mianus.
- insertion, of (A-201, Perryridge, 900), results in a bucket overflow, leading to an increase in the number of bits, and a doubling of the size of the bucket address table


000
001
010
011
100
101
110
111
hash prefix
3
bucket address table
1
A-217 Brighton 750
A-222 Redwood 700
2
A-101 Downtown 500
A-110 Downtown 600
3
A-215 Mianus 700
~~A-305 Round Hill~~ 350
3
A-102 Perryridge 400
A-201 Perryridge 900

- The insertion of the third Perryridge record, (A-218, Perryridge, 700),
- leads to another overflow. However, this overflow cannot be handled by increasing
- the number of bits, since there are three records with exactly the same hash value.
- Hence the system uses an overflow bucket,


hash prefix
3
bucket address table
1
A-217 Brighton 750
2
A-101 Downtown 500
A-110 Downtown 600
3
A-215 Mianus 700
3
A-102 Perryridge 400
A-201 Perryridge 900
3
A-218 Perryridge 700

- Advantage of dynamic/extensible hashing
  - The main advantage of extendable
  - hashing is that performance does not degrade as the file grows. Furthermore, there is minimal space overhead. Which is for bucket address table
- Disadvantage of dynamic/extensible hashing
  - disadvantage of extendable hashing is that lookup involves an additional levelof indirection, since the system must access the bucket address table before accessing the bucket itself.

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 *balance=1000*.
2. Use index on *balance* to find accounts with balances of $1000; test *branch-name* = "Perryridge".

3

3 Use index on *branch-name* to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance and find record wit balance value $1000*.

Take intersection of both sets of pointers obtained.

- There are many records pertaining to the Perryridge branch.
- There are many records pertaining to accounts with a balance of $1000.
- There are only a few records pertaining to both the Perryridge branch and accounts with a balance of $1000.

If these conditions hold, we must scan a large number of pointers to produce a small result.

Grid File

- Grid file is other alternative of multiple access key.
- A grid structure for queries on two search keys is a 2-dimensional grid, or array, indexed by values for the search keys.
- If we want to access a file on two keys,
  say Dno and Age as in our example, we can construct a grid array with one linear scale (or dimension) for each of the search attributes.
- grid array for the EMPLOYEE file with one linear scale for Dno and another for the Age attribute.
- Dno identify the row and age is search key value is used to find column so that each cell corresponding to bucket address.

Dept =2 and
age=25

sfsdfs

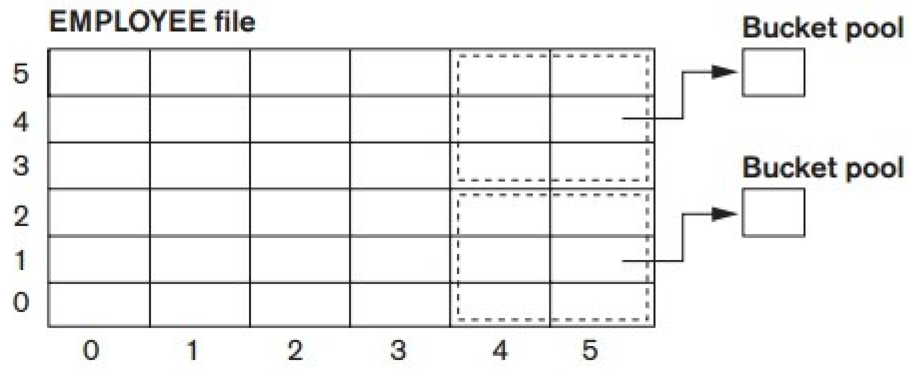
- our request for Dno = 4 and Age = 59 maps into the cell (1, 5) corresponding to the grid array
- The records for this combination will be found in the correspond-ing bucket.

Figure 18.14
Example of a grid array on Dno and Age attributes.

Dno

0	1, 2
1	3, 4
2	5
3	6, 7
4	8
5	9, 10

Linear scale
for Dno



Linear Scale for Age

0	1	2	3	4	5
< 20	21–25	26–30	31–40	41–50	> 50

End of Chapter