Chapter 11: Indexing and Storage

Modified from:

Database System Concepts, 6th Ed.

Chapter 11: Indexing and Storage

- DBMS Storage
  - Memory hierarchy
  - File Organization
  - Buffering
- Indexing
  - Basic Concepts
  - B^+-Trees
  - Static Hashing
  - Index Definition in SQL
  - Multiple-Key Access

Memory Hierarchy

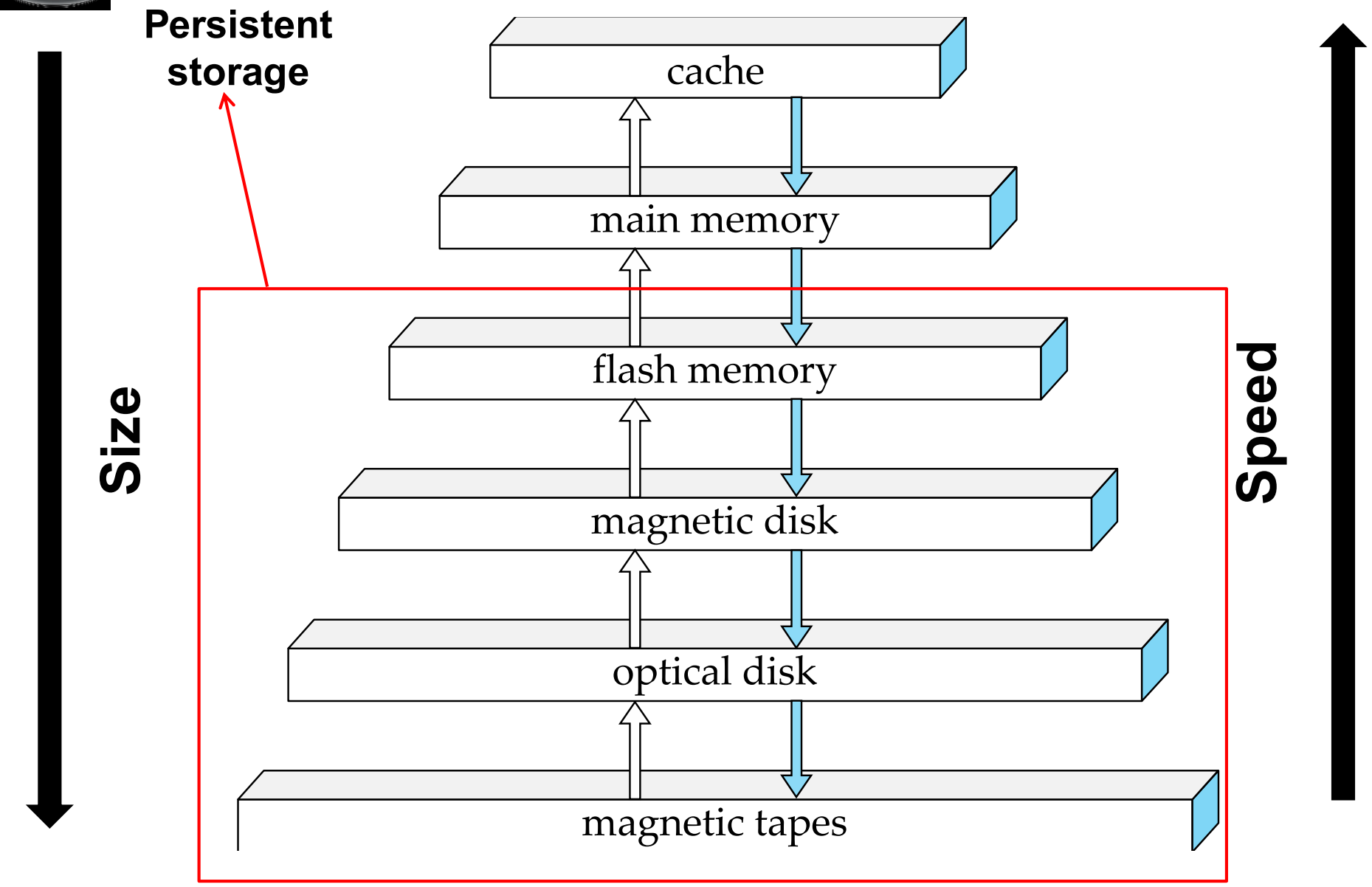
Modified from:

Database System Concepts, 6th Ed.

DBMS Storage

- Modern Computers have different types of memory
  - Cache, Main Memory, Harddisk, SSD, …
- Memory types have different characteristics in terms of
  - Persistent vs. volatile
  - Speed (random vs. sequential access)
  - Size
  - Price – this usually determines size
- Database systems are designed to be use these different memory types effectively
  - Need for persistent storage: the state of the database needs to be written to persistent storage
    - guarantee database content is not lost when the computer is shutdown
  - Moving data between different types of memory
    - Want to use fast memory to speed-up operations
    - Need slower memory for the size

Storage Hierarchy



Main Memory vs. Disk

- Why do we not only use main memory
  - What if database does not fit into main memory?
  - Main memory is volatile
- Main memory vs. disk
  - Given available main memory when do we keep which part of the database in main memory
    - **Buffer manager**: Component of DBMS that decides when to move data between disk and main memory
  - How do we ensure transaction property durability
    - Buffer manager needs to make sure data written by committed transactions is written to disk to ensure durability

Random vs. Sequential Access

- Transfer of data from disk has a minimal size = 1 block
  - Reading 1 byte is as fast as reading one block (e.g., 4KB)
- **Random Access**
  - Read data from anywhere on the disk
  - Need to get to the right track (**seek time**)
  - Need to wait until the right sector is under the arm (on avg ½ time for one rotation) (**rotational delay**)
  - Then can transfer data at ~ **transfer rate**
- **Sequential Access**
  - Read data that is on the current track + sector
  - can transfer data at ~ **transfer rate**
- Reading large number of small pieces of data randomly is very slow compared to sequential access
  - Thus, try layout data on disk in a way that enables sequential access

File Organization

Modified from:

Database System Concepts, 6th Ed.

File Organization

- The database is stored as a collection of **files**. Each file stores **records** (tuples from a table). A record is a sequence of **fields** (the attributes of a tuple).
- Reading one record of a time from disk would be very slow (random access)
  - Organize our database files in pages (size of block or larger)
  - Read/write data in units of pages
  - One page will usually contain several records
- One approach:
  - assume record size is fixed
  - each file has records of one particular type only
  - different files are used for different relations

  This case is easiest to implement; will consider variable length records later.

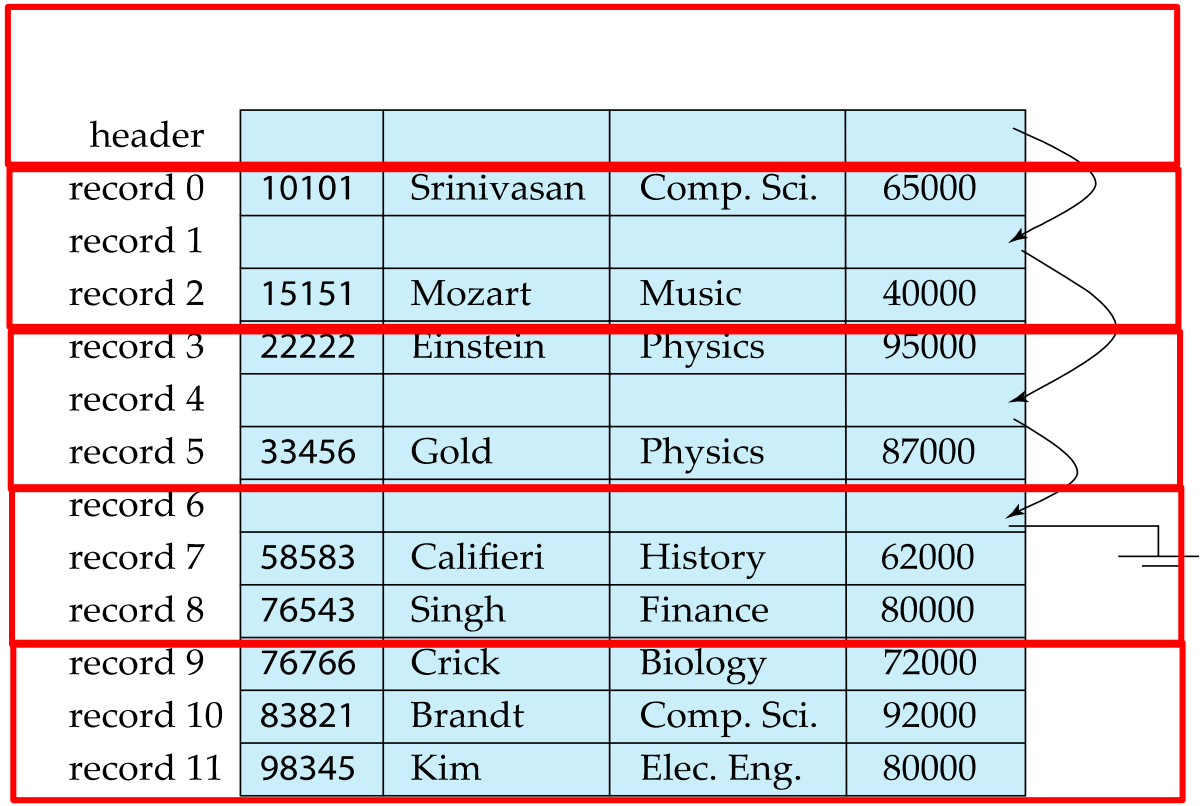
Fixed-Length Records

- Simple approach:
  - Store record *i* starting from byte $n * (i - 1)$, where *n* is the size of each record. Put maximal P / n records on each page.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries
- Deletion of record *i:* alternatives*:*
  - move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$
  - move record *n* to *i*
  - do not move records, but link all free records on a *free list*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Free Lists

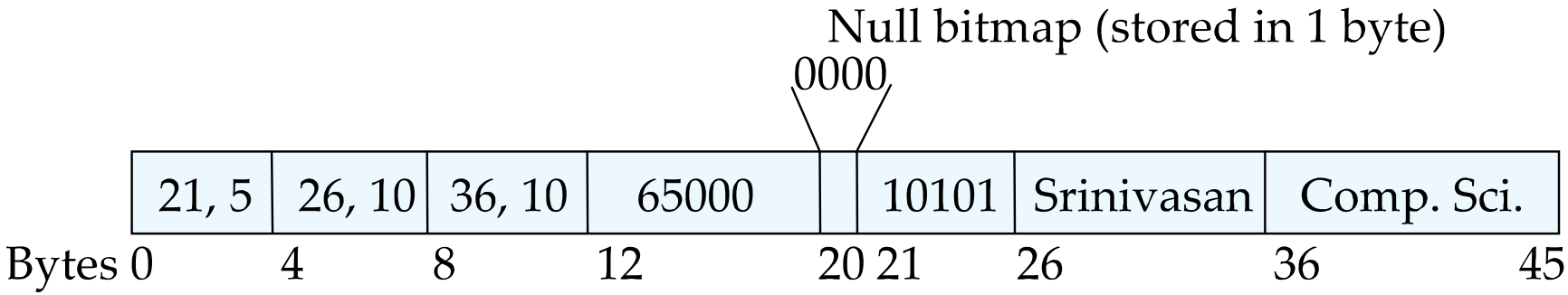
- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as pointers since they "point" to the location of a record.



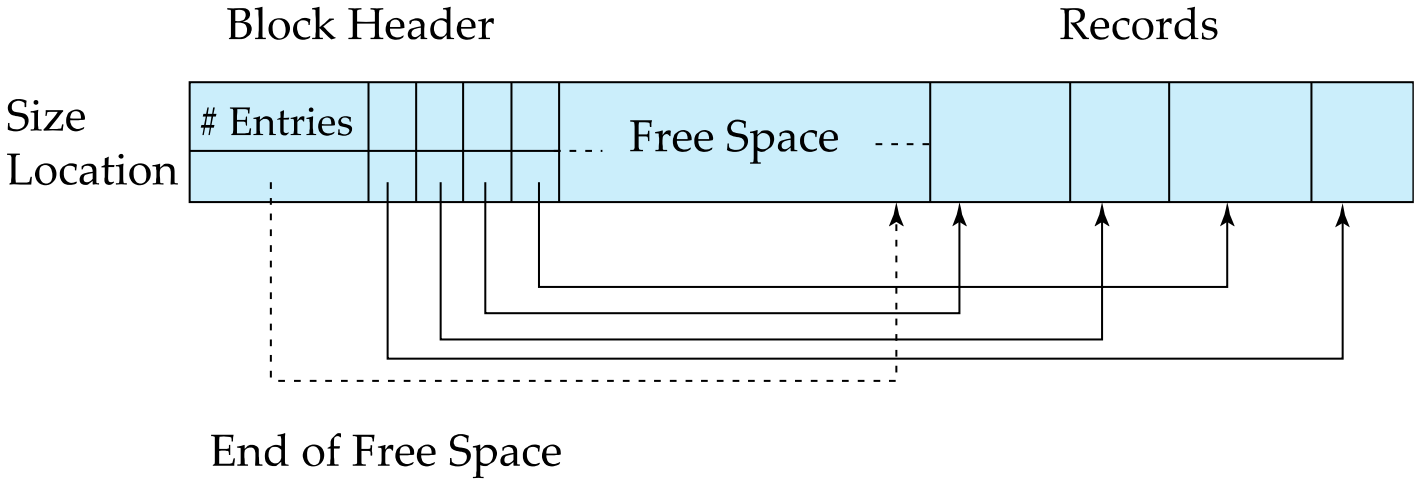
header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields such as strings (**varchar**)
  - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap



Variable-Length Records: Slotted Page Structure



- **Slotted page** header contains:
  - number of record entries
  - end of free space in the block
  - location and size of each record
- 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.

Organization of Records in Files

- **Heap** – a record can be placed anywhere in the file where there is space
  - Deletion efficient
  - Insertion efficient
  - Search is expensive
    - Example: Get instructor with name Glavic
      - Have to search through all instructors
- **Sequential** – store records in sequential order, based on the value of some search key of each record
  - Deletion expensive and/or waste of space
  - Insertion expensive and/or waste of space
  - Search is efficient (e.g., binary search)
    - As long as the search is on the search key we are ordering on

Buffering

Modified from:

Database System Concepts, 6th Ed.

Buffer Manager

- Buffer Manager
  - Responsible for loading pages from disk and writing modified pages back to disk
- Handling blocks
  1. If the block is already in the buffer, the buffer manager returns the address of the block in main memory
  2. If the block is not in the buffer, the buffer manager
     1. Allocates space in the buffer for the block
        1. Replacing (throwing out) some other block, if required, to make space for the new block.
        2. Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
     2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.

Buffer-Replacement Policies

- Most operating systems replace the block **least recently used** (LRU strategy)
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
  - LRU can be a bad strategy for certain access patterns involving repeated scans of data
    - For example: when computing the join of 2 relations r and s by a nested loops
      for each tuple *tr* of *r* do
        for each tuple *ts* of *s* do
          if the tuples *tr* and *ts* match …
  - Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable

Buffer-Replacement Policies (Cont.)

- **Pinned block** – memory block that is not allowed to be written back to disk. E.g., an operation still needs this block.
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
  - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Buffer managers also support **forced output** of blocks for the purpose of recovery (more in Chapter 16 in the textbook)

Indexing and Hashing

Modified from:

Database System Concepts, 6th Ed.

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute or 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 some sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

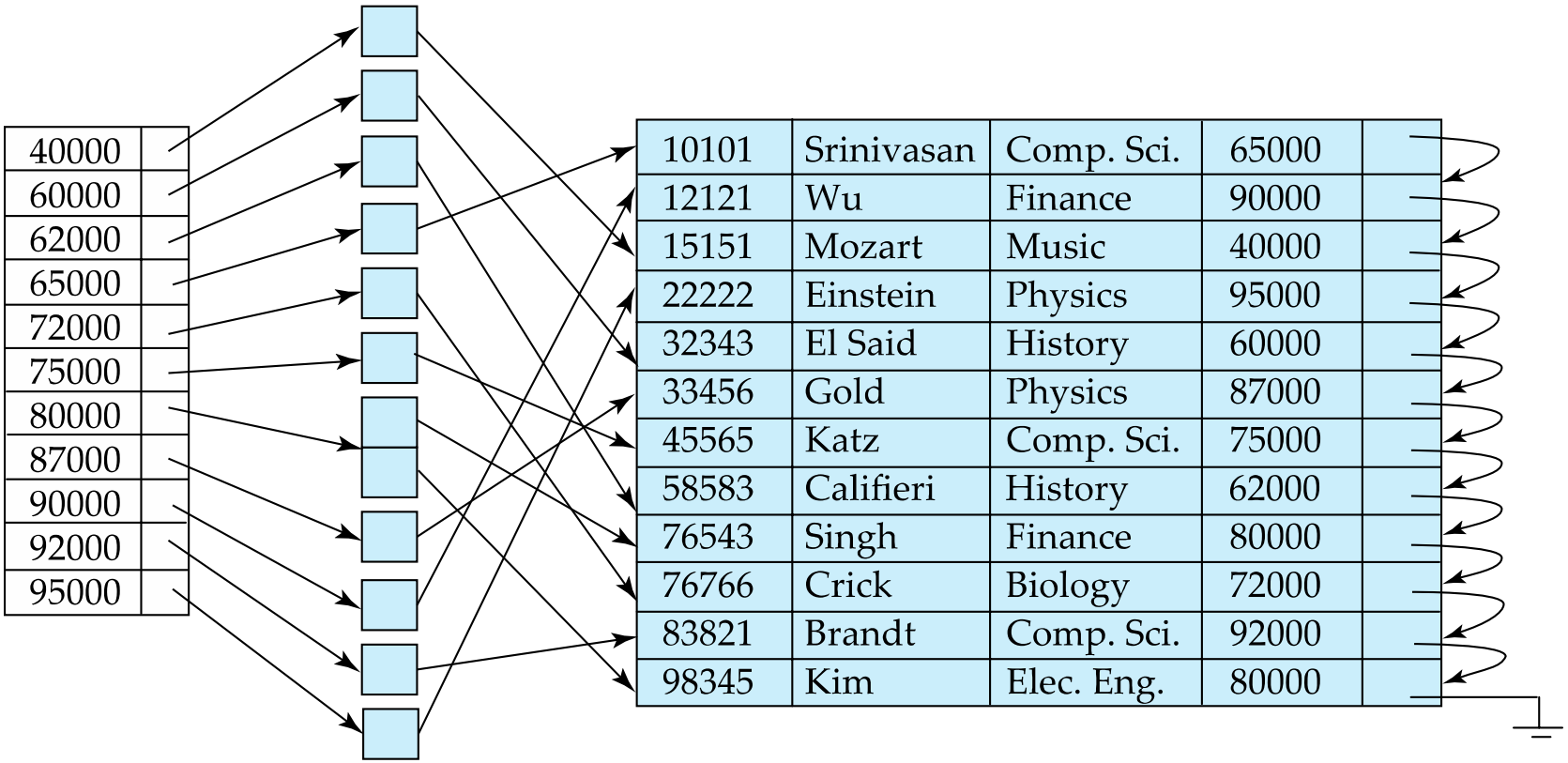
Index Evaluation Metrics

- 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.
- Access time
- Insertion time
- Deletion time
- Space overhead

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.

Secondary Indices Example



Secondary index on *salary* field of *instructor*

- 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

Primary and Secondary 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 milliseconds, versus about 100 nanoseconds for memory access

Secondary Indices

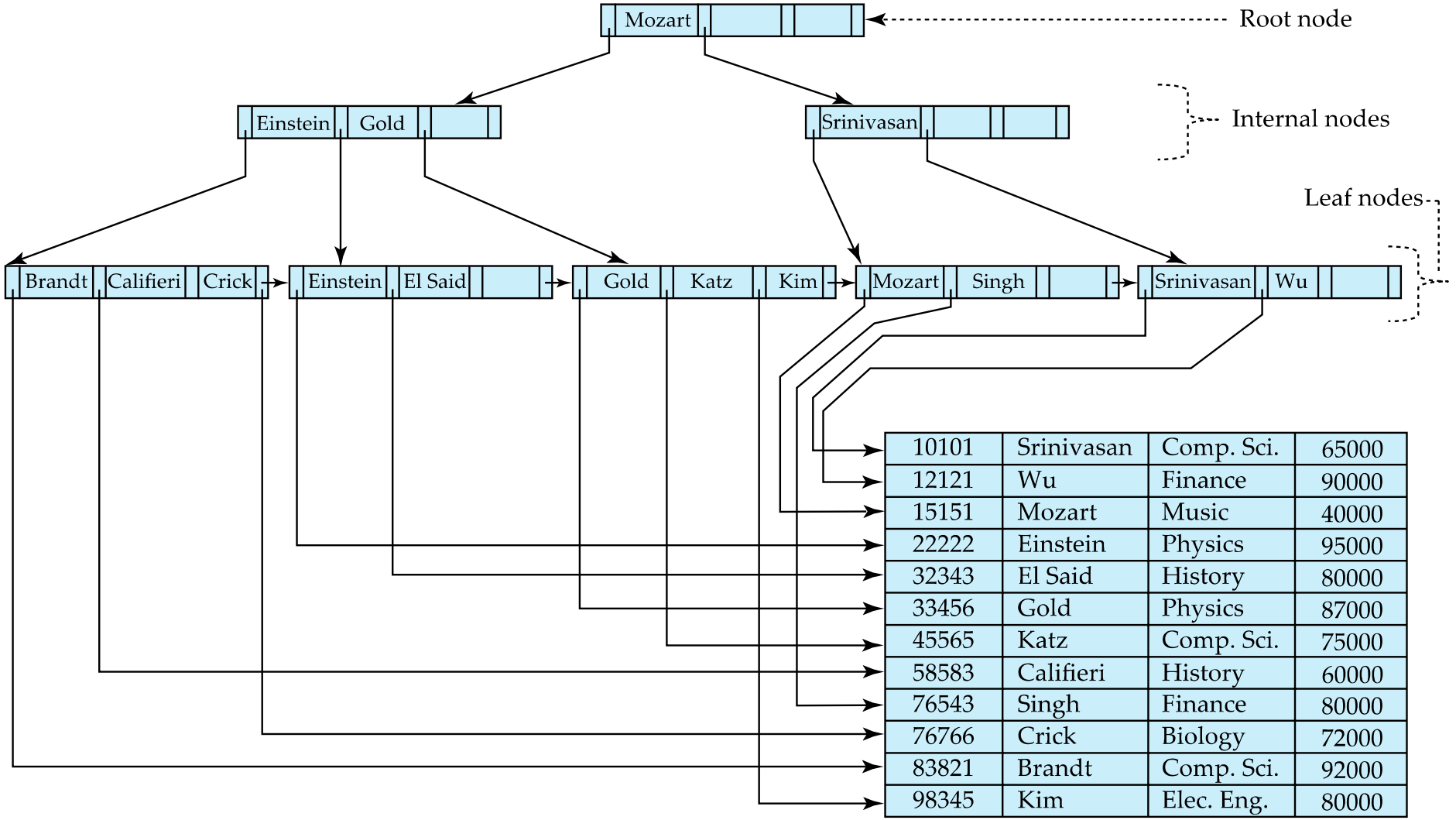
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

B^+-Tree Index

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

Example of B+-Tree



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

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

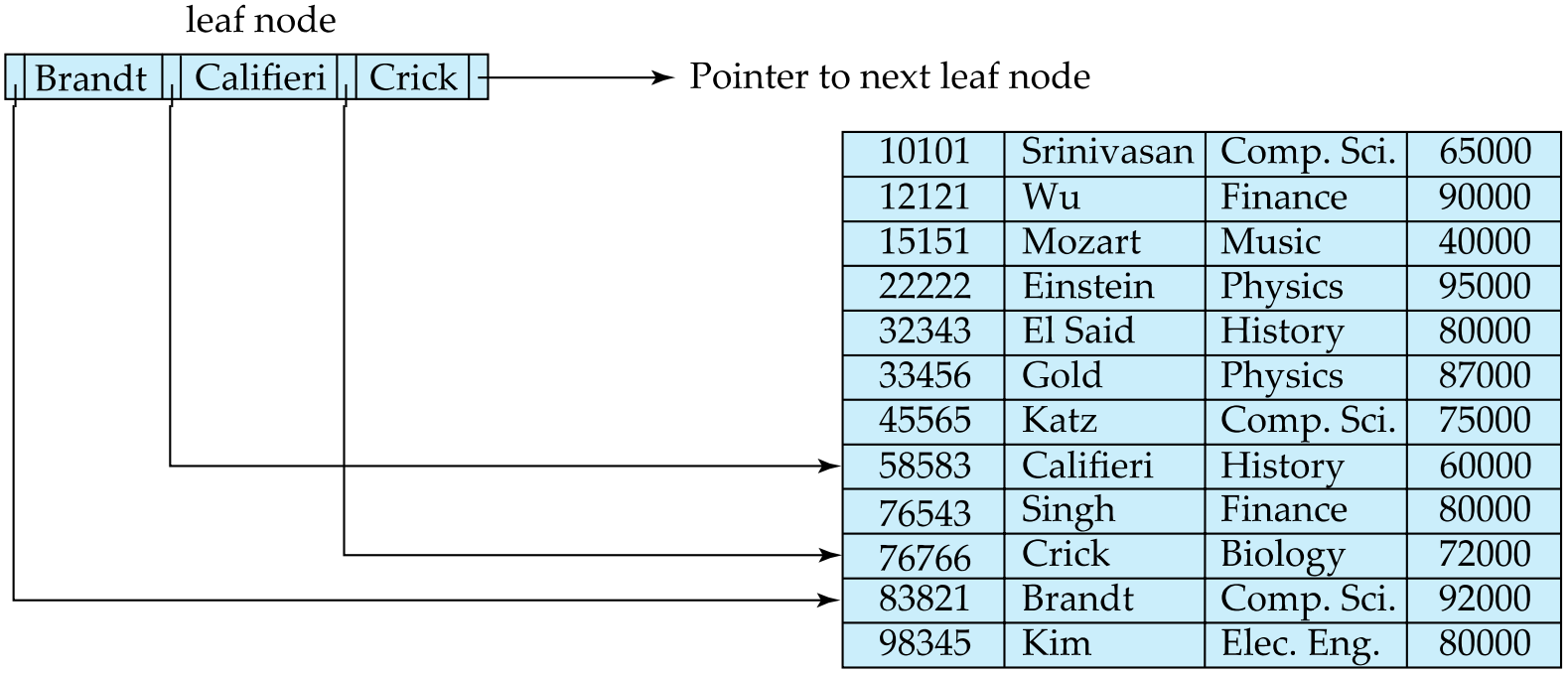
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+-Trees

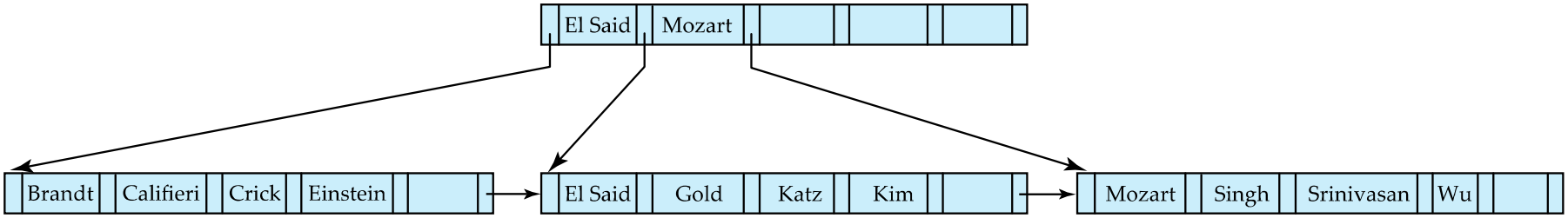
Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



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

Example of B+-tree



B+-tree for *instructor* file (n = 6)

- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with n = 6).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with n =6).
- Root must have at least 2 children.

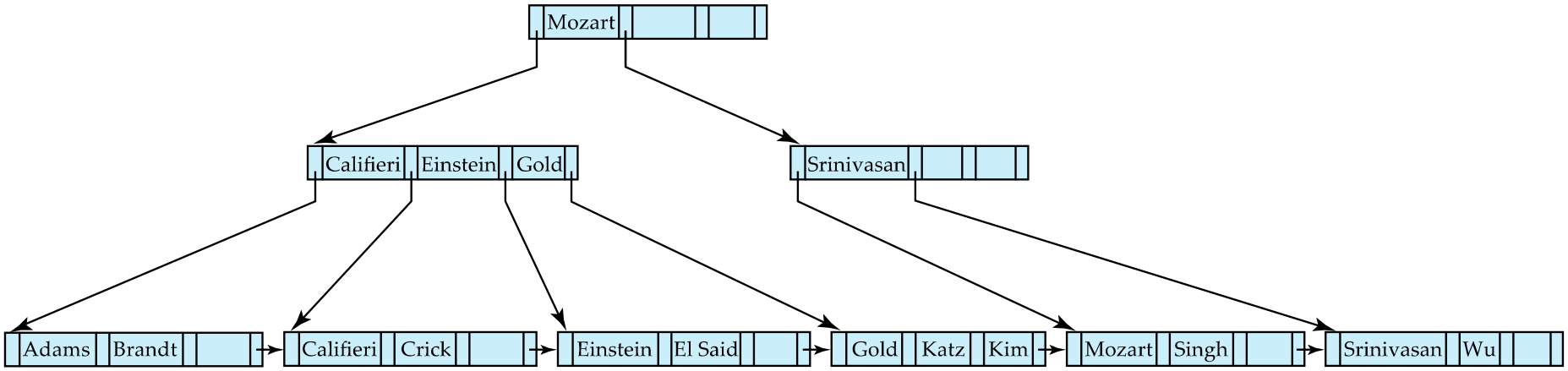
Observations about B+-trees

- 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 record with search-key value *V.*
  1. *C=root*
  2. While C is not a leaf node {
     1. Let *i* be least value s.t. $V \leq K_i$.
     2. If no such exists, set *C* = *last non-null pointer in C*
     3. Else { if (V= K_i) Set C = P_{i+1} else set *C* = P_i}

     }
  3. Let *i* be least value s.t. $K_i = V$
  4. If there is such a value *i,* follow pointer P_i to the desired record.
  5. Else no record with search-key value *k* exists.



Handling Duplicates

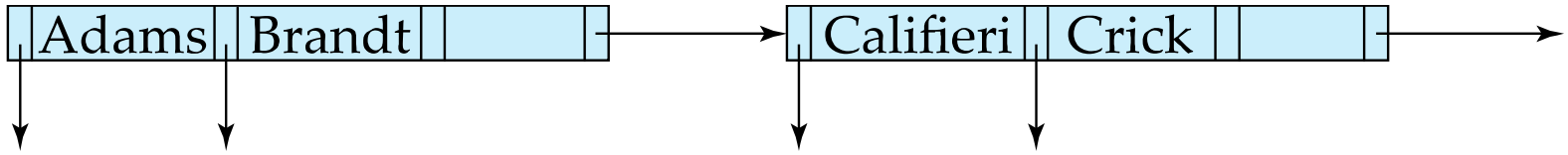
- With duplicate search keys
  - In both leaf and internal nodes,
    - we cannot guarantee that $K_1 < K_2 < K_3 < \ldots < K_{n-1}$
    - but can guarantee $K_1 \leq K_2 \leq K_3 \leq \ldots \leq K_{n-1}$
  - Search-keys in the subtree to which P_i points
    - are $\leq K_i$, but not necessarily $< K_i$,
    - To see why, suppose same search key value V is present in two leaf node L_i and L_{i+1}. Then in parent node K_i must be equal to V

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

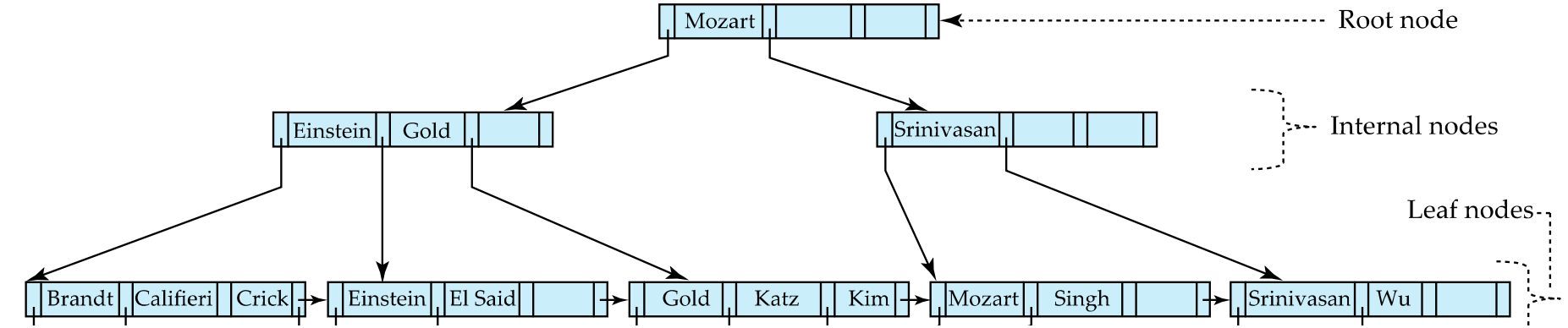
Updates on B+-Trees: Insertion (Cont.)

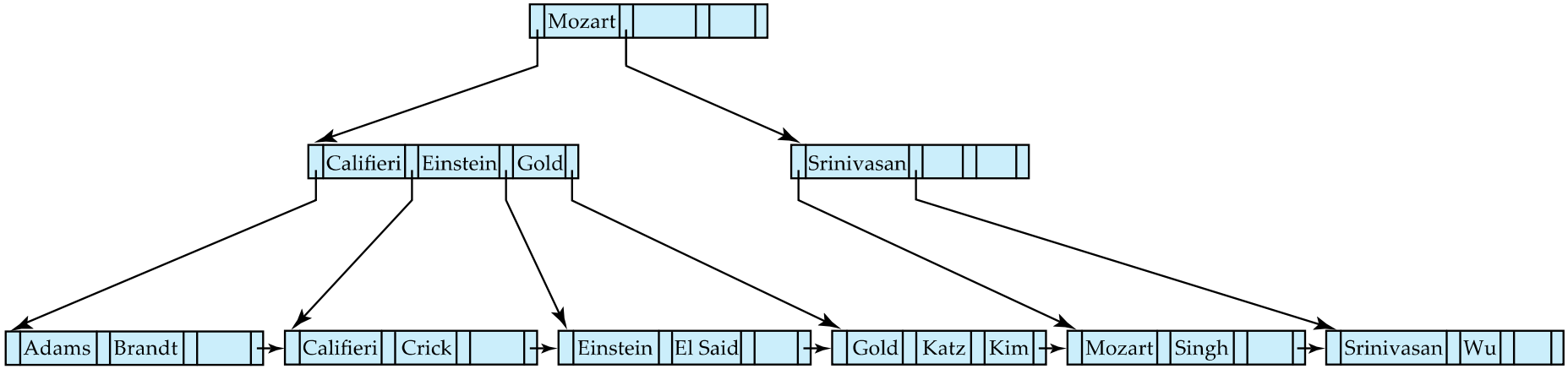
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) 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 of the tree by 1.



Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

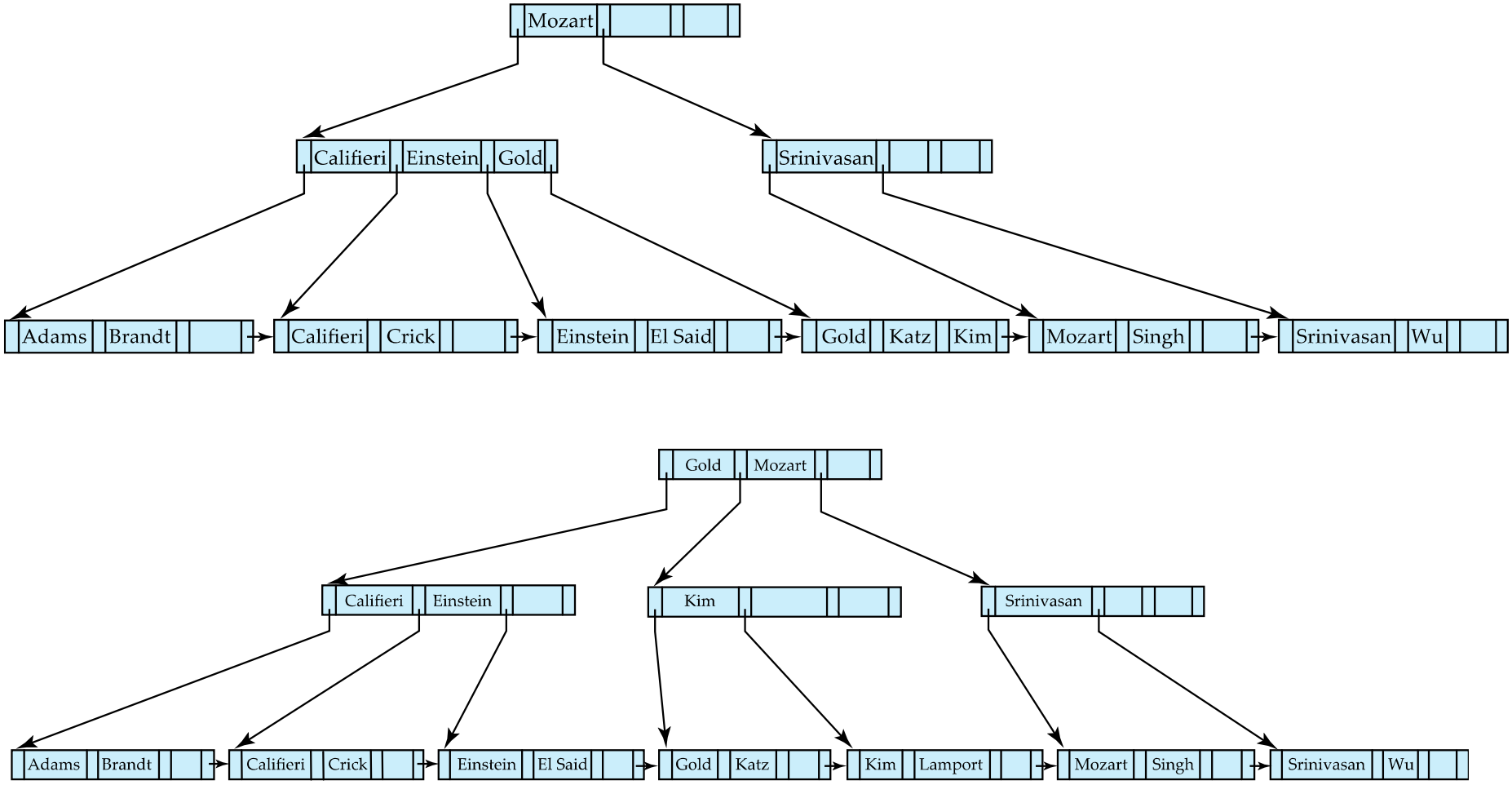
B+-Tree Insertion





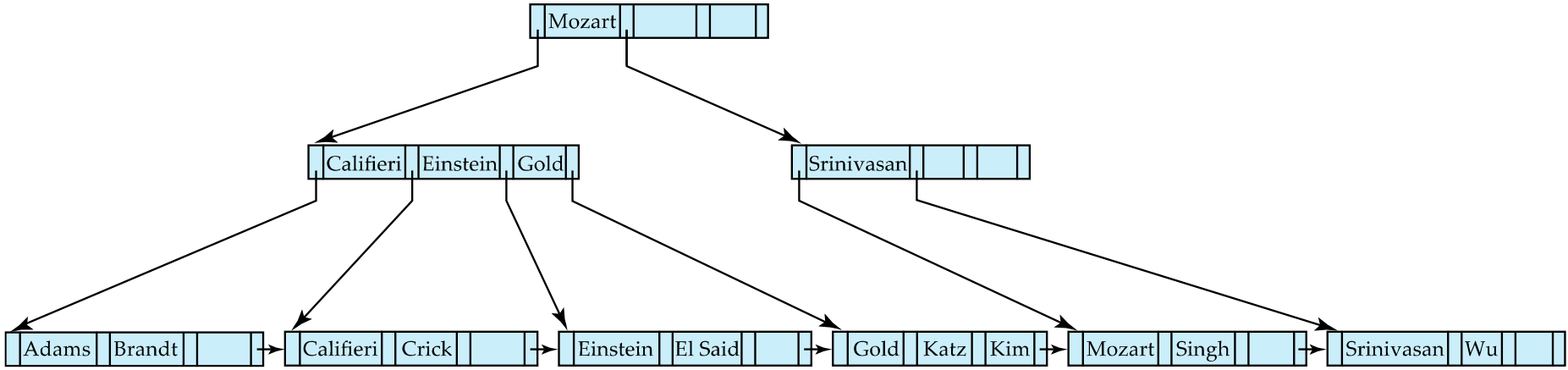
B+-Tree before and after insertion of “Adams”

B+-Tree Insertion

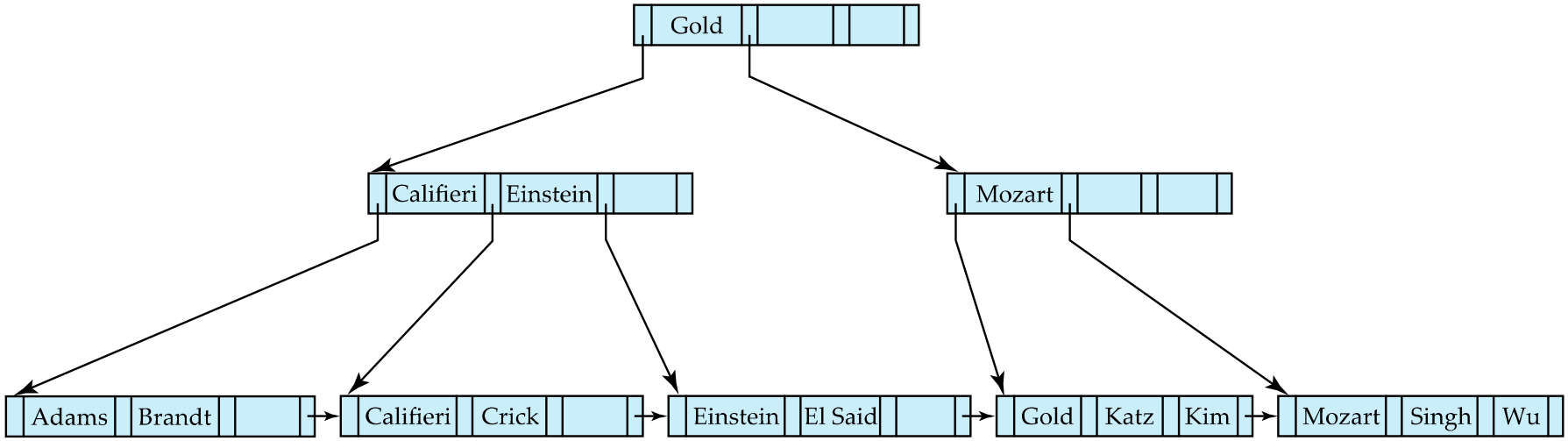


B+-Tree before and after insertion of "Lamport"

Examples of B+-Tree Deletion

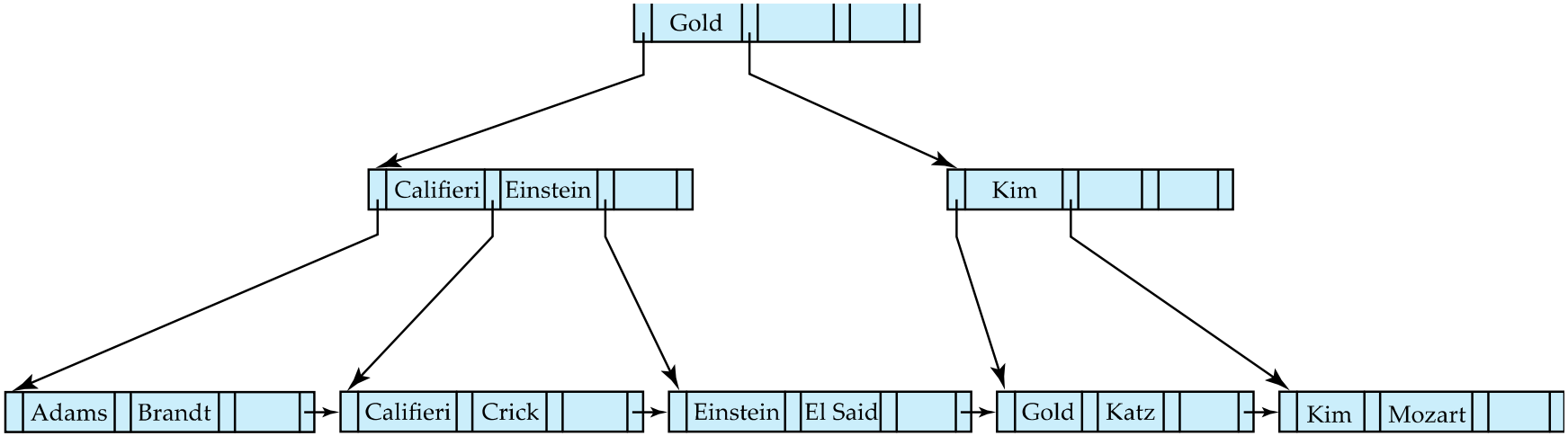


Before and after deleting "Srinivasan"



- Deleting "Srinivasan" causes merging of under-full leaves

Examples of B+-Tree Deletion (Cont.)



Deletion of "Singh" and "Wu" from result of previous example

- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result

Non-Unique Search Keys

- Alternatives to scheme described earlier
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

Hashing

Modified from:

Database System Concepts, 6th Ed.

Static Hashing

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

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key
(See figure in next slide.)

- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3

Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of *instructor* file, using *dept_name* as key (see previous slide for details).

Hash Functions

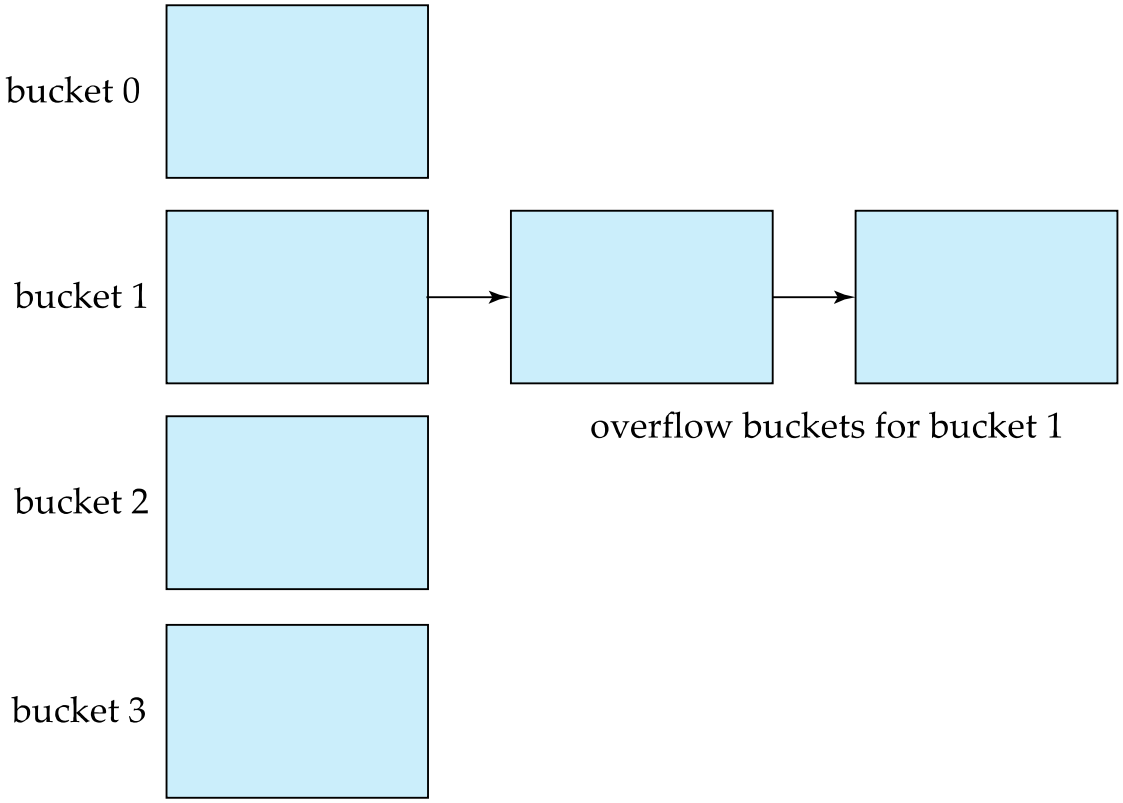
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
-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

Handling of Bucket Overflows

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

Handling of Bucket Overflows (Cont.)

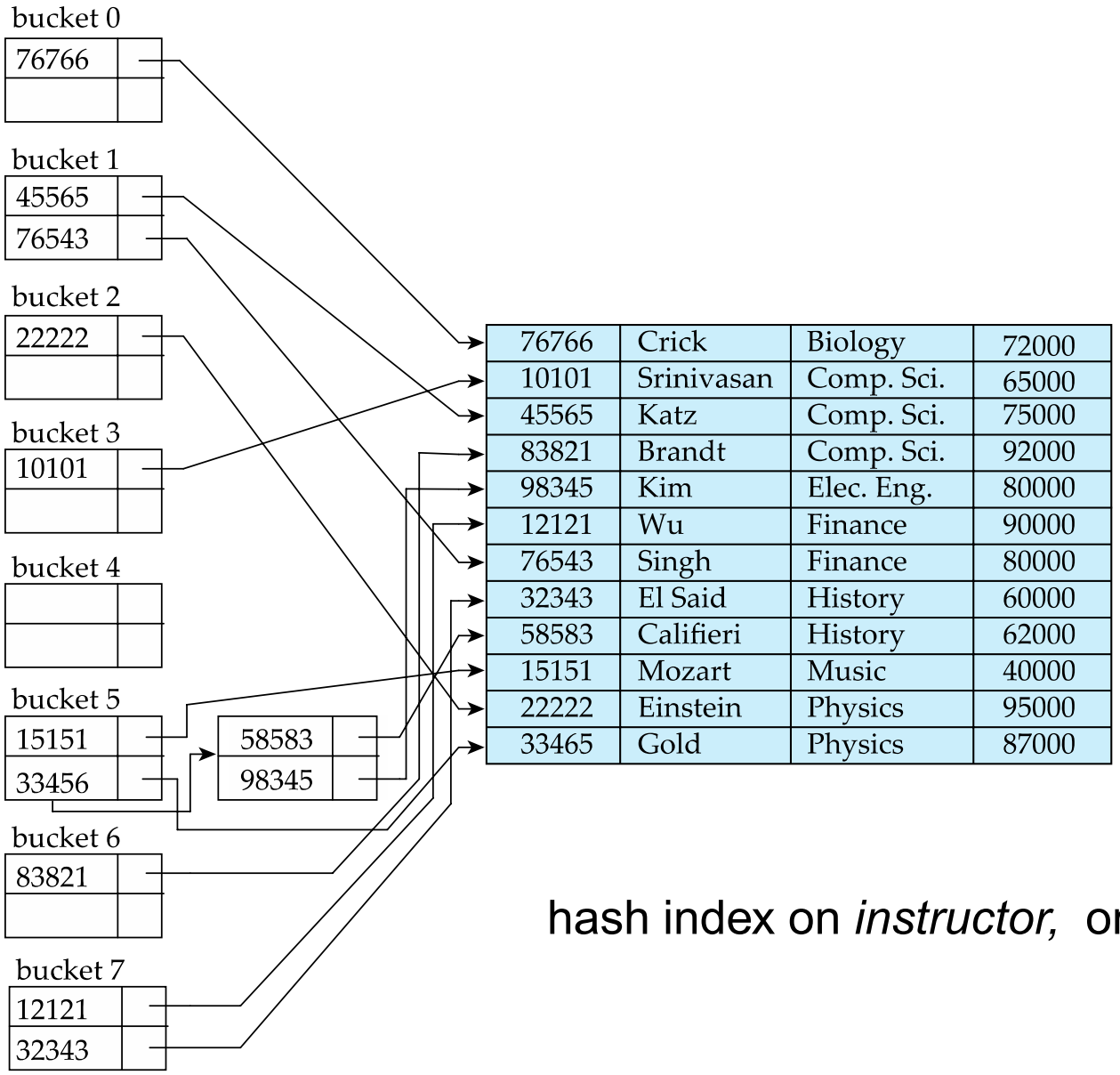
- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
  - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.



Hash Indices

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

Example of Hash Index



76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

hash index on *instructor,* on attribute *ID*

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

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.
- To drop an index

  drop index <index-name>

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

End of Chapter

Modified from:

Database System Concepts, 6th Ed.