Lecture 12
Data storage and indexing

Subject Lecturer: Kevin K.F. YUEN, PhD.

Acknowledgement: Slides were offered from Prof. Ken Yiu.
Some parts might be revised and indicated.

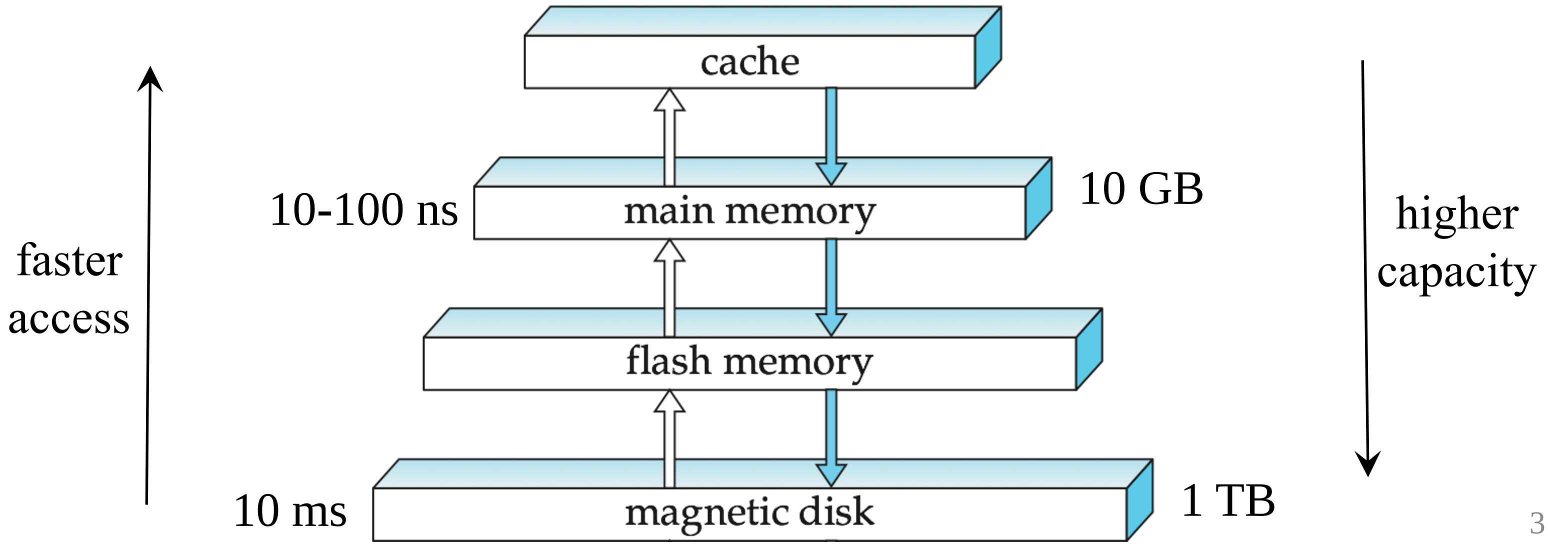
Slides content modified from
Database Systems Concepts

Outline

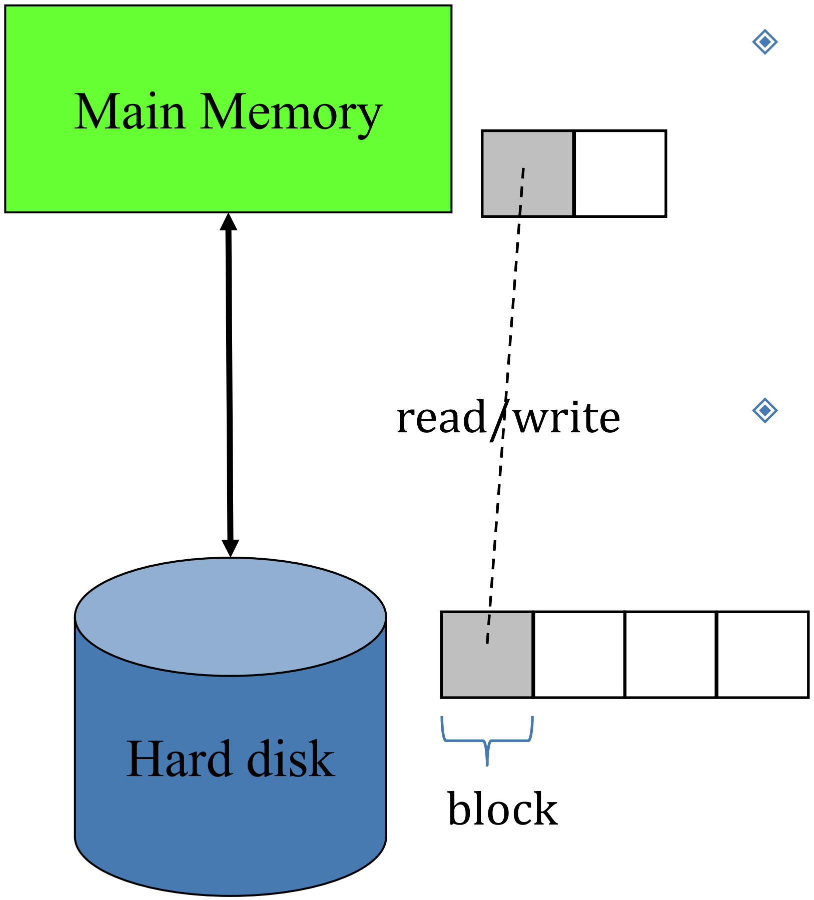
- Storage media
- File organization (on disk)
- Types of indexing
- B+-tree
- Hashing

Storage Hierarchy

- **Primary storage**: Faster but volatile
  - E.g., CPU cache, main memory
  - Loses content when power is switched off
- **Secondary storage**: non-volatile, slower, higher capacity
  - E.g., flash memory, magnetic disks
  - Content persists even when power is switched off

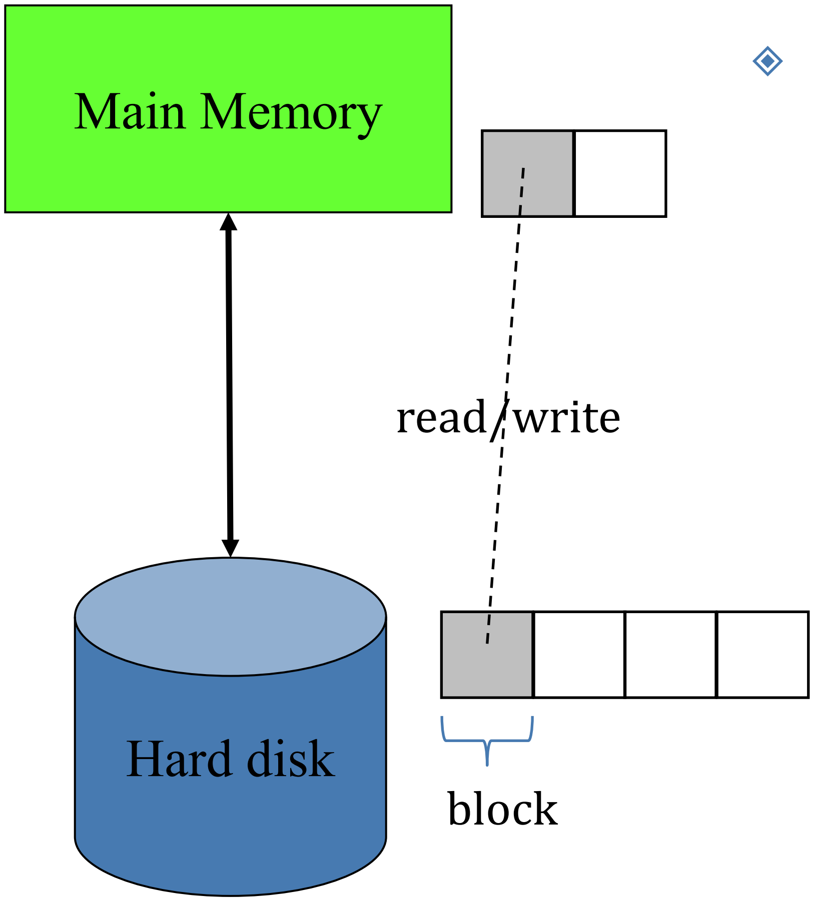


Storage



- RDBMS stores data in hard disk because
  - Large capacity (e.g., 1 TB)
  - Non-volatile (Keep data without power)
- **Disk block** – unit of data transfer between disk and main memory
  - Typical block size: 1 kilobytes (KB)
    - Much larger than an attribute’s size

Storage



- Access time = $b * t_T + s * t_S$
  - b – the number of disk block transfers
  - s – the number of seeks
  - t_T – time to transfer one disk block
  - t_S – time to seek one disk block

- Typical values:
  - seek time = 10 milliseconds (ms),
  - data transfer rate = 100 MB/s
  - → much slower than CPU!

Outline

- Storage media
- File organization (on disk)
- Types of indexing
- B+-tree
- Hashing

File Organization

- Store a **database** as a collection of *files*; store a **file** as sequence of *records*.
  - A record is a sequence of fields
- Simple approach:
  - Assume fixed record size; a file is used to store a relation
  - Store record *i* at byte $n*(i–1)$
    - *n* is the size of a record
  - but records may cross blocks
    - Modification: do not allow records to cross a block

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

How to delete record *i* ?

- Suppose we are going to delete record 3
  - Option 1: Delete record 3 and compact
  - Option 2: Delete record 3, then move last record
- Both options require additional cost

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
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

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
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

Free Lists

- Store the address of the 1^{st} deleted record in the file header
- Use 1^{st} deleted record to store the address of the 2^{nd} deleted record
  - These stored addresses are considered as **pointers** to the location of a record
- Space efficient representation:
  - reuse space for normal attributes of free records to store pointers

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

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 (Cont.)

- 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 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	

Outline

- Storage media
- File organization (on disk)
- Types of indexing
- B+-tree
- Hashing

- Example queries (in SQL)
  - select * from **instructor** where ID=22222
  - select * from **instructor** where dept_name="Music"

- How to search records efficiently?

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

Basic Concepts

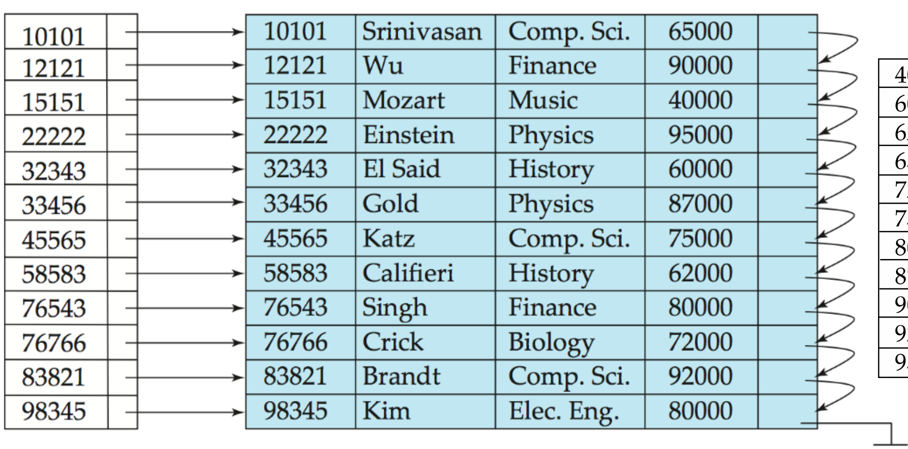
- Indexing mechanism is used to search records quickly
  - E.g., author catalog in library
- **Search key** - attribute to set of attributes used to look up records in a file
- An **index file** consists of records (called **index entries**) of the form

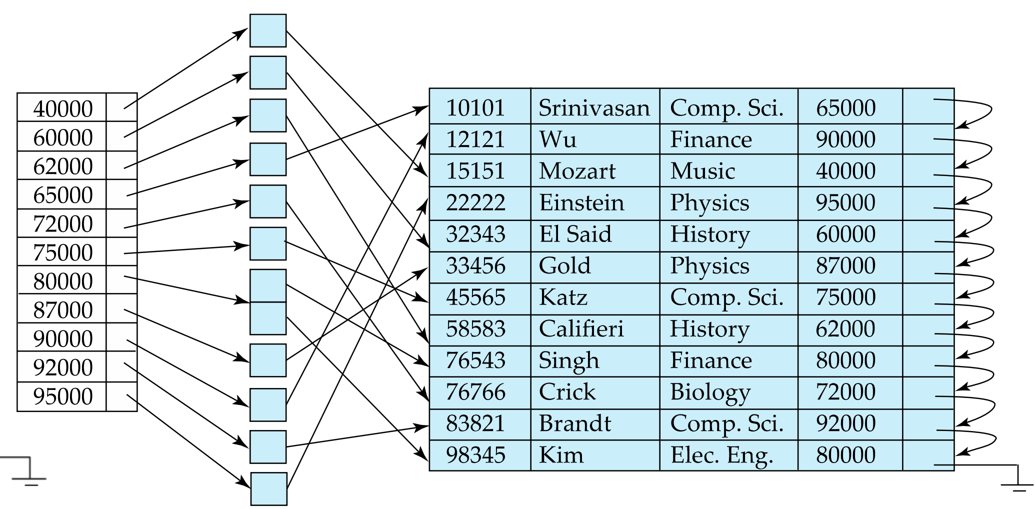
search-key	pointer

- Index files are much smaller than the original file
  - **Ordered index:** search keys are stored in sorted order
  - **Hash index:** search keys are distributed uniformly across "buckets" using a "hash function"

Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value
- **Primary index**: an index whose search key specifies the sequential order of the file
  - Also called **clustering index**
  - The search key of a primary index is usually 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





Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file

Dense index on *dept_name*, with *instructor* file sorted on *dept_name*

Biology
Comp. Sci.
Elec. Eng.
Finance
History
Music
Physics

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

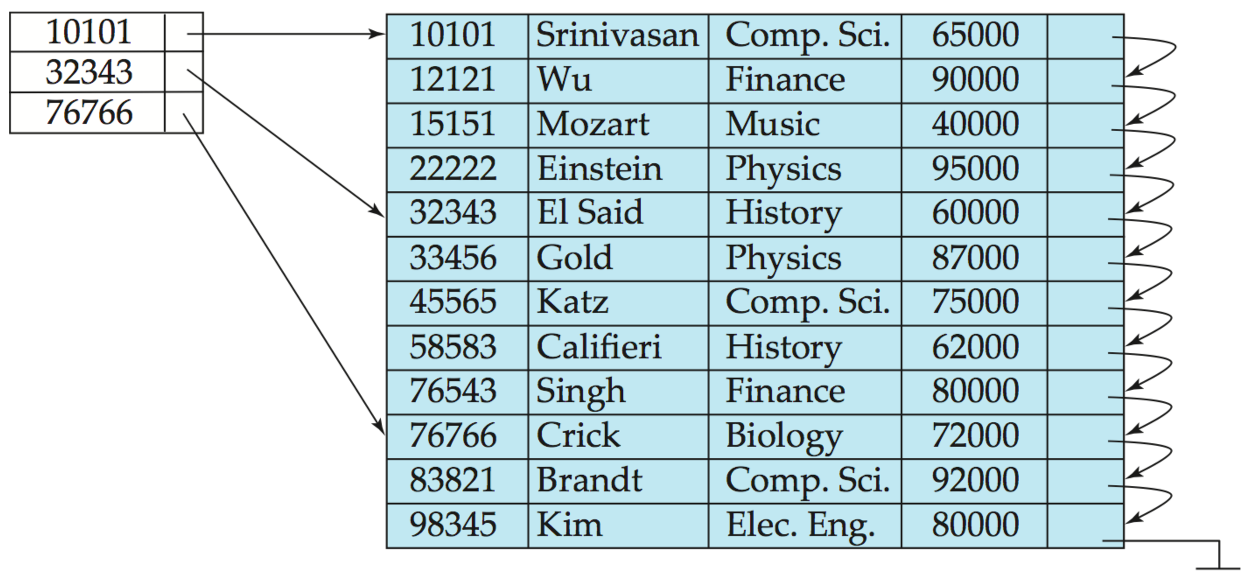
Dense index on *ID* attribute of *instructor* relation

10101
12121
15151
22222
32343
33456
45565
58583
76543
76766
83821
98345

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

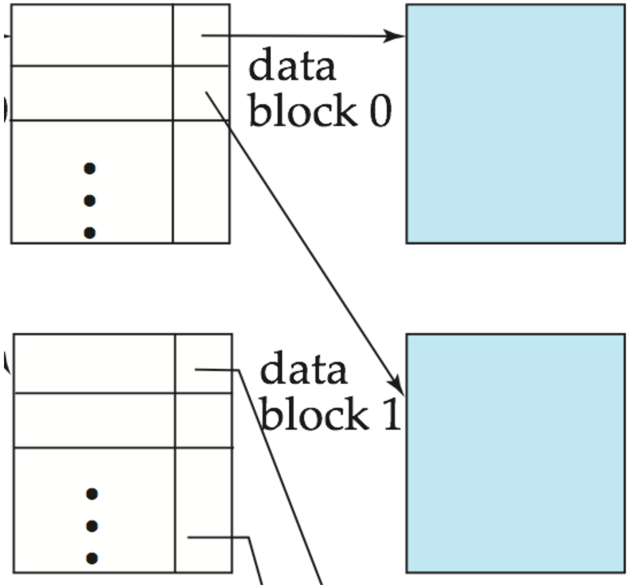
Sparse Index Files

- **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



Sparse Index Files (Cont.)

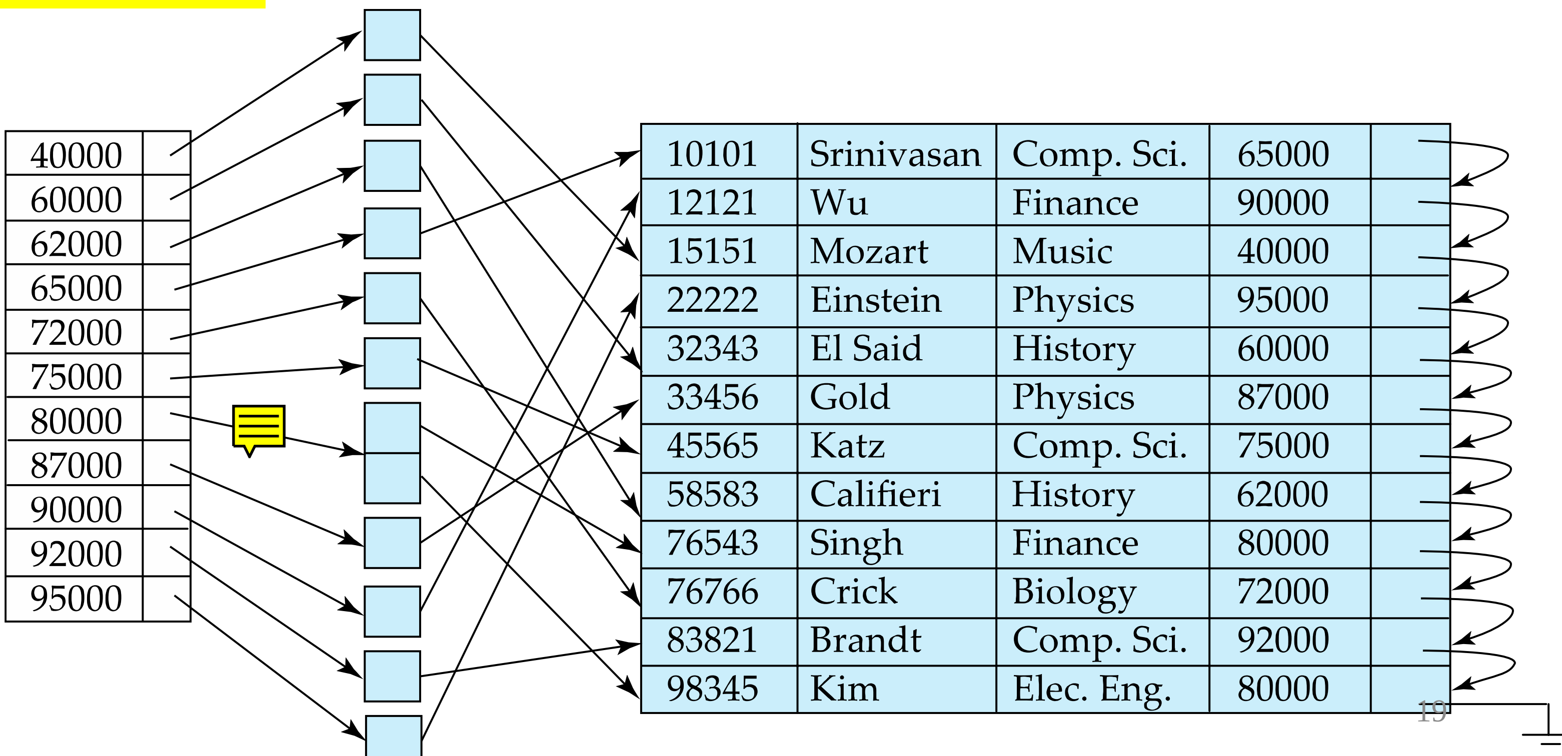
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions
  - Slower than dense index for locating records
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block



Secondary Indices Example

- The user may want to search records by an attribute which is not the search-key of the primary index, e.g.,
  - The *instructor* relation is stored sequentially by ID
  - Query: find all instructors with *salary* in a specified range of values
- 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. Why?

Secondary index on *salary* field of *instructor*



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

Drawbacks of previous index files

- Drawback: expensive to update an index file
- E.g., when we delete a record of key "58583", what happens?

10101
12121
15151
22222
32343
33456
45565
58583
76543
76766
83821
98345

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

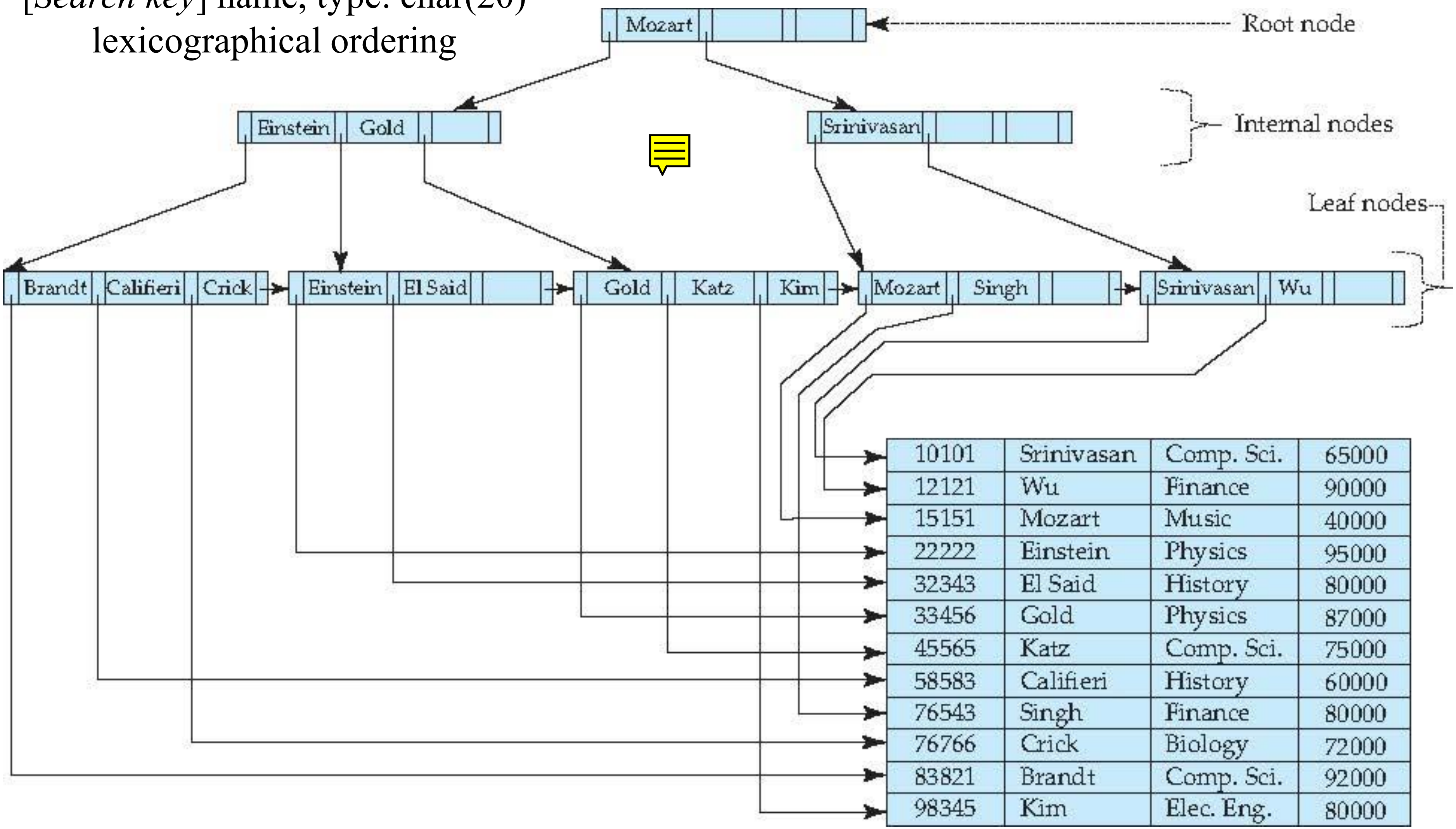
- That's why we need a better structure for index file (e.g., B^+-tree index file)
  - Fast to organize the structure during insertions/ deletions

Outline

- Storage media
- File organization (on disk)
- Types of indexing
- B+-tree
- Hashing

Example of B$^+$-Tree

[*Search key*] name, type: char(20)
lexicographical ordering



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.)

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

- Each node occupies a disk block
- 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 (i.e., no other nodes in the tree), it can have between 0 and $(n-1)$ values

Note that the definitions of B^+-tree node and B^+-tree height are different from Lecture 3!

B+-Tree Node Structure

- Typical node

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - K_i is a search-key value
  - P_i is a pointer to a child node (for non-leaf node) or a pointer to a record (for leaf node)
    - Any search-key $SK_?$ in the subtree of P_i must satisfy:

$$K_{i-1} \leq SK_? < K_i$$

- The search-keys in a node are ordered

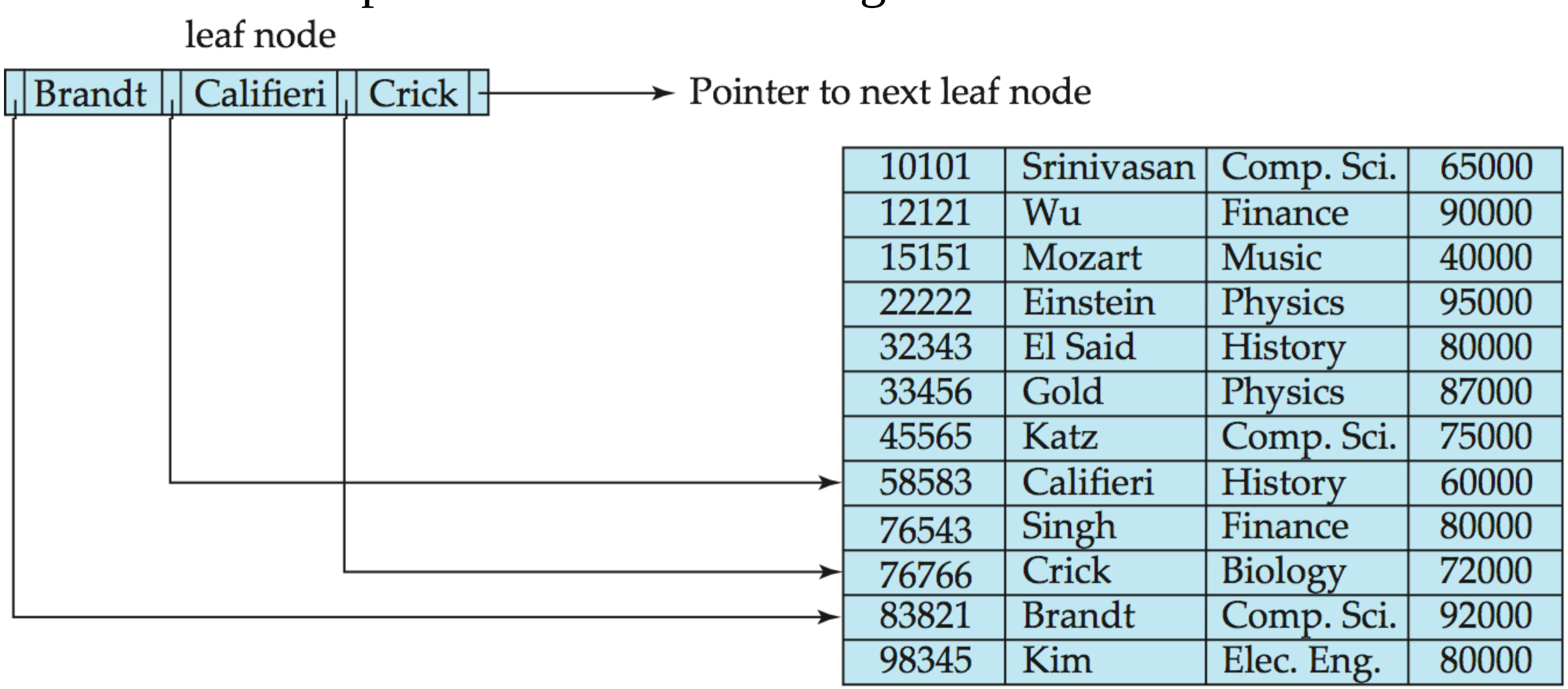
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(assume no duplicate keys)

Leaf Nodes in B$^+$-Trees

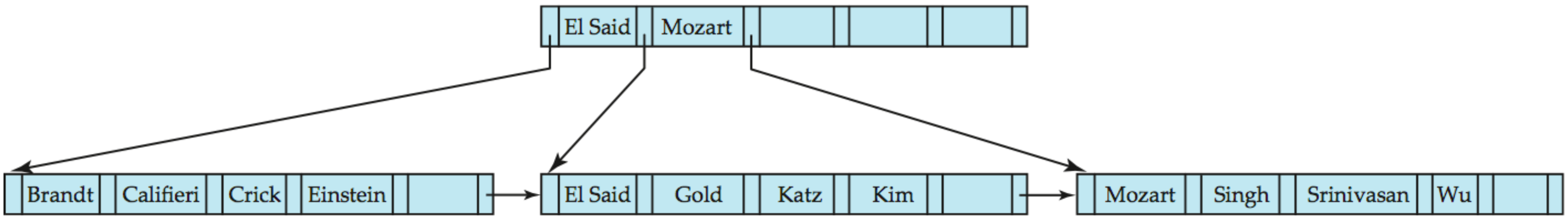
Properties of a leaf node:

- 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
  - This pointer is useful for range search



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 ($n = 6$)

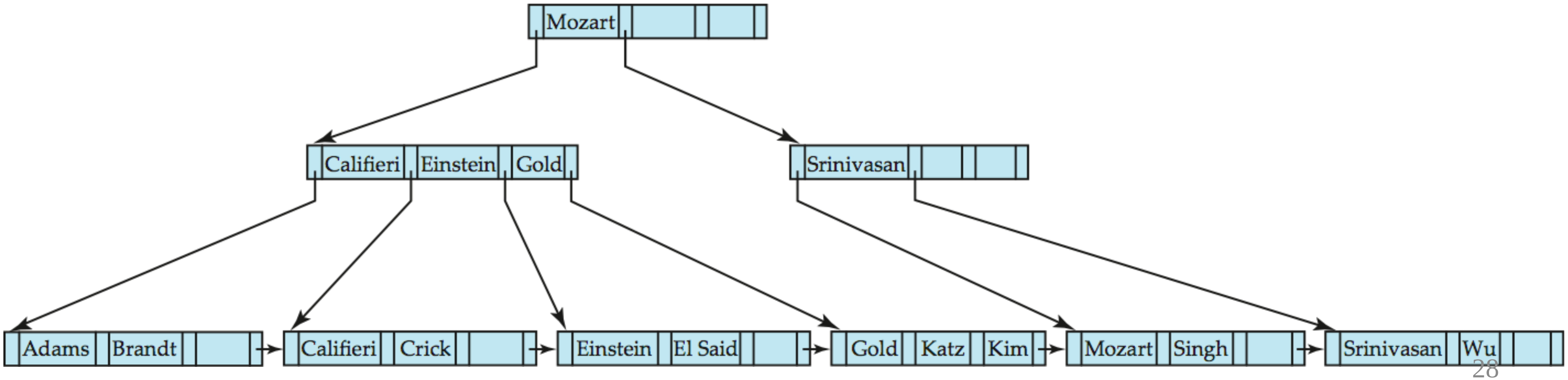
- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$)
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with $n = 6$)
- Root must have at least 2 children

Observations about B^+-trees

- The tree height of a B^+-tree is $\lceil \log_{\lceil n/2 \rceil}(NK) \rceil$
  - NK is the number of search-key values in the file
  - n is the maximum number of children of a node
- Cost to search a record with search-key value V
  - Cost = $\lceil \log_{\lceil n/2 \rceil}(NK) \rceil$ disk block accesses
  - Example:
    - Given: 1 million search keys, disk block size = 4000 bytes, pointer size = 4 bytes, key size = 36 bytes
    - $4n + 36(n-1) \leq 4000 \rightarrow 40n \leq 4036$
      $\rightarrow n \leq 100.9$
    - Lookup cost = $\lceil log_{50}(1{,}000{,}000) \rceil = \lceil 3.532 \rceil = 4$ block accesses
- Insertions and deletions can be handled efficiently, as the index can be restructured in logarithmic time

Queries on B$^+$-Trees

- Find record with search-key value *V*.
  1. $C = root$
  2. While C is not a leaf node {
     2.1. Let *i* be least value s.t. $V \leq K_i$.
     2.2. If no such exists, set C = *last non-null pointer in C*
     2.3. 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.



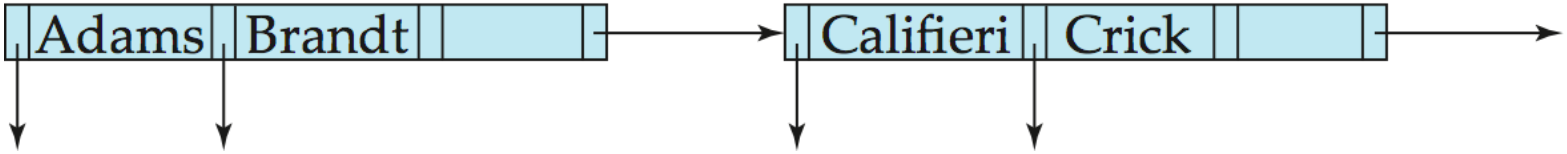
Updates on B^+-Trees: Insertion

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
    2.1. Add record to the file
    2.2. If necessary add a pointer to the bucket.
3. If the search-key value is not present, then
    3.1. add the record to the main file (and create a bucket if necessary)
    3.2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
    3.3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide

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

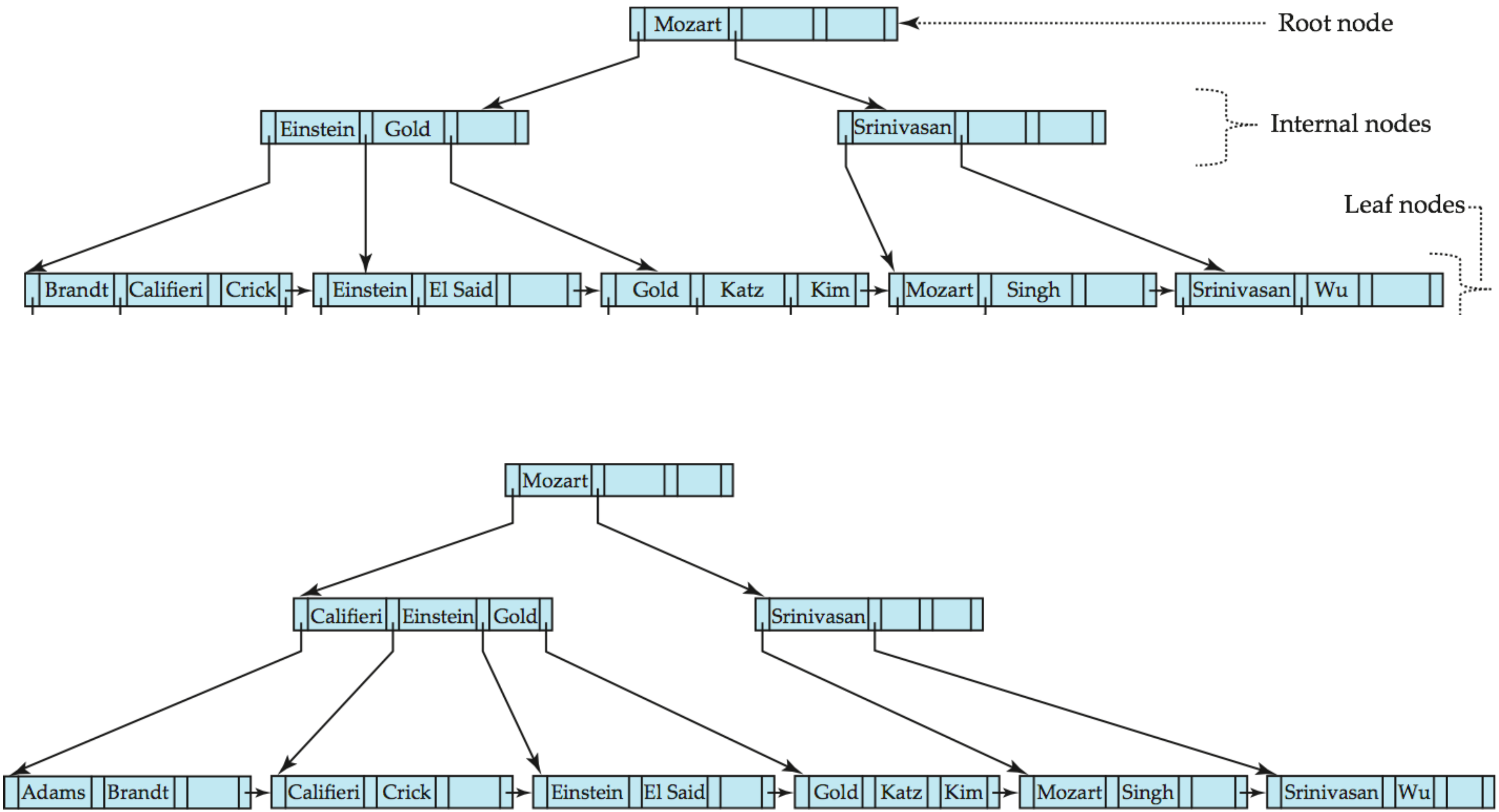
- Splitting a leaf node:
  - Sort the *n* (search-key value, pointer) pairs (including the inserted pair). Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - Let the new node be *p*, and let *k* be the least key value in *p*. Insert (*k*,*p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.

- Splitting of nodes proceeds upwards till reaching a non-full node
  - In the worst case, the root node may be split → increase the tree height by 1



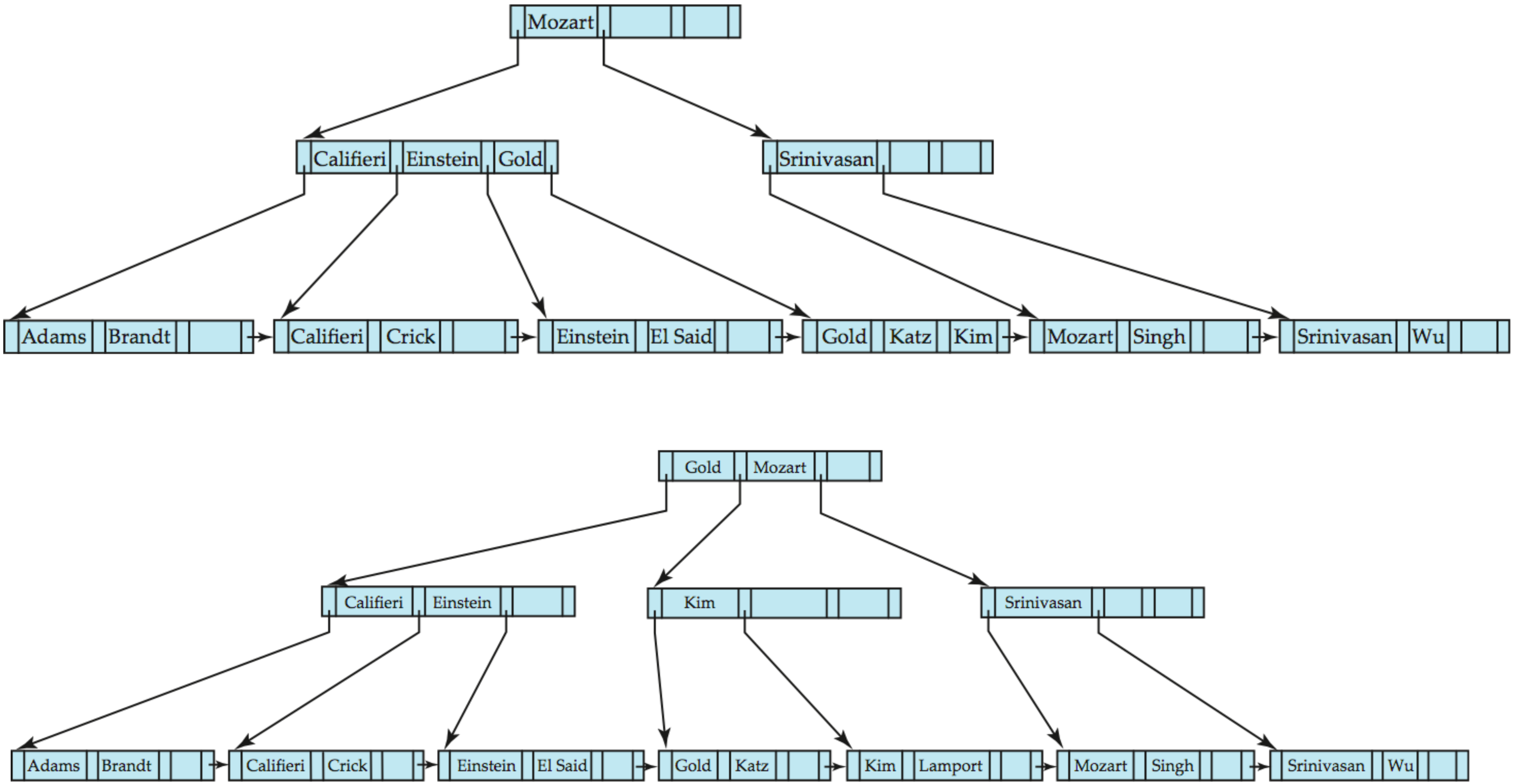
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

B$^+$-Tree Insertion



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

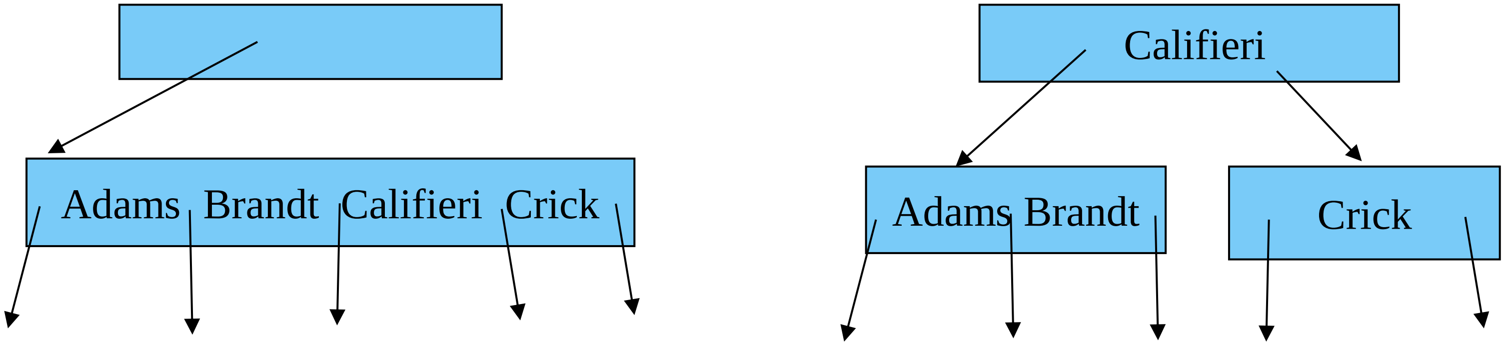
B+-Tree Insertion



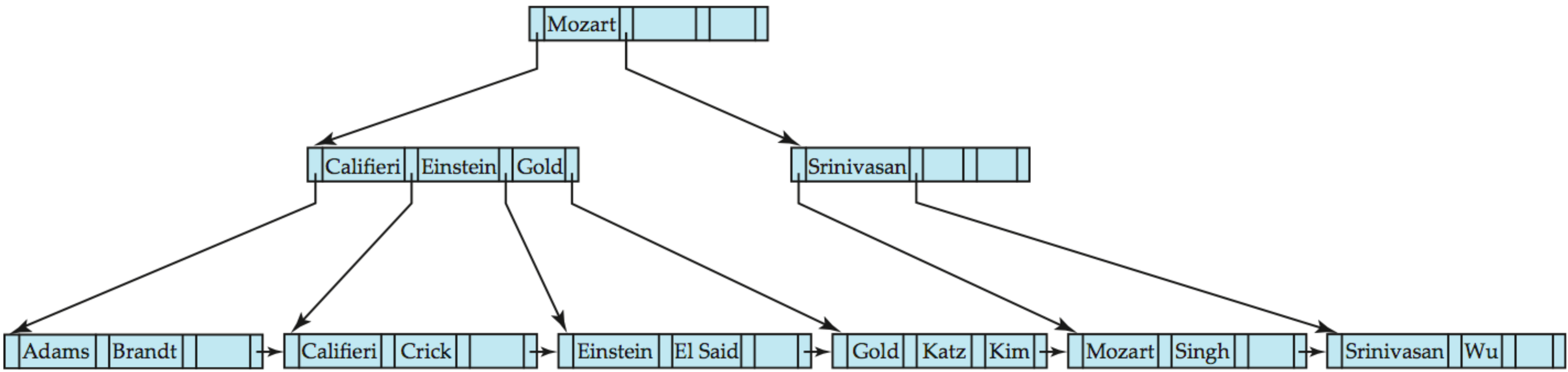
B+-Tree before and after insertion of "Lamport"

Insertion in B^+-Trees (Cont.)

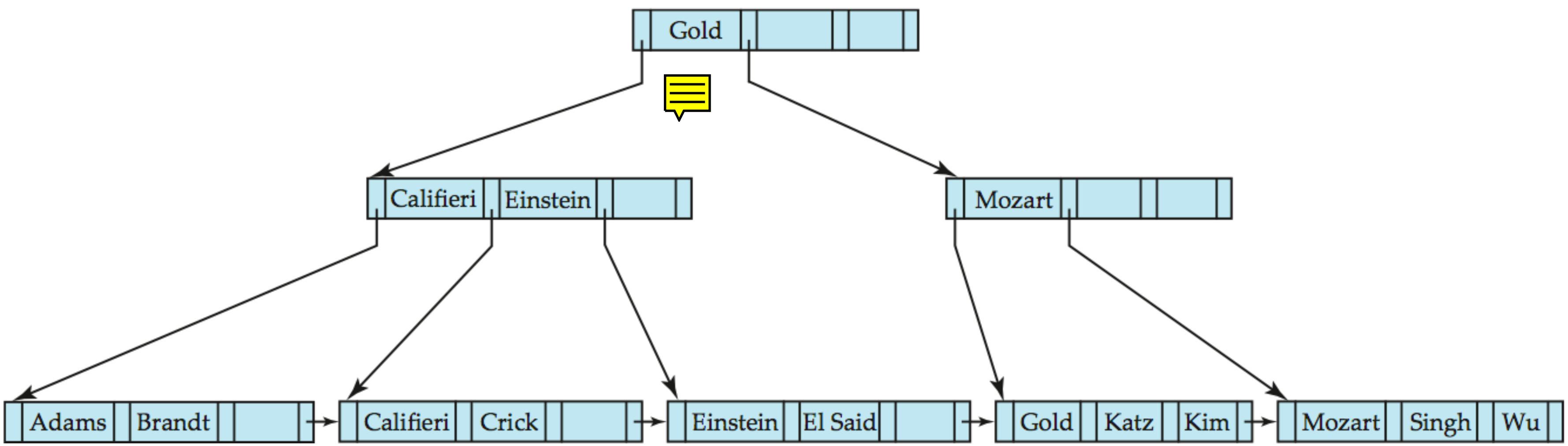
- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil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



Examples of B$^+$-Tree Deletion

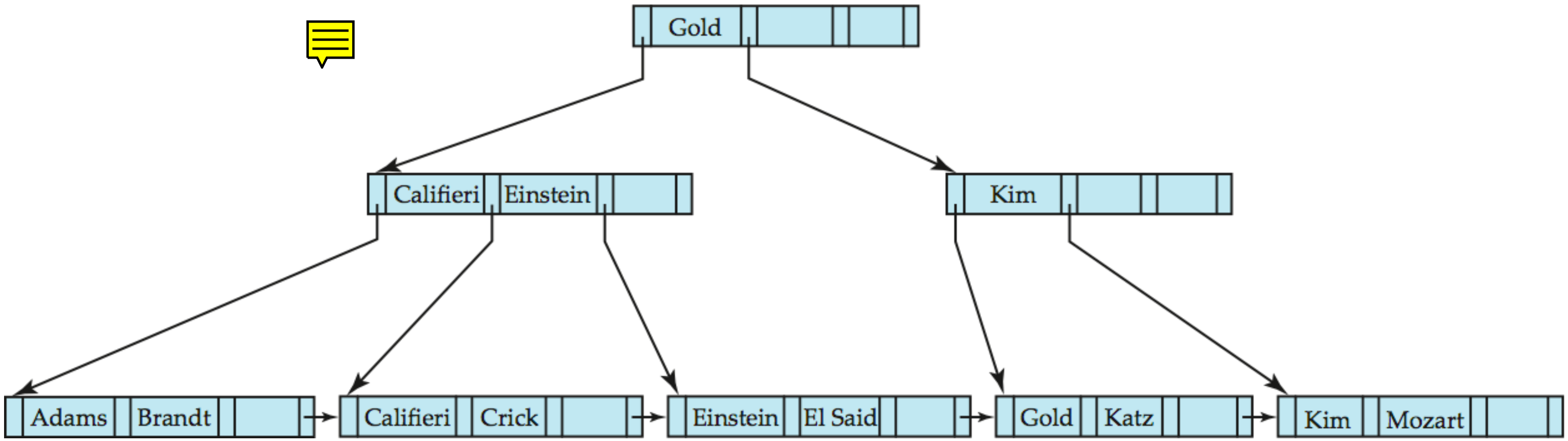


Before and after deleting "Srinivasan"



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

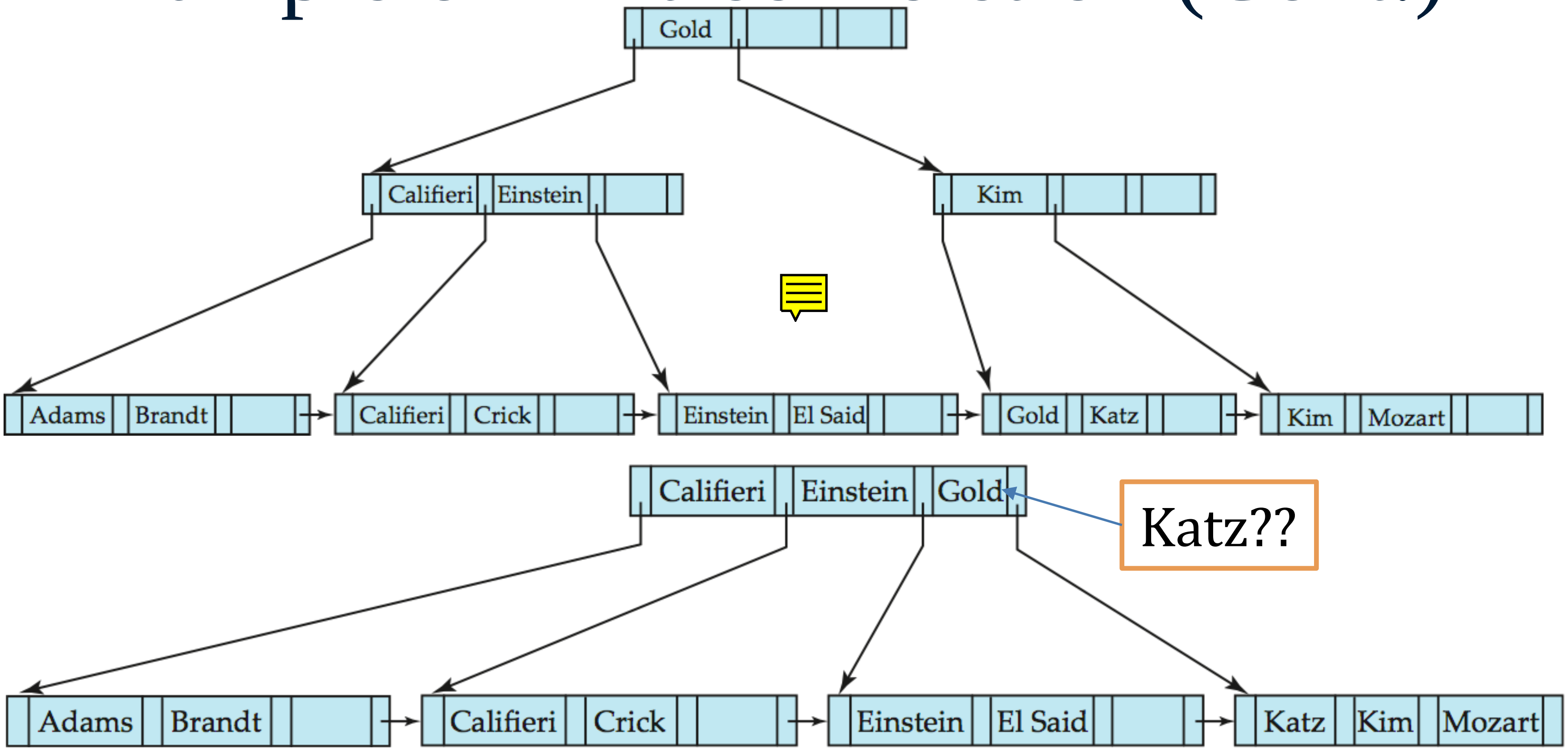
Examples of B+-Tree Deletion (Cont.)



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

Example of B+-tree Deletion (Cont.)



Before and after deletion of "Gold" from earlier example

- Node with "Gold" and "Katz" became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

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

- After removal, if the node has too few entries, what should be done?

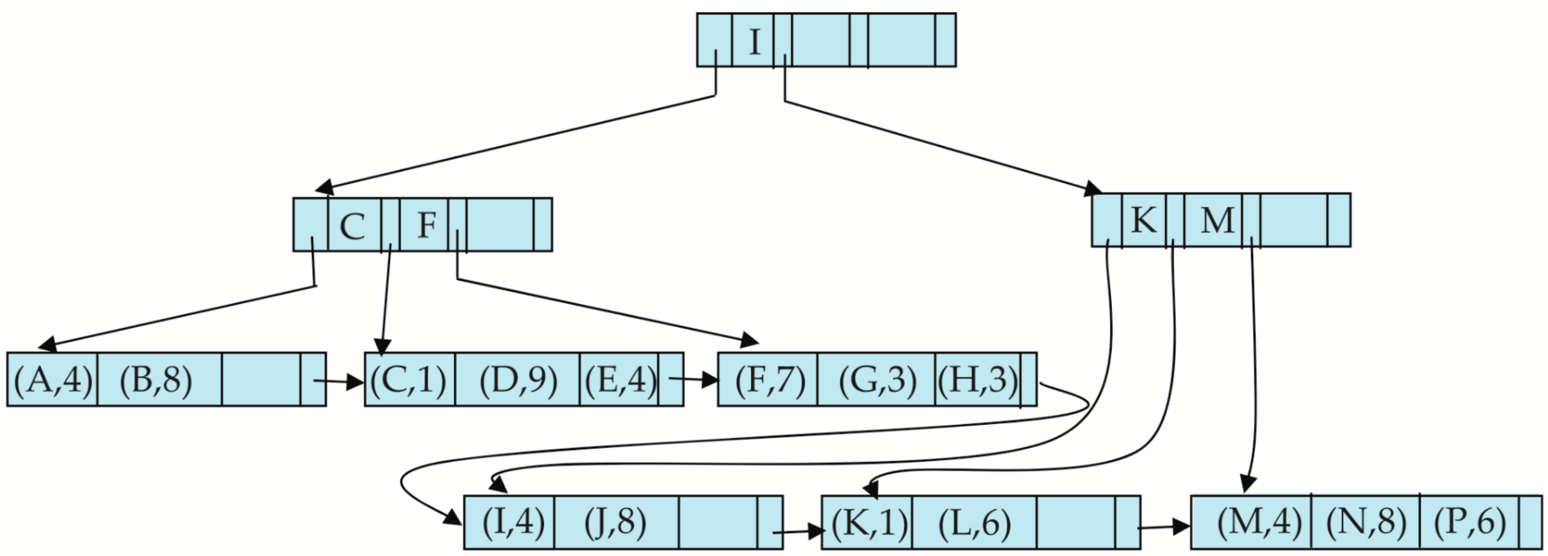
Updates on B$^+$-Trees: Deletion

How to process an underfull node?

- Option 1: if the entries in the node and a sibling fit into a single node, then ***merge a node with a sibling***
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent
- Option 2: if 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 File Organization

- B⁺-Tree File Organization
  - The leaf nodes store records, instead of pointers
  - Leaf nodes are still required to be half full
- Handle insertion & deletion in the same way as a B⁺-tree index
- How to improve space utilization of leaf nodes?
  - During node splitting/merging, involve 2 siblings in redistribution → each node having at least $\lfloor 2n/3 \rfloor$ entries



Outline

- Storage media
- File organization (on disk)
- Types of indexing
- B+-tree
- Hashing

Static Hashing

- A **bucket** is a unit of storing records
  - A bucket occupies 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 that takes a search-key value K as input, and returns a bucket identifier $h(K)$
  - To search a record with search-key K,
    go to the bucket $h(K)$,
    then use sequential search in that bucket

Example of Hash File Organization

- Suppose that there are 8 buckets
- Consider the hash function H(*dept_name*) $= (\Sigma_{i=0..dept\_name.\text{length}-1}\ \text{Code}(dept\_name[i])\ )\ \text{mod}\ 8$
  - where Code(*ch*) = toLowerCase(*ch*) – 'a' + 1
- Examples of hash values
  - H(Music) = 1
    - (13+21+19+9+3) mod 8
  - H(History) = 2
    - (8+9+19+20+15+18+25) mod 8
  - H(Physics) = 3
    - (16+8+25+19+9+3+19) mod 8

Hash file organization of *instructor* file, using *dept_name* as key

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

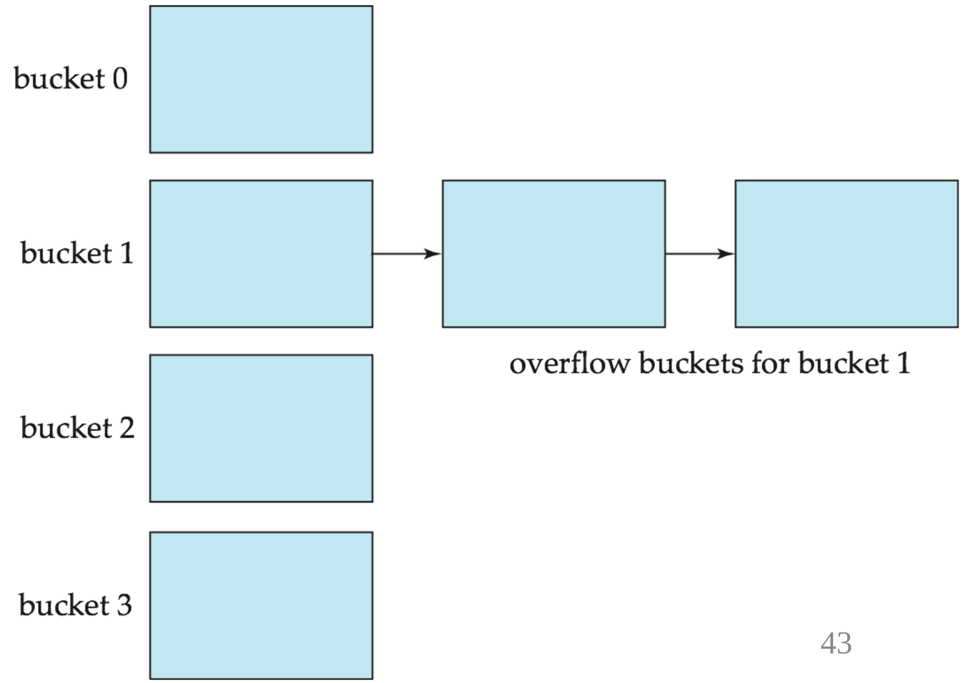
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

How to handle bucket overflows?

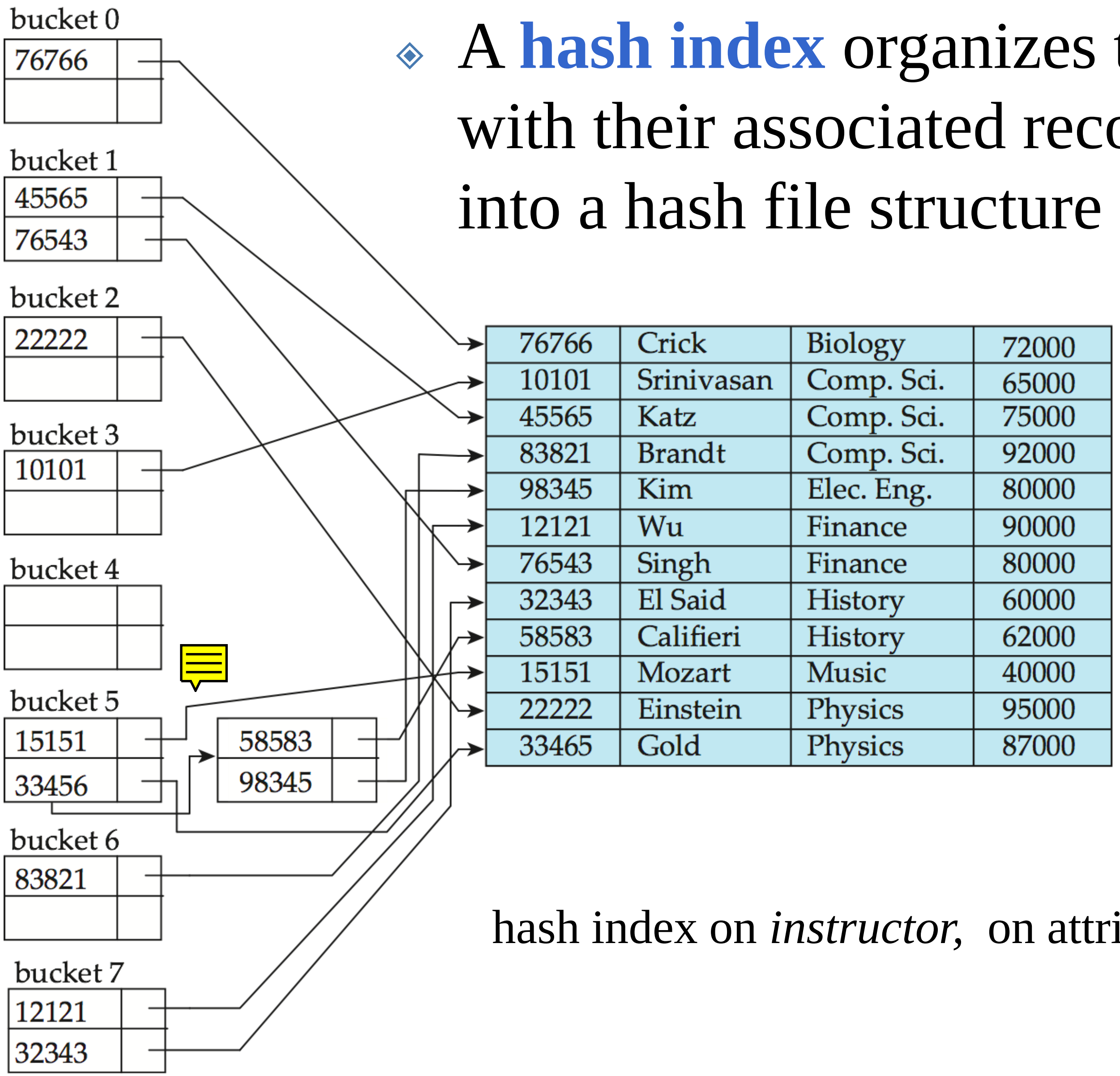
- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records, due to:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values

- **Overflow chaining** – chain the overflow buckets of a given bucket by a linked list
  - This is also called **closed hashing**



Hash Index

- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure



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*

Which type of index (ordered index, hash index) should be used?

- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is better at retrieving records having a specified value of the key
  - If range queries are common, ordered indices are preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

Information about Quiz 2

- *Date/Time*:
- around 8:00 pm, 26 (Tuesday class) and 27 (Wednesday class) November
- *Scope*: Lectures 7 – 11
  - Relational model, SQL, ER diagram, database normalization, Storage
- *Format*: short questions
- You are allowed to bring FIVE A4 papers (both sides) of reference material