Intro to DB

CHAPTER 11
INDEXING & HASHING

Chapter 11: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

Basic Concepts

- to speed up access to desired data
- **Search Key**
  -
- **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

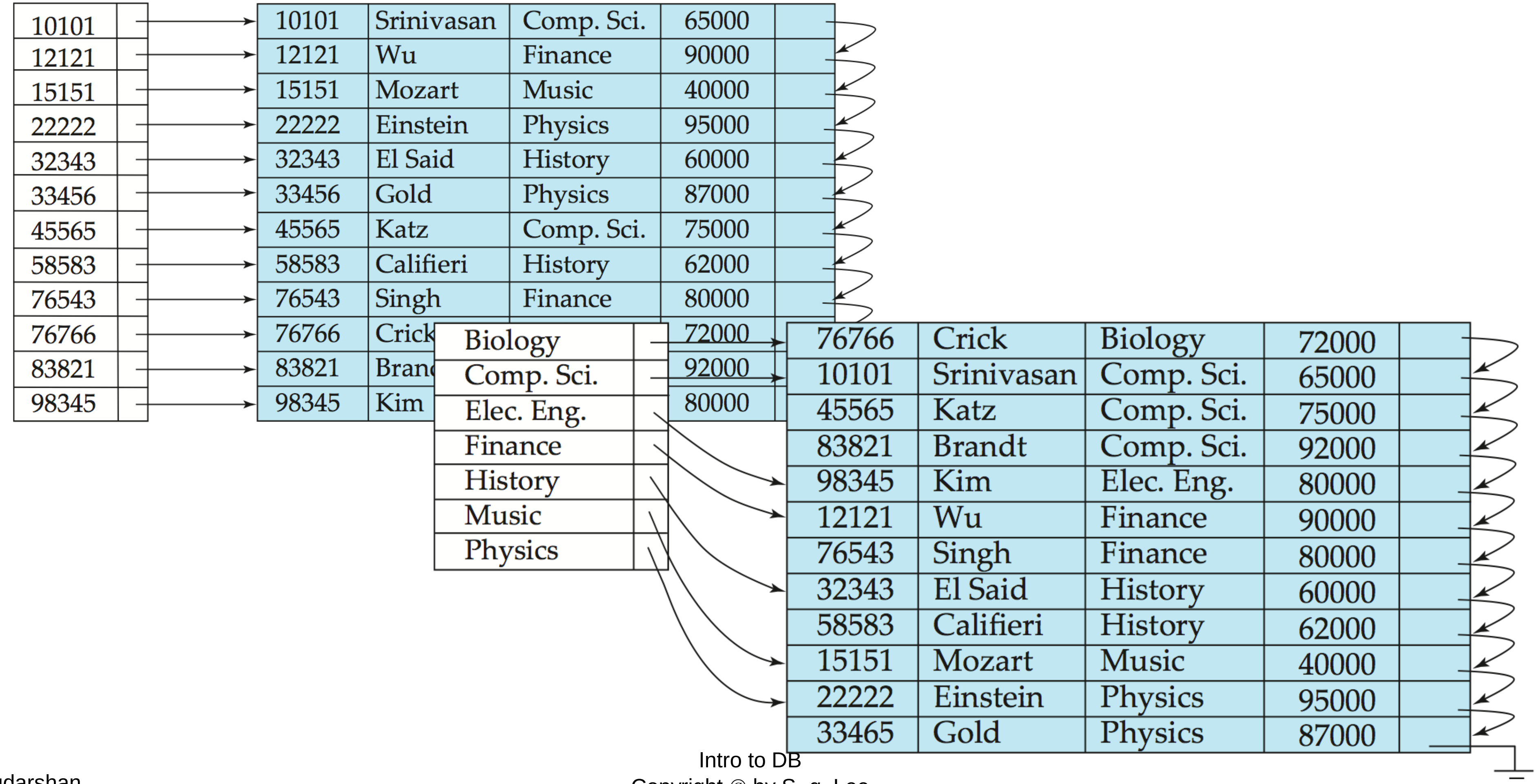
- Access types supported
  - Point queries: specific value for search key
  - Range queries: search key value falling in a specified range
- Time
  - Access time
  - Insertion time
  - Deletion time
- Space overhead

Ordered Indices

- **Primary index**
  -
  - 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**
  -

Dense Index Files

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



Sparse Index Files

- Sparse Index

  contains index records for only some search-key values

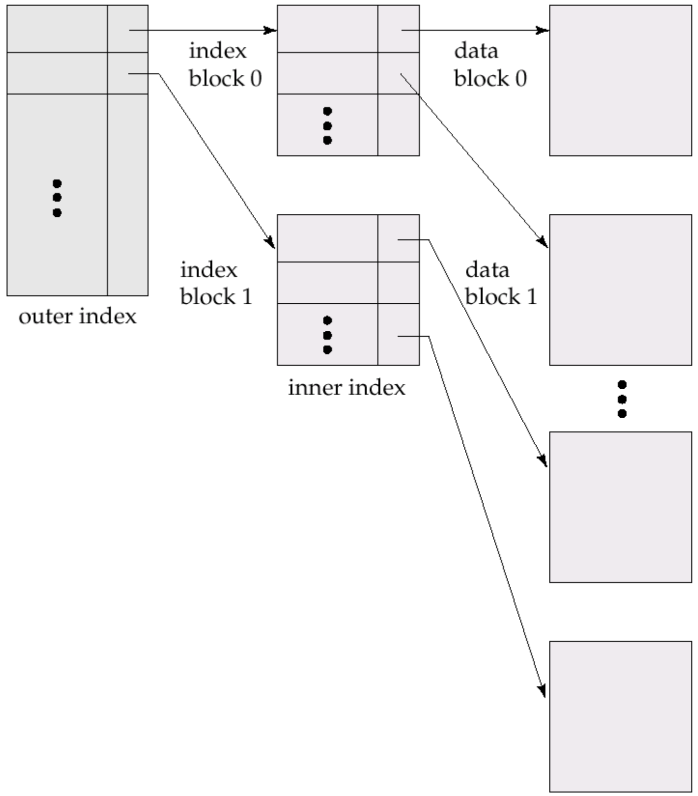
  - Applicable when records are sequentially ordered on search-key
  - Less space and less maintenance overhead for insertions/deletions.
  - Generally slower than dense index for locating records.

10101	
32343	
76766	

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	

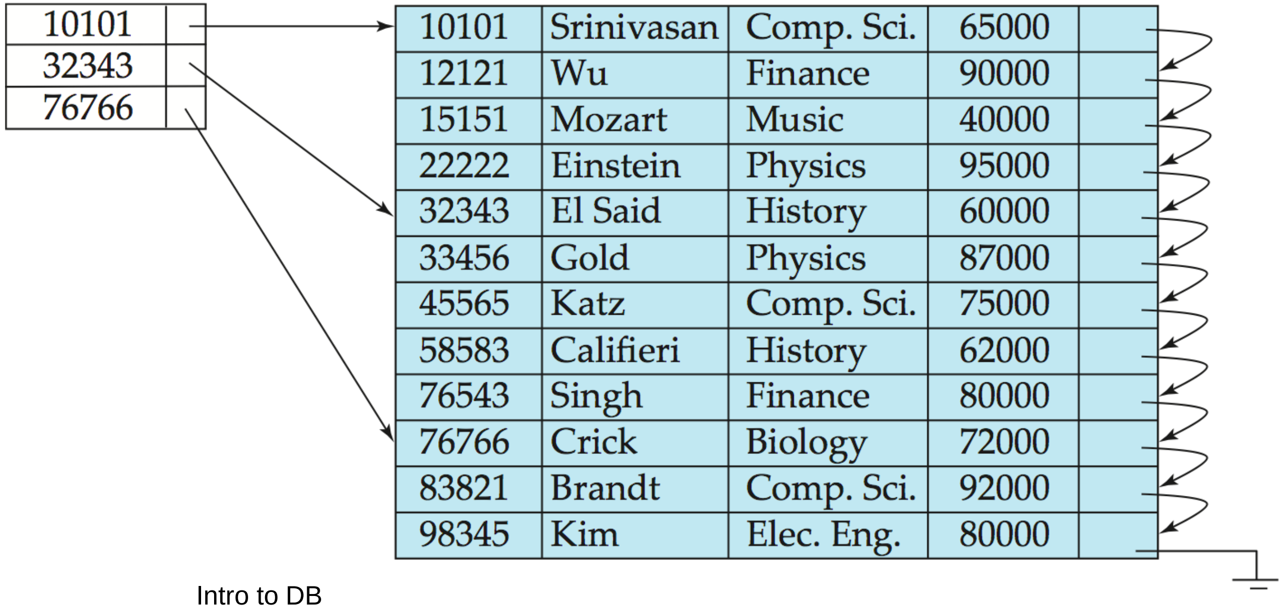
Multilevel Index

- If primary index does not fit in memory, access becomes expensive
- 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 outer index is still too large to fit in main memory, another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file



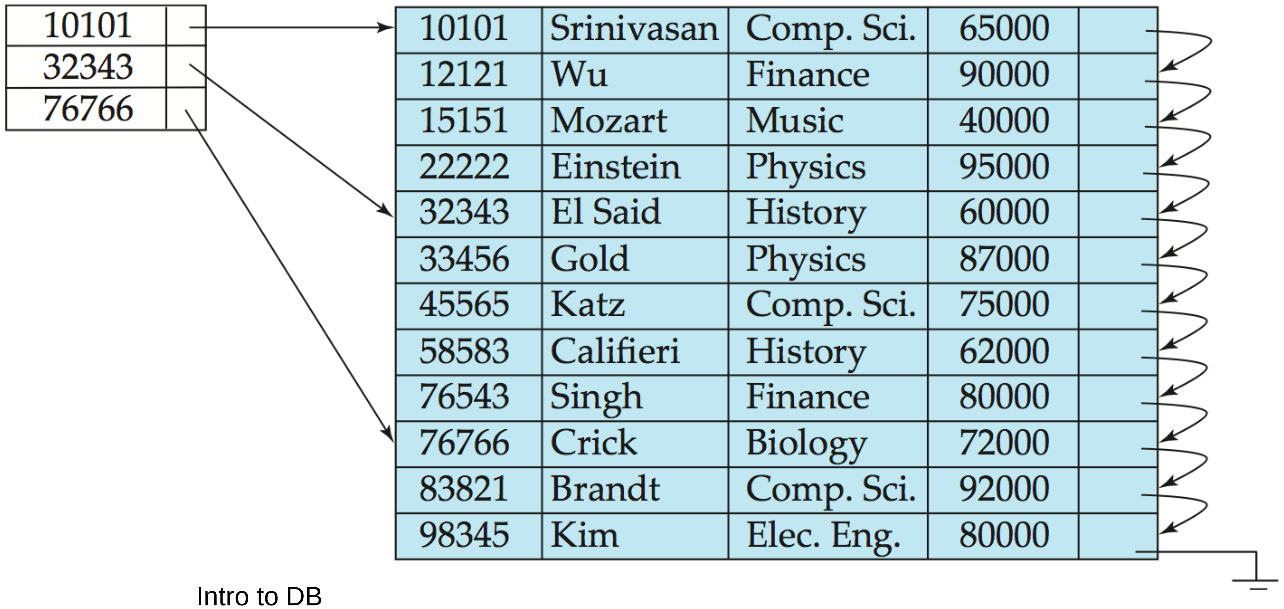
Index Update: Deletion

- Index entry must be updated accordingly
  - 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.
  - **for 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.



Index Update: Insertion

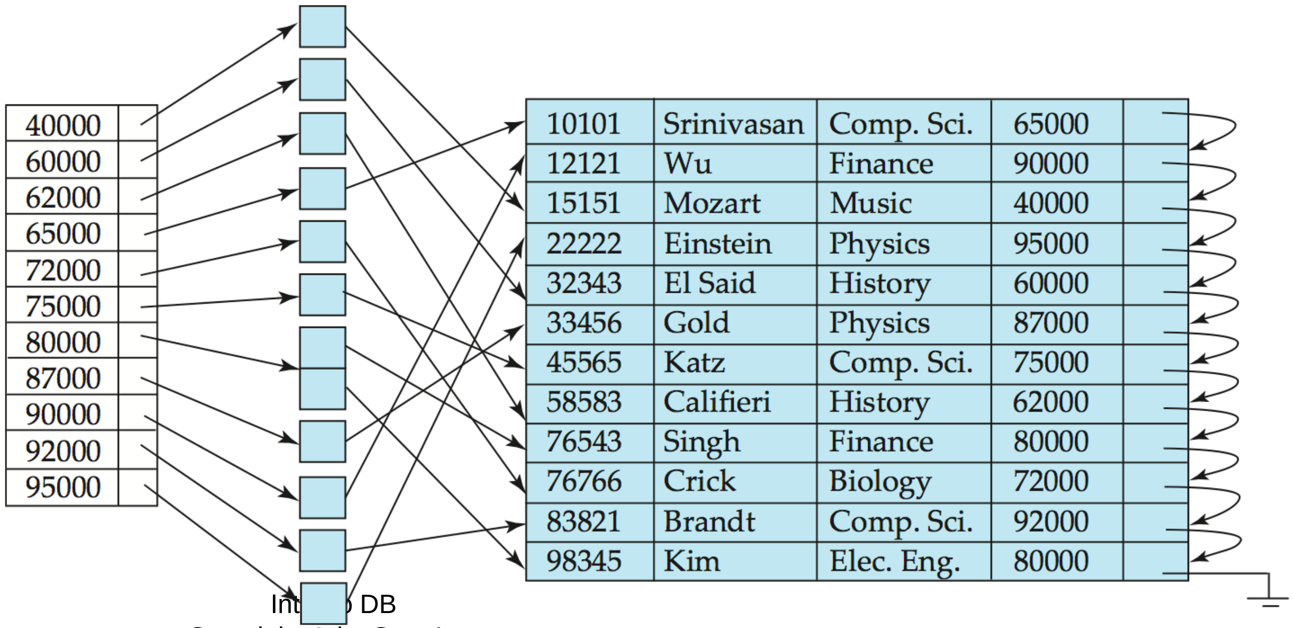
- Perform a lookup using the search-key value appearing in the record to be inserted.
- **Dense indices** – if search-key value does not appear in the index, insert it.
- **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
  - If a new block is created, the first search-key value appearing in the new block is inserted into the index.



Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated
  - Updating indices imposes overhead on database modification
- 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
- Index takes up space

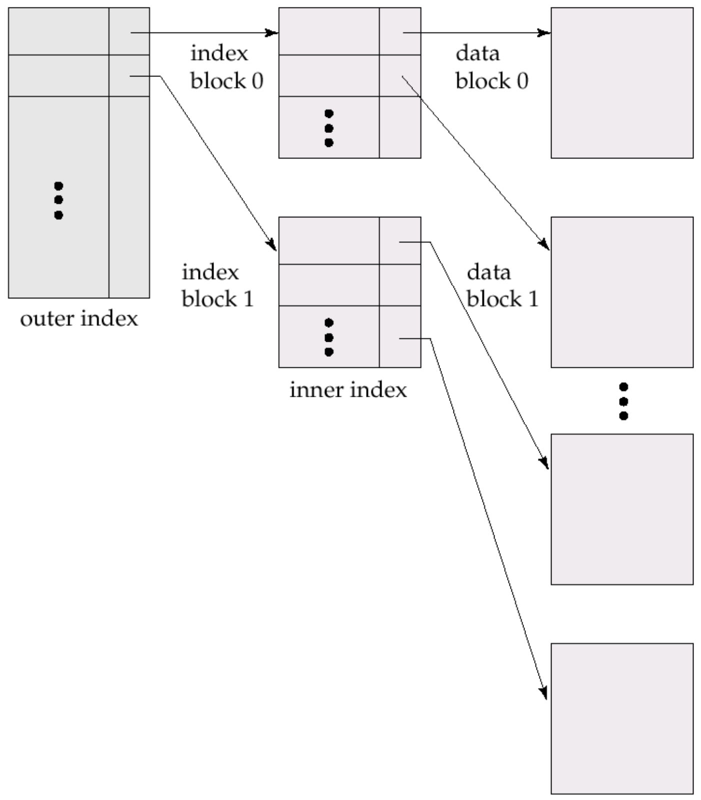
Secondary index on *salary* field of *instructor*



Ordered Index Performance

- Access types supported
  - Point queries: specific value for search key
  - Range queries: search key value falling in a specified range
- Time
  - Access time: depends on height of index tree
  - Insertion/Deletion time: also depends on height
- *n* key values & *k* children/node
  - Best case: height = $\log_k(n)$
  - Worst case: height = n

=> *We want to have* **balanced** *index trees!*



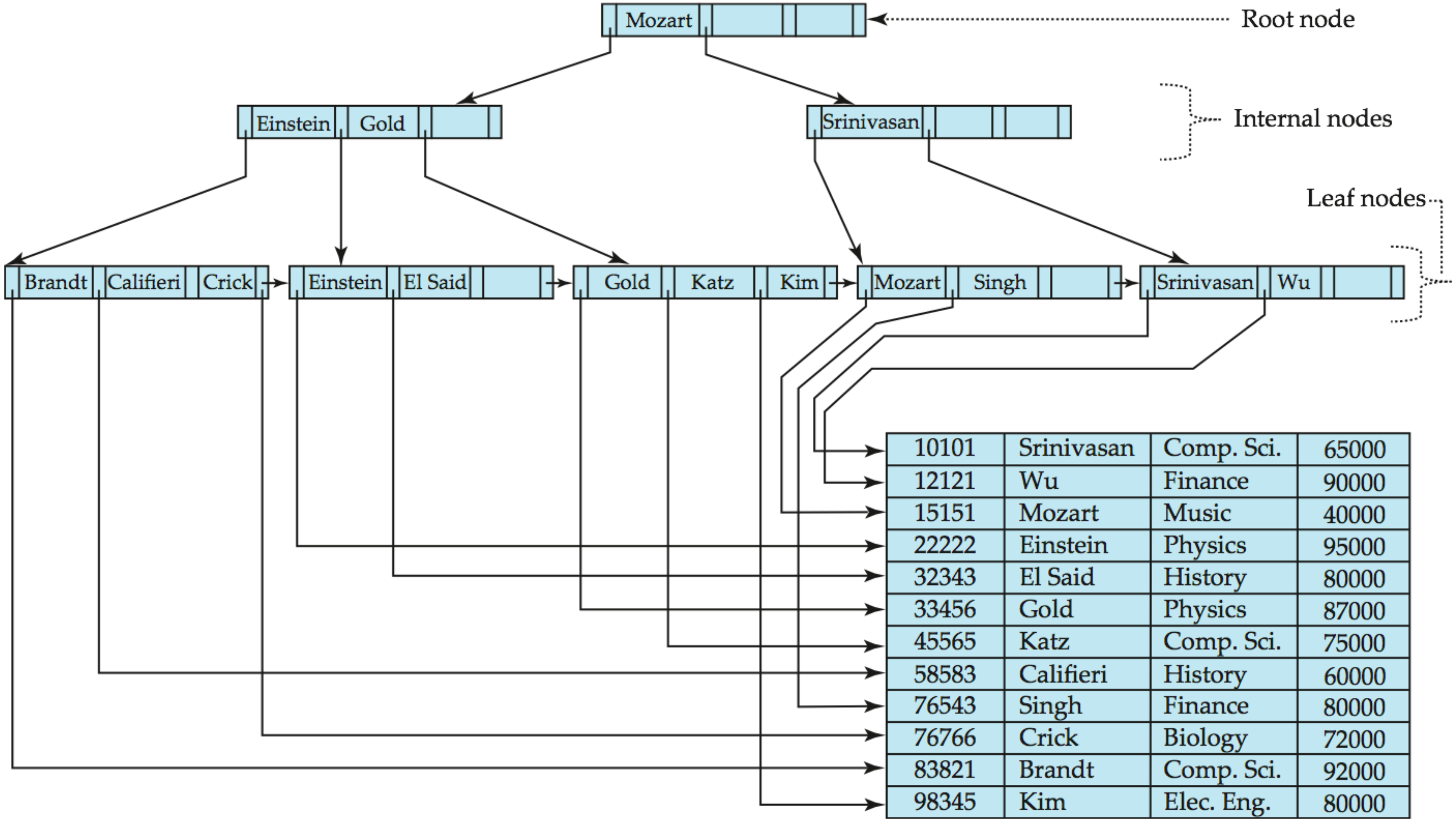
B+-Tree Index

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

- Each node (that is not a root) has between $\lceil n/2 \rceil$ and n pointers

- Special case: *root* node
  - If 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.

Example of B$^+$-Tree



Advantages of B+-Tree Index

- Advantages
  - 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) Disadvantages
  -
  - space overhead
- Advantages of B+-trees outweigh disadvantages
- B+-trees are used extensively

B^+-Tree Node Structure

- Typical node

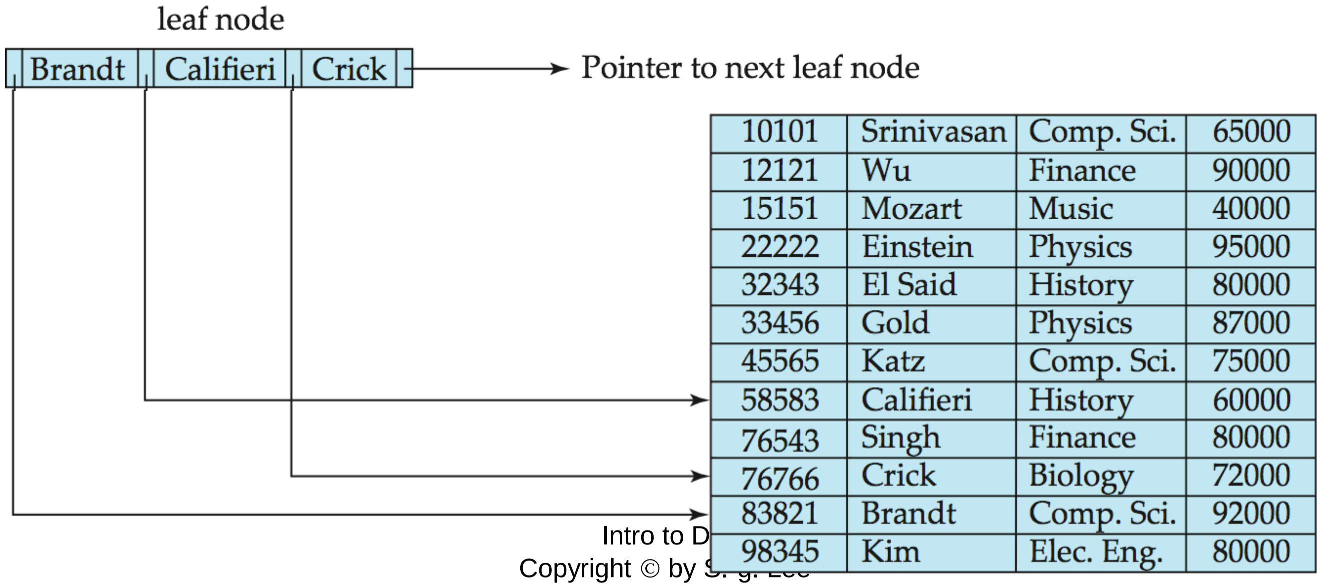
P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

  - 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

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

Leaf Nodes

- 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.
    - need bucket structure only if search-key does not form a primary key
- For leaf nodes L_i and L_j, $i < j$,
  -
- P_n points to next leaf node in search-key order

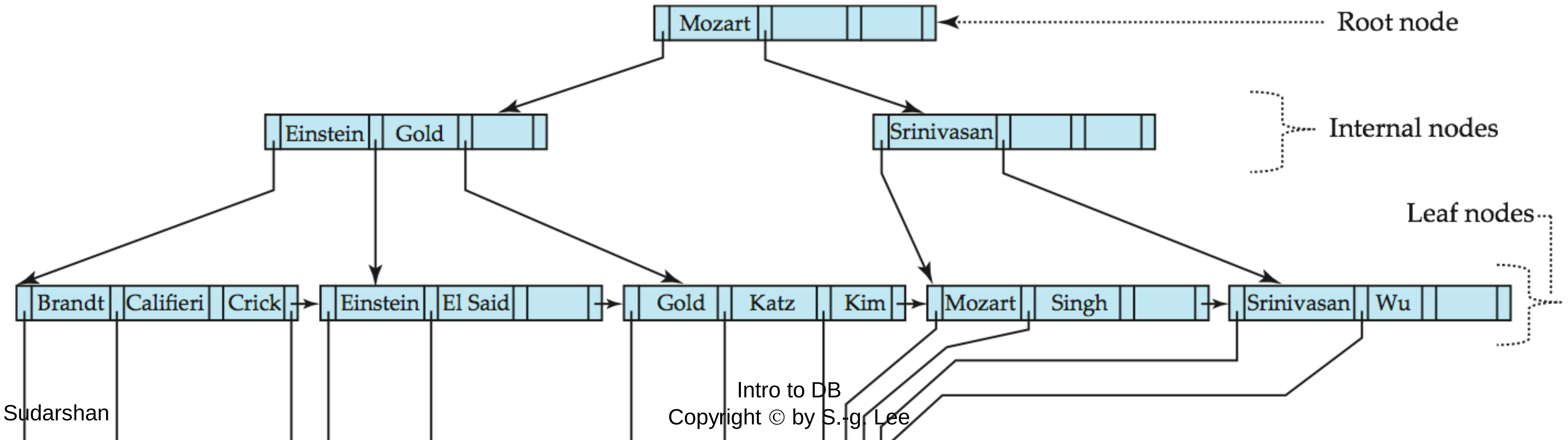


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Non-Leaf Nodes

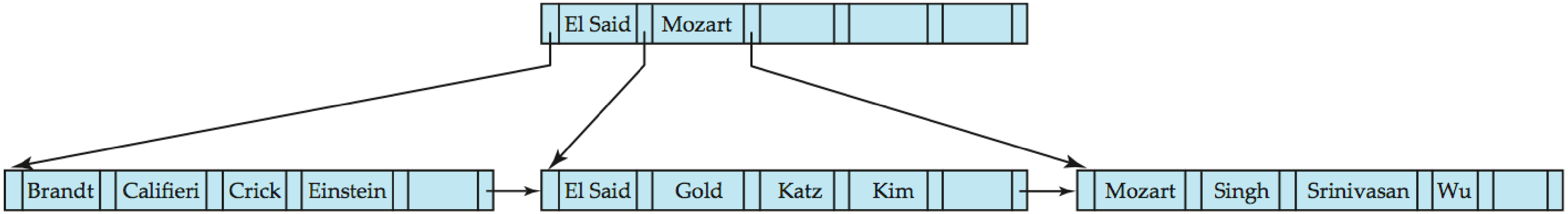
- For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq m$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i

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



Example of a B$^+$-tree

- $n = 5$
- All nodes other than root must have
  -
  - ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 5$).
- Non-leaf nodes other than root must have
  -
  - ($\lceil (n/2 \rceil$ and n with n =5; or 1 more than number of key values).
- Root must have at least 2 children.



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

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
- 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
  - Level h has at least $2*\lceil n/2 \rceil^{h\text{-}1}$ values
  - 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.

B$^+$-Tree: Search

Find all records with a search-key value of *V.*

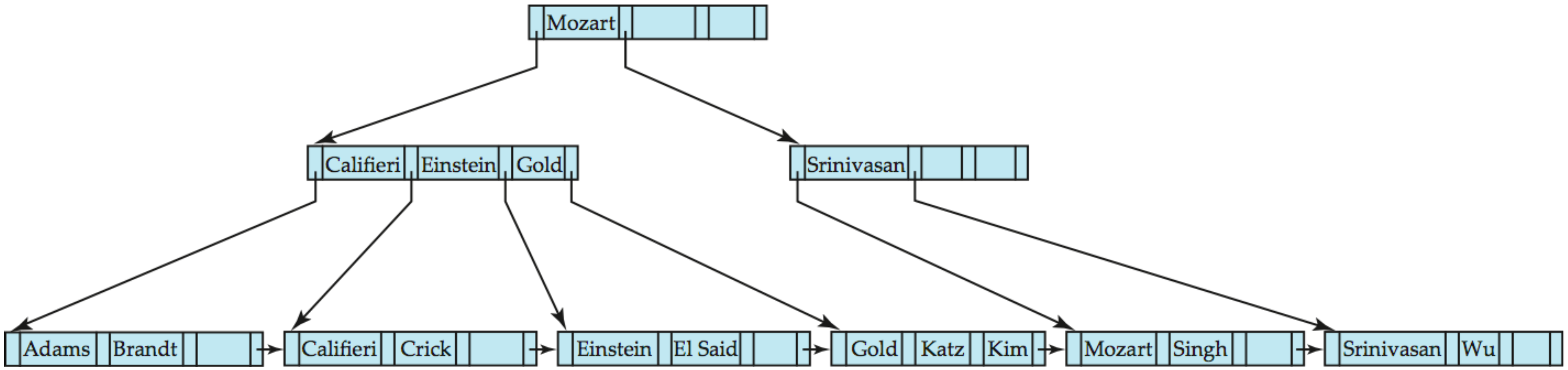
1. *C=root*
2. **While** *C* is a *nonleaf node* {

   Let K_m be the *last non-null search-key value* in *C* ; **Set** K_0= *-inf* ; K_{m+1}= *+inf*

   Let *i* be the smallest value s.t. $K_{i-1} \leq V < K_i$

   $C = P_i$ }
3. **If** there is a value *j* s.t. $K_j = V$, follow pointer P_j to the desired record

   else $\boxed{}$



B^+-Trees: Search Performance

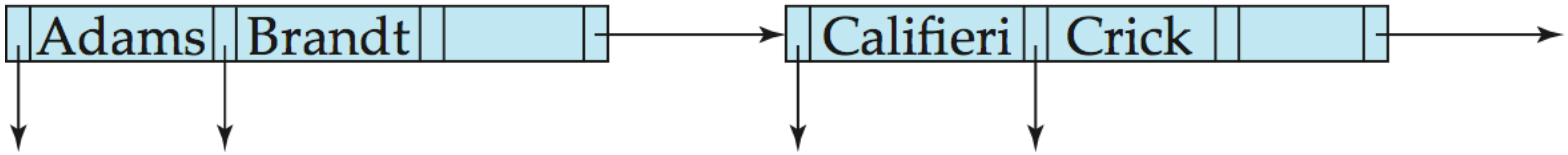
- If there are K search-key values in the file, the height of the tree is
- A node is generally the same size as a disk block
  - typically 2~4 kilobytes
  - and n is typically around 50~100 (40 bytes per index entry).
- With 1 million search key values and n = 100
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.
- Contrast this with a balanced binary tree (or an unbalanced tree with degree 100)
  - around 20 nodes are accessed in a lookup
  - 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 L* in which the search-key value would appear (Search)
2. **If** the search-key value is already present in the leaf node
   1.
   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 *L*, insert <*key-value, pointer*> pair in *L*
      else **split** the node (along with the new <*key-value, pointer*> entry)
      - as discussed in the next slide

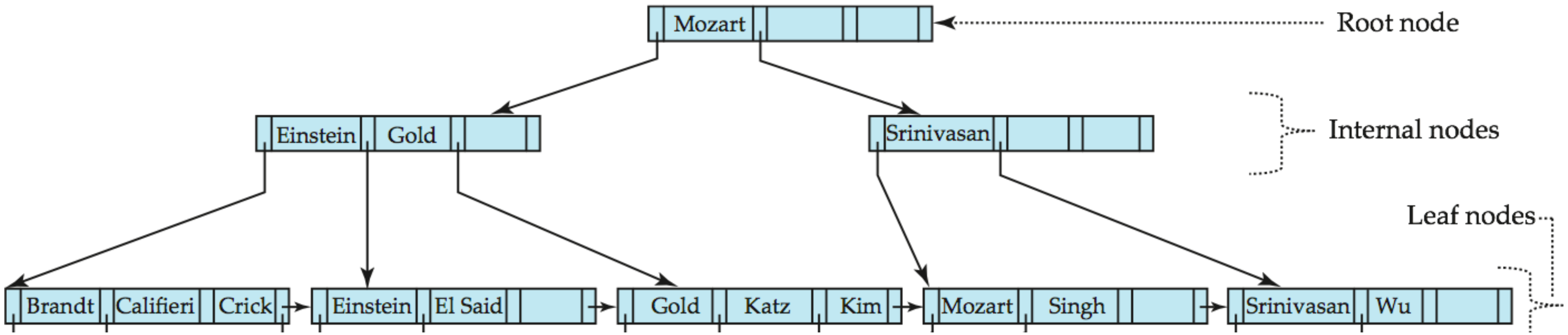
Updates on B+-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - Take the *n* <*key-value, pointer*> pairs (including the one being inserted) in sorted order
  - Place the first $\lceil n/2 \rceil$ in the original node *L*, and the rest in a new node *P*.
  - Let *k* be the least key value in *P*.
    Insert <*k, P*> into the parent of the node being split.
  - If the parent is full,
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - worst case: root node may be split increasing the height of the tree by 1

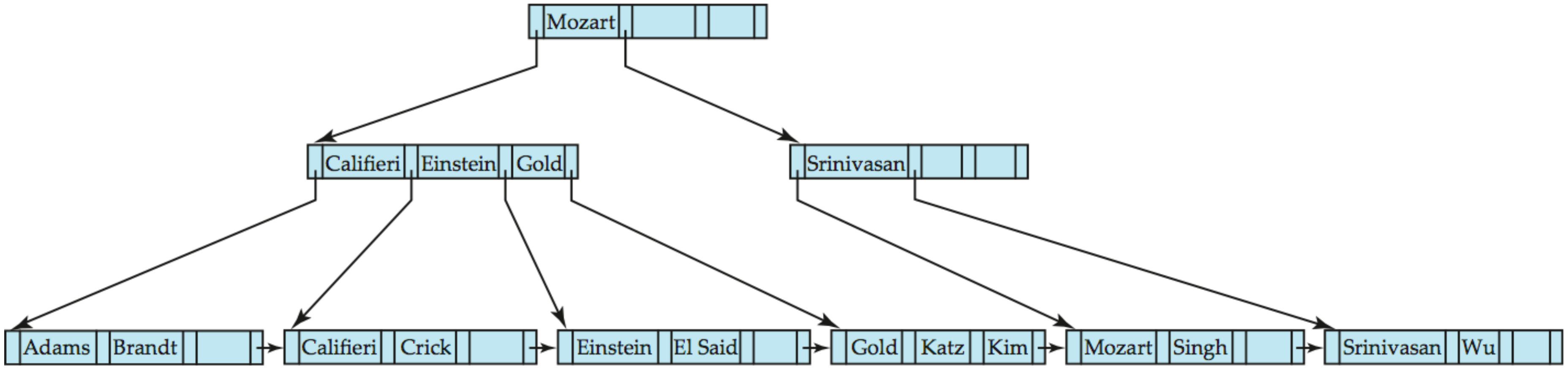


Result of splitting node after inserting Adams into node (Brandt, Califieri, Crick).
Next step: insert entry with <*Califieri, pointer-to-new-node*> into parent

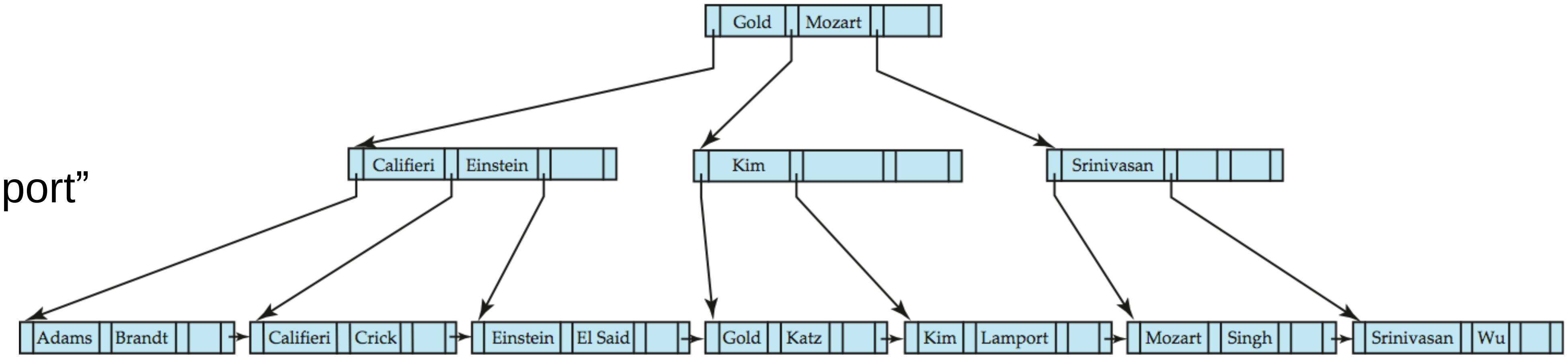
B+-Tree Insertion Example



- *Insert* “Adams”



- *Insert* “Lamport”



Updates on B$^+$-Trees: Deletion

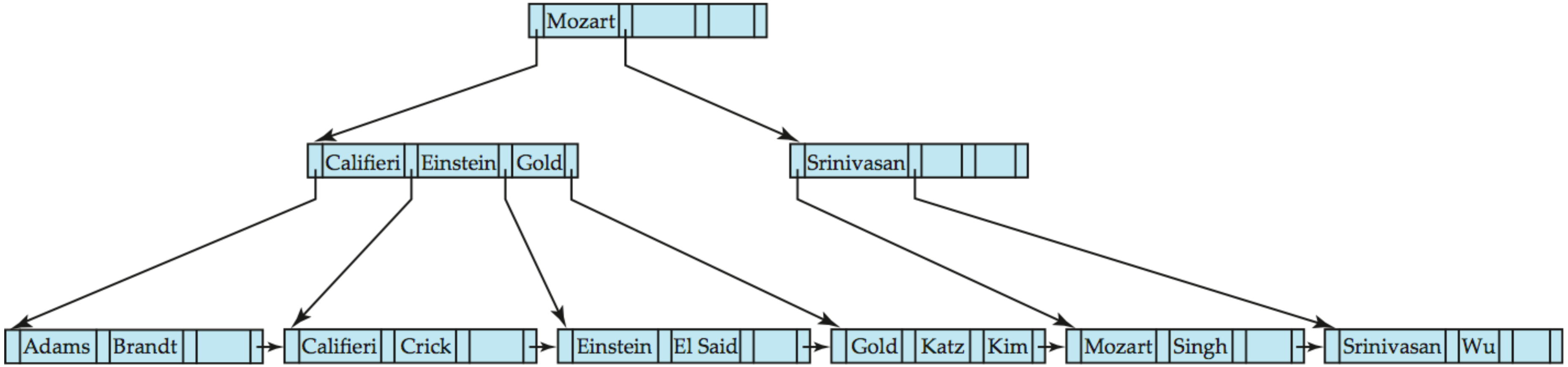
1. **Find** the record to be deleted (using the B+-tree, leaf node *L*)
   Remove the record from the main file (and from the bucket, if present)
2. **Remove** <*key-value, pointer*> from the leaf node *L*
   If this was the last record with the *key-value* (e.g., the bucket has become empty)
3. **If** *L* has too few entries (less than $\lceil n/2 \rceil$ pointers) due to the removal
   then **merge siblings** or **redistribute entries** (as explained in next slide)
4.
   - **If** the root node has only one pointer after deletion,
     then delete it and the sole child becomes the root

Updates on B+-Trees: Deletion

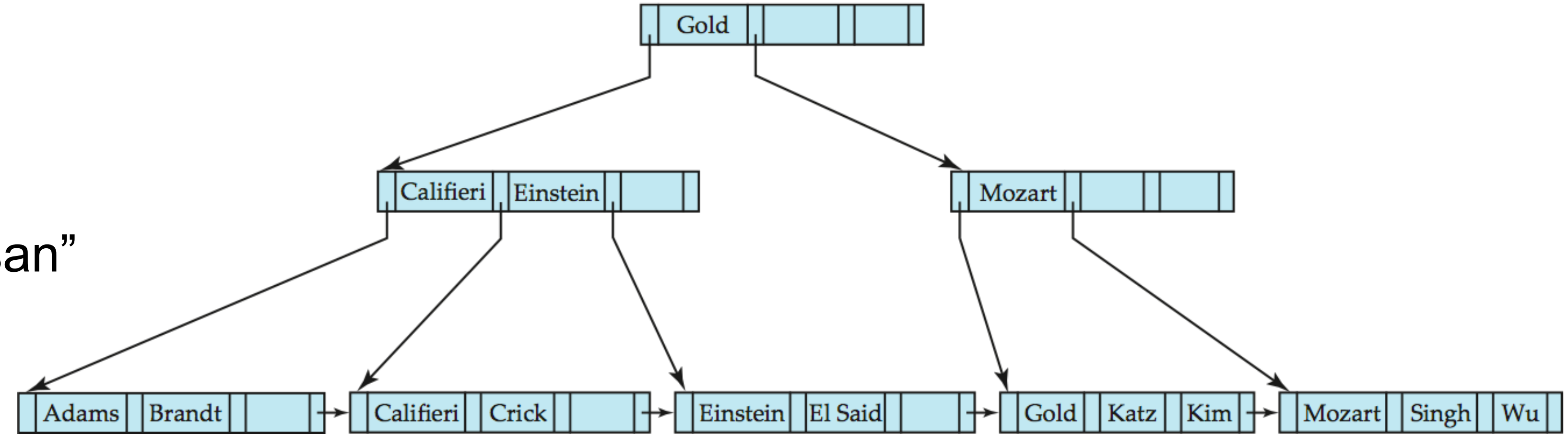
If L has too few entries (less than $\lceil n/2 \rceil$ pointers) due to the removal {

- **If** entries in L 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, P.
  - Delete the pair $< K_{i-1}, P >$ from its parent, recursively using the above procedure.
- **Else**, ***redistribute pointers***:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  -

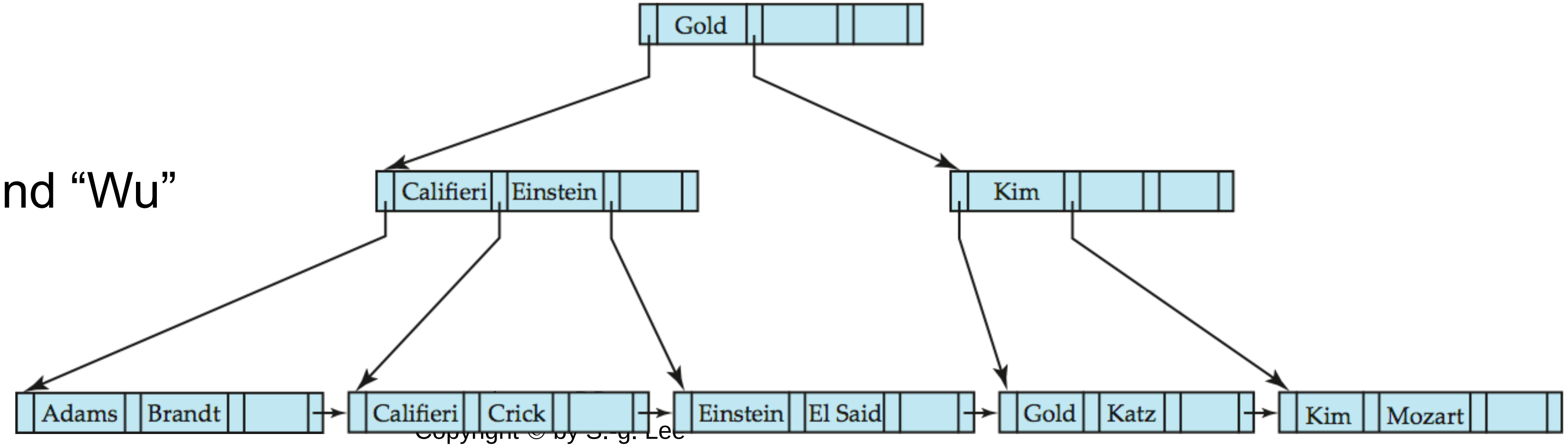
B+-Tree Deletion Example



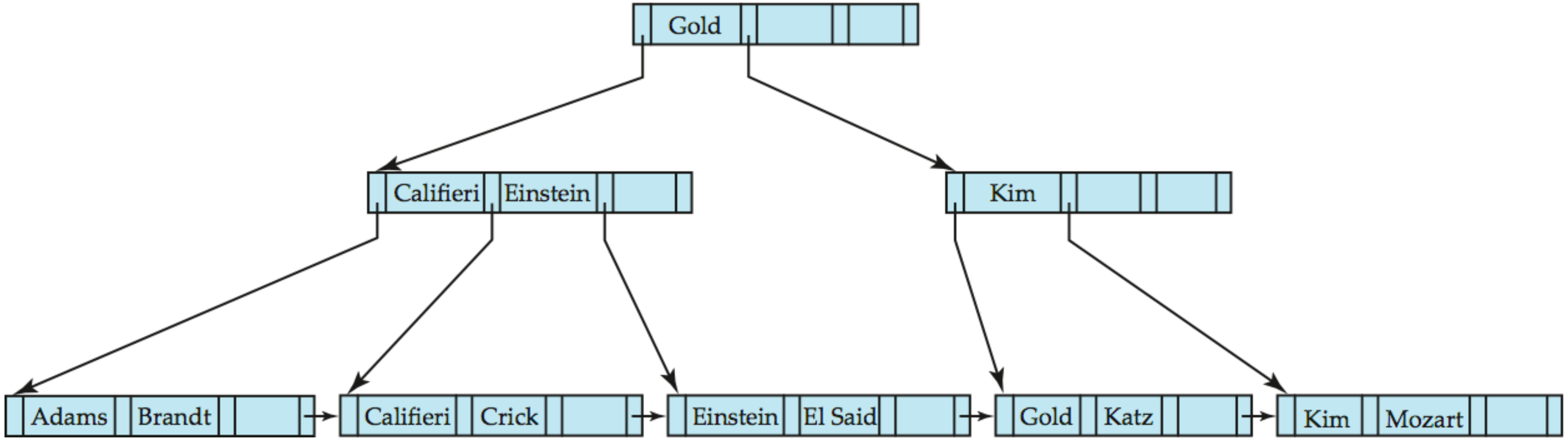
- *Delete* “Srinivasan”



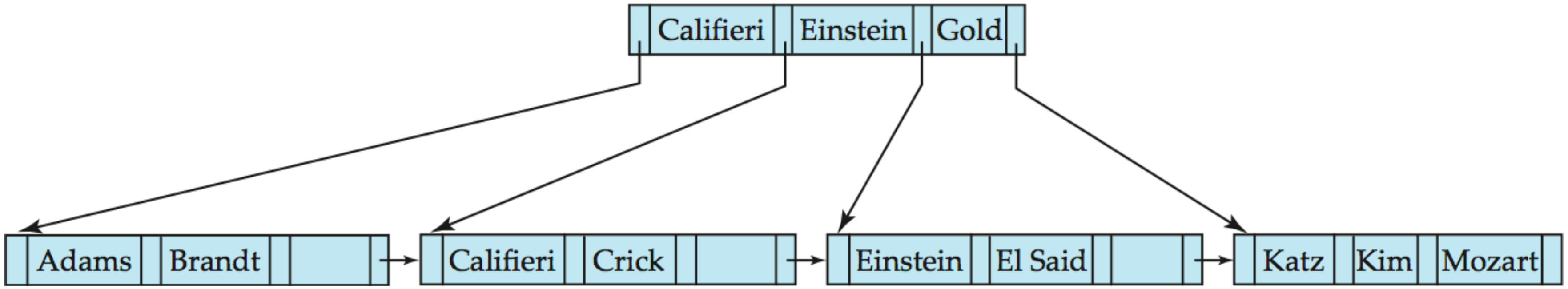
- *Delete* “Singh” and “Wu”



B+-Tree Deletion Example



- *Delete* "Gold"



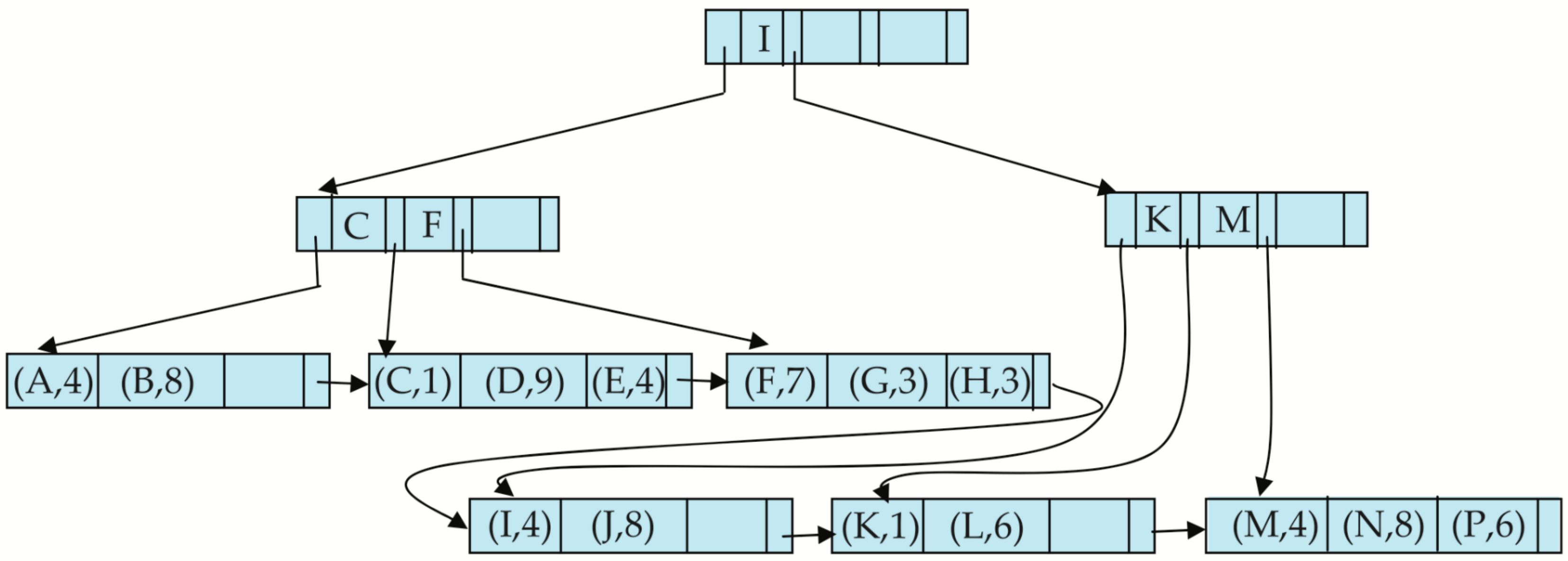
- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

B^+-Tree File Organization

- The leaf nodes in a B^+-tree file organization **store records**
  - instead of pointers.
- Data file degradation problem can be solved by using B^+-Tree File Organization.
  - Just as index file degradation problem is solved by using B^+-Tree indices.
-
  - 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.)

- Good space utilization important since records use more space than pointers.
  => involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution results in each node having at least $\lfloor 2n/3 \rfloor$ entries



Static Hashing

- **bucket**
  - unit of storage containing one or more records
  -
- **hash file organization**
  - obtain the bucket of a record directly from its search-key value using a **hash function**
- Hash function *h*
  - function from the set of all search-key values *K* to the set of all bucket addresses *B*
  - 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.

Hash File Organization

- 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

  h(Music) = 1

  h(History) = 2

  h(Physics) = 3

  h(Elec. Eng.) = 3

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

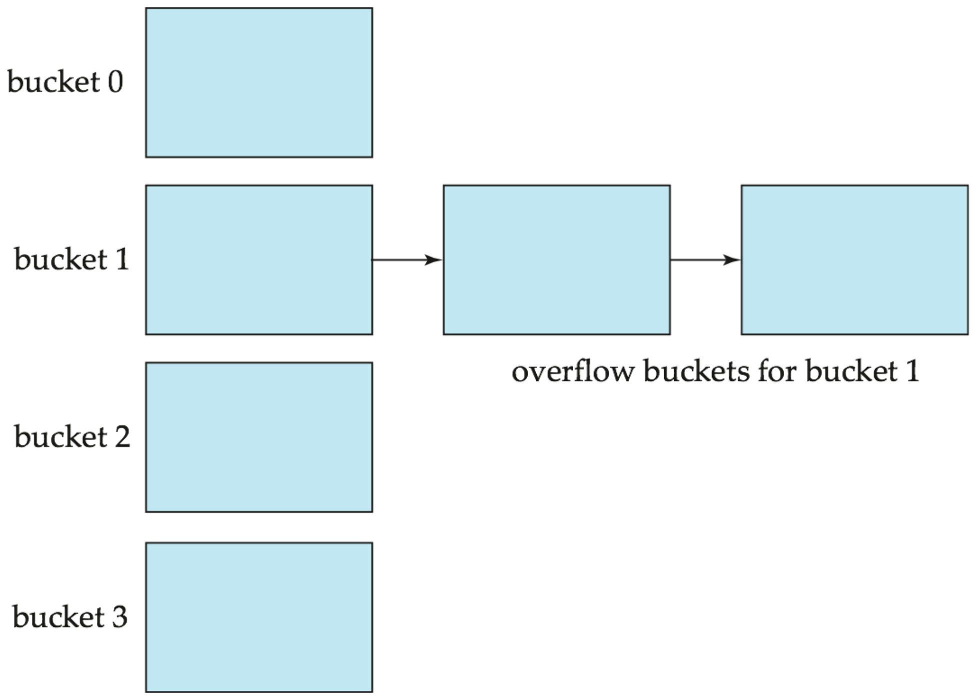
bucket 7

Hash Functions

- Worst hash function
  -
  - access time becomes $O(n)$
- An ideal hash function is
  - **uniform**: each bucket is assigned the same number of search-key values from the set of *all* possible values.
  - **random**: 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

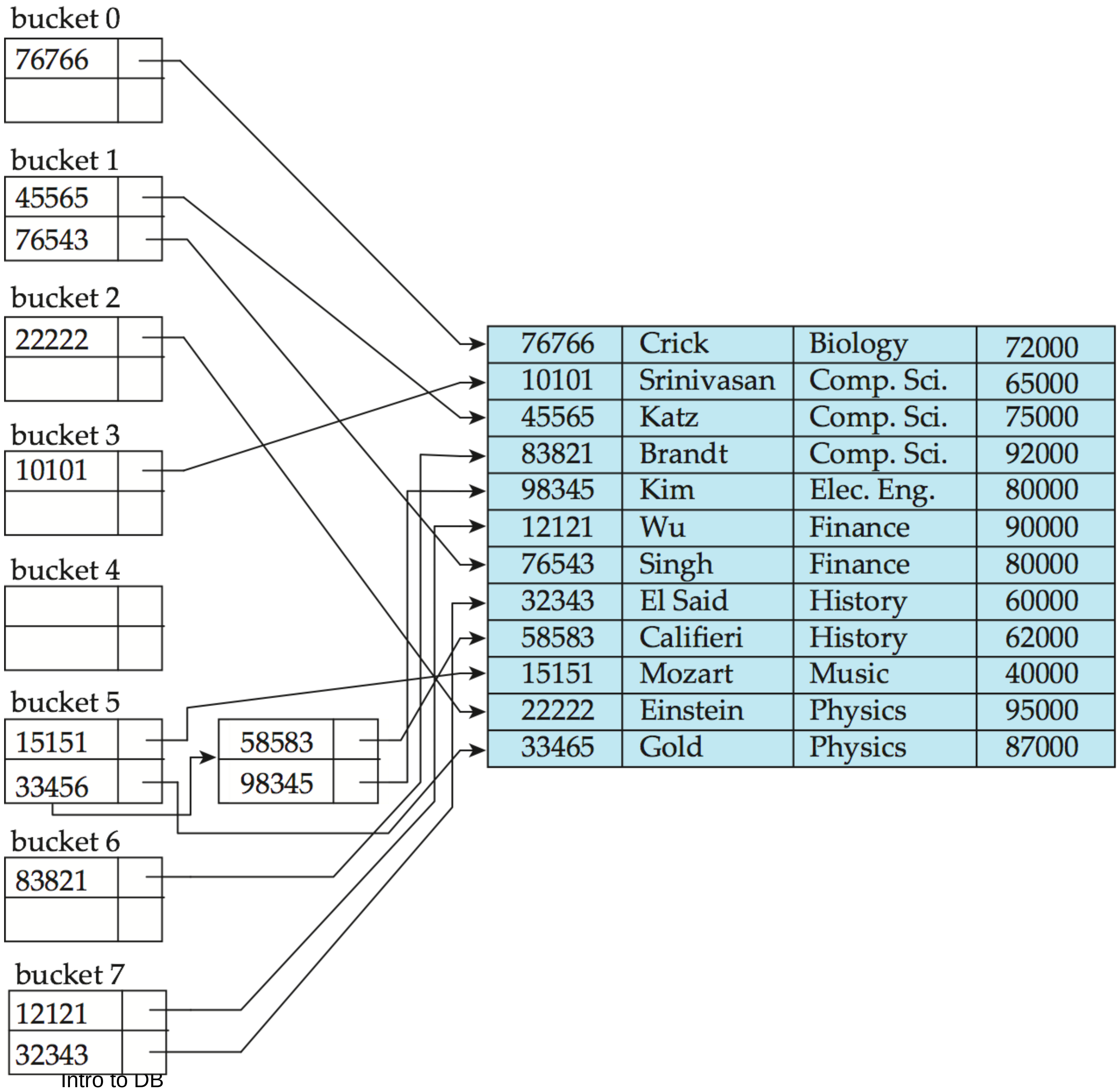
Handling of Bucket Overflows

- 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*****.**



Hash Indices

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



hash index on *ID* of *instructor*

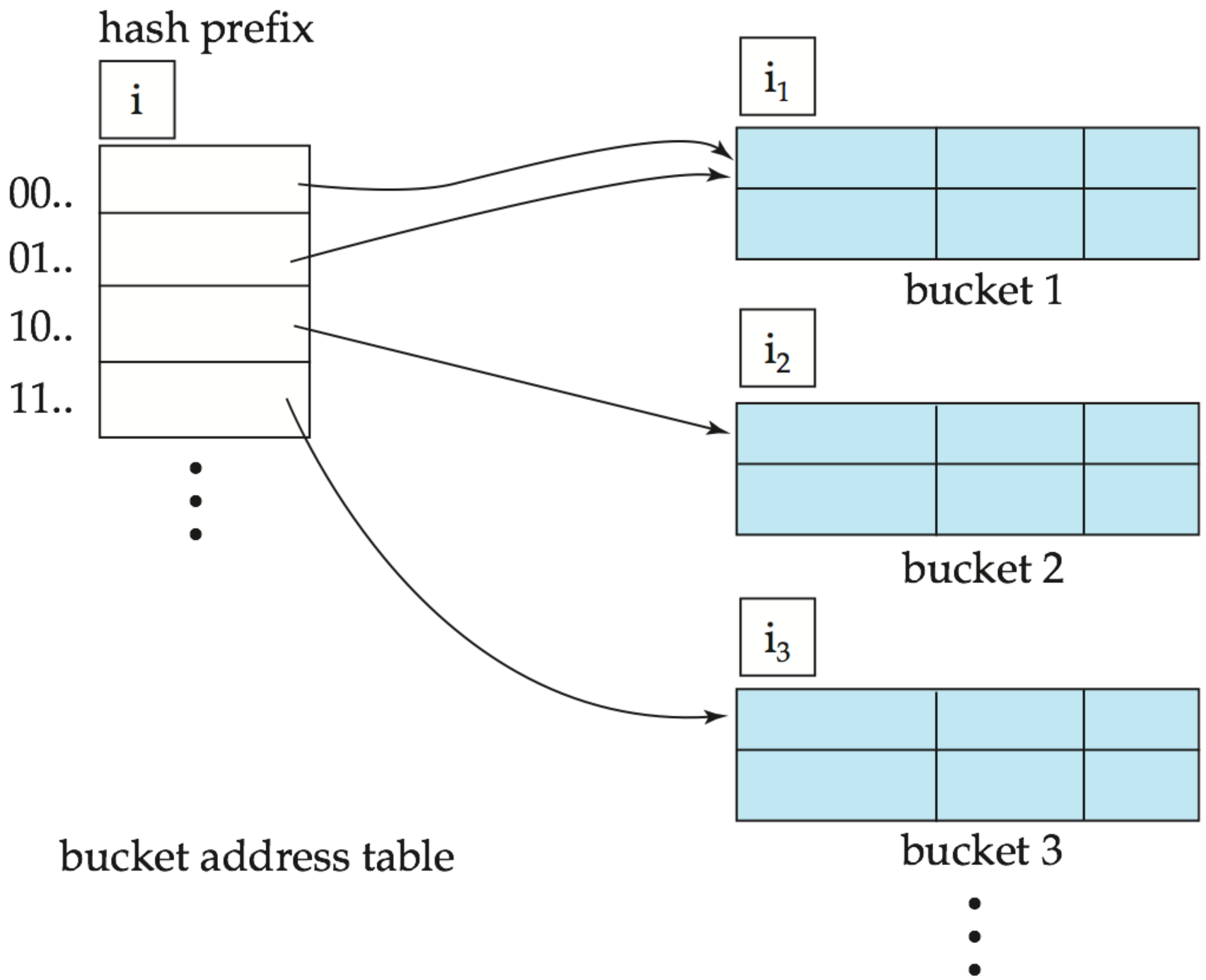
Deficiencies of Static Hashing

- Hash function *h* maps key values to a fixed set of *B* of bucket addresses
- Databases grow or shrink with time.
  - If number of buckets is too small, performance will degrade due to too much overflows.
  - Too many buckets wastes space (buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file
  - a new hash function and bucket address space
  - Expensive and disrupts normal operations
- Better solution: *dynamic hashing*
  - allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Allows the hash function to be modified dynamically
  - Good for database that grows and shrinks in size
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range — typically *b*-bit integers, with $b = 32$.
  - Use only a prefix (i bits) of the hash function
    - Bucket address table size = 2^i. Initially $i = 0$
    - Value of i grows and shrinks as the size of the database grows and shrinks.
  - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Extendable Hashing

- Each bucket *j* stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits.
- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say *j*.
  - If there is room in the bucket *j* insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide)
    - Overflow buckets used instead in some cases (will see shortly)

Extendable Hashing: Insertion

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

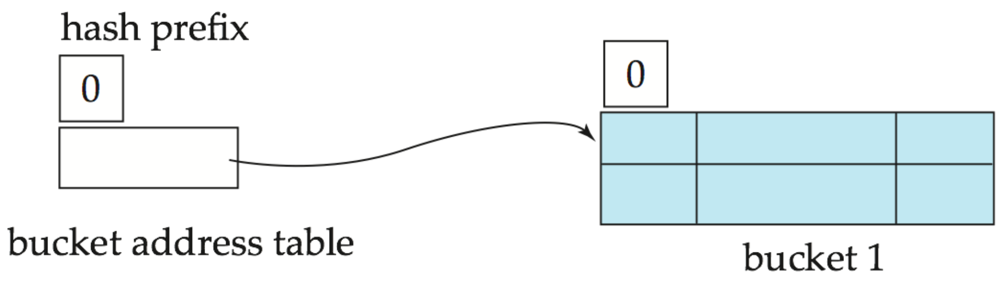
- If $i > i_j$ (more than one pointer to bucket *j*)
  - allocate a new bucket *z*, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to *j*, to point to *z*
  - remove each record in bucket *j* and reinsert (in *j* or *z*)
  - recompute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket *j*)
  - If *i* reaches some limit *b*, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment *i* and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j
      Now $i > i_j$ so use the first case above.

Extendable Hashing: Deletion

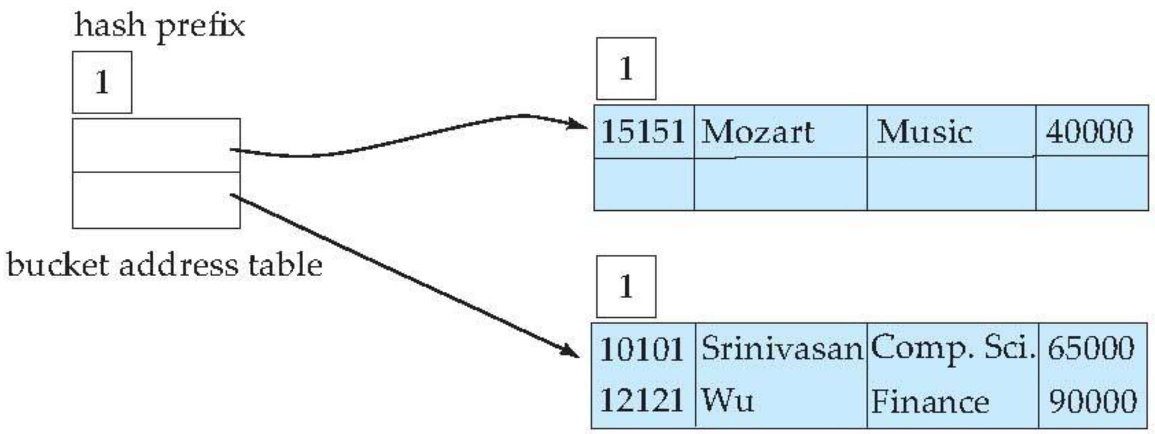
- Locate it the key value in its bucket and remove it.
- The bucket itself can be removed if it becomes empty
  - with appropriate updates to the bucket address table
- Coalescing of buckets can be done
  - coalesce only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ prefix
- Decreasing bucket address table size is also possible
  - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Extendable Hashing: Example

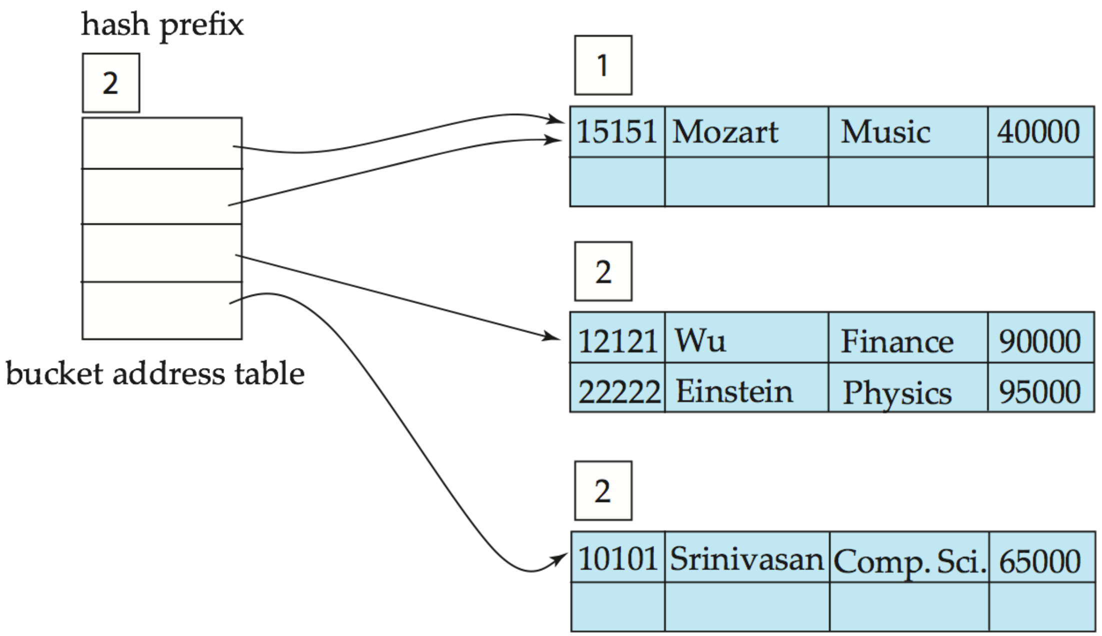
- Initial Hash structure



- Insert “Mozart”, “Srinivasan”, and “Wu”

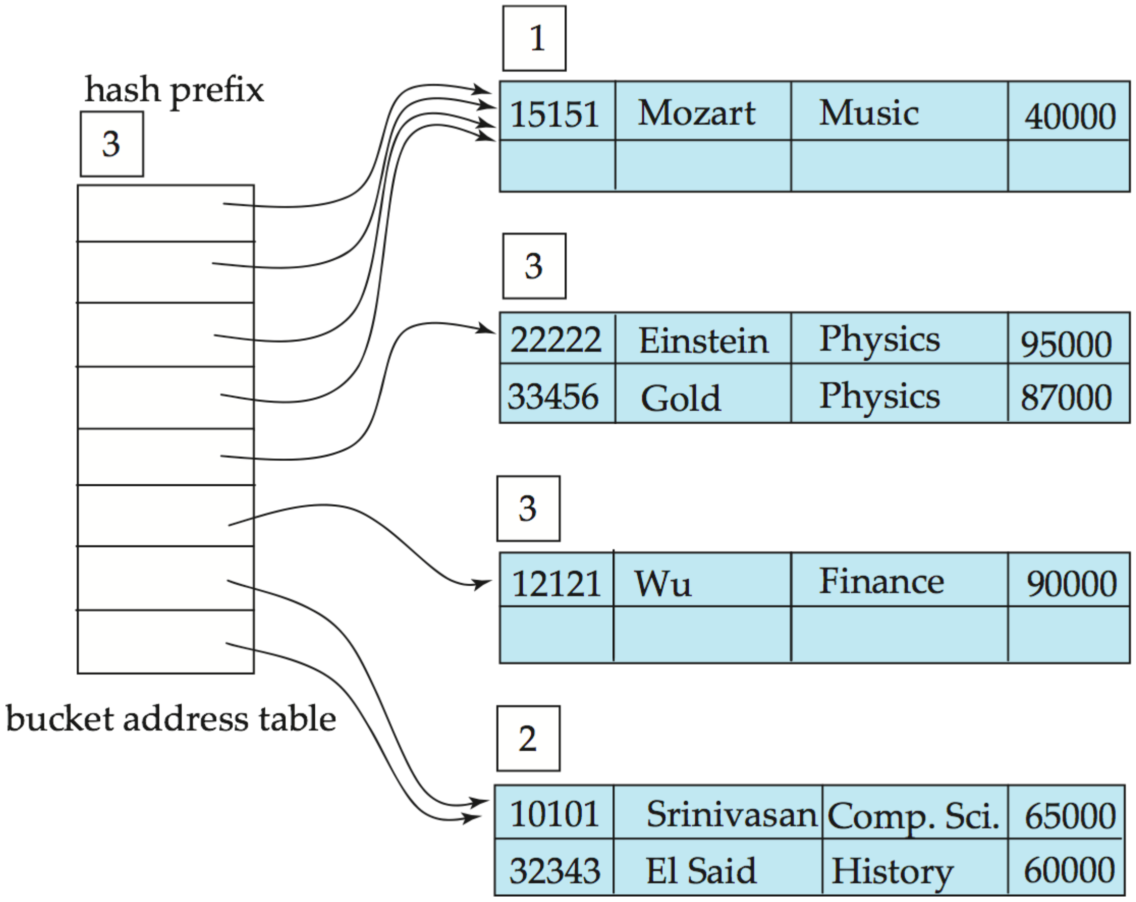


- Insert “Einstein”

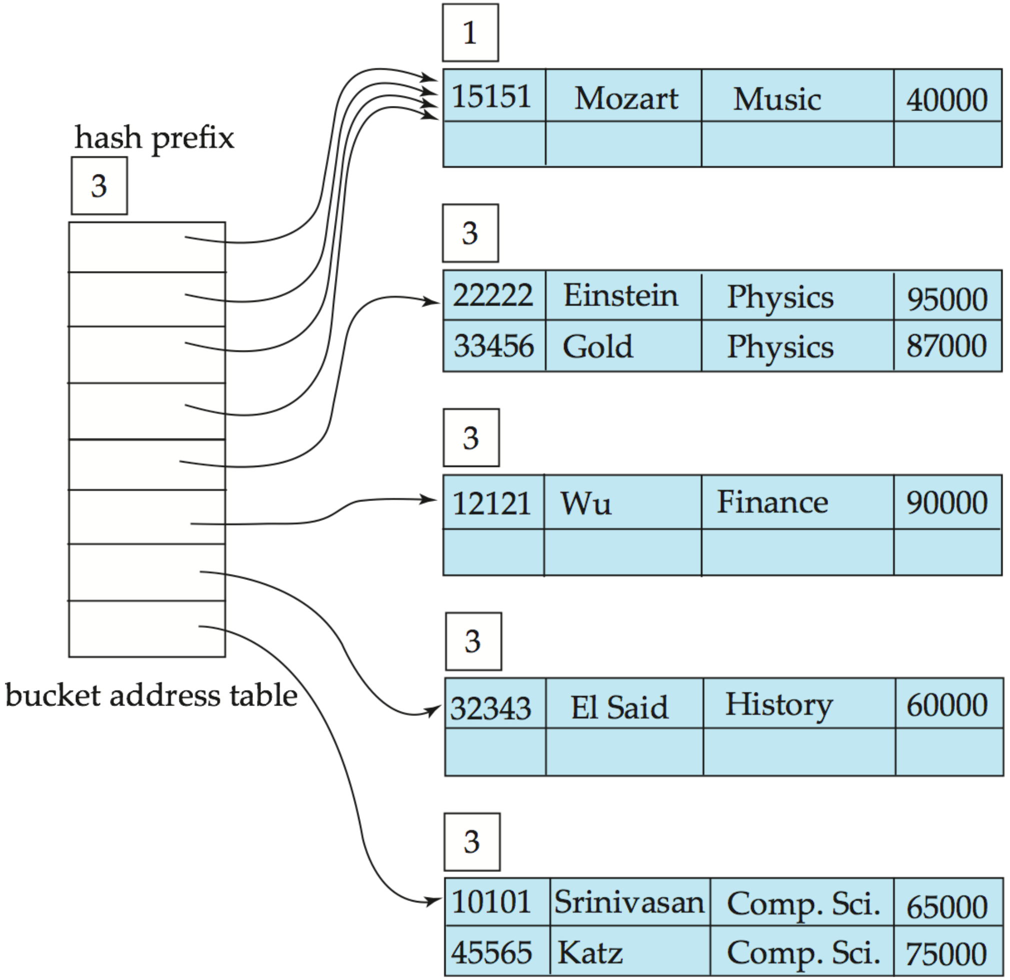


Example (Cont.)

- Insert “Gold” and “El Said”

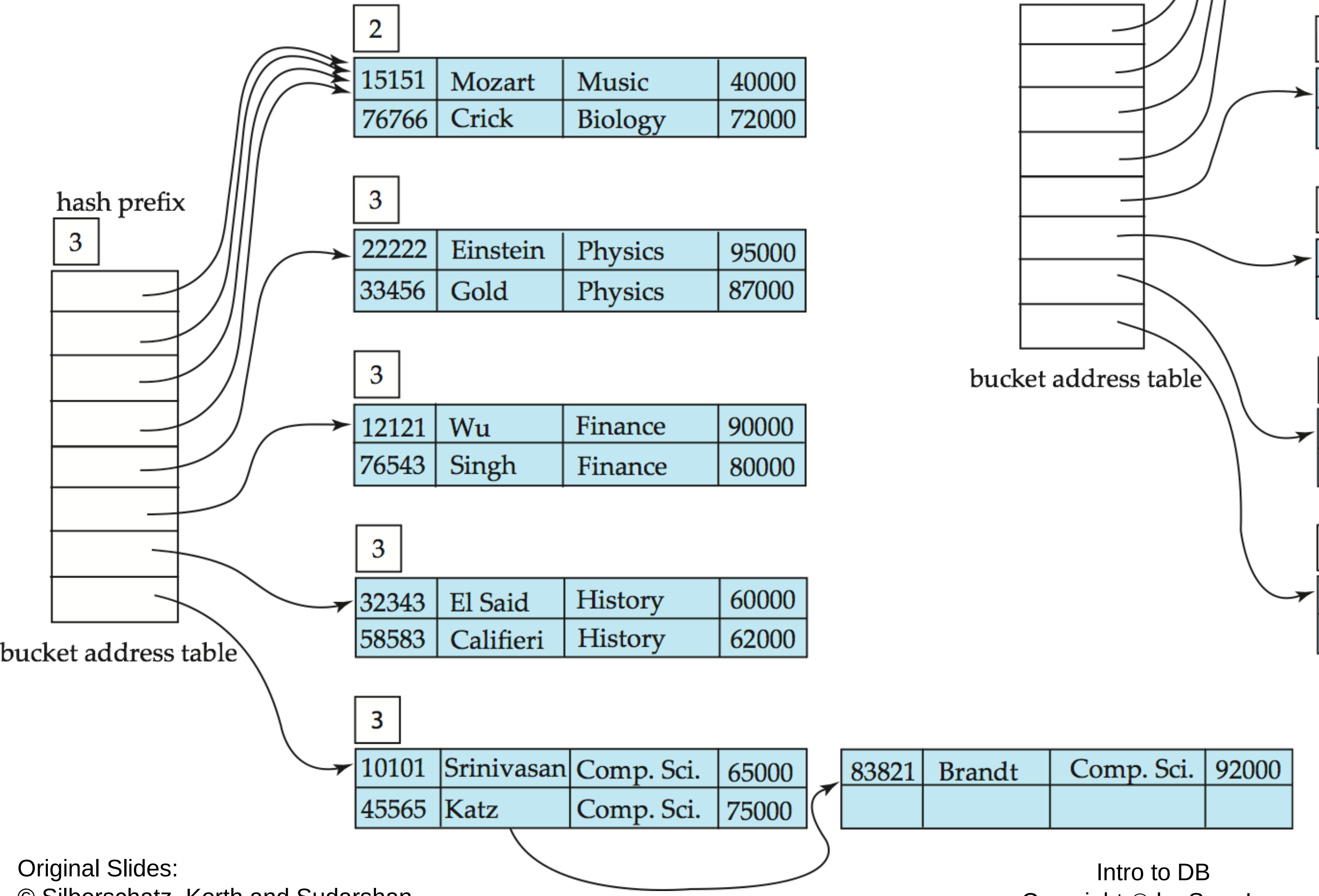


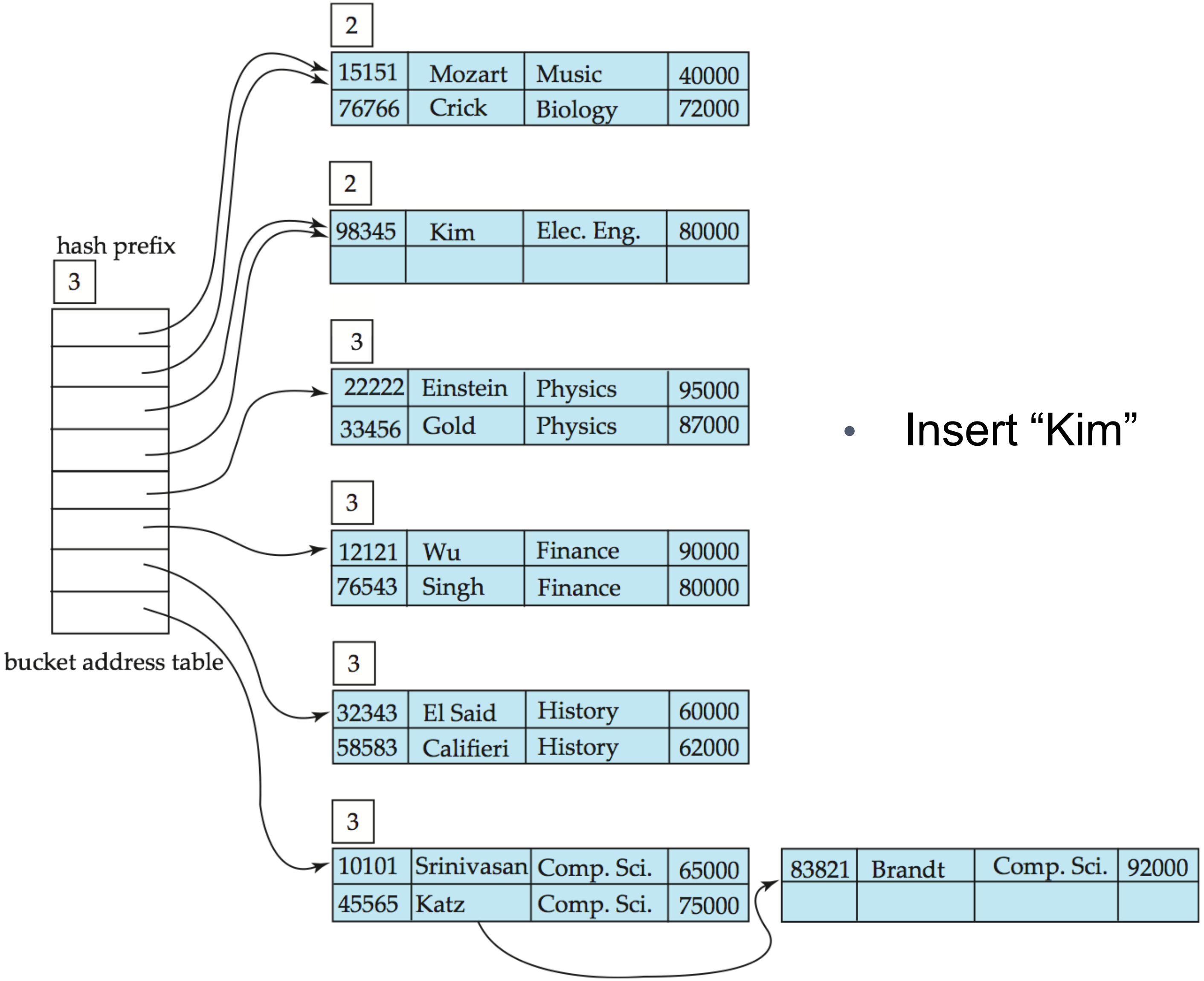
- Insert “Katz”



Example (Cont.)

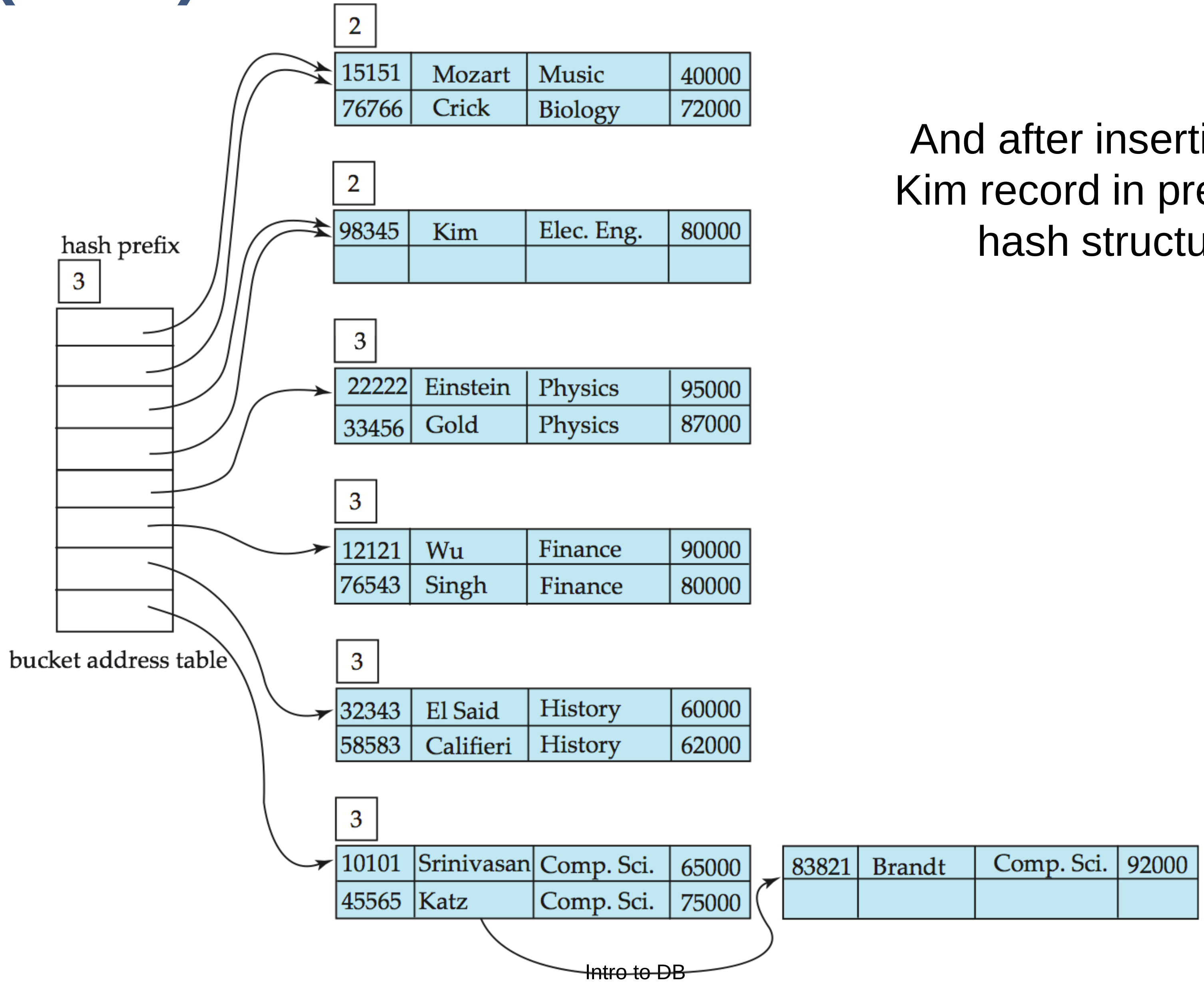
And after insertion of eleven records





- Insert "Kim"

Example (Cont.)



And after insertion of Kim record in previous hash structure

Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
  - Changing size of bucket address table is an expensive operation

Ordered Indexing vs Hashing

- Cost of periodic re-organization
  -
- 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 generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred

Other Issues in Indexing

- Record relocation and secondary indices
  - If a record moves, all secondary indices that store record pointers have to be updated
  - *Solution*: use primary-index search key instead of record pointer in secondary index
    - Extra traversal of primary index to locate record
      - Higher cost for queries, lower costs for node splits
    - Add record-id if primary-index search key is non-unique
- Duplicate keys
  - Buckets used in both hashing and tree indicies
  - Extra care needed in algorithm implementations

Index Definition in SQL

- Create an index

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

  create index *b-index* **on** *branch(branch-name)*

- **create unique index**
  - Not really required if SQL **unique** integrity constraint is supported

- To drop an index

  drop index <index-name>

END OF CHAPTER 11