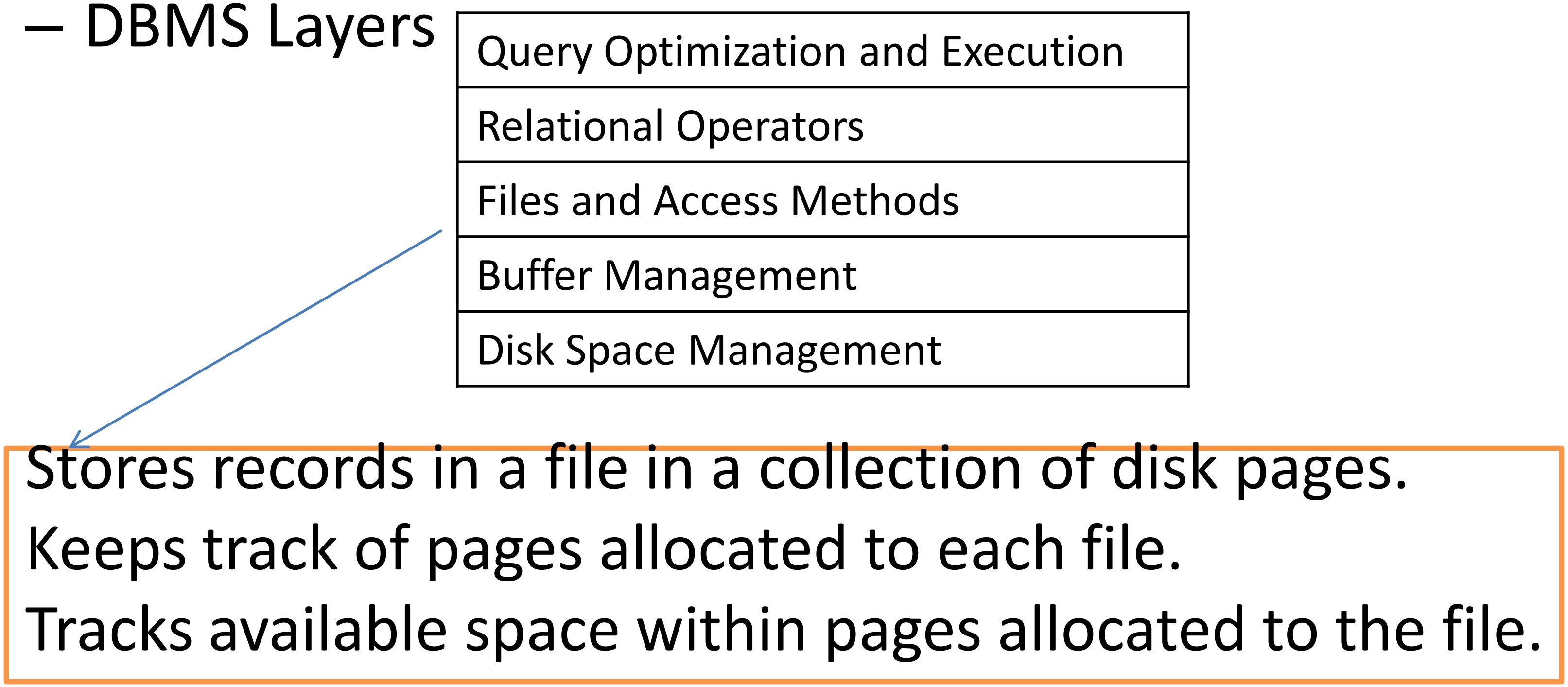
Chapter 8: File Organization and Indexing

File Organization

- File organization: is a method of arranging the records in a file when the file is stored on disk. A relation is typically stored as a file of records.
  - DBMS Layers



Query Optimization and Execution

Relational Operators

Files and Access Methods

Buffer Management

Disk Space Management

Stores records in a file in a collection of disk pages.
Keeps track of pages allocated to each file.
Tracks available space within pages allocated to the file.

Data on External Storage

- A DBMS stores vast amounts of data and the data has to be preserved across program executions.
- Therefore, data is stored on external storage and fetched into main memory as needed for processing.
- The unit of information that is read and written to a disk is Page.
- Higher layer of DBMS views this pages as unified files and can read or write records to these files.

Classification of Physical Storage Media

Several types of data storage exist in most computer system

These storage are classified by

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
  - data loss on power failure or system crash
  - physical failure of the storage device
- Can differentiate storage into:
  - volatile storage: loses contents when power is switched off
  - non-volatile storage: Contents persist even when power is switched off.

Storage Media

Among the media typically available are these:

- Cache:
  - The fastest and most costly form of storage.
- Main Memory:
  - The general purpose machine instructions operate in main memory
  - used for data that are available to be operated.
  - Is volatile in nature.
- Flash Memory:
  - Is non-volatile in nature
  - Reading data from flash memory is at speed of that of main memory

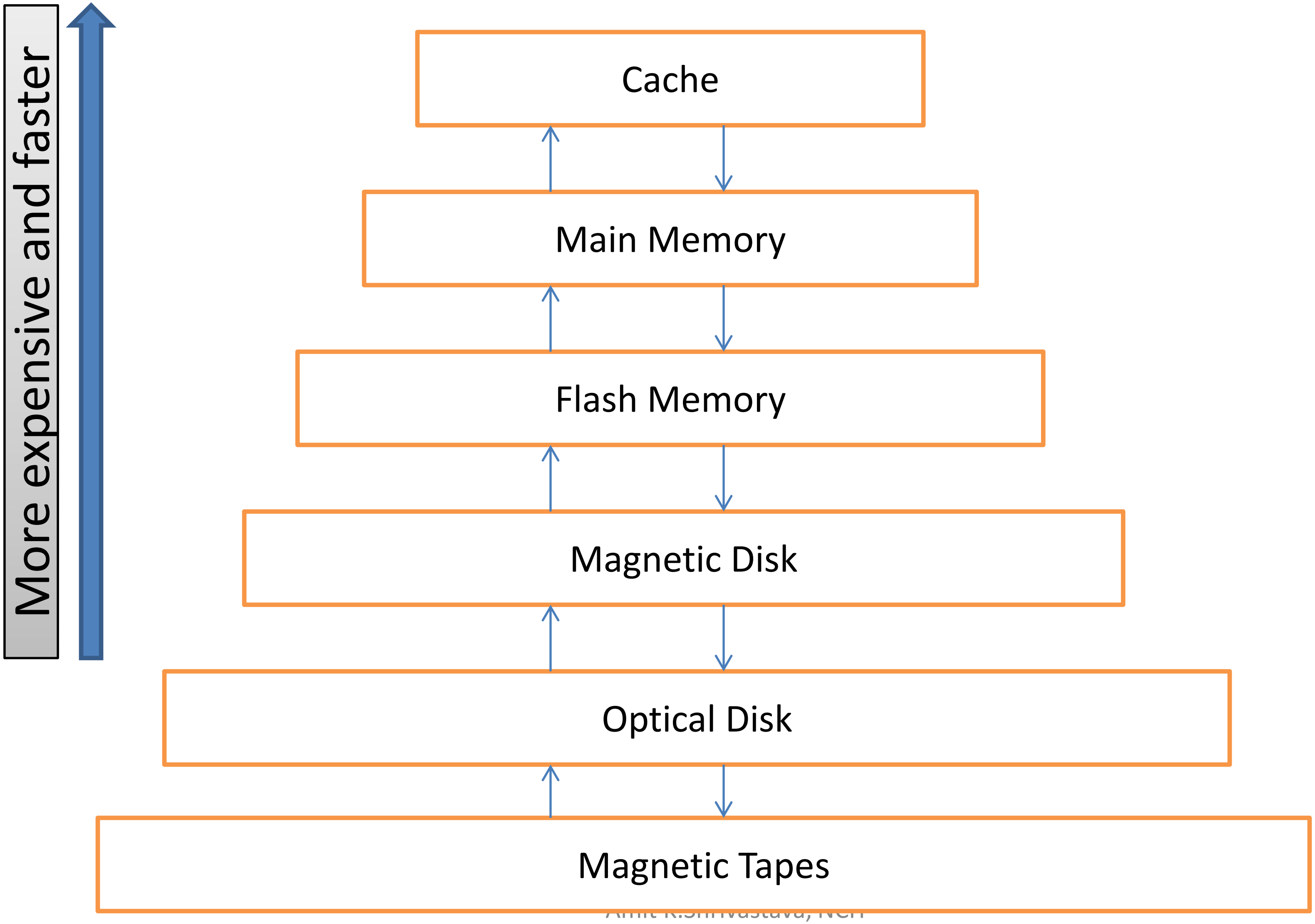
Storage Media

- Flash Memory:
  - Writing data to flash memory is slower
  - More expensive than disks
  - Used for applications with read workloads that require fast random accesses.
  - Mostly used in embedded systems like in hand held devices and other digital electronic devices.
- Magnetic Disk :
  - Secondary storage, used for long term storage of data.
  - Data should be moved to main memory before in use.
  - It is non volatile in nature.
  - Direct Memory Access.

Storage Media

- Optical Storage:
  - Random access of data
  - Used as secondary access of data for long term .
  - Mostly found in compact disk (CD) and digital video disk(DVD).
  - Data are stored optically on a disk and read by a laser.
  - Capable of storing high data in Gigabytes.
  - Found both in read only and read write form as CD-R, CD-RW,DVD-R,DVD-RW
- Magnetic Tape :
  - Is referred as sequential access storage.
  - Primarily used for backup and archival data .
  - Cheaper than disks but also access to data is slower.

Storage Media



More expensive and faster

Cache

Main Memory

Flash Memory

Magnetic Disk

Optical Disk

Magnetic Tapes

Fig.: Storage device Hierarchy

Buffer Manager

- Subsystem that is responsible for loading pages from external storage to the main memory (buffer pool) when needed.
- File & Index layers make calls to the buffer manager.
- If block is already in buffer, BM passes address of block in main memory.
- Else, BM first allocates space in the buffer for block.
- BM reads in requested block from disk to buffer and passes address of block in main memory.
- Idea is to Keep as many blocks(pages) in memory as possible to reduce disk accesses.
- Replacement Policies
  - LRU(Least-Recently-Used pages … are discarded => the oldest are discarded first)
  - MRU(Most-Recently-Used, the newest are discarded first)
  - LFU(Least-Frequently-Used)

File Organization

- Method for arranging a collection of records and supporting the concept of a file.
- A file is organized logically as a sequence of records. These records are mapped into the disk blocks.
- Each file is also logically partitioned into fixed length storage units called blocks, which are the units of both storage allocation and data transfer.
- Normally the block size range from 4 to 8 kilo bytes by default to Giga Bytes.
- Blocks contains records that is determined by the form of physical data organization being used.
- The size of the records should not be greater than block considering the fact that the field might be of image and it can varies in size dramatically.
- The another fact is that the entire record should be contained in one block rather than divided into separate blocks. This will benefits in speeding up the data access.
- Records may varies in size in one particular file system, so it is to be addressed.

Fixed Length Records

- Giving the attributes fixed size for an entity in terms of byte.
- For the file student, record may contain id int, name varchar(20), age int. For each record the space required is 24 bytes.
- **Problems**
  - The file records are of the same record type, but one or more of the fields are of varying size.
  - The file records are of the same record type but one or more of the fields may have multiple values for individual's records.
  - The file records are of the same records type but one or more of the fields are optional.
  - But it is very hard to occupy the block size of multiple of 24 bytes. So, it may be possible that when the block size obtain its max limit, it might be possible that the information of records may shift to another block.
  - While deleting the records from file, it should be either marked as deleted or should be occupied by next new record.

Fixed Length Records

- **Solutions**
  - The records should be entered into the block after the block size computation (dividing block size with the record size), by leaving the remaining block size unused.
  - While deleting record, we could move the record that came after it into the space formerly occupied by the deleted record and so on, until every record following the deleted record has been moved ahead. Or it might be easier to move the final record of the file into the space occupied by the deleted record.
  - The file header is used for storing the information of deleted records as well as available records. While inserting the new record, if the file contains the deleted spaces it will insert the record in that position and so on, if it does not contain any deleted position the record is placed at the end of the file

Fixed Length Records

Record 0	A-102	Kalimati	4000
Record 1	A-201	Patan	5000
Record 2	A-302	Bhaktapur	6000
Record 3	A-402	kalanki	4000
Record 4	A-103	kalimati	5000
Record 5	A-502	kirtipur	3000
Record 6	A-105	kalimati	5000
Record 7	A-202	patan	4000
Record 8	A-403	kalanki	4000

Record 0	A-102	Kalimati	4000
Record 1	A-201	Patan	5000
Record 3	A-402	kalanki	4000
Record 4	A-103	kalimati	5000
Record 5	A-502	kirtipur	3000
Record 6	A-105	kalimati	5000
Record 7	A-202	patan	4000
Record 8	A-403	kalanki	4000

Record 2 deleted and all records moved up

Fixed Length Records

Record 0	A-102	Kalimati	4000
Record 1	A-201	Patan	5000
Record 8	A-403	kalanki	4000
Record 3	A-402	kalanki	4000
Record 4	A-103	kalimati	5000
Record 5	A-502	kirtipur	3000
Record 6	A-105	kalimati	5000
Record 7	A-202	patan	4000

Record 2 deleted and final record moved

header			
Record 0	A-102	Kalimati	4000
Record 1			
Record 2	A-302	Bhaktapur	6000
Record 3	A-402	kalanki	4000
Record 4			
Record 5	A-502	kirtipur	3000
Record 6			
Record 7	A-202	patan	4000
Record 8	A-403	kalanki	4000

Free list after deletion of records 1,4,6

Variable Length Records

- It is used for
  - storing multiple record type in a file
  - record type that allow variable lengths for one or more fields
  - record types that allow repeating fields, such as arrays.
- Byte String Representation:
  - Attach a special *end of record* (⊥) symbol to the end of each record.
  - Store each record as a string of consecutive bytes.
  - Disadvantages:
    - Not easy to reuse space occupied formerly by a deleted record.
    - No space for records to grow longer.

Variable Length Records

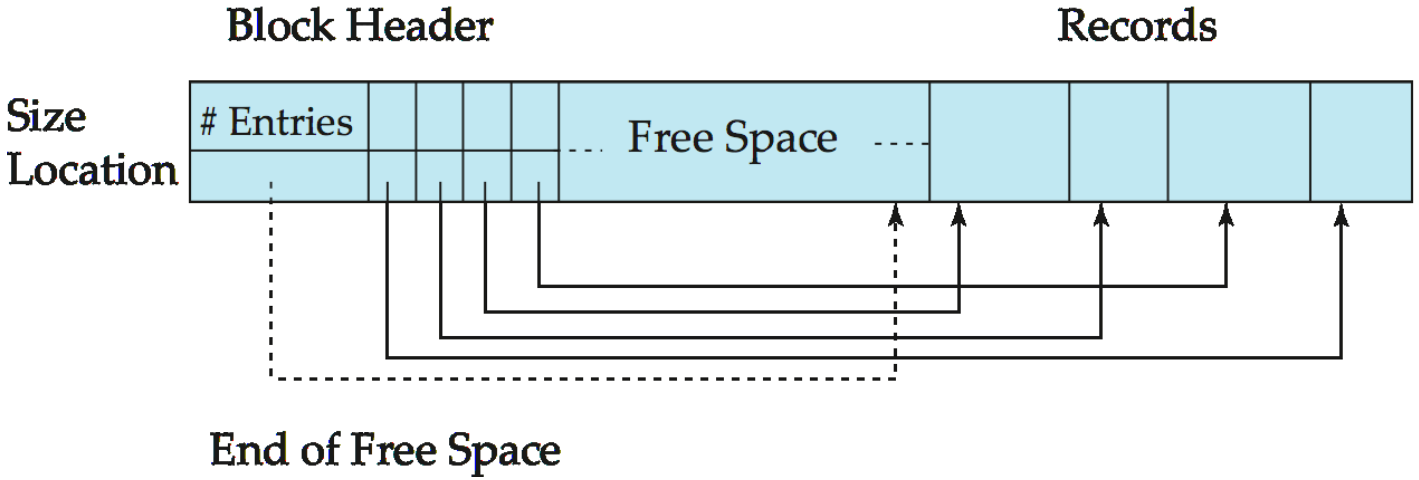
- Byte String Representation:

0	Kalimati	A-102	4000	A-103	5000	A-105	5000	⊥
1	Patan	A-201	5000	A-202	4000	⊥		
2	Bhaktapur	A-303	6000	⊥				
3	Kalanki	A-402	4000	A-403	4000	⊥		
4	Kirtipur	A-502	3000	⊥				

- Slotted page structure:
  - Header consists of
    - number of record entries
    - end of free space in the block
    - location and size of each record

Variable Length Records

- Slotted page structure:
  - Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
  - Pointers should not point directly to record — instead they should point to the entry for the record in header.



Variable Length Records

- Fixed Length Representation:
  - Two ways
    - Reserved Space
    - List Representation

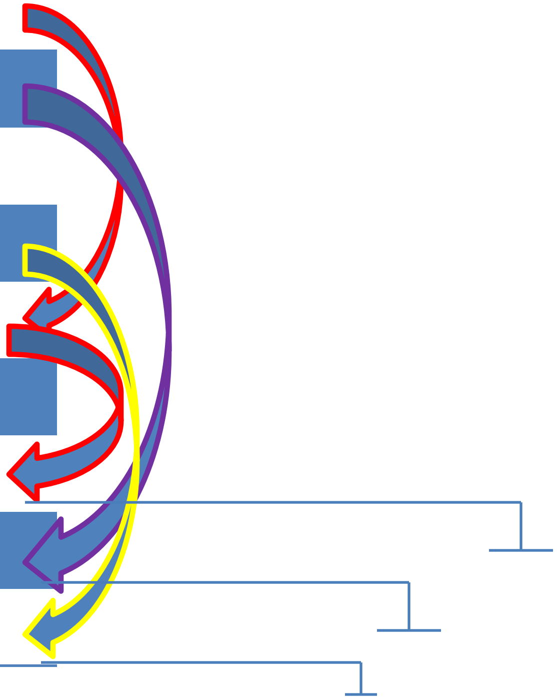
0	Kalimati	A-102	4000	A-103	5000	A-105	5000
1	Patan	A-201	5000	A-202	4000	⊥	⊥
2	Bhaktapur	A-303	6000	⊥	⊥	⊥	⊥
3	Kalanki	A-402	4000	A-403	4000	⊥	⊥
4	Kirtipur	A-502	3000	⊥	⊥	⊥	⊥

Reserved Space Method

Variable Length Records

- Fixed Length Representation:

Record 0	Kalimati	A-102	4000
Record 1	Patan	A-201	5000
Record 2	Bhaktapur	A-302	6000
Record 3	kalanki	A-402	4000
Record 4		A-103	5000
Record 5	kirtipur	A-502	3000
Record 6		A-105	5000
Record 7		A-202	4000
Record 8		A-403	4000



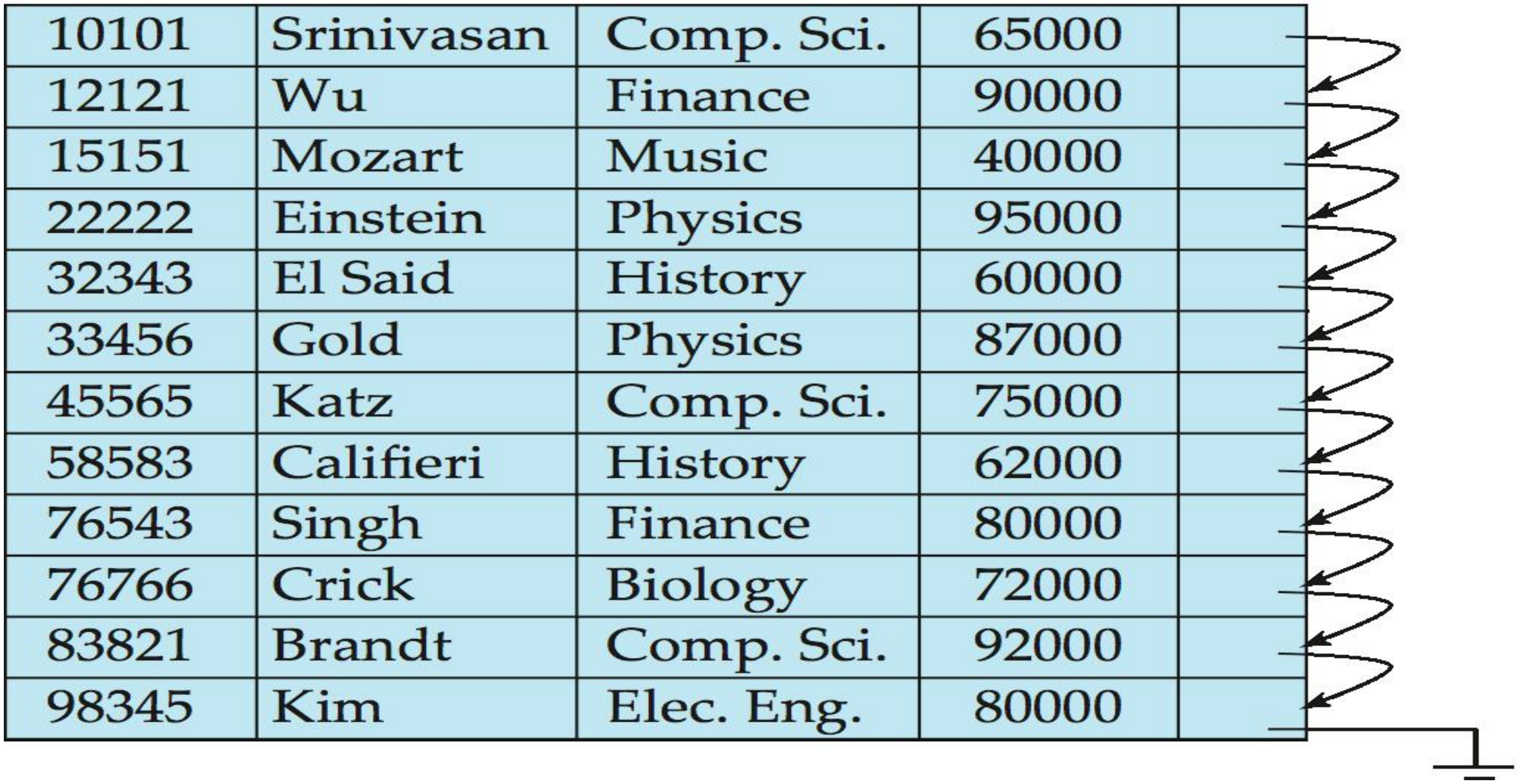
Linked List Method

Organization of Records in Files

- **Heap**
  - a record can be placed anywhere in the file where there is space
- **Sequential**
  - store records in sequential order, based on the value of the search key of each record
- **Hashing**
  - a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - Motivation: store related records on the same block to minimize I/O

Sequential File Organization

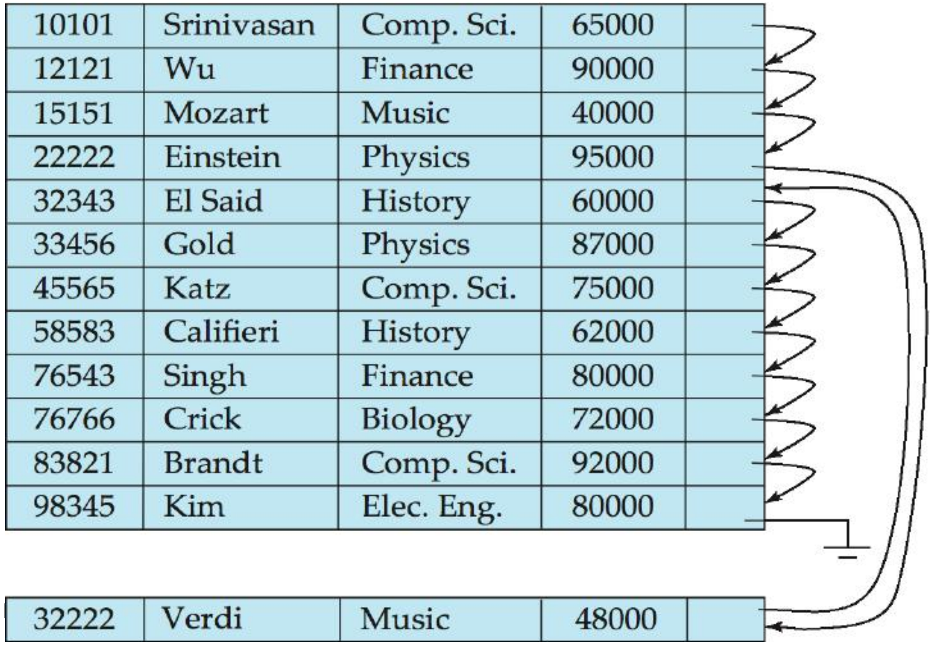
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

Sequential File Organization

- Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - 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



10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	

Data Dictionary Storage

- The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

I. Information about relations
   - names of relations
   - names, types and lengths of attributes of each relation
   - names and definitions of views
   - integrity constraints

II. User and accounting information, including passwords

III. Statistical and descriptive data
   - number of tuples in each relation

IV. Physical file organization information
   - How relation is stored (sequential/hash/…)
   - Physical location of relation

V. Information about indices

Indexing

- Indexing mechanisms are used to speed up access to desired data.
- **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

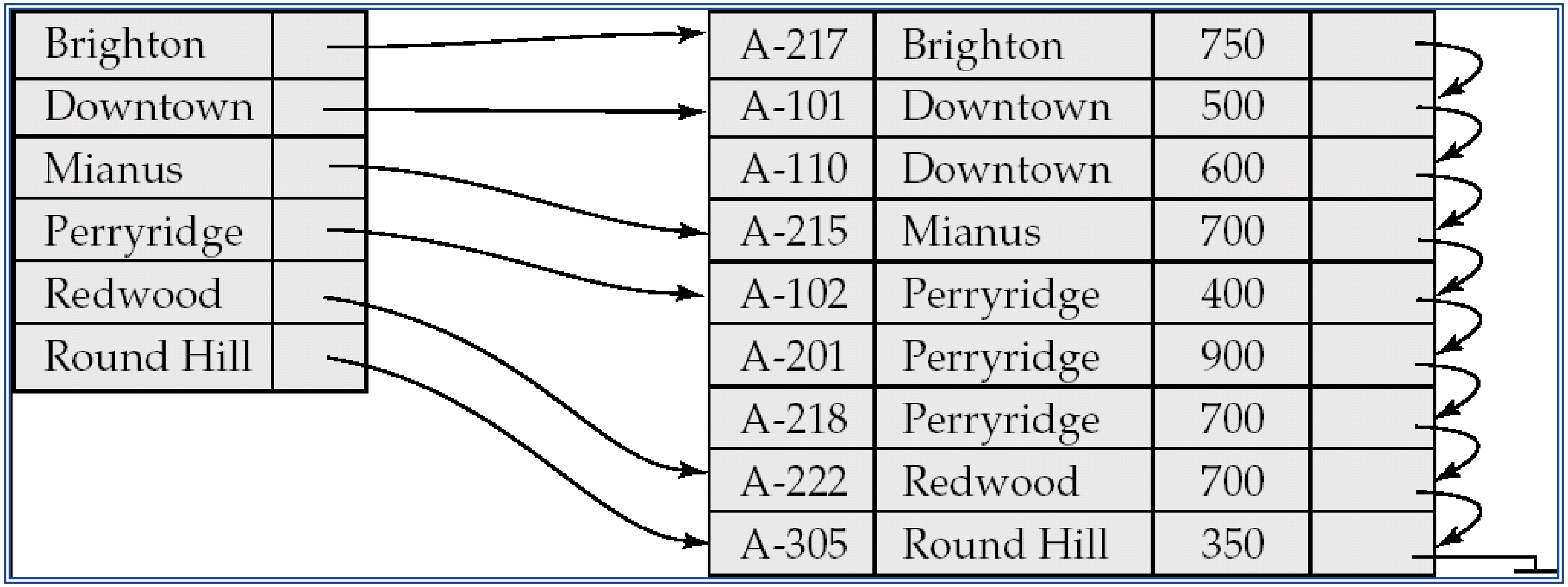
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".
- Index Evaluation Metrics:
  - Access time
  - Insertion time
  - Deletion time
  - Space overhead
  - Access types supported efficiently. E.g.,
    - records with a specified value in the attribute
    - or records with an attribute value falling in a specified range of values.
    - This strongly influences the choice of index, and depends on usage.

1.Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **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-sequential file: ordered sequential file with a primary index.*

1.1.1Dense index

- Index record appears for every search-key value in the file.
- In dense primary index, the index record contains the search key value and a pointer to the first data record with that search key value.

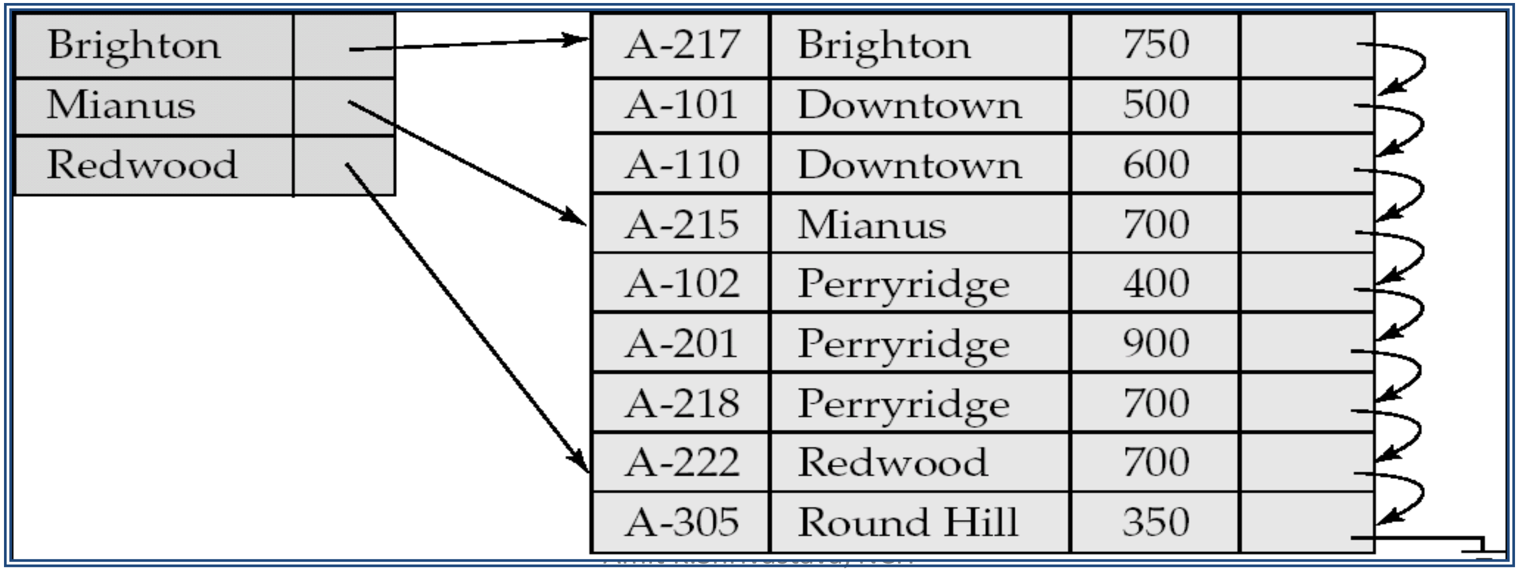


Brighton	
Downtown	
Mianus	
Perryridge	
Redwood	
Round Hill	

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	

1.1.2 Sparse Index

- Index records appears for only some search-key values.
- Here also the index record contains the search key value and a pointer to the first data record with that search key value.
- Only 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



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	

1.1.3 Multilevel Indices

- 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.
- To locate a record, we first use binary search on the outer index to find the record for the largest search-key value less than or equal to the one that we desire.
- The pointer points to the block of inner index. We scan this block until we find the record that has the largest search key value less than or equal to the one that we desire. The pointer in this record points to the block of the file that contains the record for which we are looking

1.1.4 Index Update

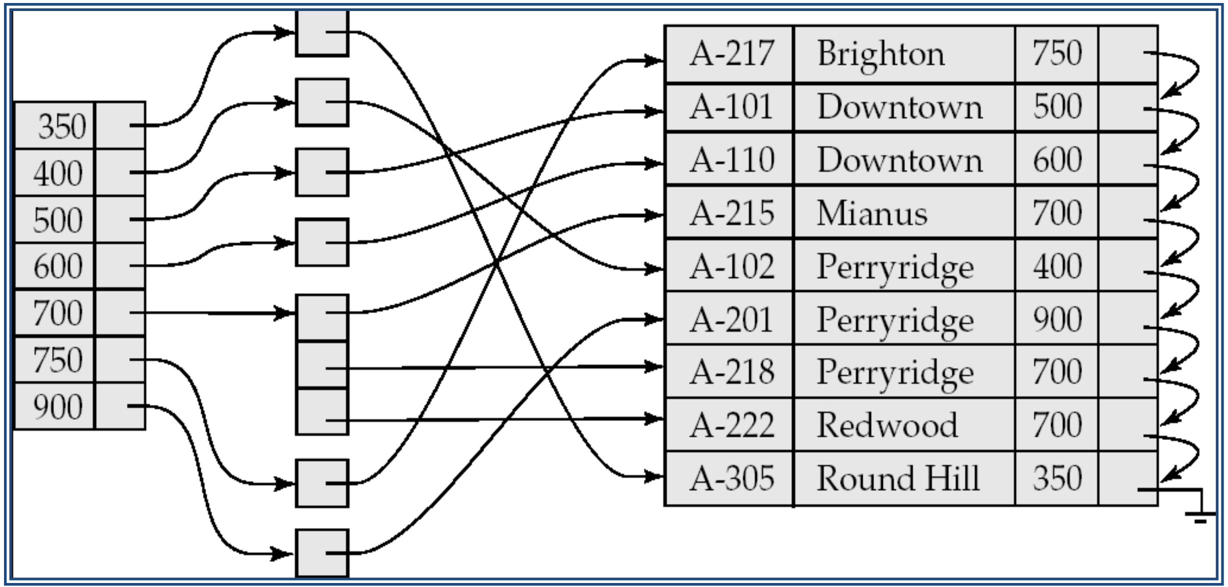
- **Deletion**
  - If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
  - Single-level index deletion:
    - **Dense indices** – deletion of search-key: similar to file record deletion.
    - **Sparse indices** –
   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

1.1.4 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.
      - 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

1.2 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* relation 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 indices have to be dense, since the file is not sorted by the search key.



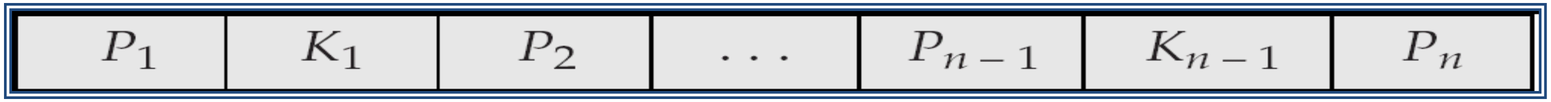
A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

1.2.1 Secondary and Primary Indices

- Indices offer substantial benefits when searching for records, but updating indices imposes overhead on database modification - when a file is modified, every index on the file must be updated,
- 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
  - Block fetch requires about 5 to 10 micro seconds, versus about 100 nanoseconds for memory access

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
- Typical node



P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

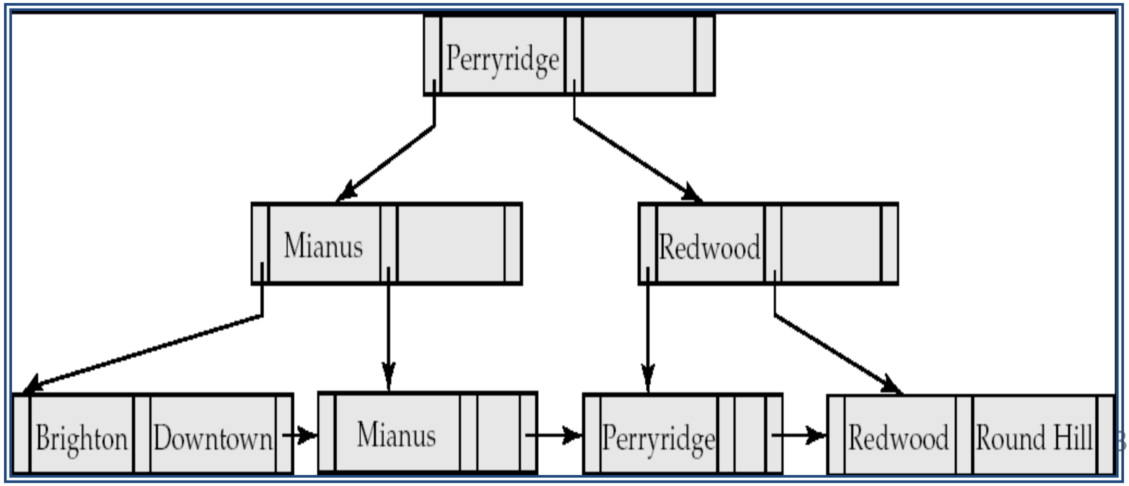
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$

- Usually the size of a node is that of a block

B^+-Tree Index Files

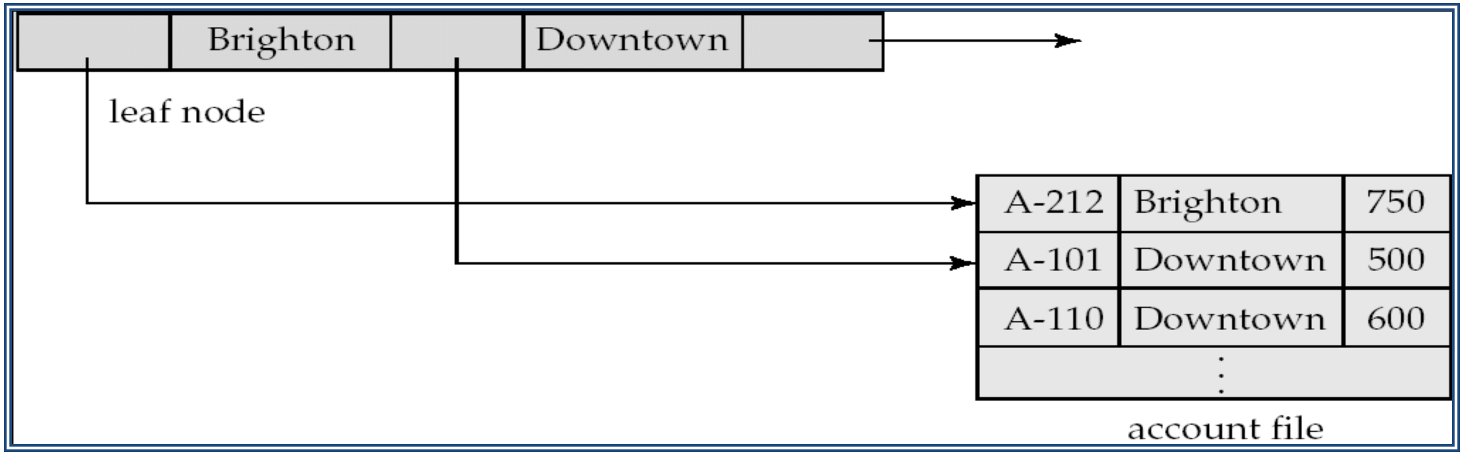
- A B^+-tree is a rooted tree satisfying the following properties:
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
  - A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
  - Special cases:
    - 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.
  - All values must be at leaf node.



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

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}
- *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.*

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 some details, and more in the book).

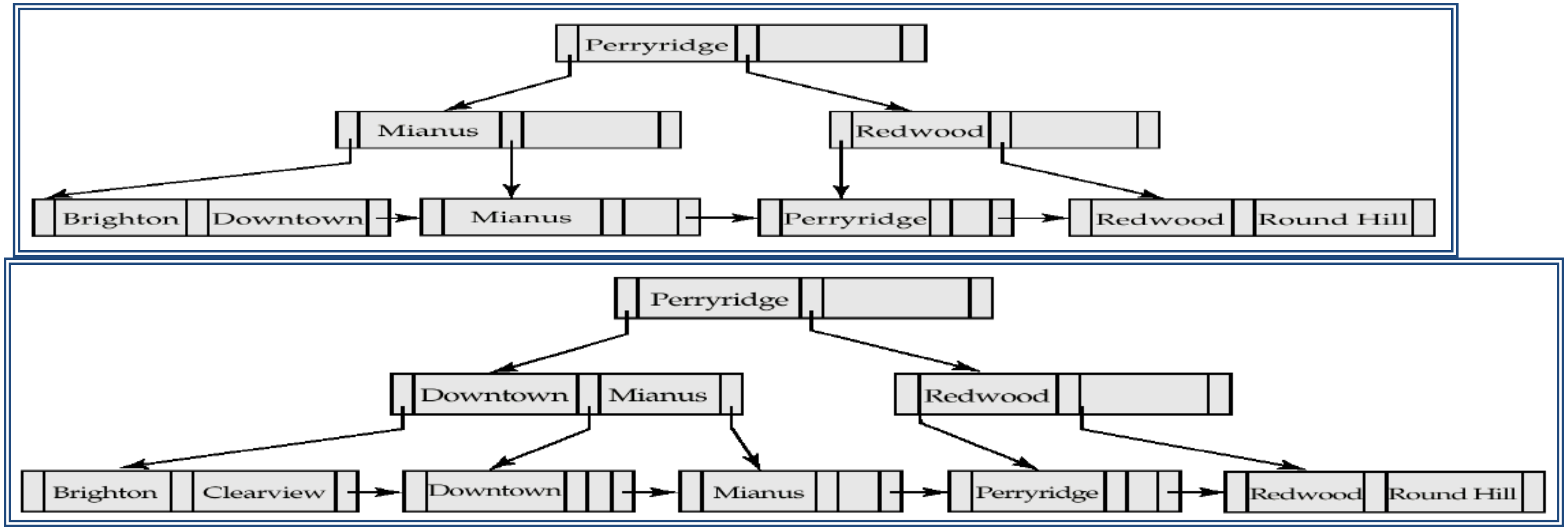
Queries on B$^+$-Trees

- 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.
- 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 4Kbytes
  - *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.
    - I.e. at most 4 accesses to disk blocks are needed
- 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
   2. If necessary add a pointer to the bucket.
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)



B^+-Tree before and after insertion of “Clearview”

Updates on B^+-Trees: Insertion

- 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 of the tree by 1.
- 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, …, $K_{\lceil n/2 \rceil -1}$,$P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil +1}$,$K_{\lceil n/2 \rceil +1}$,…,K_n,P_{n+1} from M into newly allocated node N’
  - Insert ($K_{\lceil n/2 \rceil}$,N’) into parent N

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.
- 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.

B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B^+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B^+-Tree
  - Insertion and deletion more complicated than in B^+-Trees
  - Implementation is harder than B^+-Trees.
  - Not possible to sequentially scan a table by just looking at leafs.
- Typically, advantages of B-Trees do not out weigh disadvantages.
  - In DBMSs B+-Trees are favored.

Hashing

- Hashing is an effective technique to calculate direct location of data record on the disk without using index structure.
- It uses a function, called hash function and generates address when called with search key as parameters. Hash function computes the location of desired data on the disk.
- Hash Organization:
  - **Bucket:** Hash file stores data in bucket format. Bucket is considered a unit of storage. Bucket typically stores one complete disk block, which in turn can store one or more records.
  - **Hash Function:** A hash function h, is a mapping function that maps all set of search-keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.
    - Choose hash function that assign search key values to buckets in such a way that distribution is either uniform or random.

Static Hashing

- In static hashing, when a search-key value is provided the hash function always computes the same address.
- For example, if mod-4 hash function is used then it shall generate only 5 values.
- The output address shall always be same for that function. The numbers of buckets provided remain same at all times.
- Operation:
  - **Insertion:** When a record is required to be entered using static hash, the hash function h, computes the bucket address for search key K, where the record will be stored.
    - Bucket address = h(K)
  - **Search:** When a record needs to be retrieved the same hash function can be used to retrieve the address of bucket where the data is stored.
  - **Delete:** This is simply search followed by deletion operation.

Bucket Overflows

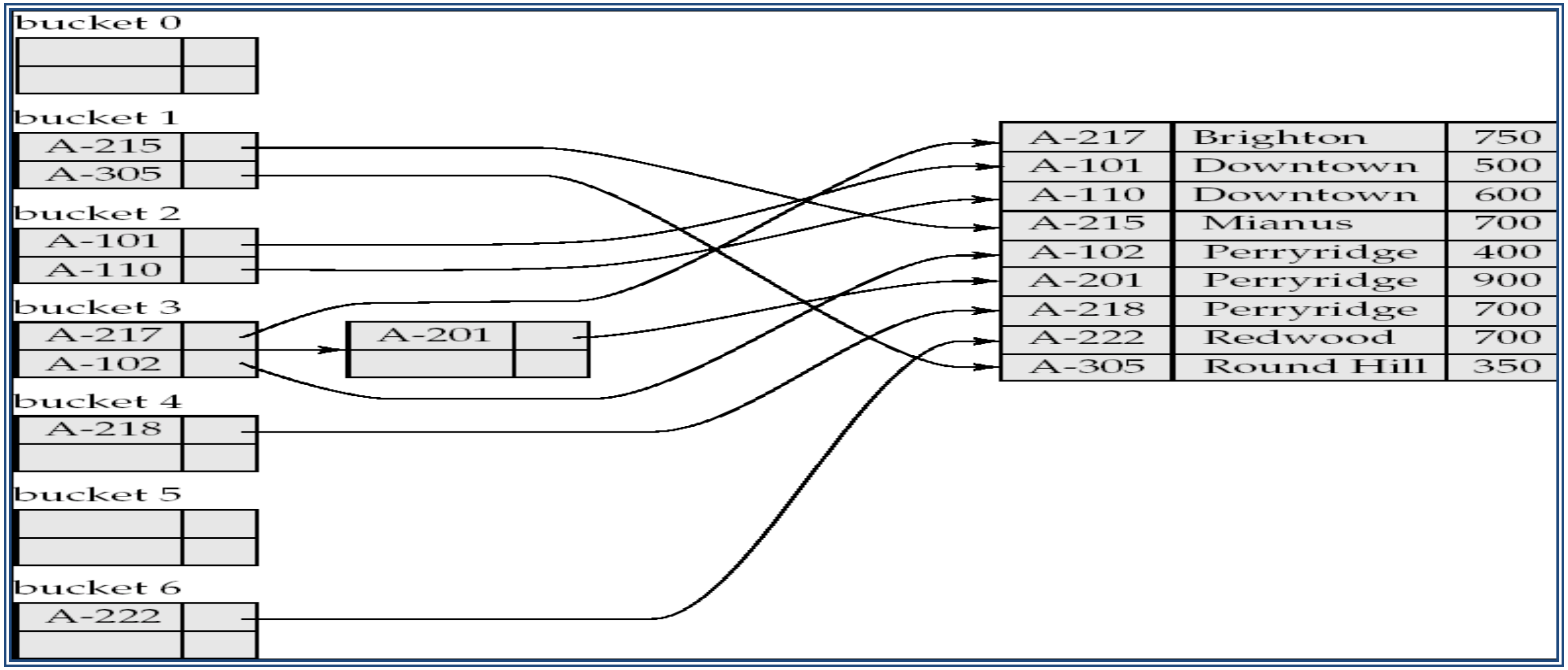
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets.*
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
  - Also called *closed hashing*
- Linear Probing - does not use overflow buckets, is not suitable for database applications.
  - Also called *open hashing*

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.
- We construct a hash index as follows. We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).

Hash Indices(contd…)

- Figure below shows a secondary hash index on the *account file, for the search key account-number. The hash function in the figure* computes the sum of the digits of the account number modulo 7. The hash index has seven buckets, each of size 2. One of the buckets has three keys mapped to it, so it has an overflow bucket



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