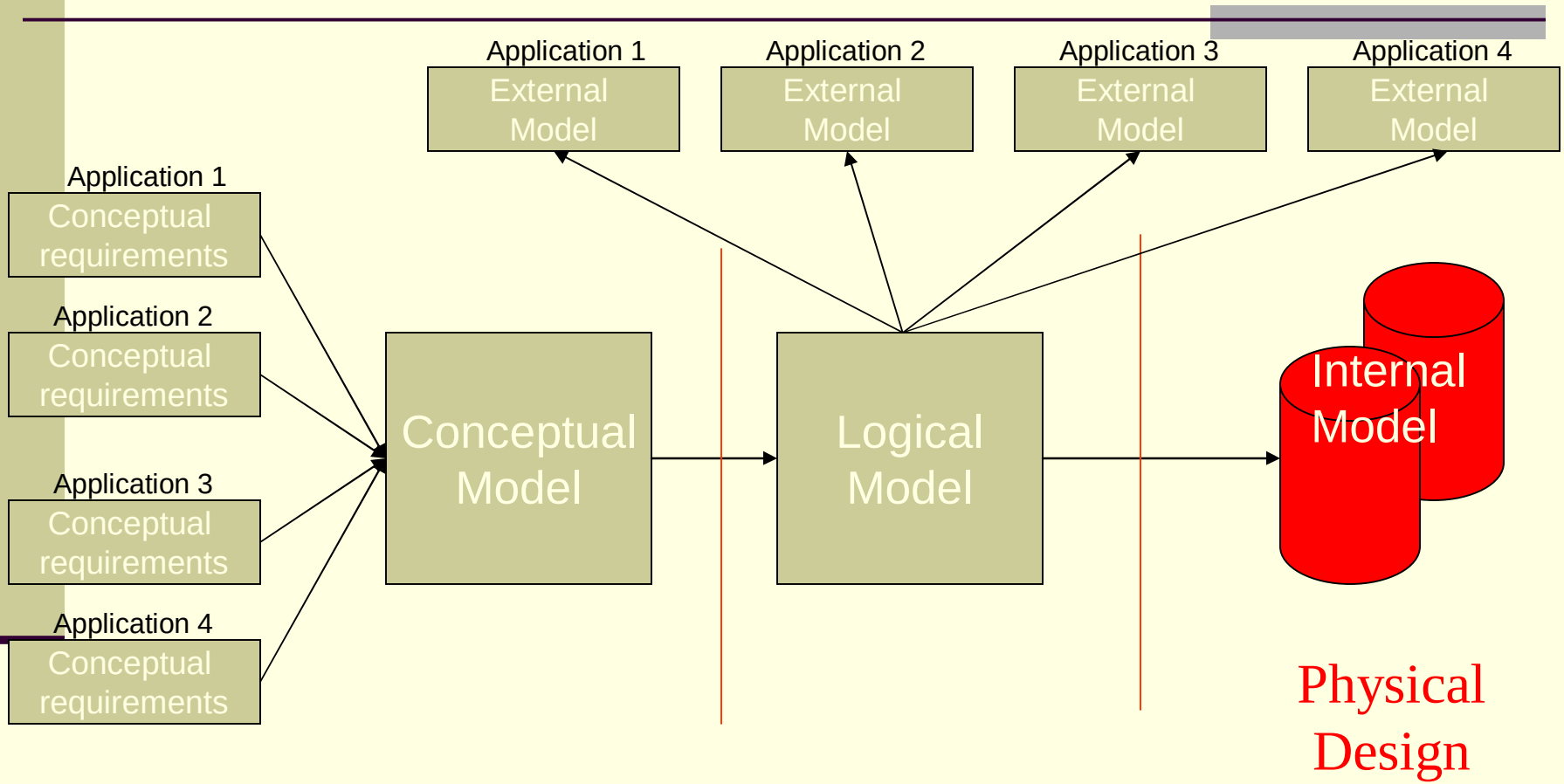
Data Storage and Access Methods

Dr. Ajanta De Sarkar
Reader, CSE Dept.
BIT Mesra, Kolkata Campus

Database Design Process



Physical Database Design

- The primary goal of physical database design is *data processing efficiency*
- We will concentrate on choices often available to optimize performance of database services
- Physical Database Design requires information gathered during earlier stages of the design process

Physical Design Information

- Information needed for physical file and database design includes:
  - Normalized relations plus size estimates for them
  - Definitions of each attribute
  - Descriptions of where and when data are used
    - entered, retrieved, deleted, updated, and how often
  - Expectations and requirements for response time, and data security, backup, recovery, retention and integrity
  - Descriptions of the technologies used to implement the database

Physical Design Decisions

- There are several critical decisions that will affect the integrity and performance of the system
  - Storage Format
  - Physical record composition
  - Data arrangement
  - Indexes
  - Query optimization and performance tuning

Storage Format

- Choosing the storage format of each *field* (attribute). The DBMS provides some set of data types that can be used for the physical storage of fields in the database

- Data Type (format) is chosen to minimize storage space and maximize data integrity

Objectives of data type selection

- Minimize storage space
- Represent all possible values
- Improve data integrity
- Support all data manipulations
- The correct data type **should**, in minimal space, represent every possible value (but eliminate illegal values) for the associated attribute *and* can support the required data manipulations (e.g. numerical or string operations)

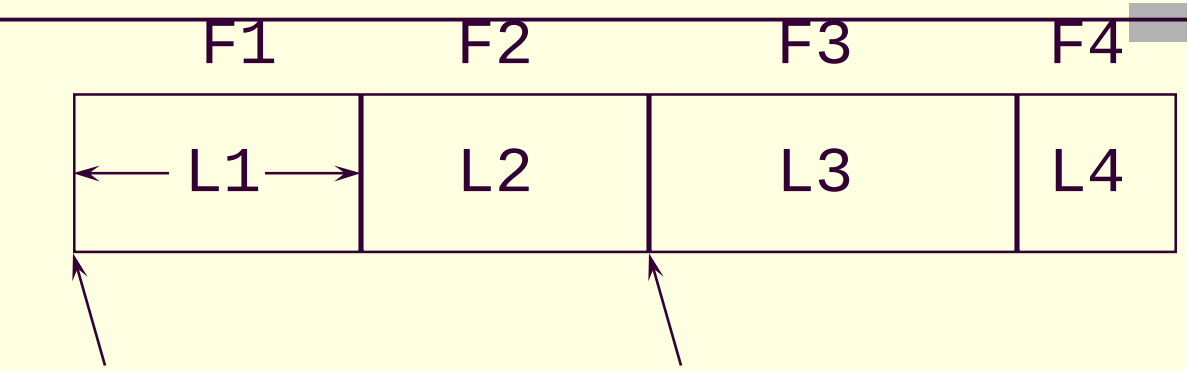
Designing Physical Records

- A physical record is a group of fields stored in adjacent memory locations and retrieved together as a unit
- Fixed Length and variable fields

Data Storage

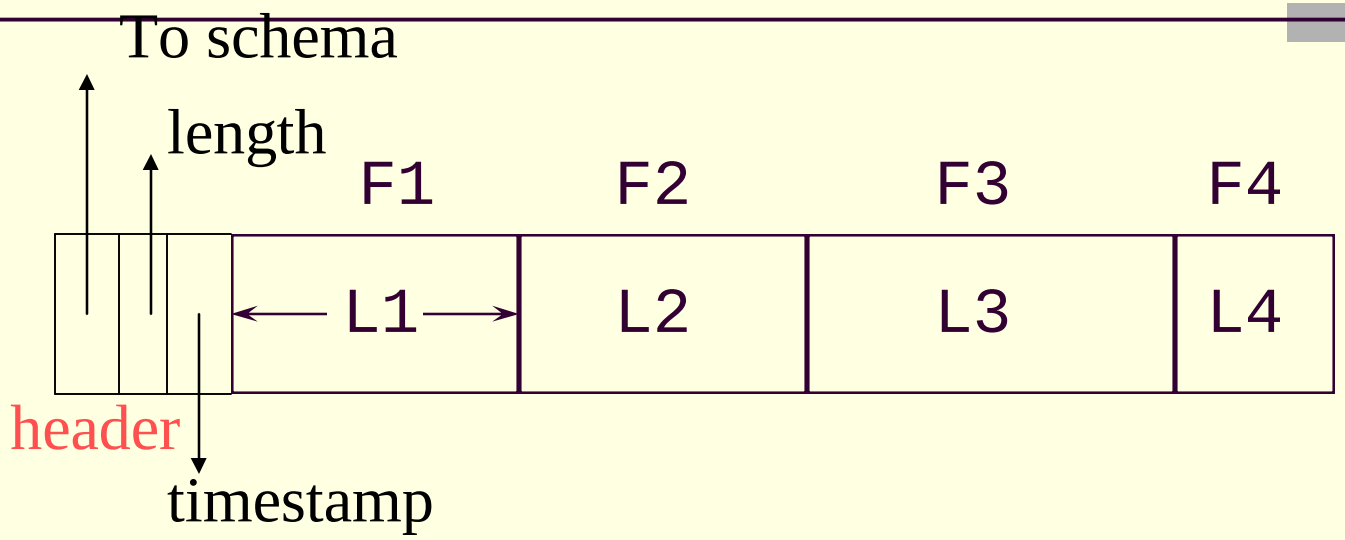
- Storing Data: Disks
- Buffer manager
- Representing relational data in a disk

Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.
- **Note the importance of schema information!**

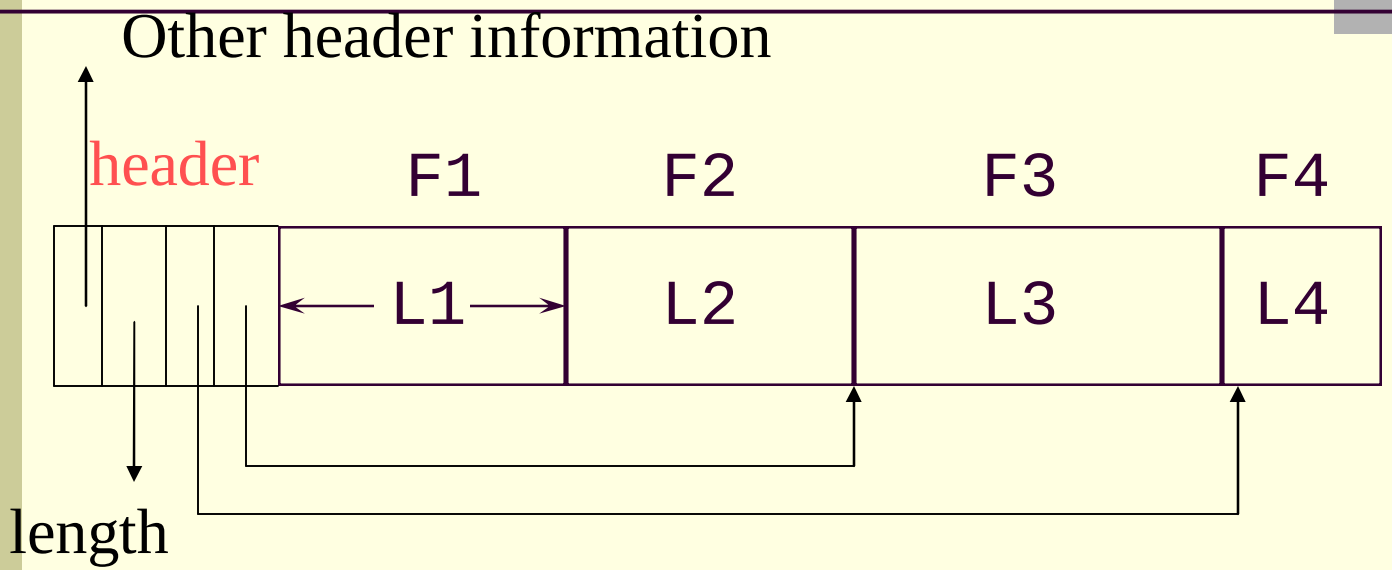
Record Header



Need the header because:

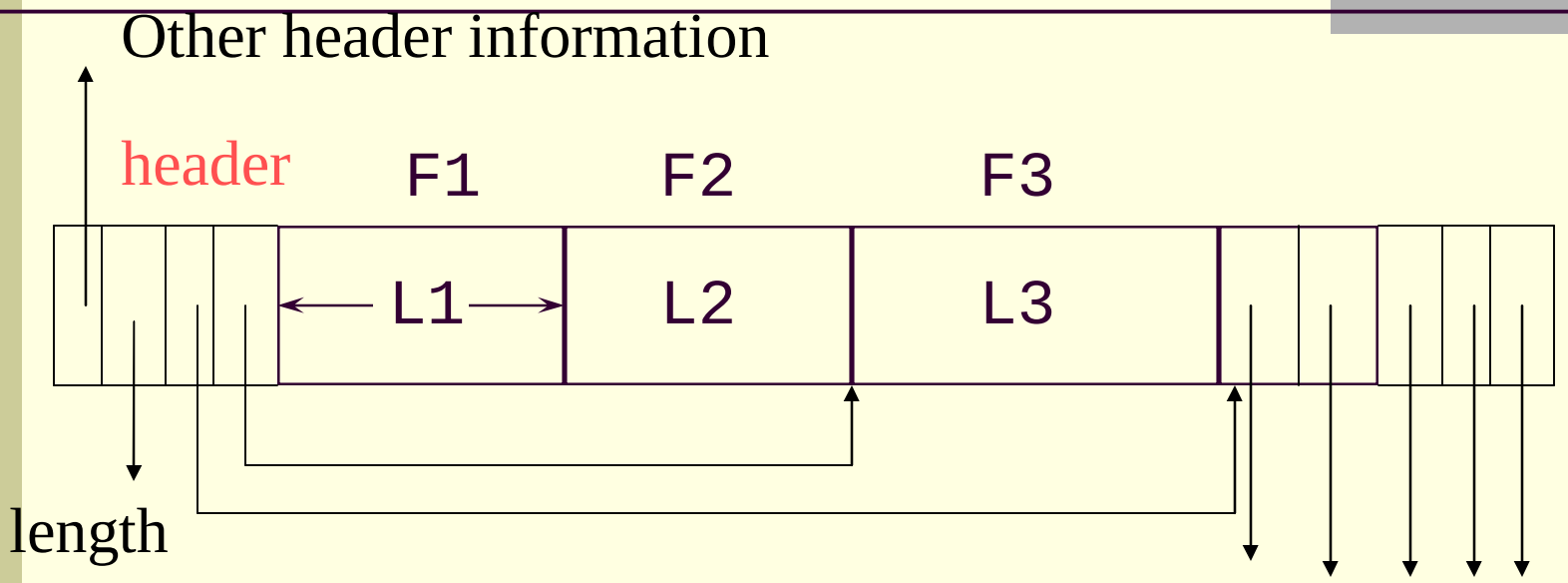
- The schema may change
  for a while new+old may coexist
- Records from different relations may coexist

Variable Length Records



Place the fixed fields first: F1, F2
Then the variable length fields: F3, F4
Null values take 2 bytes only
Sometimes they take 0 bytes (when at the end)

Records With Referencing Fields

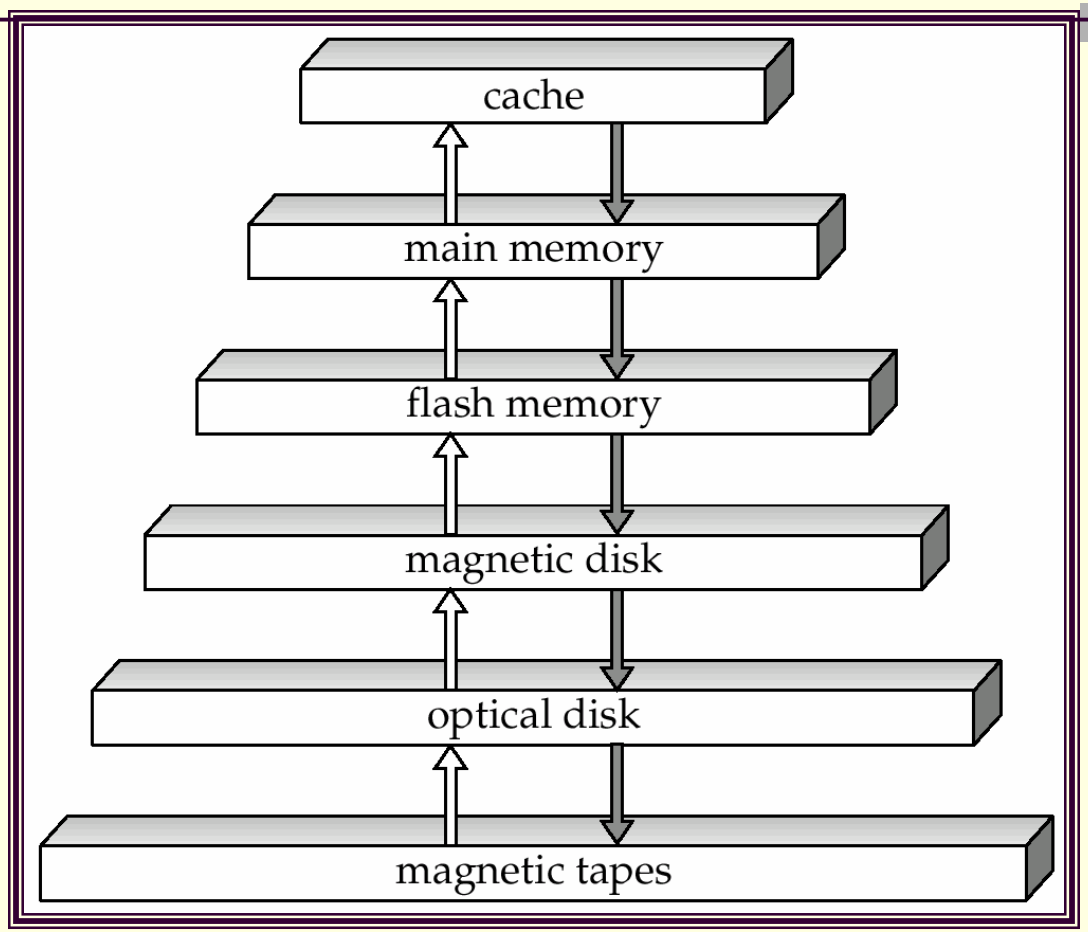


E.g. to represent one-many or many-many relationships

BLOB

- Binary large objects
- Supported by modern database systems
- E.g. images, sounds, etc.
- Storage: attempt to cluster blocks together

Storage Hierarchy



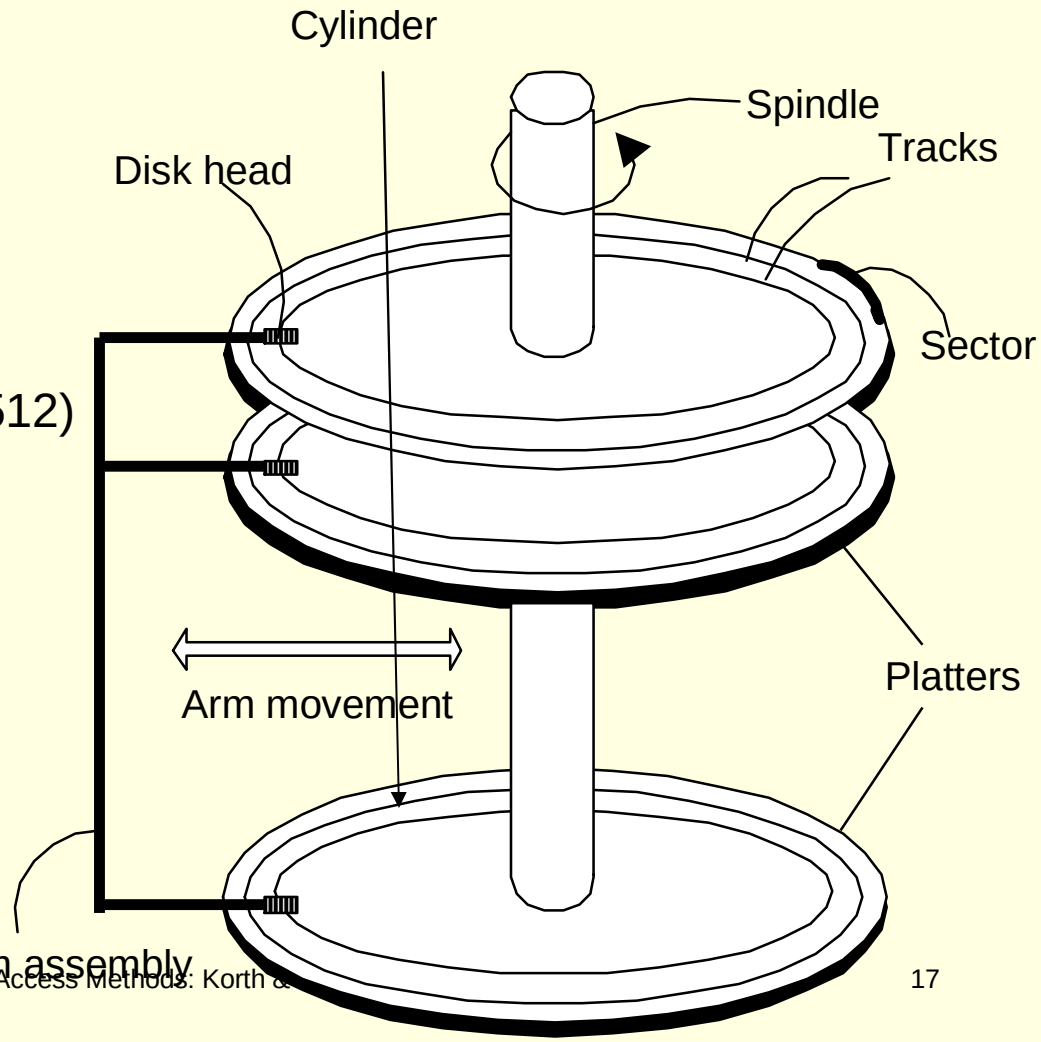
Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage

The Mechanics of Disk

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platters (1-30)
- Number of tracks (<=10000)
- Number of sectors (256/track)
- Number of bytes / sector (2^9=512)
- Block size (2^{12}=4096)



Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - Average latency is 1/2 of the worst case latency.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 4 to 8 MB per second is typical
  - Multiple disks may share a controller, so rate that controller can handle is also important
    - E.g. ATA-5: 66 MB/second, SCSI-3: 40 MB/s
    - Fiber Channel: 256 MB/s

Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a
    "theoretical MTTF" of 30,000 to 1,200,000 hours for a new disk
    - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
  - MTTF decreases as disk ages

Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.
- Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.

Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  1. If the block is already in the buffer, the requesting program is given the address of the block in main memory
  2. If the block is not in the buffer,
     1. the buffer manager allocates space in the buffer for the block, replacing (throwing out) some other block, if required, to make space for the new block.
     2. The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
     3. Once space is allocated in the buffer, the buffer manager reads the block from the disk to the buffer, and passes the address of the block in main memory to requester.

Important Disk Access Characteristics

- Blocking factor for a file is the average number of records stored in a disk block.

- Block access time = Disk latency + transfer time
- Disk latency = seek time + rotational latency
- Seek time = time for the head to reach the right track
  - 10ms – 40ms
- Rotational latency = rotation time to get to the right sector
  - Time for one rotation = 10ms
  - Average rotation latency = 10ms/2
- Transfer time = typically 5-10MB/s
- Disks read/write one block at a time (typically 4kB)

ACCESS METHODS

When we design a file, the important issue is how we will retrieve information (a specific record) from the file. Sometimes we need to process records one after another, whereas sometimes we need to access a specific record quickly without retrieving the preceding records. The access method determines how records can be retrieved: **sequentially** or **randomly**.

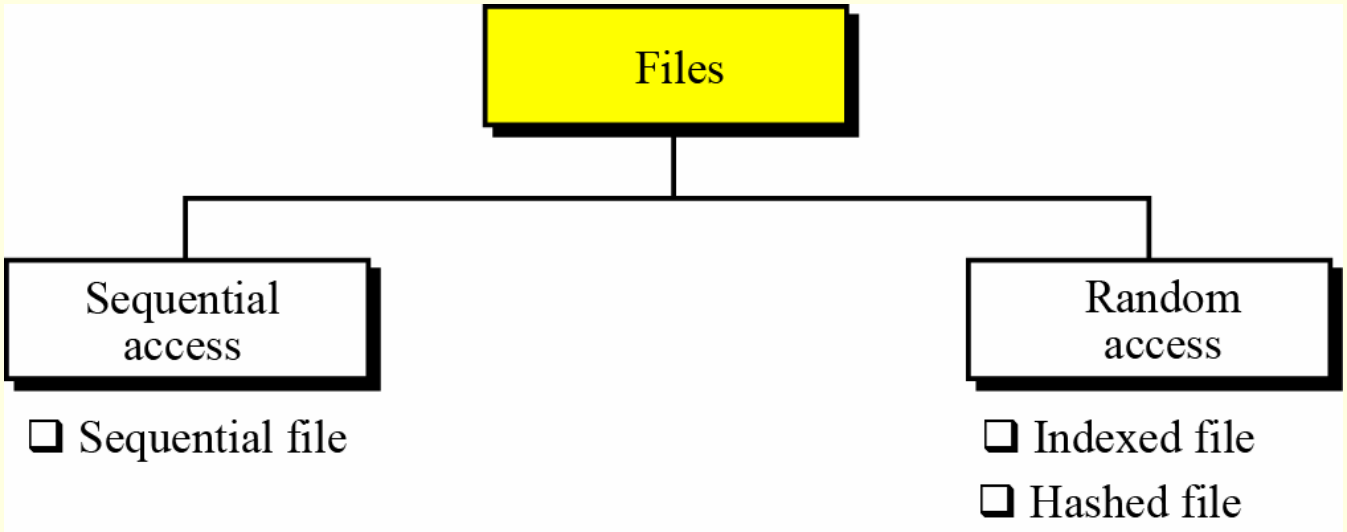
Sequential access

If we need to access a file sequentially—that is, one record after another, from beginning to end—we use a **sequential file structure**.

Random access

If we need to access a specific record without having to retrieve all records before it, we use a file structure that allows random access. Two file structures allow this: indexed files and hashed files. This taxonomy of file structures is shown in Figure.

A taxonomy of file structures

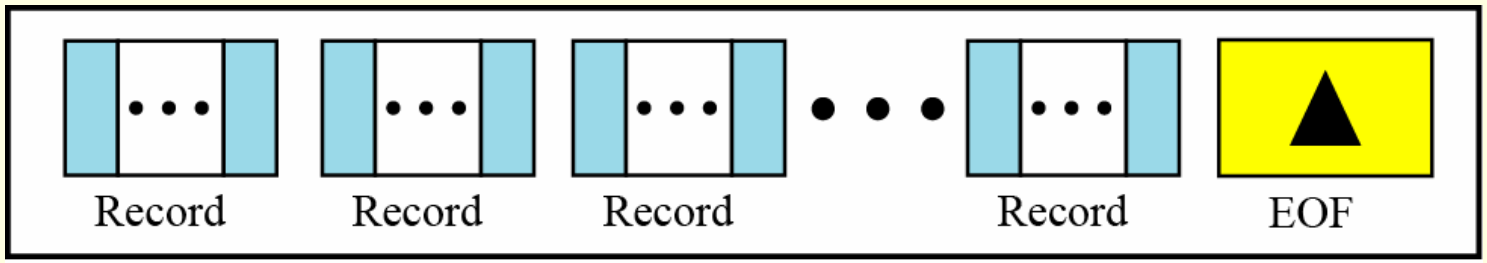


A taxonomy of file structures

SEQUENTIAL FILES

A sequential file is one in which records can only be accessed one after another from beginning to end. The next Figure shows the layout of a sequential file. Records are stored one after another in auxiliary storage, such as tape or disk, and there is an EOF (end-of-file) marker after the last record. The operating system has no information about the record addresses, it only knows where the whole file is stored. The only thing known to the operating system is that the records are sequential.

SEQUENTIAL FILES



Sequential file

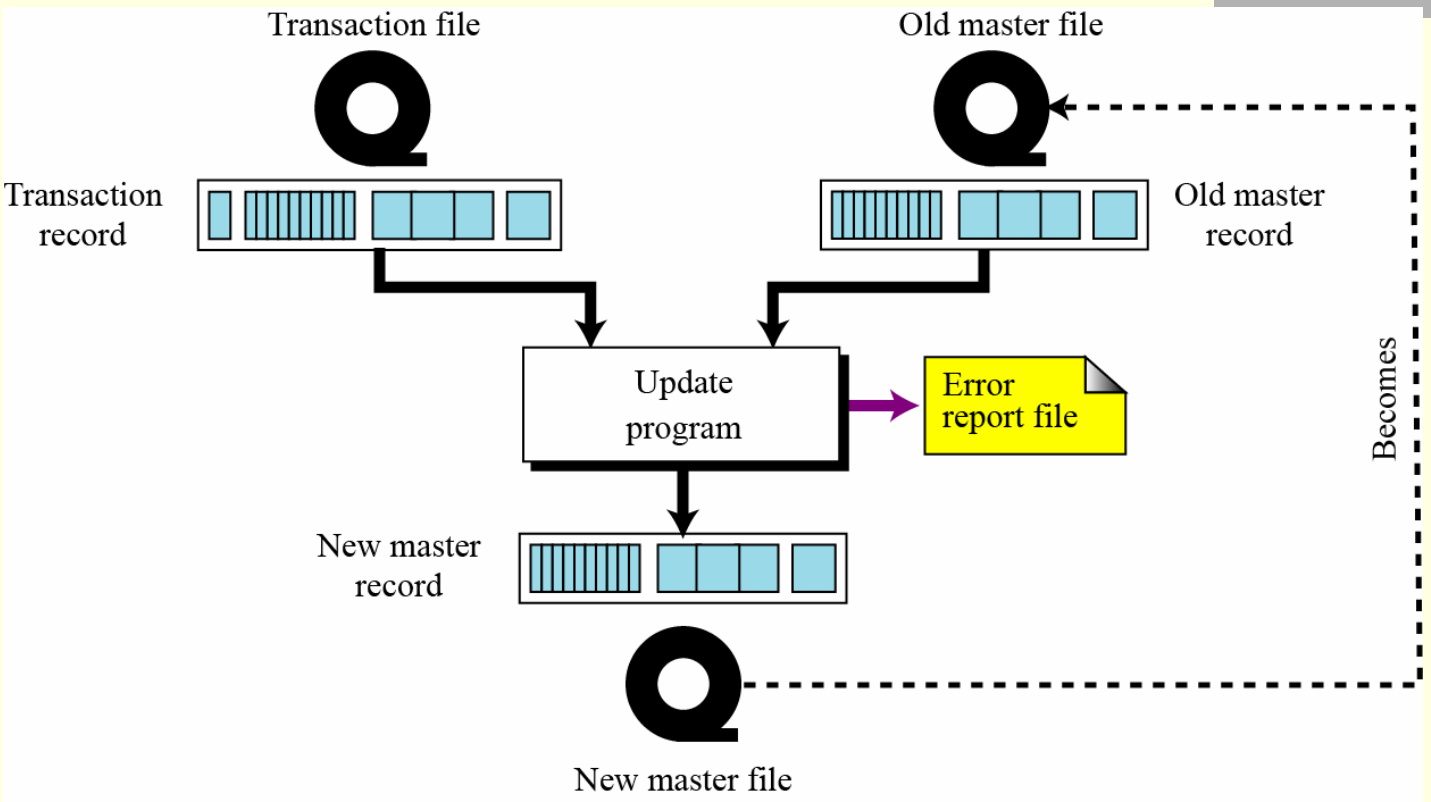
Updating sequential files

Sequential files must be updated periodically to reflect changes in information. The updating process is very much involved because all the records need to be checked and updated (if necessary) sequentially.

Files involved in updating

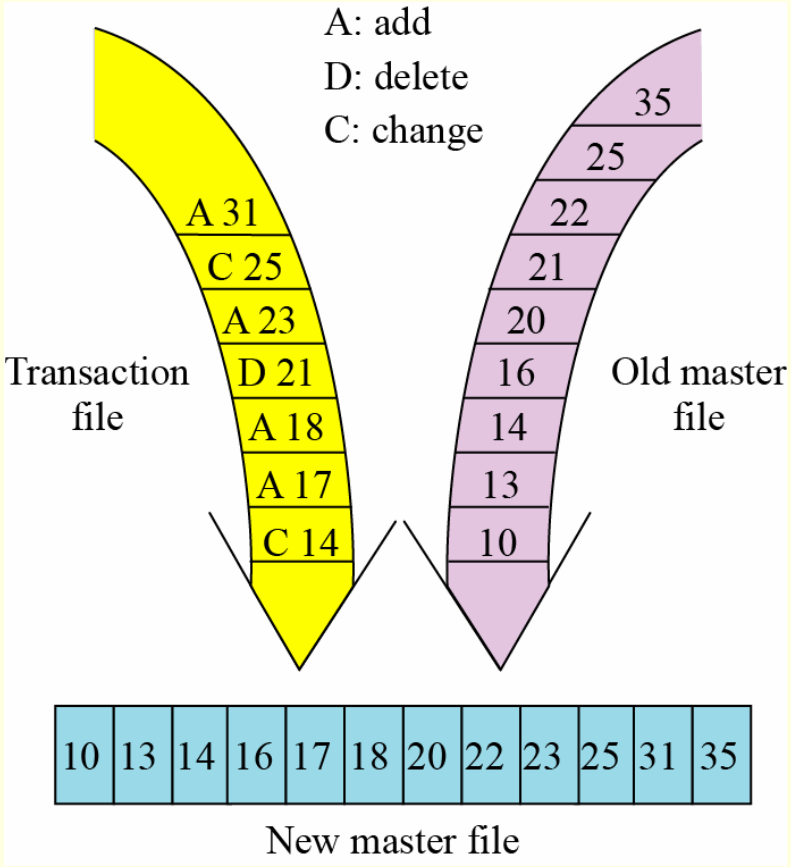
There are four files associated with an update program: the **new master file**, the **old master file**, the **transaction file** and the **error report file**. All these files are sorted based on key values. Next figure is a pictorial representation of a sequential file update.

Updating a sequential file

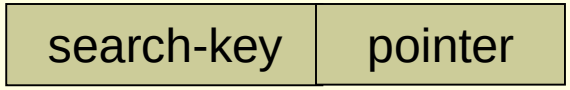


Processing file updates

To make the updating process efficient, all files are sorted on the same key. This updating process is shown in Figure



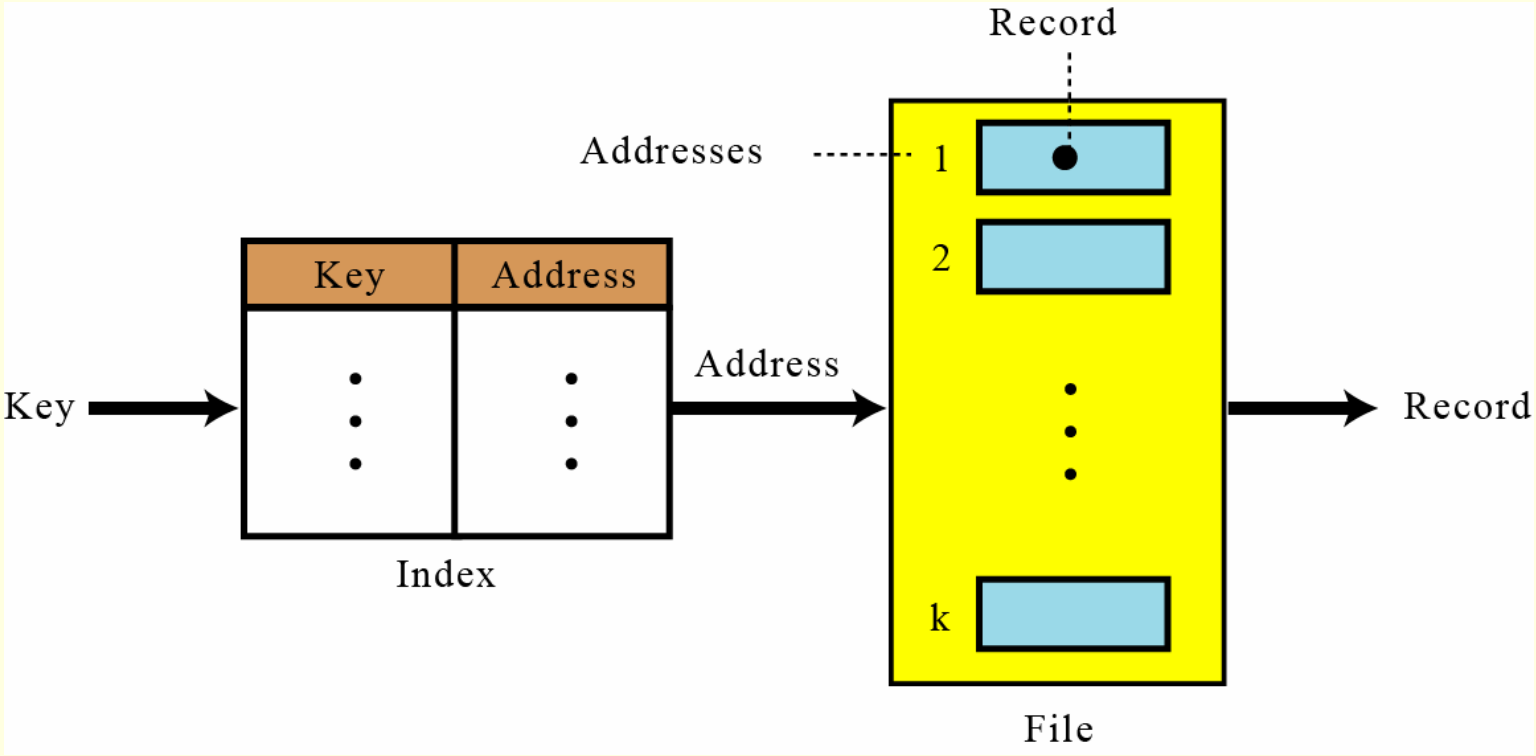
Indexing

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **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 

  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".

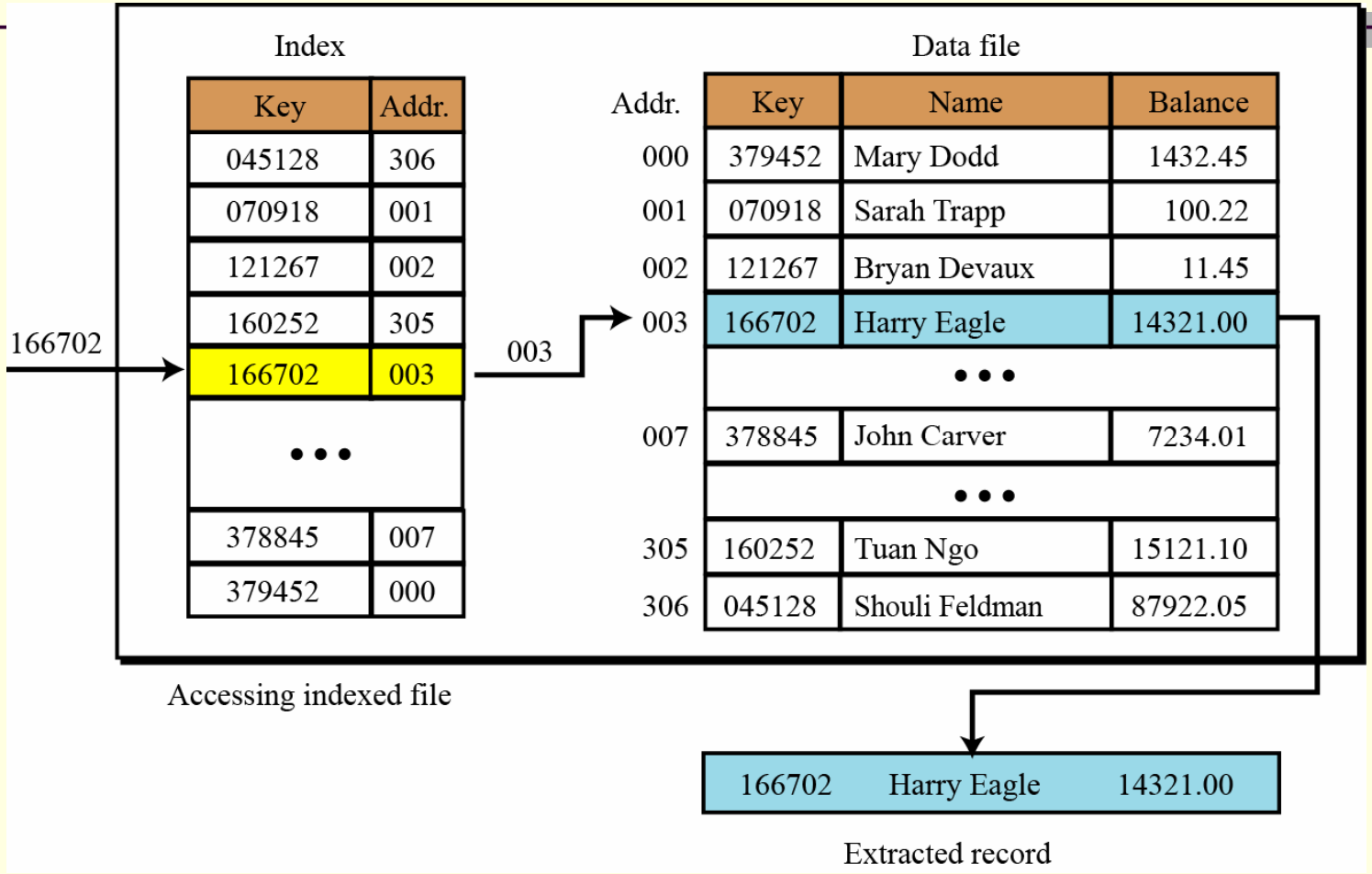
INDEXED FILES

To access a record in a file randomly, we need to know the address of the record.



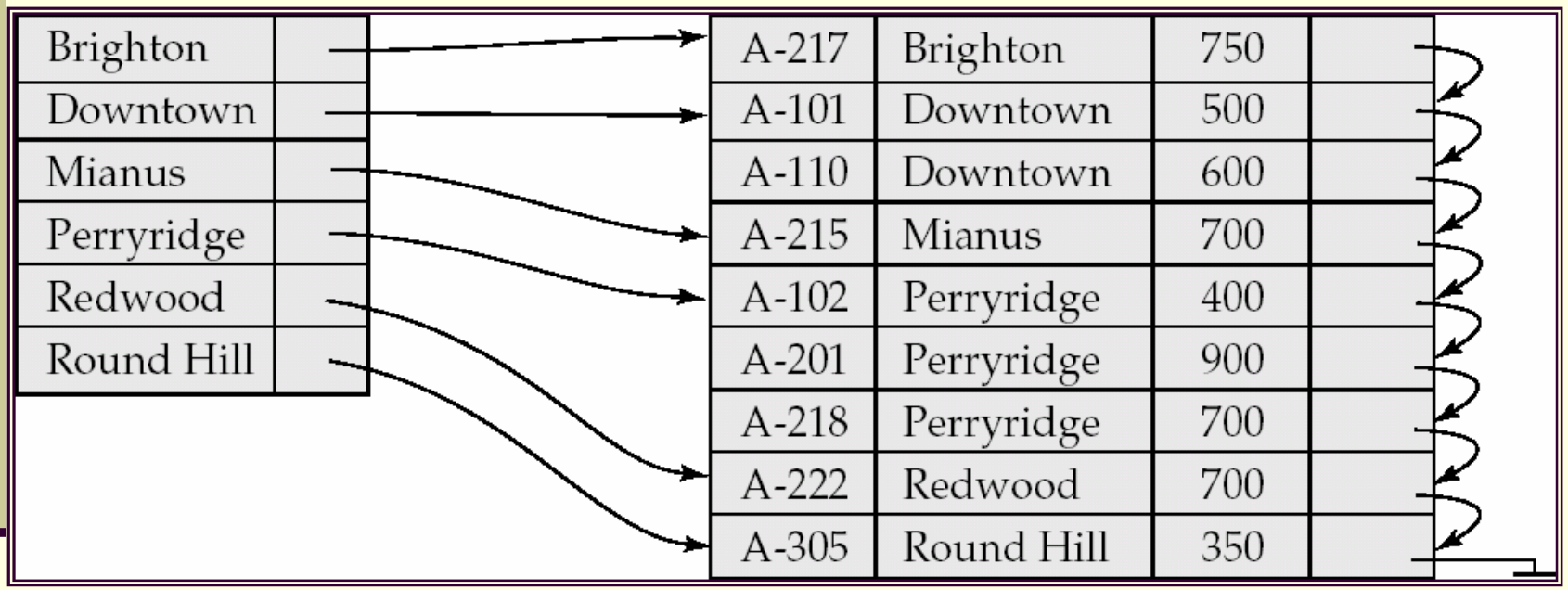
Mapping in an indexed file

Logical view of an indexed file



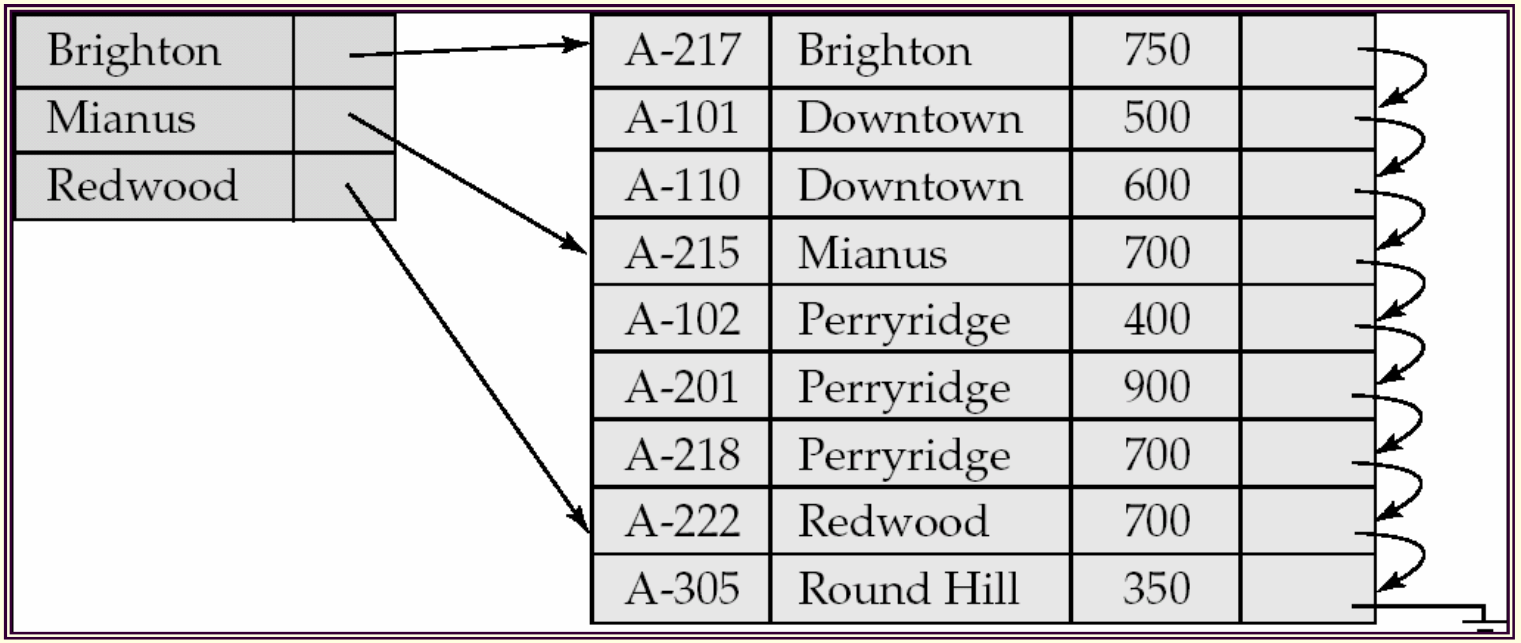
Dense Index Files

- Dense index — Index record appears



Sparse Index Files

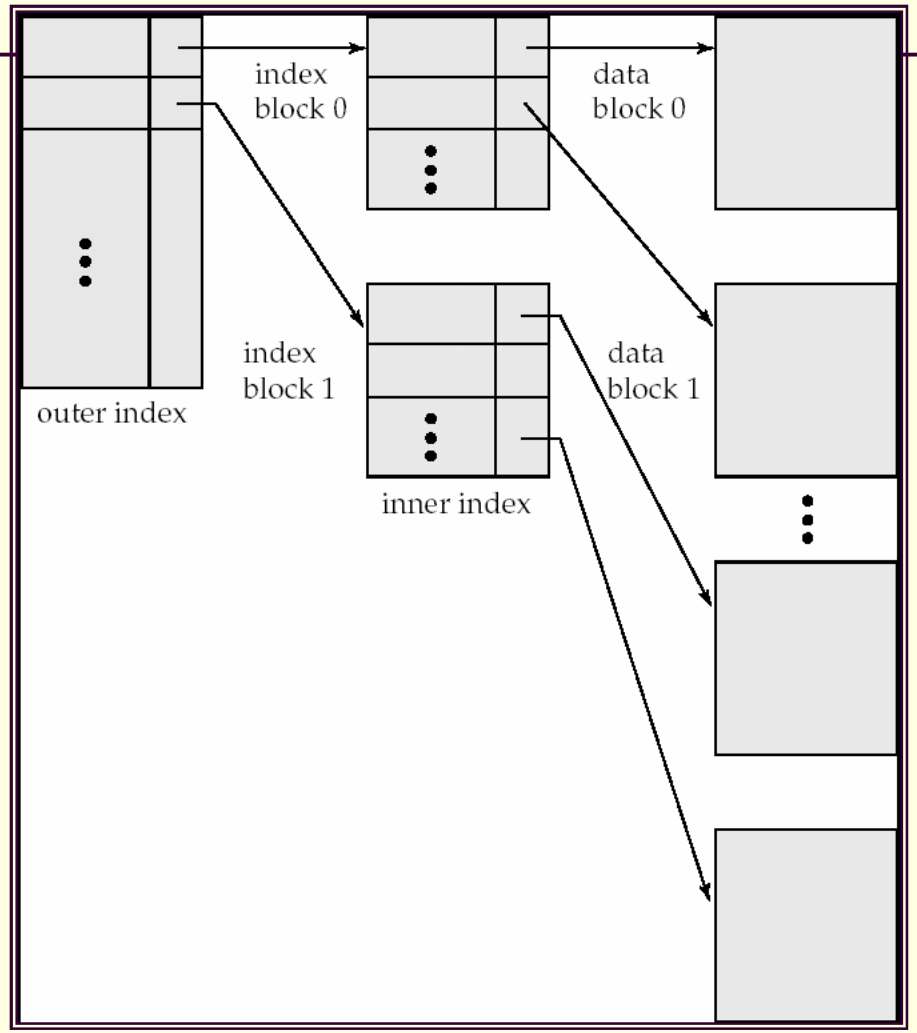
- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points



Multilevel Index

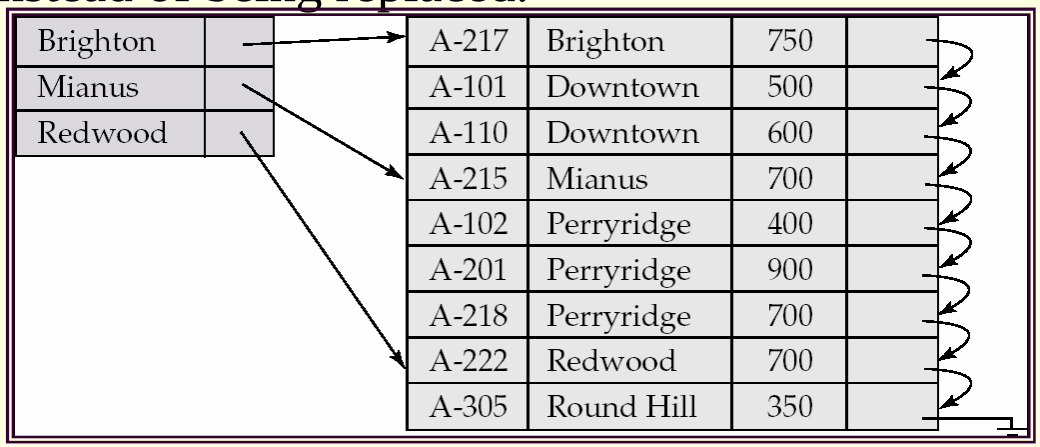
- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: Record 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: similar to file record deletion.
  - **Sparse indices** –
    - if deleted key value exists in the index, the value is replaced by 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.



Index Update: Record Insertion

- Single-level index insertion:
  - Perform a lookup using the key value from inserted record
  - **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.
    - If a new block is created, 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

B^+-Tree Index Files

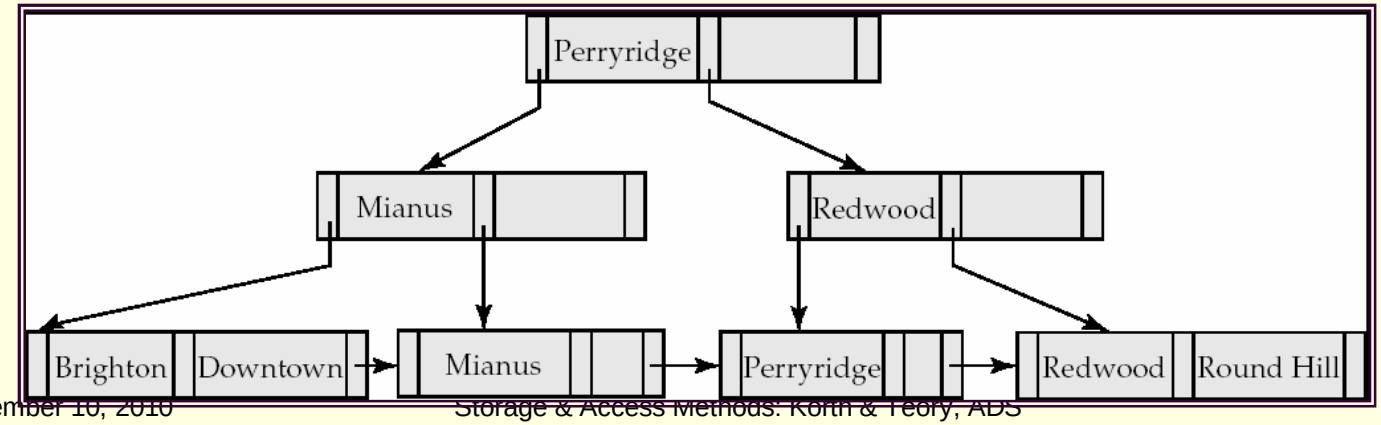
B^+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B^+-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B^+-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B^+-trees outweigh disadvantages
  - B^+-trees are used extensively

B+-Tree Index Files (Cont.)

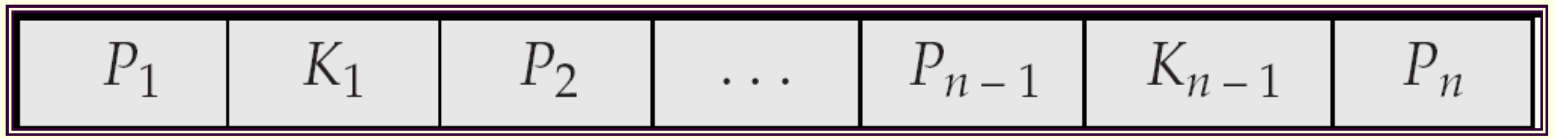
A B+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.



B^+-Tree Node Structure

- Typical node



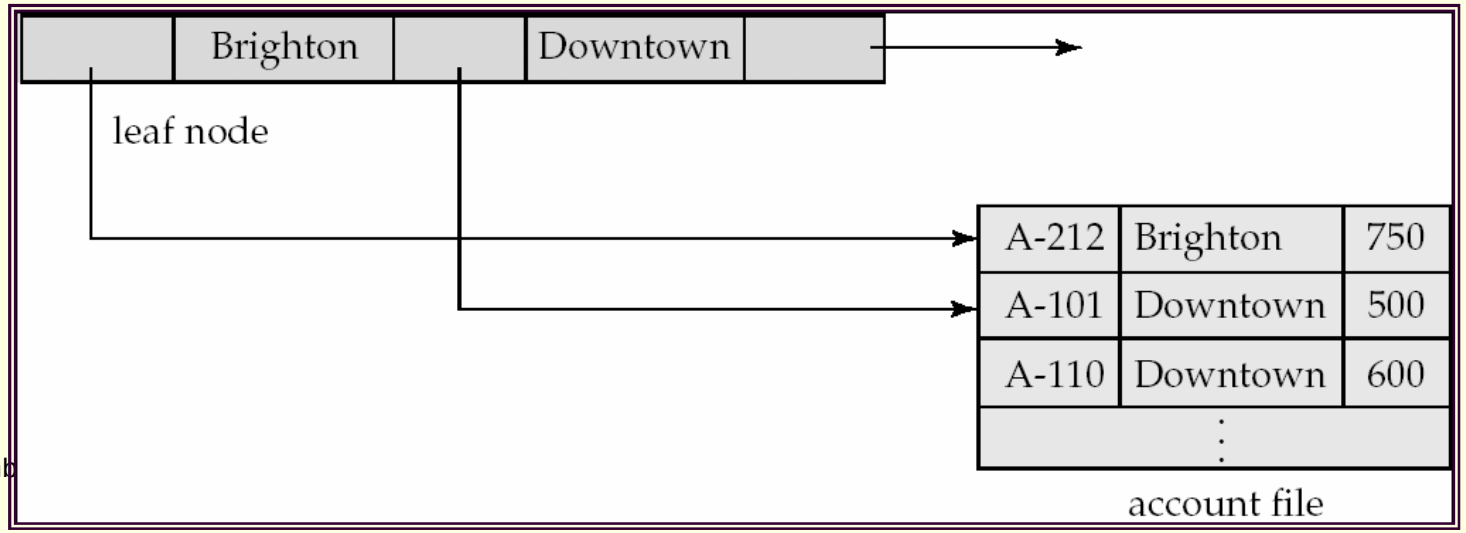
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Leaf Nodes in B^+-Trees

Properties of a leaf node:

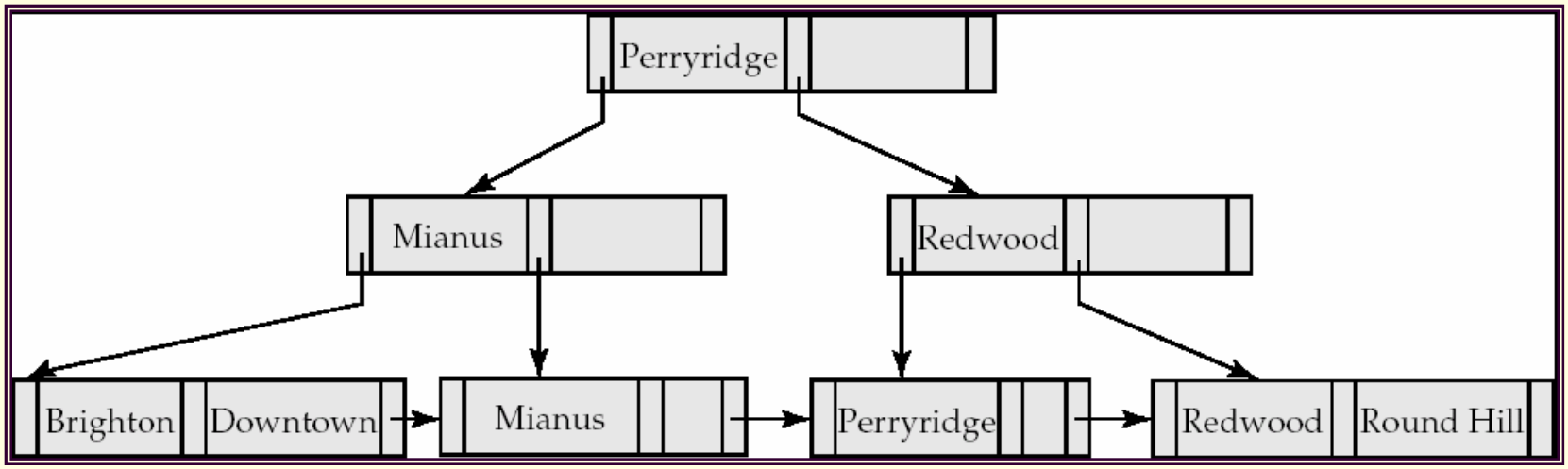
- For $i = 1, 2, \ldots, n-1$, pointer P_i either points to a file record with search-key value K_i, or to a bucket of pointers to file records, each record having search-key value K_i. Only need bucket structure if search-key does not form a primary key.
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than L_j's search-key values
- P_n points to next leaf node in search-key order



Non-Leaf Nodes in B$^+$-Trees

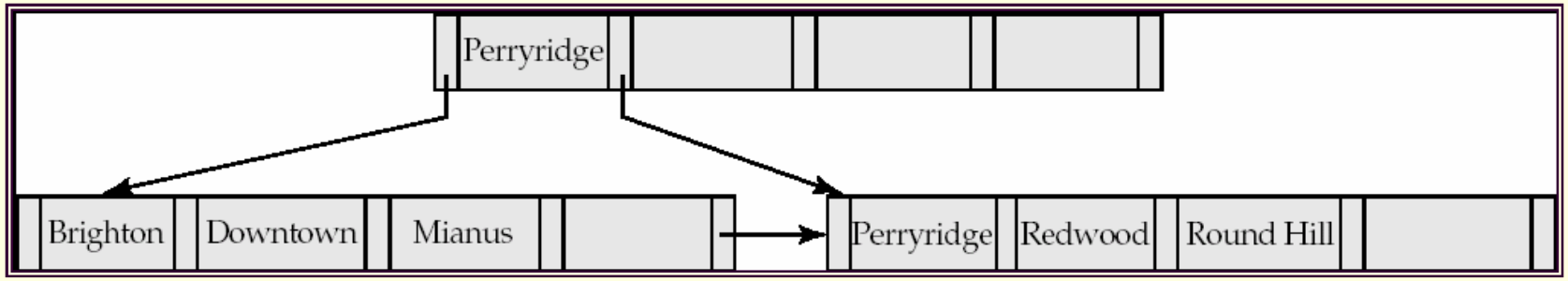
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}

Example of a B$^+$-tree



B$^+$-tree for *account* file (n = 3)

Example of B$^+$-tree



B$^+$-tree for *account* file (n = 5)

- Leaf nodes must have between 2 and 4 values ($\lceil (n–1)/2 \rceil$ and n –1, with n = 5).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil (n/2 \rceil$ and n with n =5).
- Root must have at least 2 children.

Observations about B^+-trees

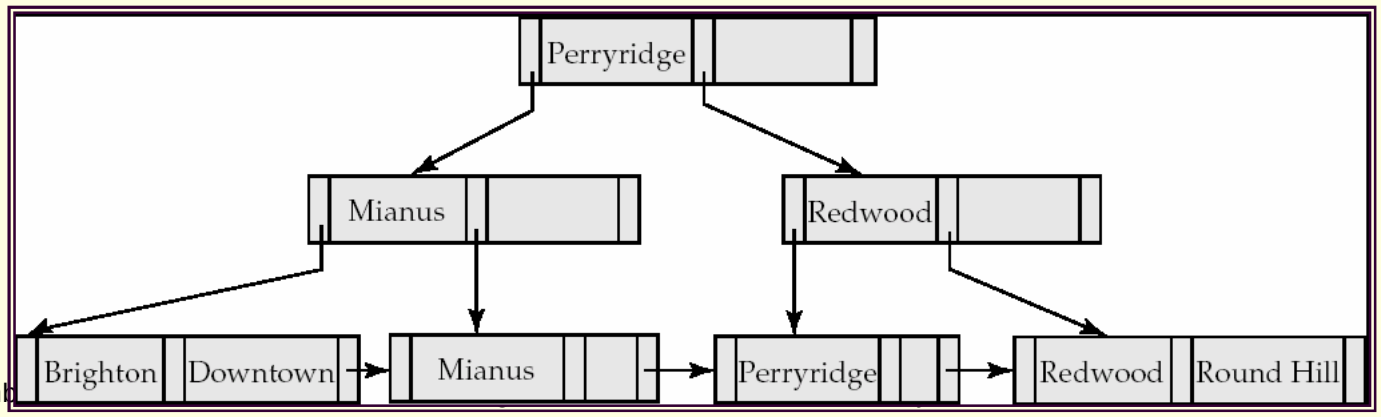
- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
  - Level below root has at least $2*\lceil n/2 \rceil$ values
  - Next level has at least $2*\lceil n/2 \rceil * \lceil n/2 \rceil$ values
  - .. etc.
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Queries on B^+-Trees

- Find all records with a search-key value of k.

1. N=*root*
2. Repeat
   1. Examine N for the smallest search-key value > k.
   2. If such a value exists, assume it is K_i. Then set $N = P_i$
   3. Otherwise $k \geq K_{n-1}$. Set $N = P_n$

   Until N is a leaf node
3. If for some i, key $K_i = k$ follow pointer P_i to the desired record or bucket.
4. Else no record with search-key value k exists.



Queries on B^{+}-Trees (Cont.)

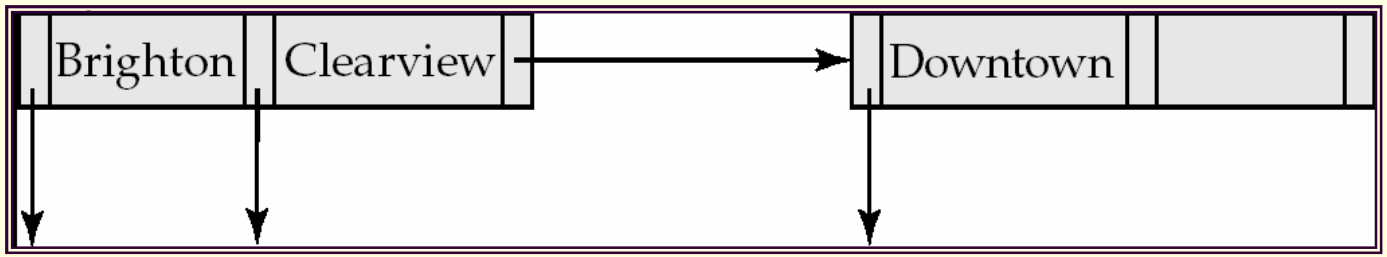
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B^+-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
   1. Add record to the file
3. If the search-key value is not present, then
   1. add the record to the main file (and create a bucket if necessary)
   2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
   3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

Updates on B$^+$-Trees: Insertion (Cont.)

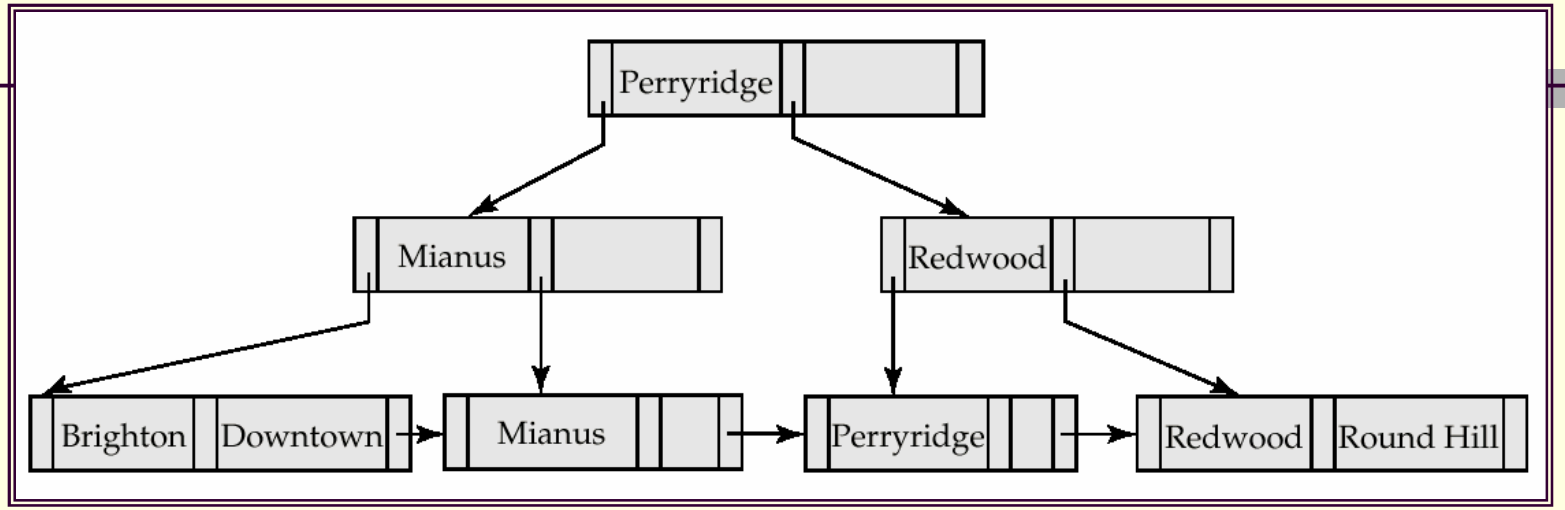
- Splitting a leaf node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height

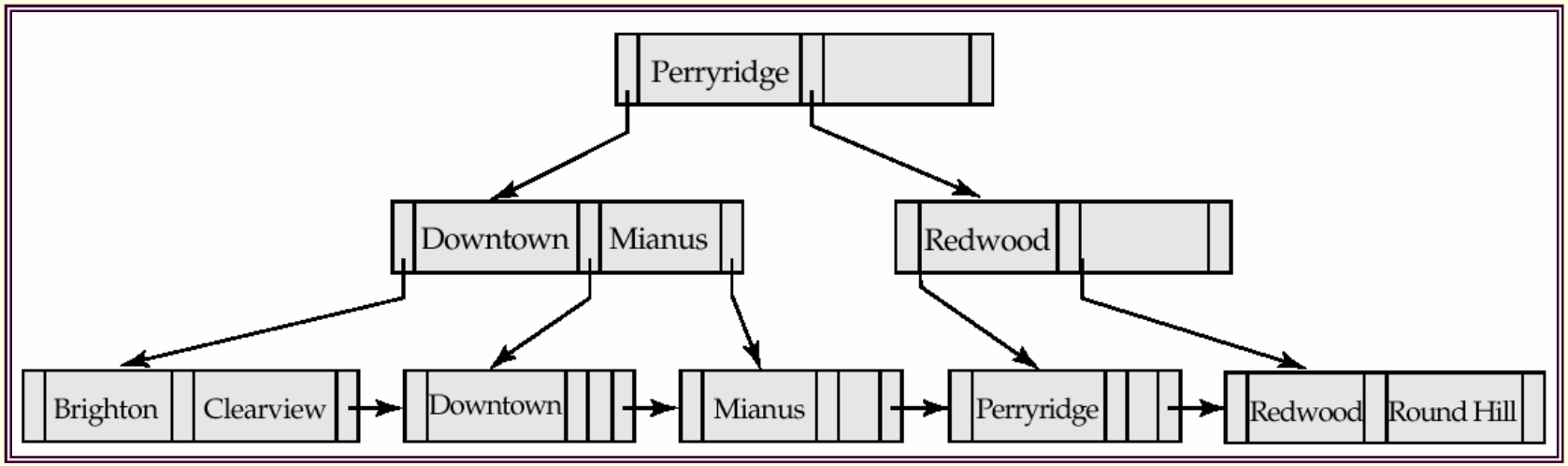


Result of splitting node containing Brighton and Downtown on inserting Clearview

Next step: insert entry with (Downtown,pointer-to-new-node) into parent

Updates on B+-Trees: Insertion (Cont.)

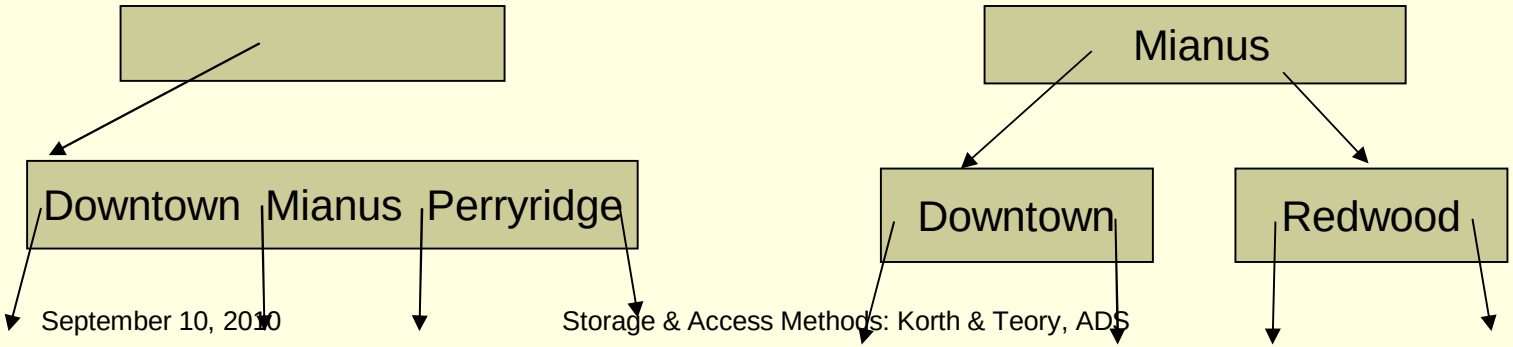




B+-Tree before and after insertion of "Clearview"

Insertion in B^+-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1,K_1, \ldots, K_{\lceil n/2 \rceil -1},P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil +1},K_{\lceil n/2 \rceil +1},\ldots,K_n,P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil},N')$ into parent N
- **Read pseudocode in book!**



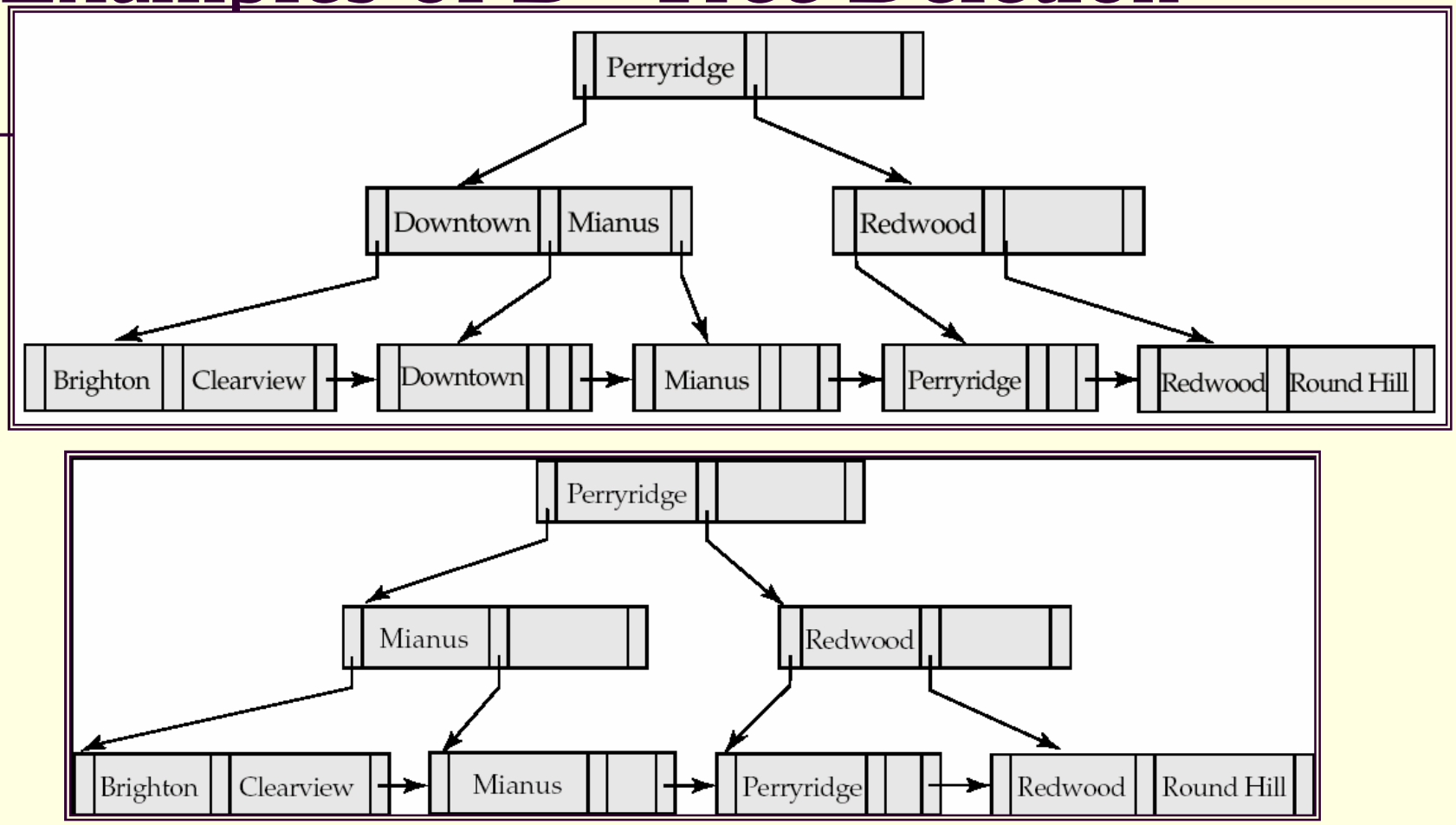
Updates on B^+-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

Updates on B^+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

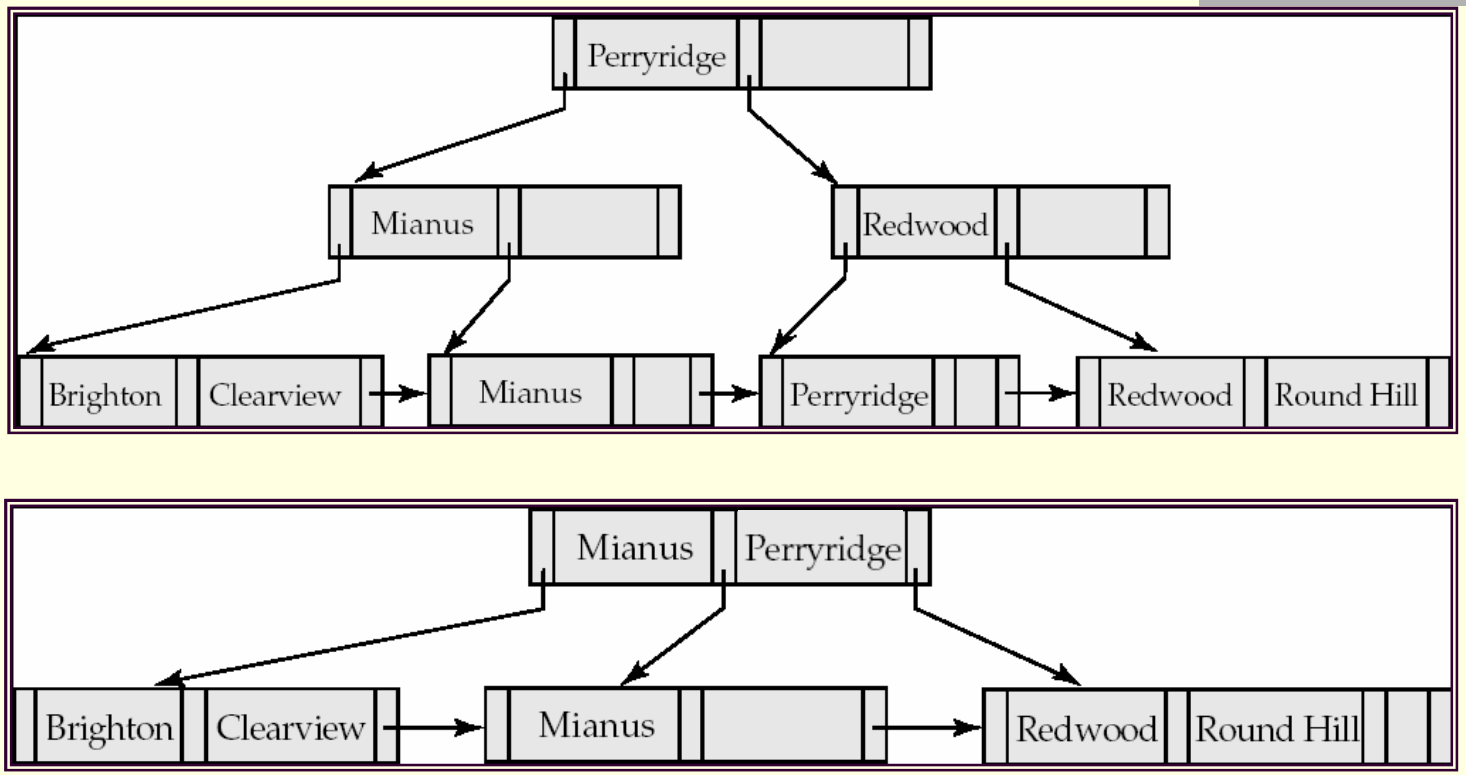
Examples of B$^+$-Tree Deletion



Before and after deleting "Downtown"

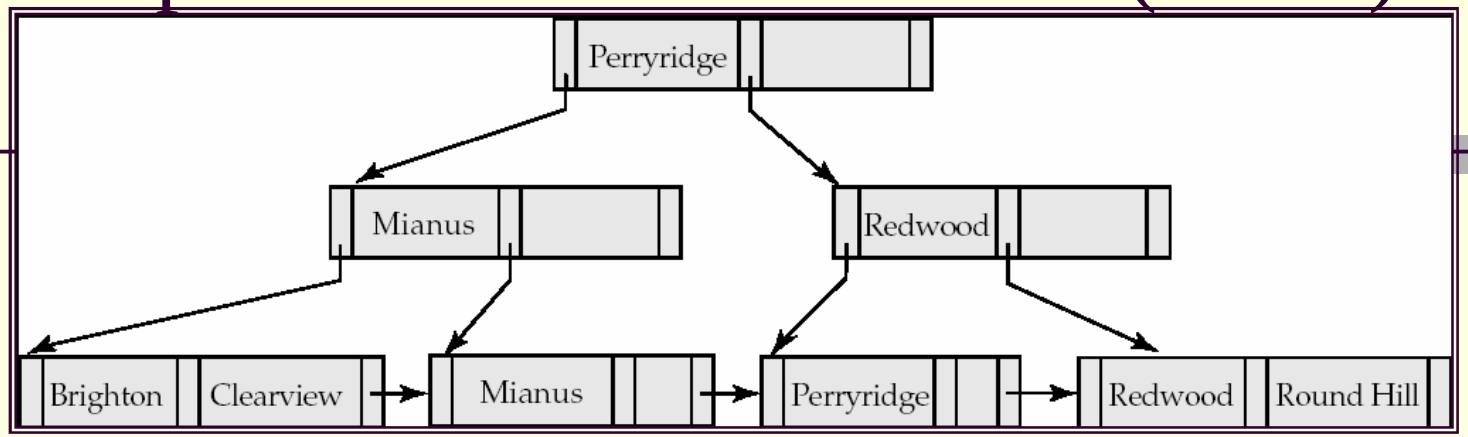
- Deleting "Downtown" causes merging of under-full leaves
  - leaf node can become empty only for n=3!

Examples of B+-Tree Deletion (Cont.)

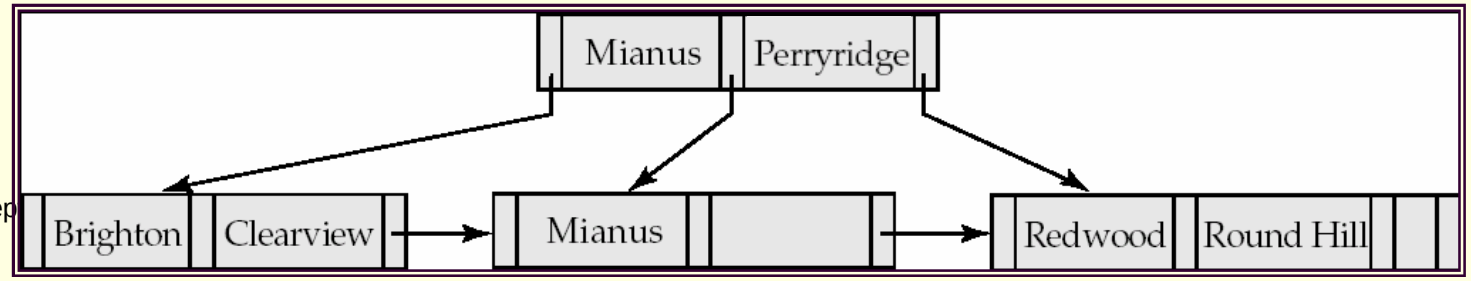


Before and After deletion of “Perryridge” from result of previous example

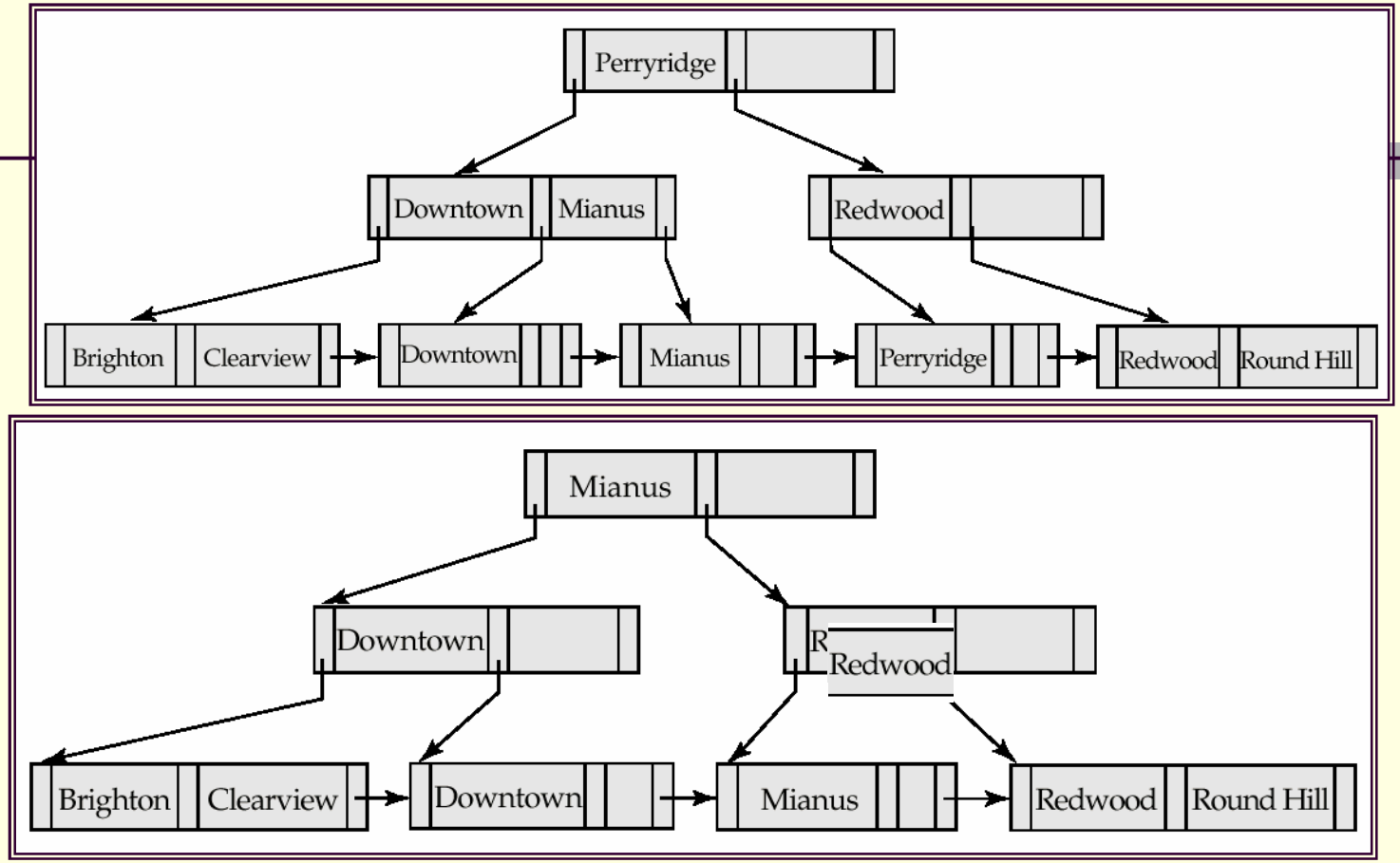
Examples of B+-Tree Deletion (Cont.)



- Leaf with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling.
- As a result "Perryridge" node's parent became underfull, and was merged with its sibling
  - Value separating two nodes (at parent) moves into merged node
  - Entry deleted from parent
- Root node then has only one child, and is deleted



Example of B+-tree Deletion (Cont.)



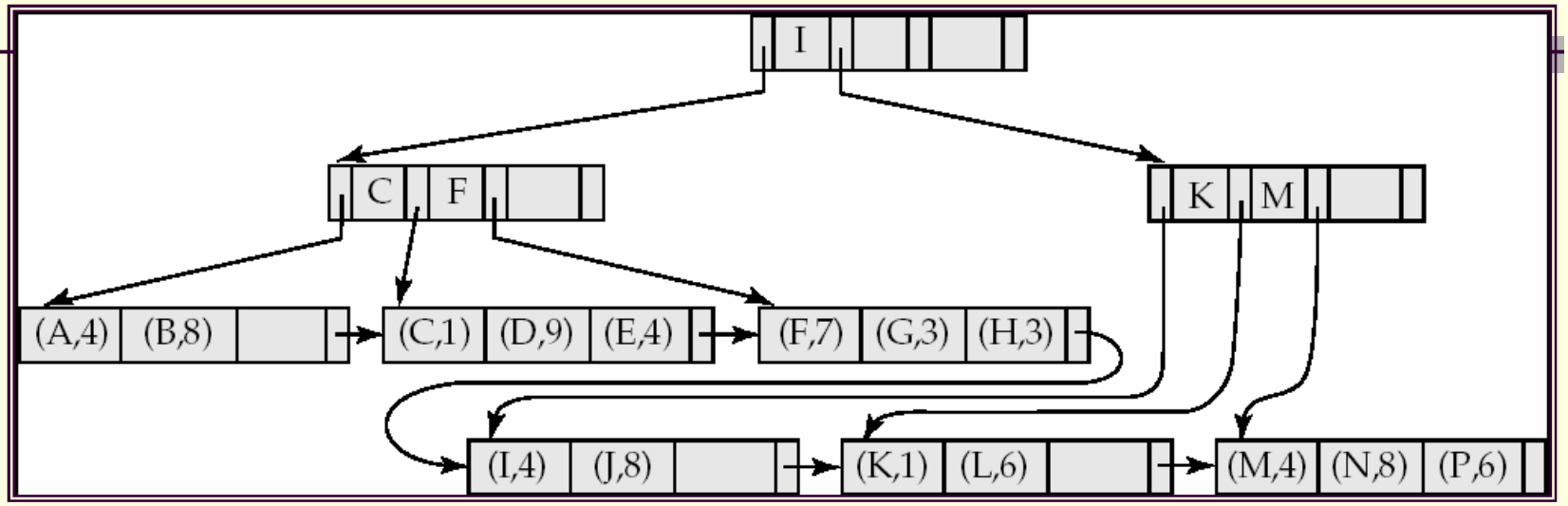
Before and after deletion of "Perryridge" from earlier example

- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

B^+-Tree File Organization

- Index file degradation problem is solved by using B^+-Tree indices.
- Data file degradation problem is solved by using B^+-Tree File Organization.
- The leaf nodes in a B^+-tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B+-Tree File Organization (Cont.)



Example of B+-tree File Organization

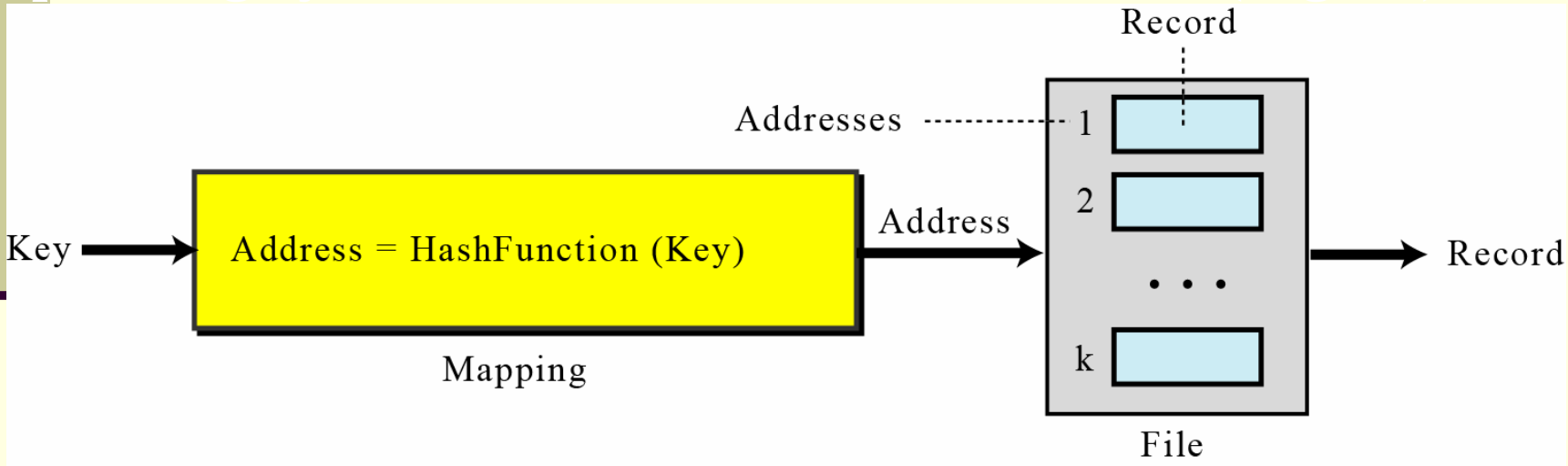
- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Inverted files

One of the advantages of indexed files is that we can have more than one index, each with a different key. For example, an employee file can be retrieved based on either social security number or last name. This type of indexed file is usually called an **inverted file**.

HASHED FILES

A **hashed file** uses a mathematical function to accomplish this mapping. The user gives the key, the function maps the key to the address and passes it to the operating system, and the record is retrieved (Figure).



Mapping in a hashed file

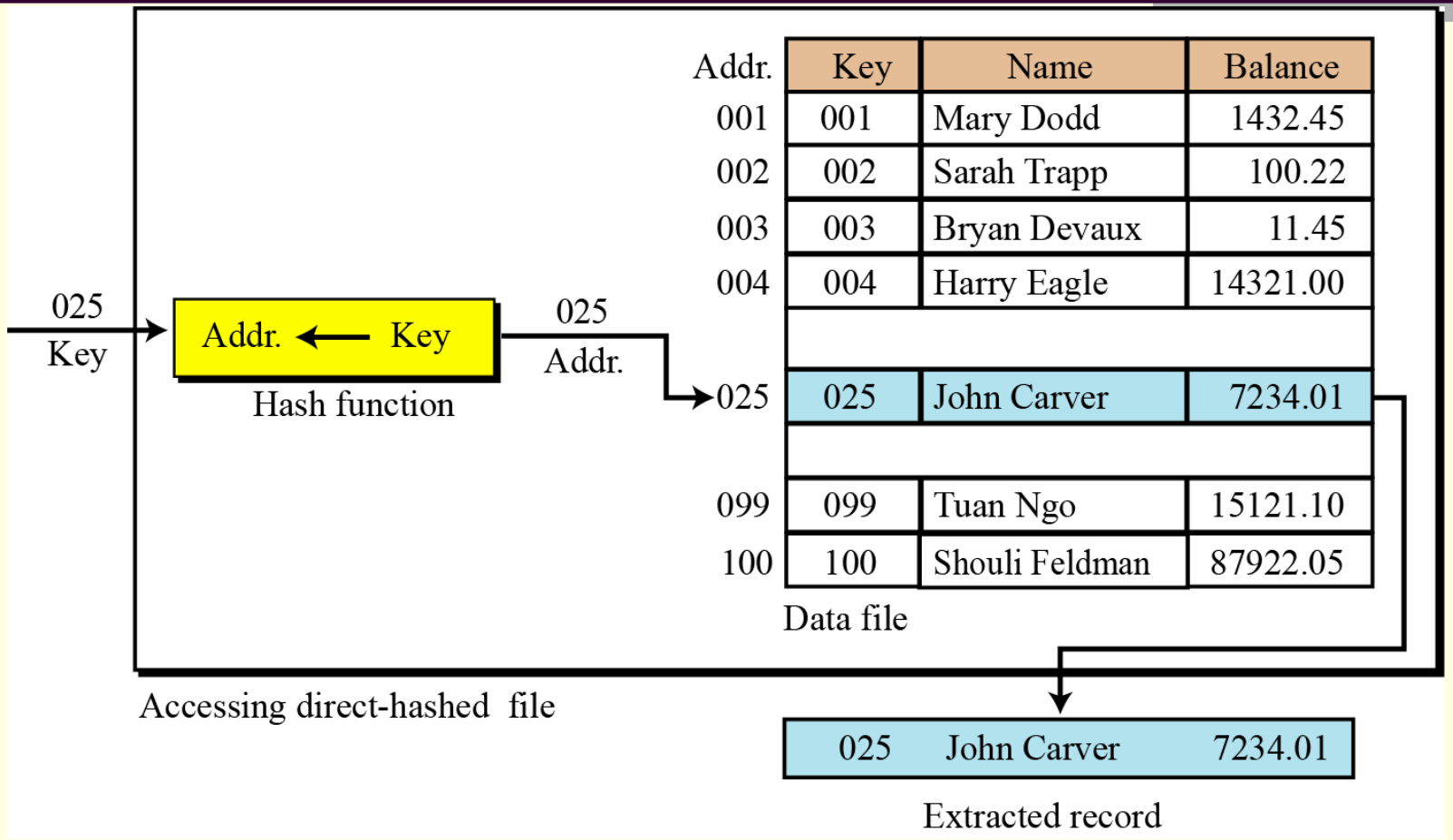
Hashing methods

For key-address mapping, we can select one of several hashing methods. We discuss a few of them here.

Direct hashing

In direct hashing, the key is the data file address without any algorithmic manipulation. The file must therefore contain a record for every possible key. Although situations suitable for direct hashing are limited, it can be very powerful, because it guarantees that there are no synonyms or collisions (discussed later in this chapter), as with other methods.

Direct hashing

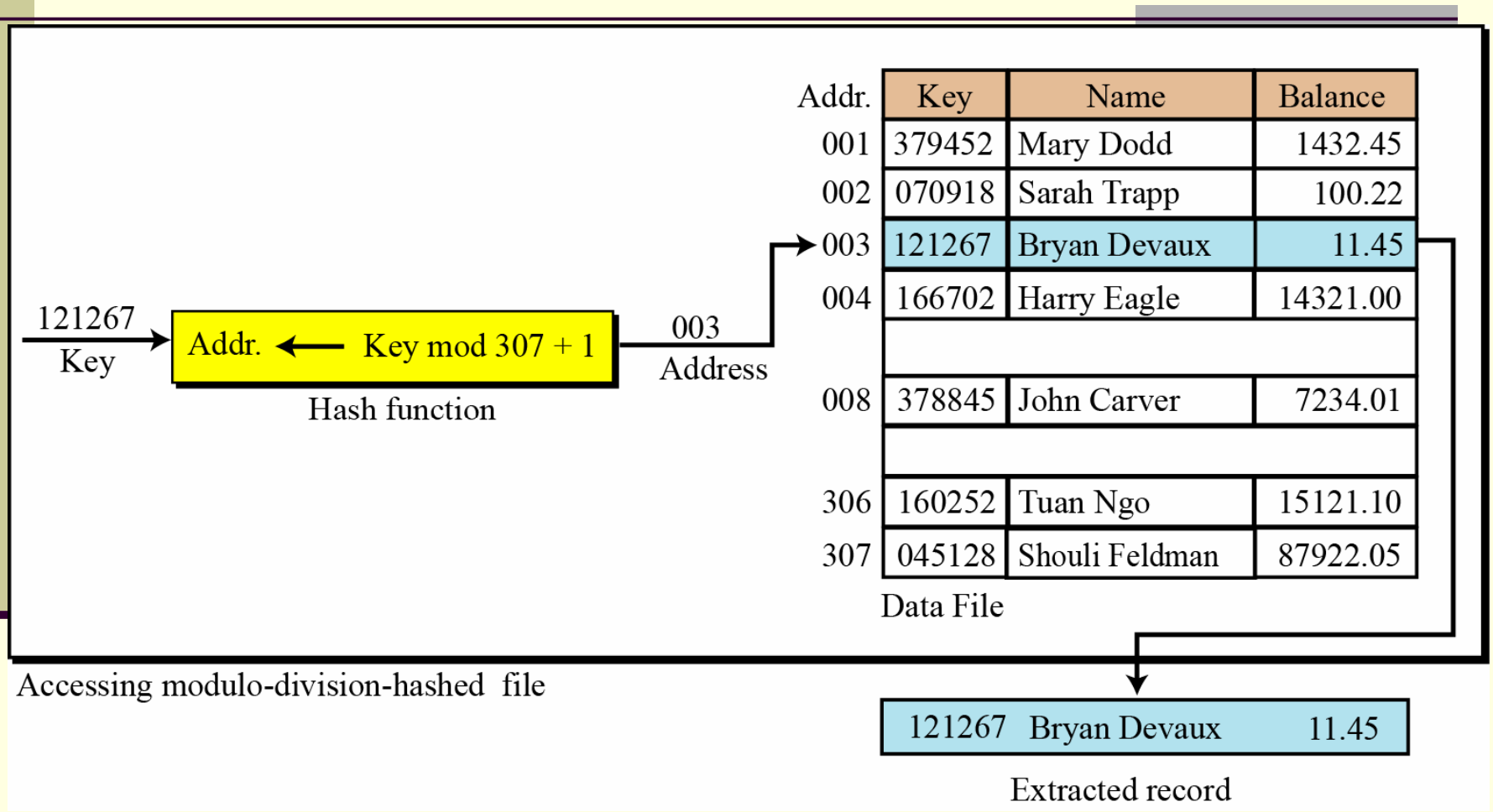


Modulo division hashing

Also known as division remainder hashing, the modulo division method divides the key by the file size and uses the remainder plus 1 for the address. This gives the simple hashing algorithm that follows, where list_size is the number of elements in the file. The reason for adding a 1 to the mod operation result is that our list starts with 1 instead of 0.

```
address = key mod list_size + 1
```

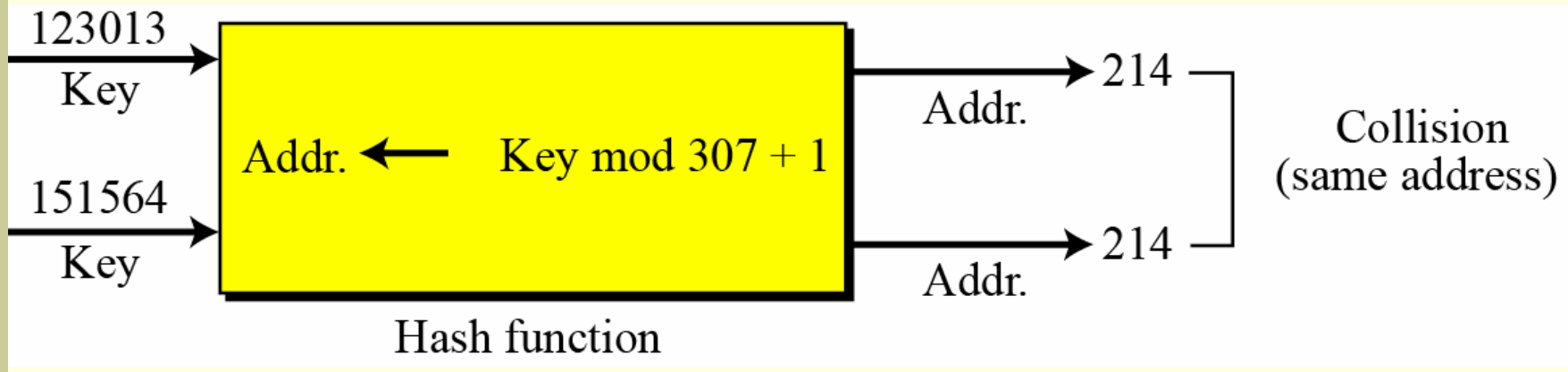
Modulo division



Collision

Generally, the population of keys for a hashed list is greater than the number of records in the data file. For example, if we have a file of 50 students for a class in which the students are identified by the last four digits of their social security number, then there are 200 possible keys for each element in the file (10,000/50). Because there are many keys for each address in the file, there is a possibility that more than one key will hash to the same address in the file. We call the set of keys that hash to the same address in our list synonyms. The collision concept is illustrated in Figure.

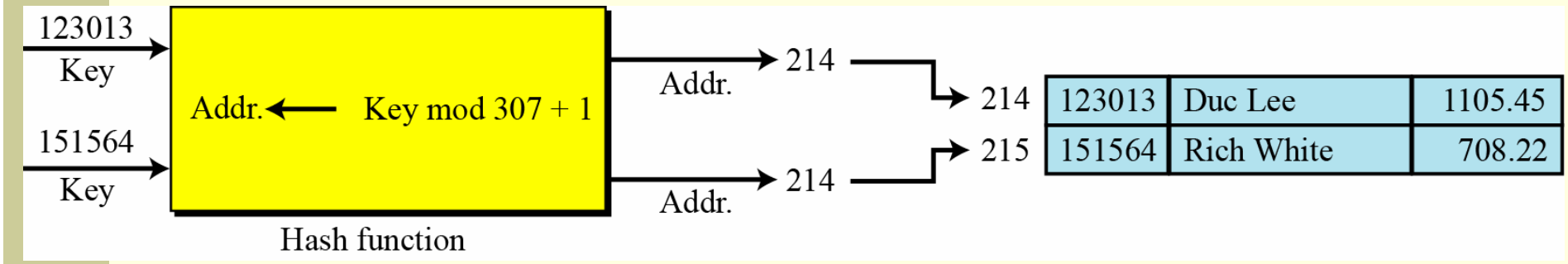
Collision


123013
Key
151564
Key
Addr.
Key mod 307 + 1
Hash function
Addr.
214
Addr.
214
Collision
(same address)

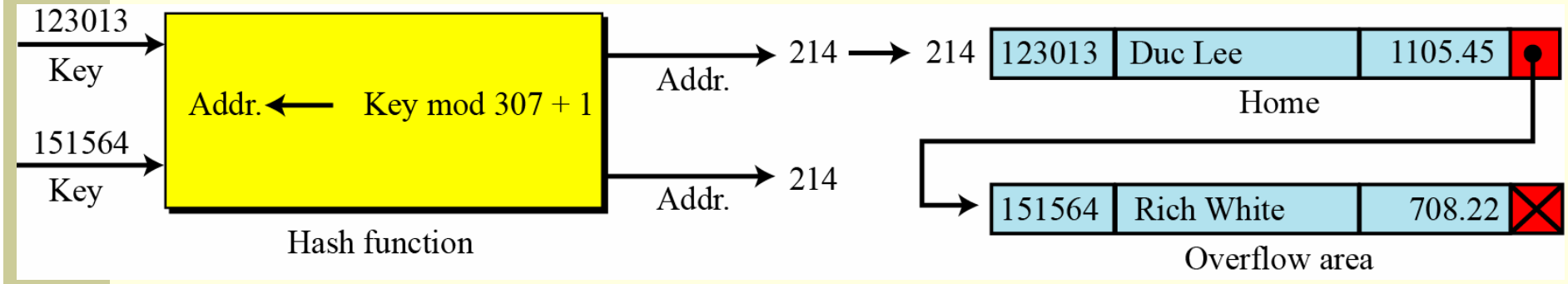
Collision resolution

With the exception of the direct method, none of the methods we have discussed for hashing creates one-to-one mappings. This means that when we hash a new key to an address, we may create a collision. There are several methods for handling collisions, each of them independent of the hashing algorithm. That is, any hashing method can be used with any collision resolution method. In this section, we discuss some of these methods.

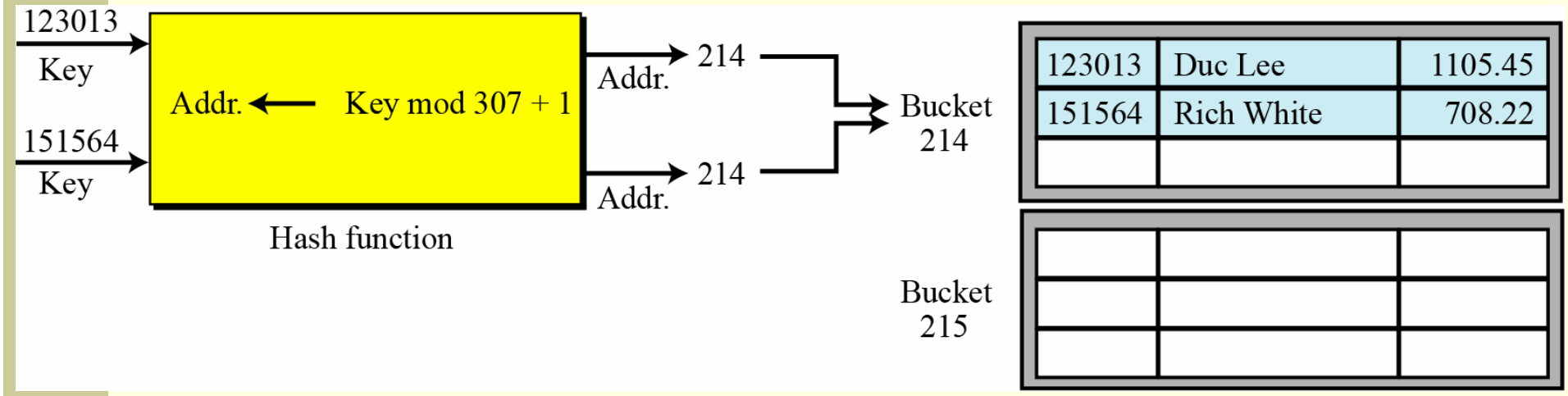
Open addressing resolution



Linked list resolution



Bucket hashing resolution



Comparative Access Methods

Factor	Sequential	Indexed	Hashed
Storage space	No wasted space	No wasted space for data but extra space for index	more space needed for addition and deletion of records after initial load
Sequential retrieval on primary key	Very fast	Moderately Fast	Impractical
Random Retr.	Impractical	Moderately Fast	Very fast
Multiple Key Retr.	Possible but needs a full scan	Very fast with multiple indexes	Not possible
Deleting records	can create wasted space	OK if dynamic	very easy
Adding records	requires rewriting file	OK if dynamic	very easy
Updating records	usually requires rewriting file	Easy but requires Maintenance of indexes	very easy

Index Definition in SQL

- Create an index

  create index <index-name> **on** <relation-name> (<attribute-list>)

  E.g.: **create index** *b-index* **on** *branch(branch_name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>

- Most database systems allow specification of type of index, and clustering.