Chapter 11: Indexing and Hashing

Database System Concepts, 6th Ed.

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

Indexing mechanisms used to speed up access to desired data.

E.g., author catalog in library

Search Key - attribute to set of attributes used to look up records in a file.

An **index file** consists of records (called **index entries**) of the form

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 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.

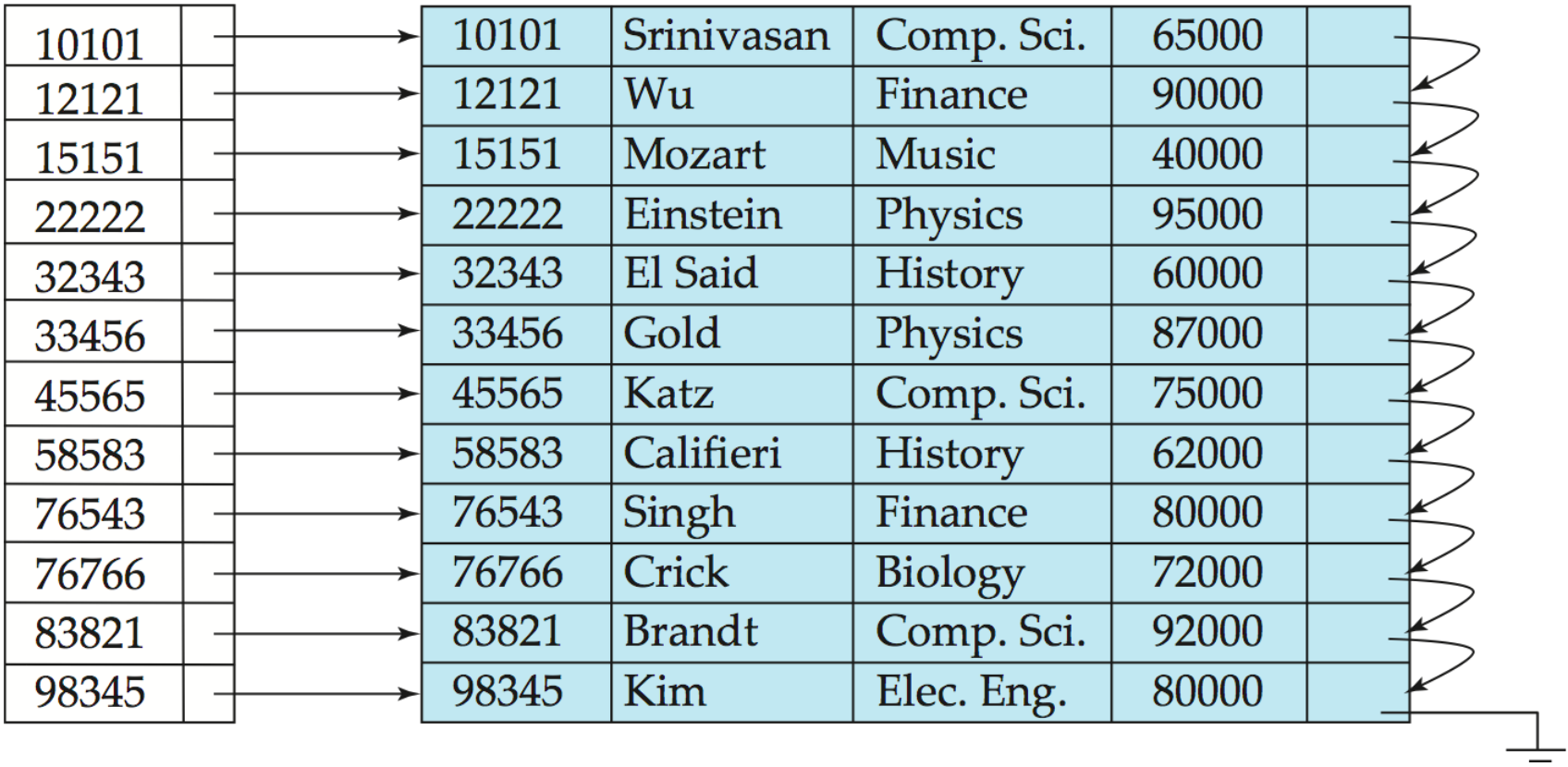
Secondary index: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**

Index-sequential file**:** ordered sequential file with a primary index.

Dense Index Files

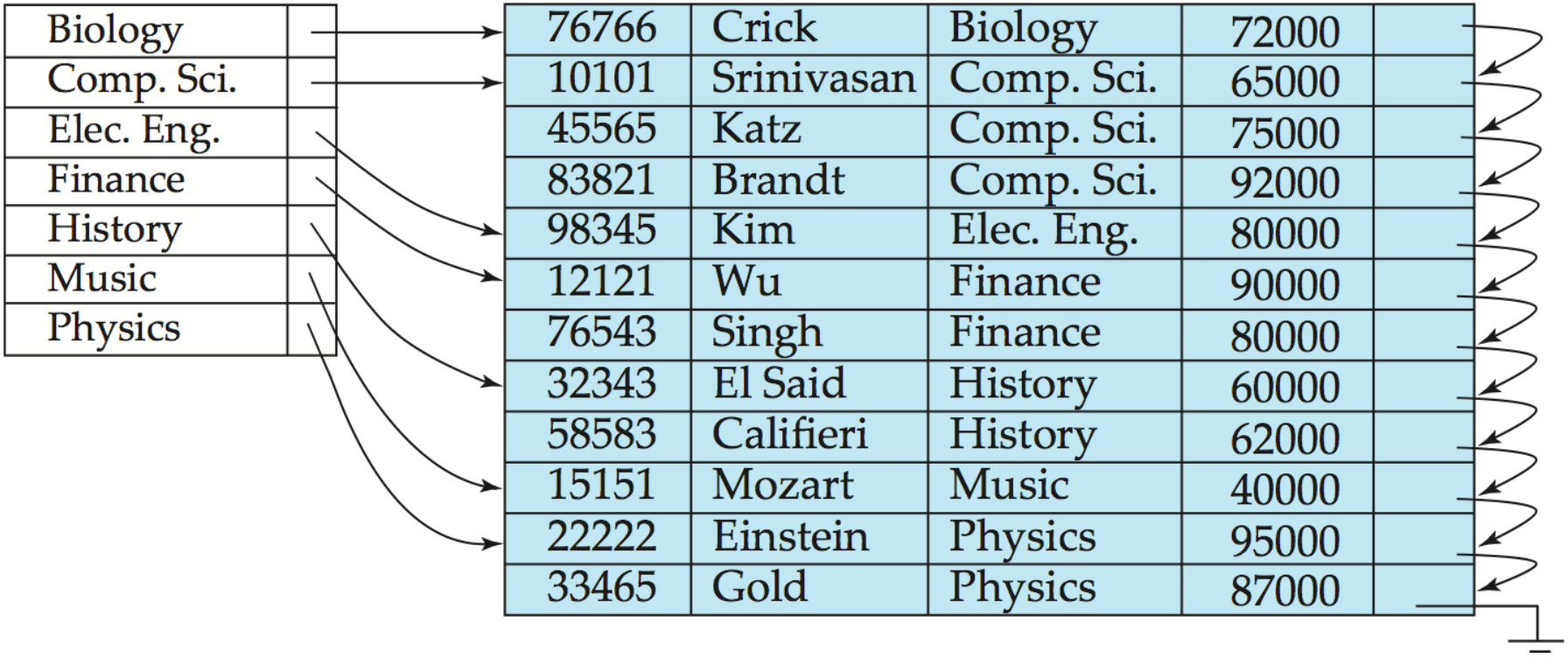
Dense index — Index record appears for every search-key value in the file.

E.g. index on *ID* attribute of *instructor* relation



Dense Index Files (Cont.)

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



Sparse Index Files

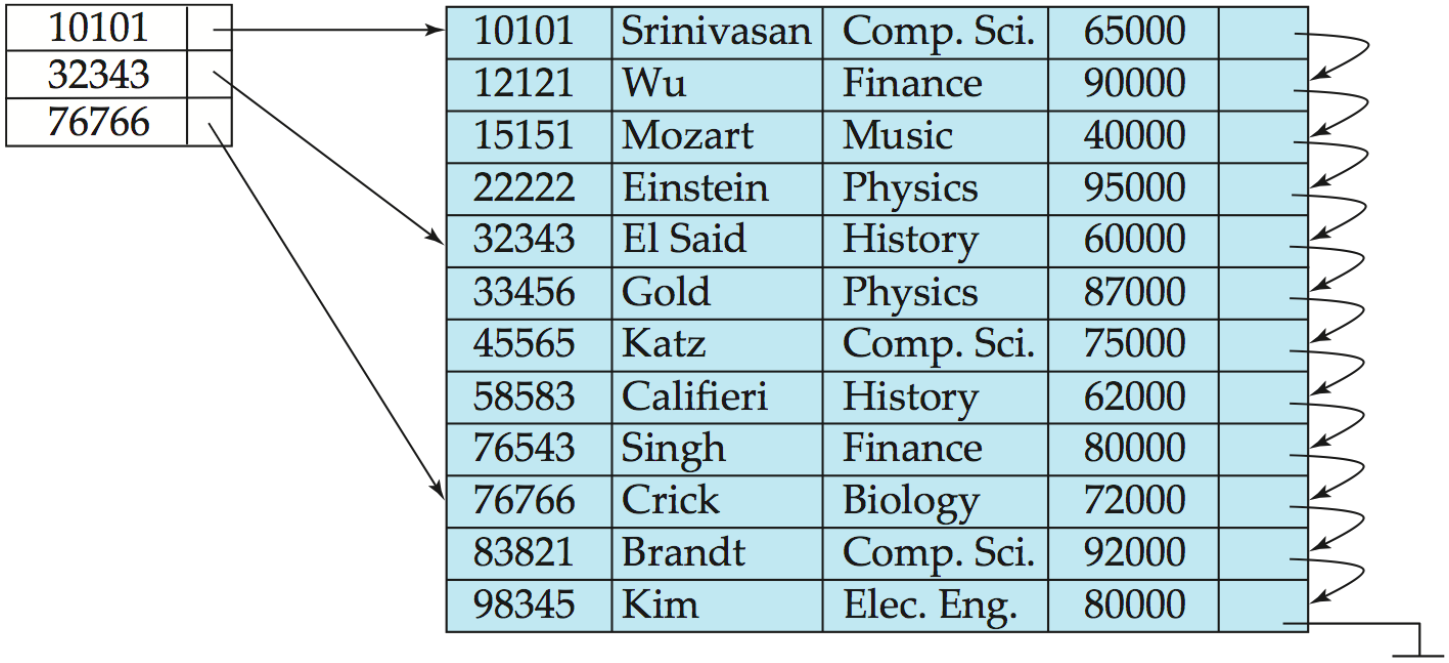
Sparse Index: contains index records for only some search-key values.

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

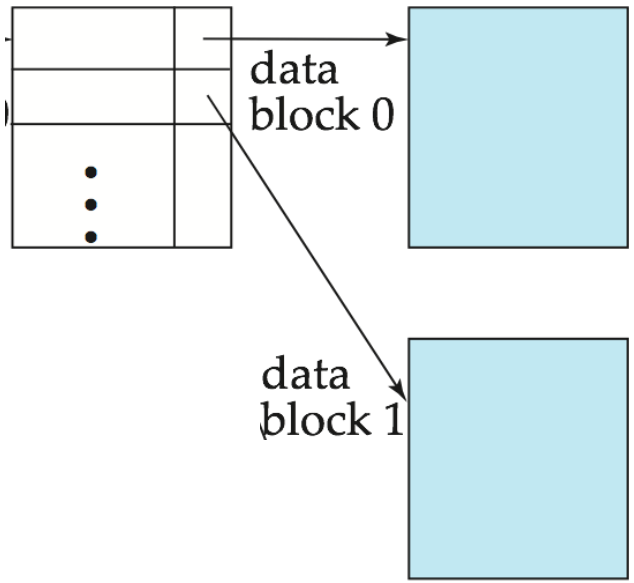


Sparse Index Files (Cont.)

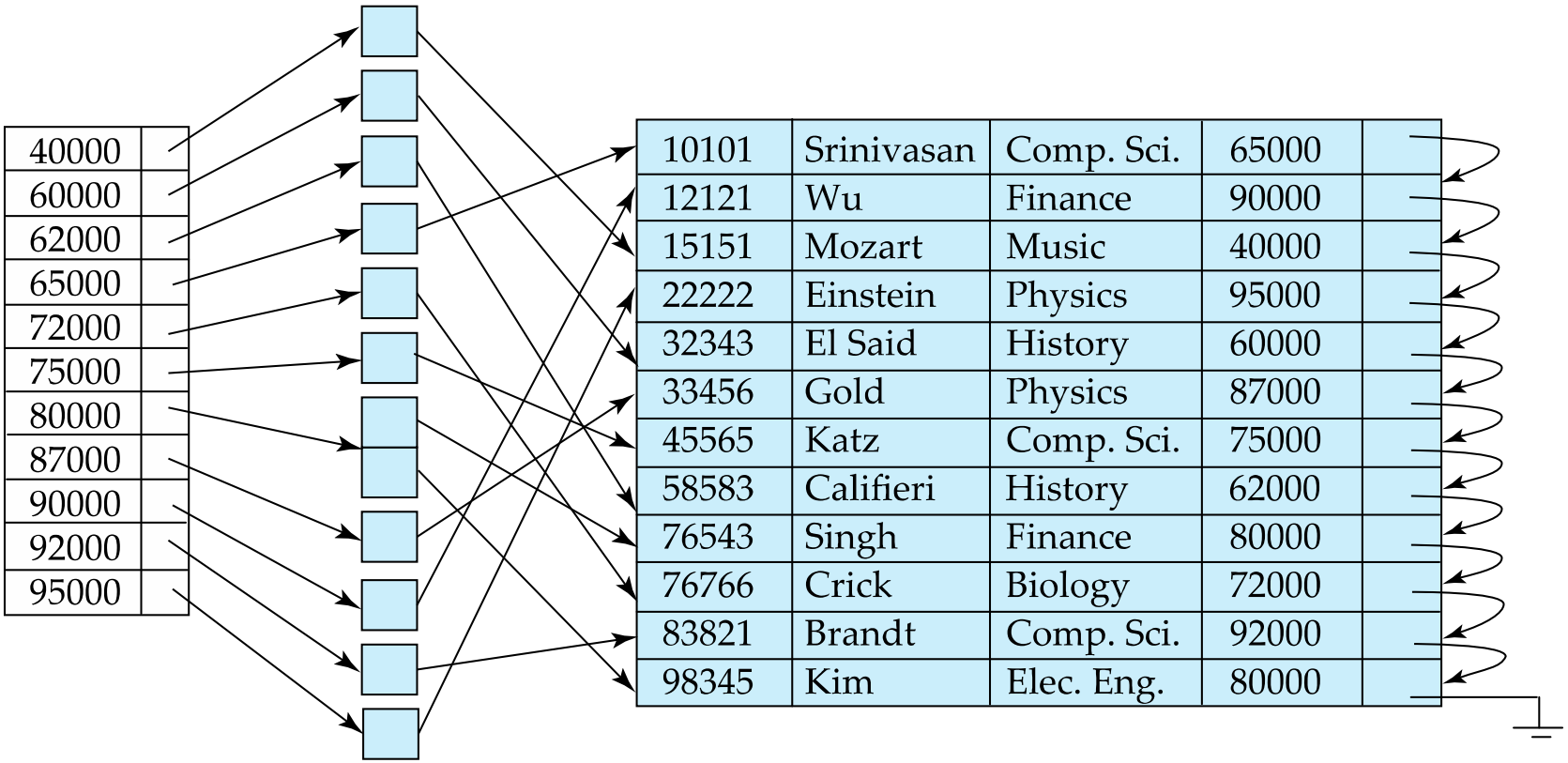
Compared to dense indices:

- Less space and less maintenance overhead for insertions and deletions.
- Generally 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



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

Multilevel Index

If primary index does not fit in memory, access becomes expensive.

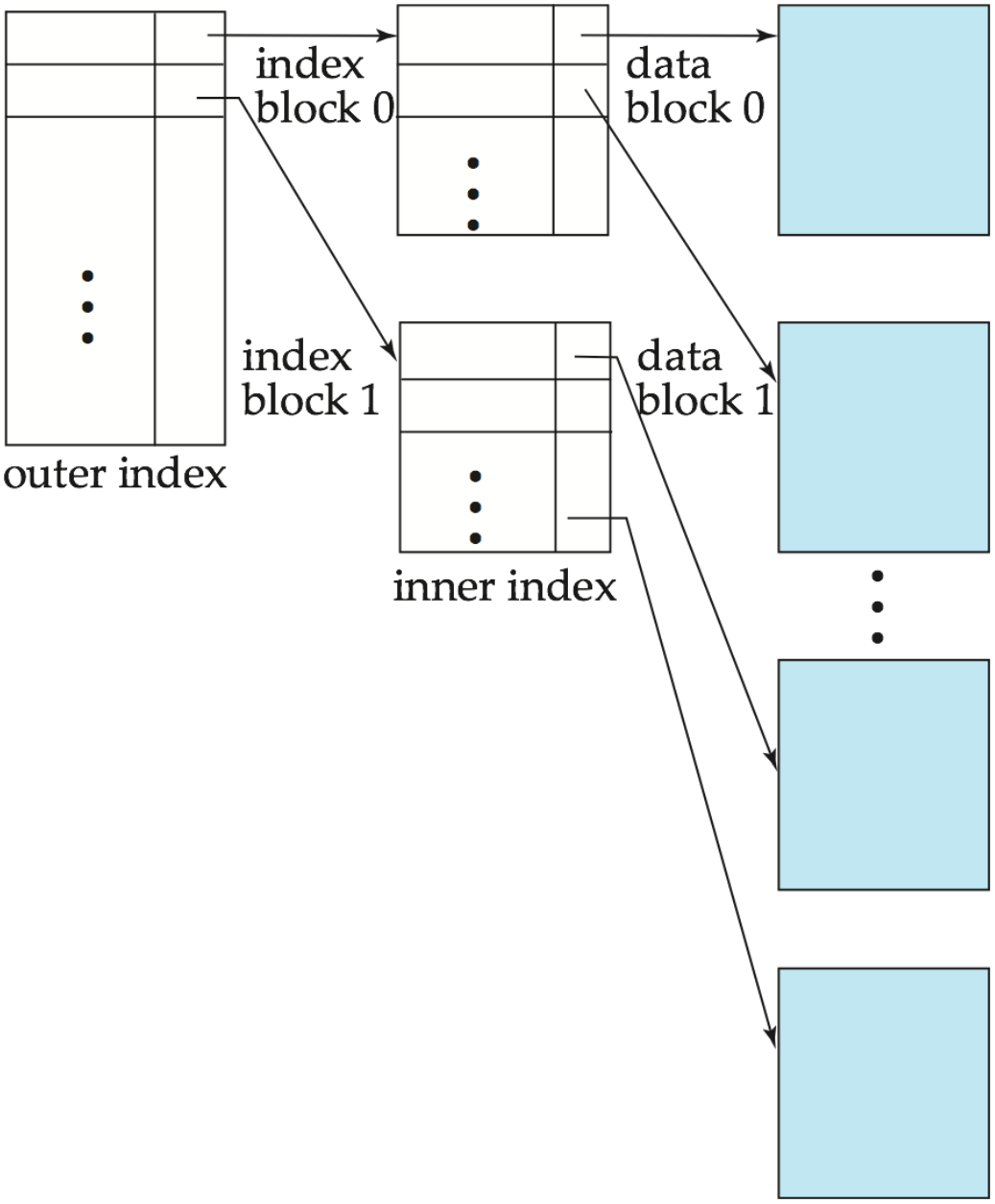
Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.

- outer index – a sparse index of primary index
- inner index – the primary index file

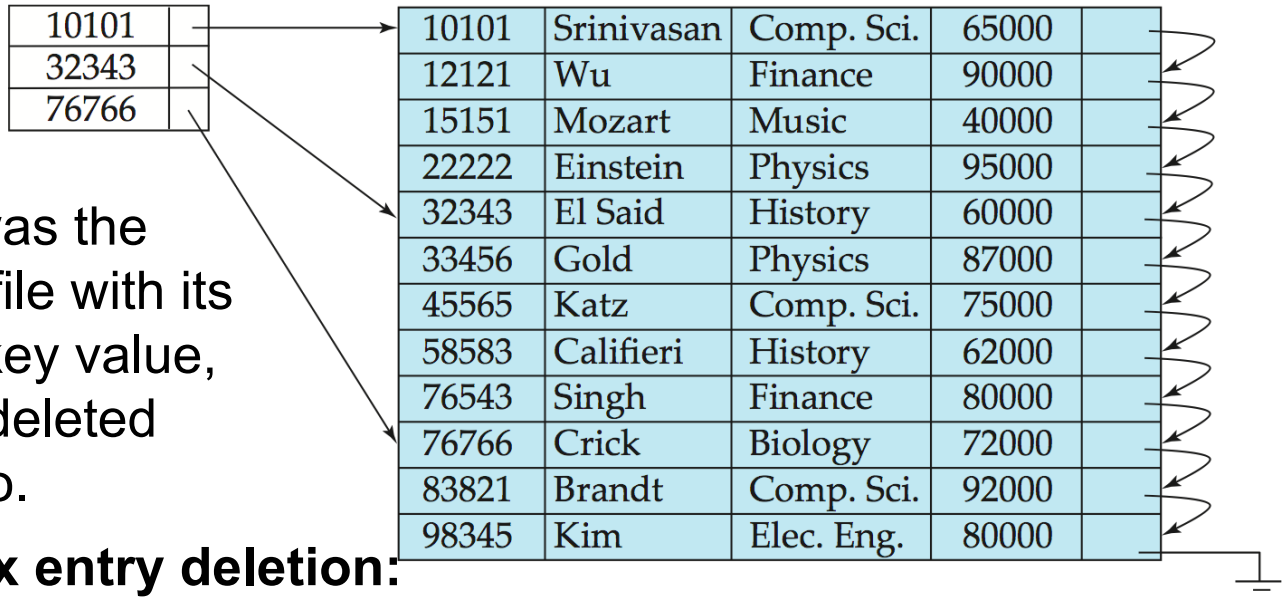
If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.

Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)



Index Update: 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 entry deletion:

Dense indices – deletion of search-key is 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.

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 and deletion: algorithms are simple extensions of the single-level algorithms

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

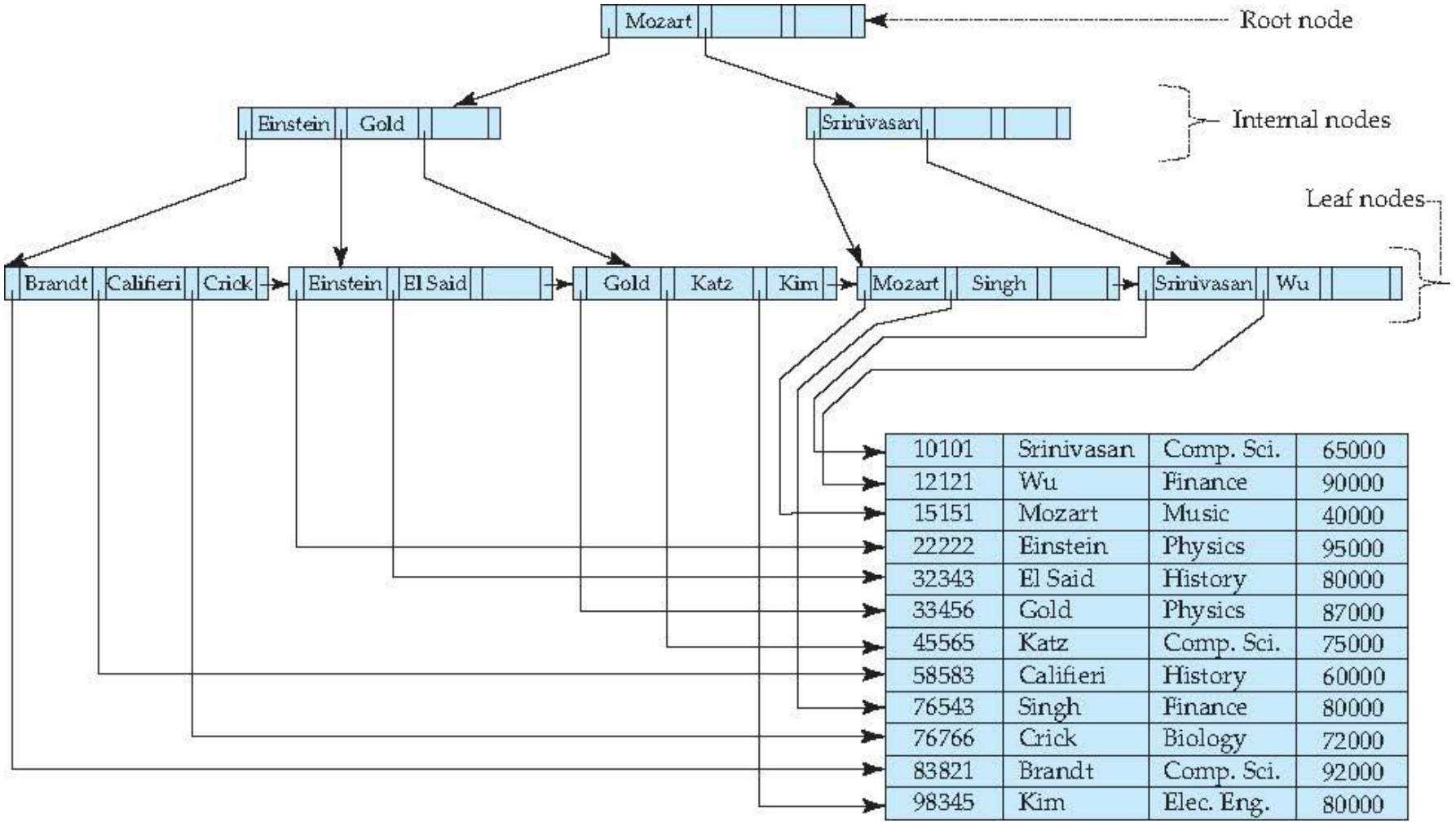
We can have a secondary index with an index record for each search-key value

B^+-Tree Index 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

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

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

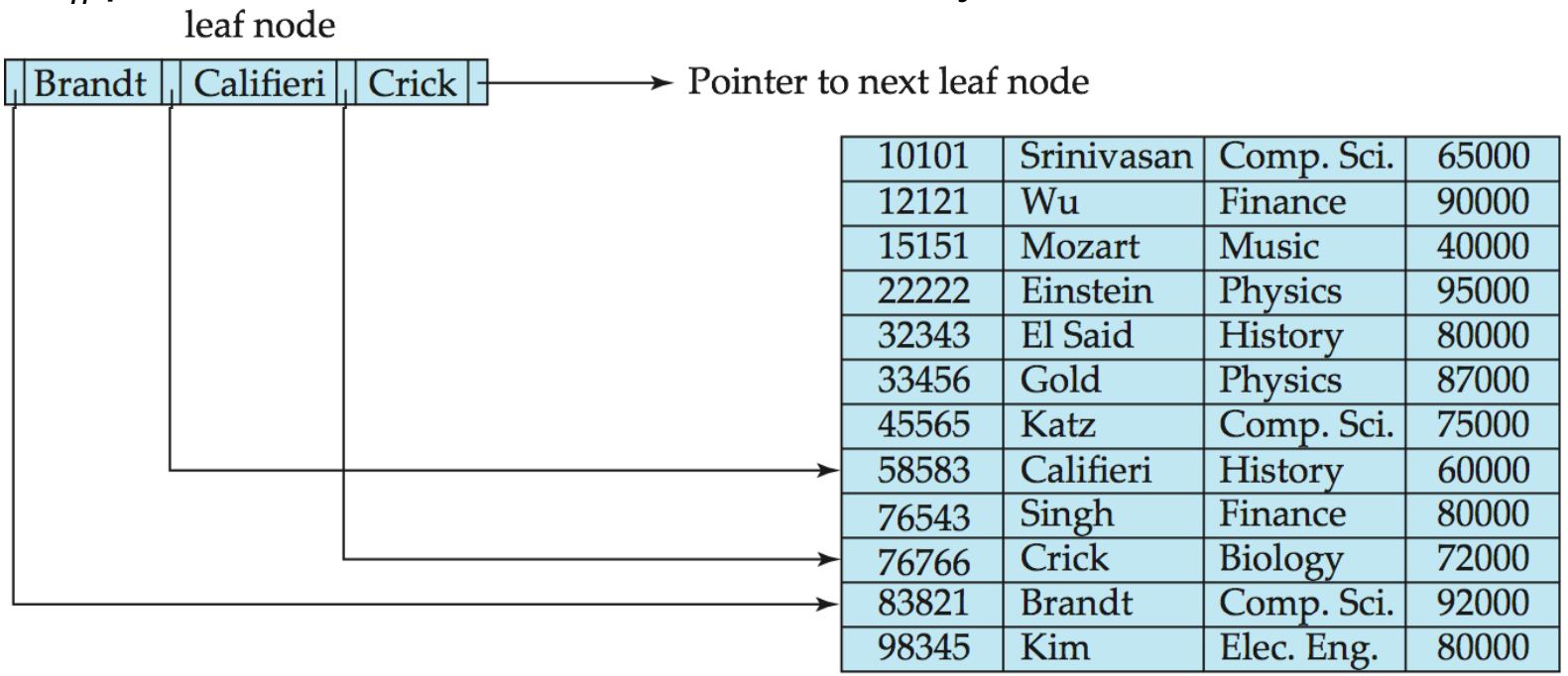
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(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

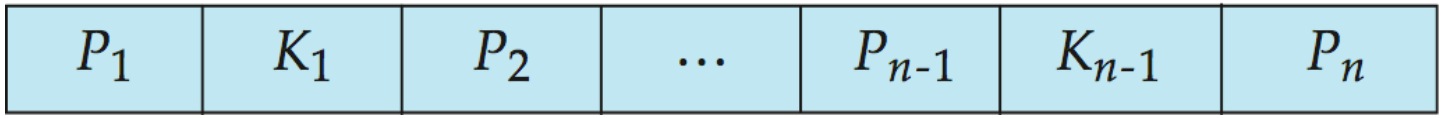


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

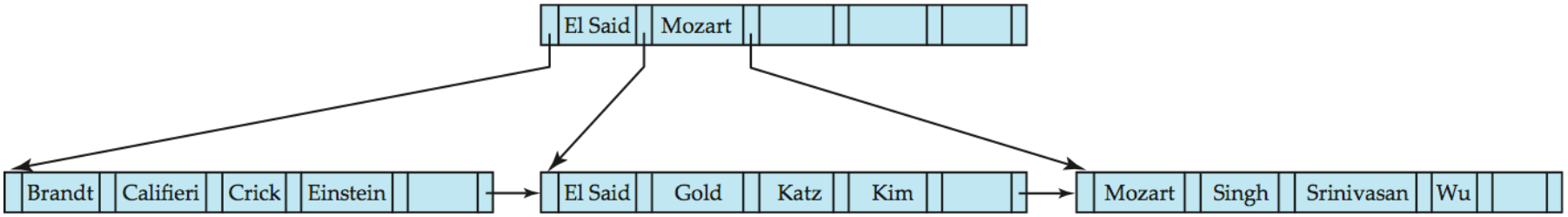
Non-Leaf Nodes in B+-Trees

Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:

- All the search-keys in the subtree to which P_1 points are less than K_1
- For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
- All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}



Example of 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.

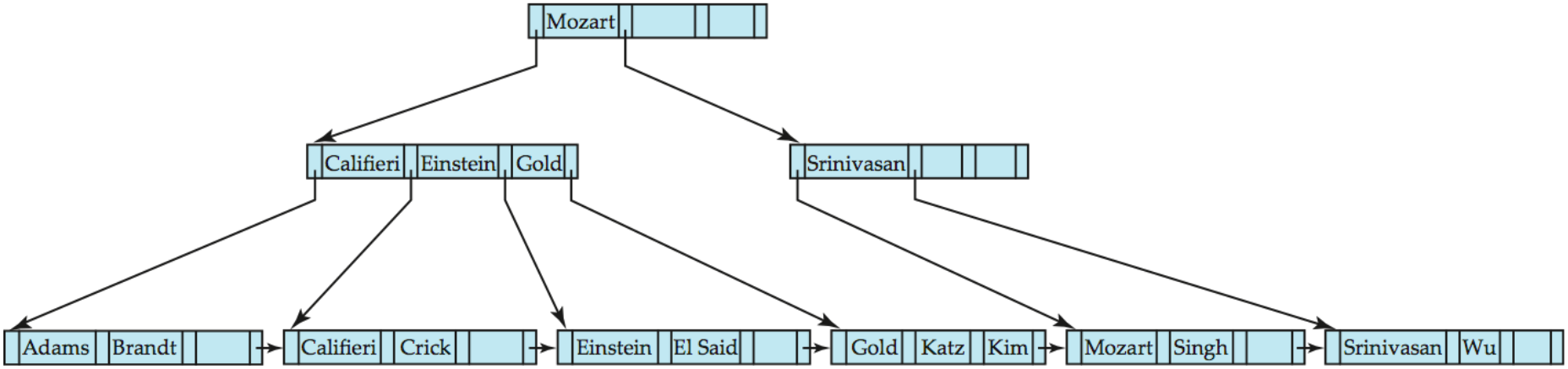
Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

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

Handling Duplicates

We modify find procedure as follows

traverse P_i even if $V = K_i$

As soon as we reach a leaf node C check if C has only search key values less than V

- if so set C = right sibling of C before checking whether C contains V

Procedure printAll

uses modified find procedure to find first occurrence of V

Traverse through consecutive leaves to find all occurrences of V

** Errata note: modified find procedure missing in first printing of 6th edition

Queries on B+-Trees (Cont.)

If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

A node is generally the same size as a disk block, typically 4 kilobytes

and n is typically around 100 (40 bytes per index entry).

With 1 million search key values and n = 100

at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup.

Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup

above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B+-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
    1. Add record to the file
    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) as discussed in the next slide.

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

Splitting a leaf node:

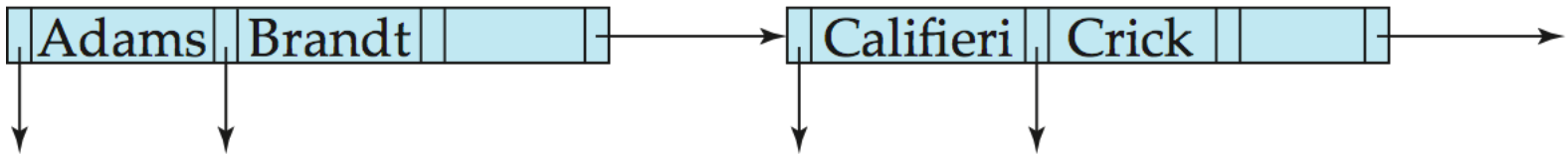
take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.

let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.

If the parent is full, split it and **propagate** the split further up.

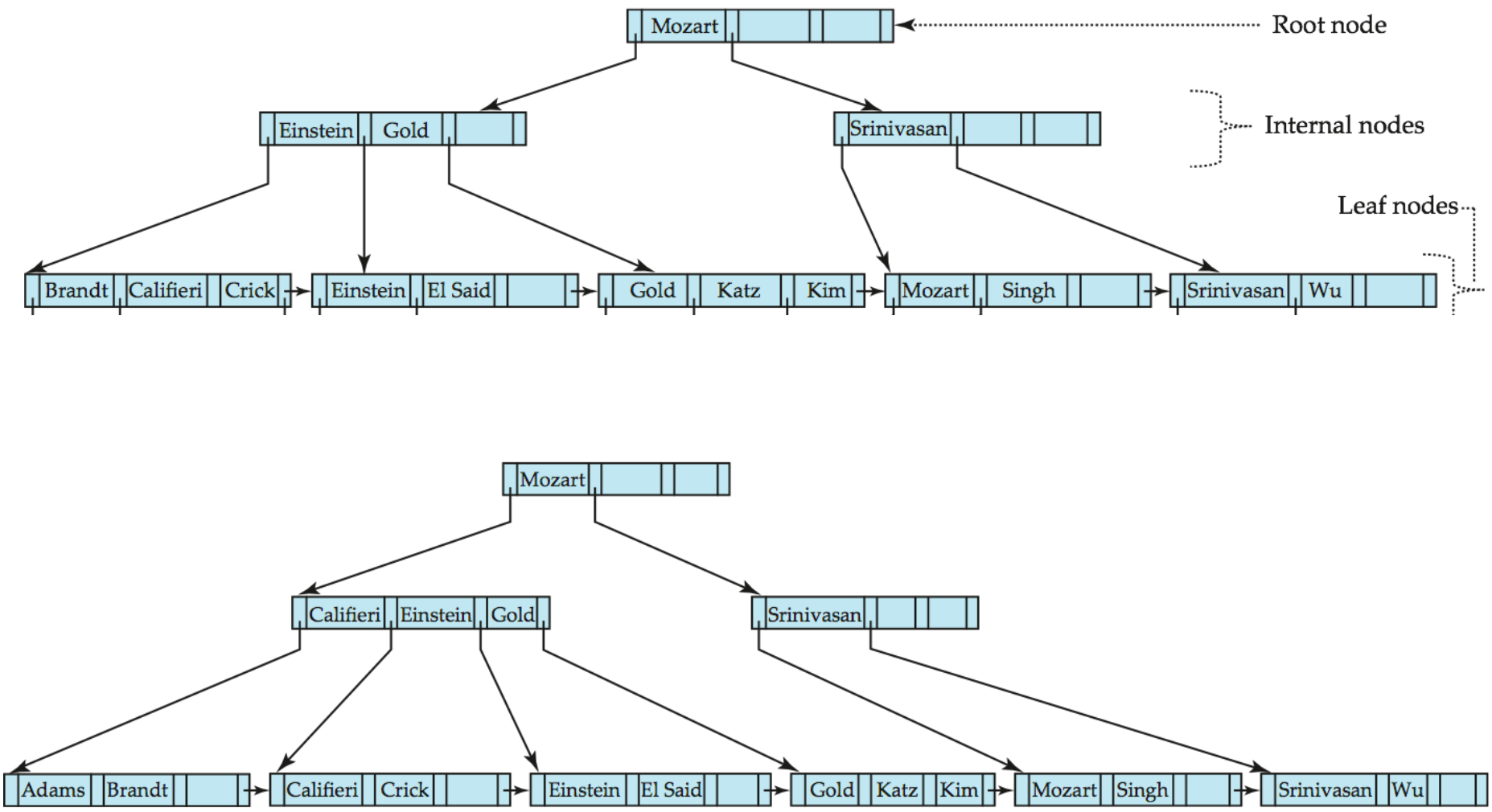
Splitting of nodes proceeds upwards till a node that is not full is found.

In the worst case the root node may be split increasing the height of the tree by 1.



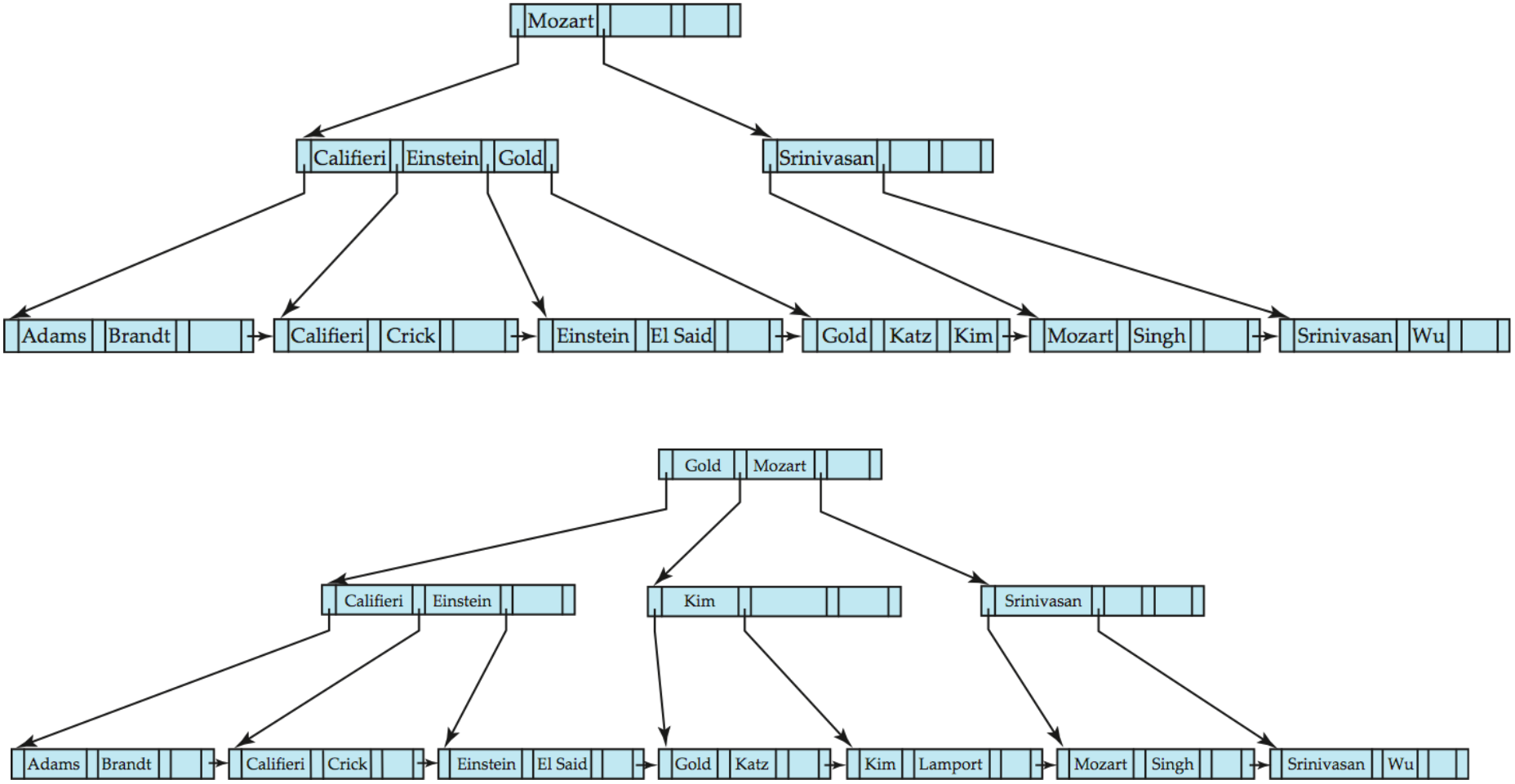
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

B+-Tree Insertion



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

B+-Tree Insertion



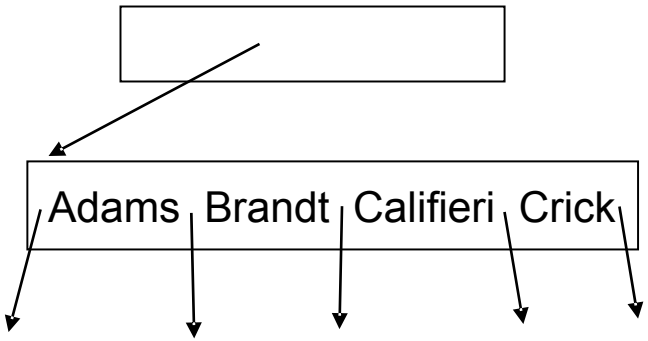
B+-Tree before and after insertion of “Lamport”

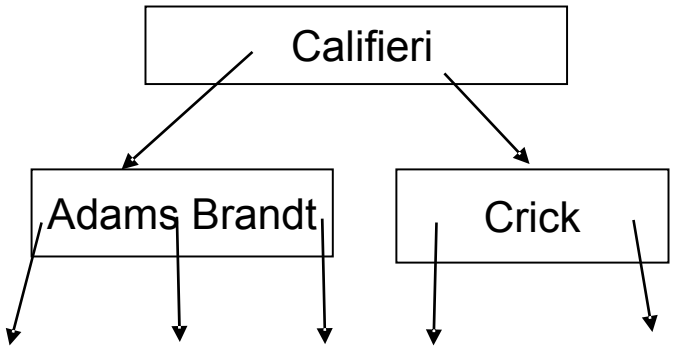
Insertion in B+-Trees (Cont.)

Splitting a non-leaf node: when inserting (k,p) into an already full internal node N

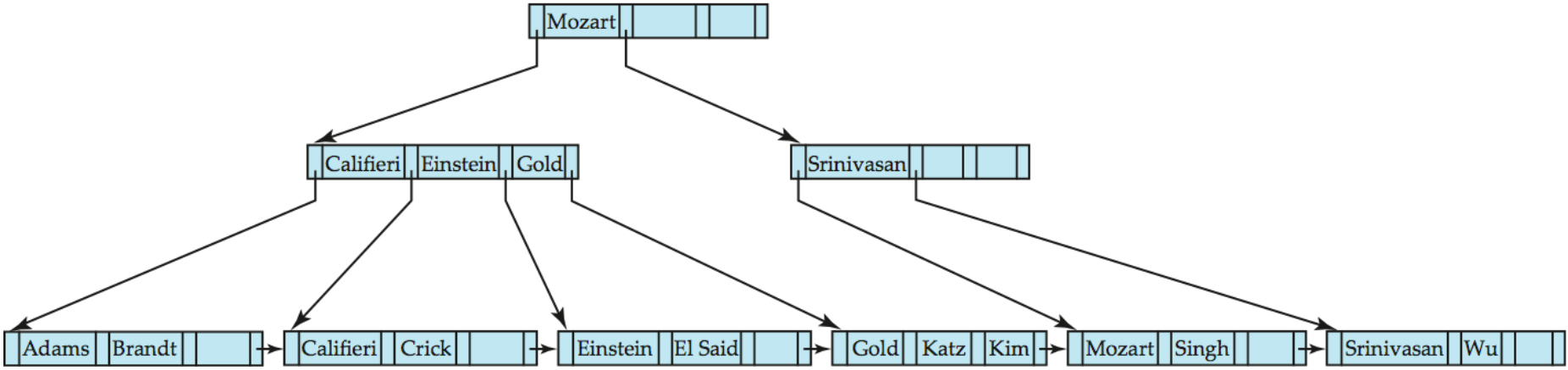
- Copy N to an in-memory area M with space for n+1 pointers and n keys
- Insert (k,p) into M
- Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
- Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
- Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N

Read pseudocode in book!

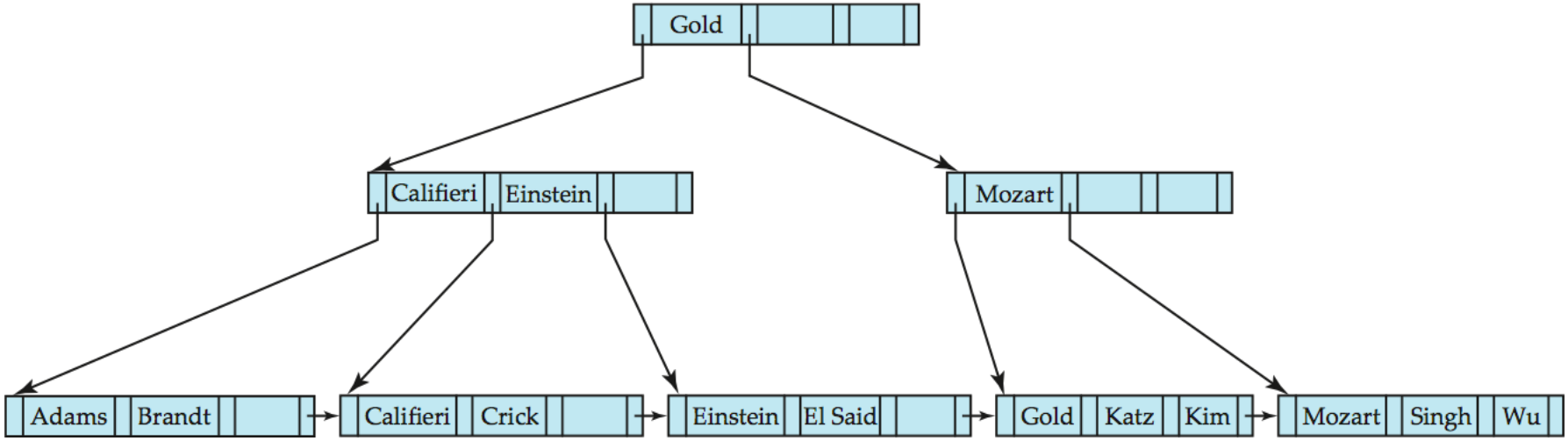




Examples of B+-Tree Deletion

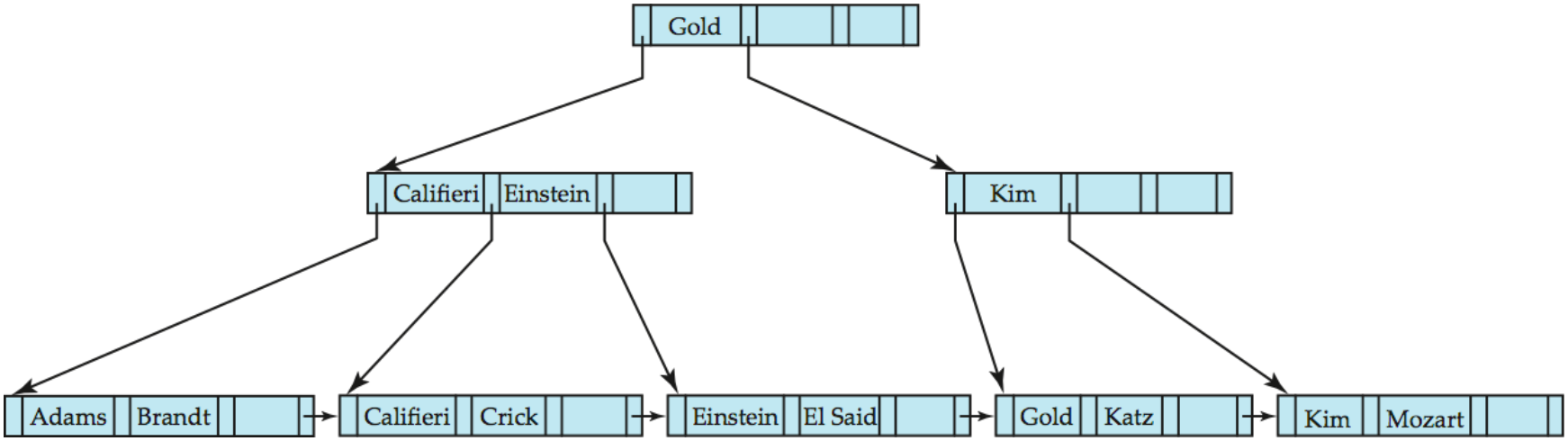


Before and after deleting “Srinivasan”



Deleting “Srinivasan” causes merging of under-full leaves

Examples of B+-Tree Deletion (Cont.)

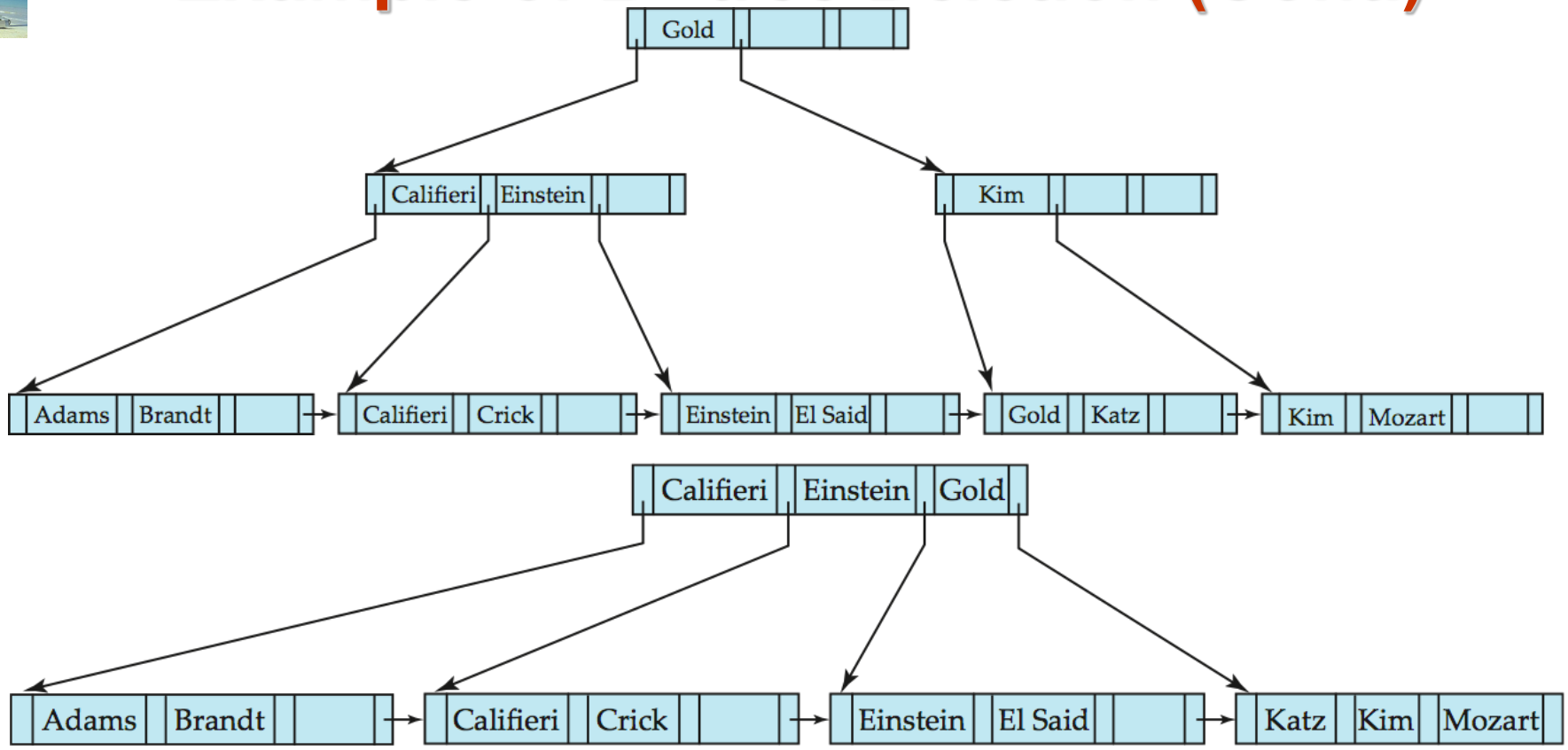


Deletion of “Singh” and “Wu” from result of previous example

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

If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***

Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.

Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

Updates on B+-Trees: Deletion

Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:

- Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
- Update the corresponding search-key value in the parent of the node.

The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.

If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

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

B^+-Tree File Organization

Index file degradation problem is solved by using B^+-Tree indices.

Data file degradation problem is solved by using B^+-Tree File Organization.

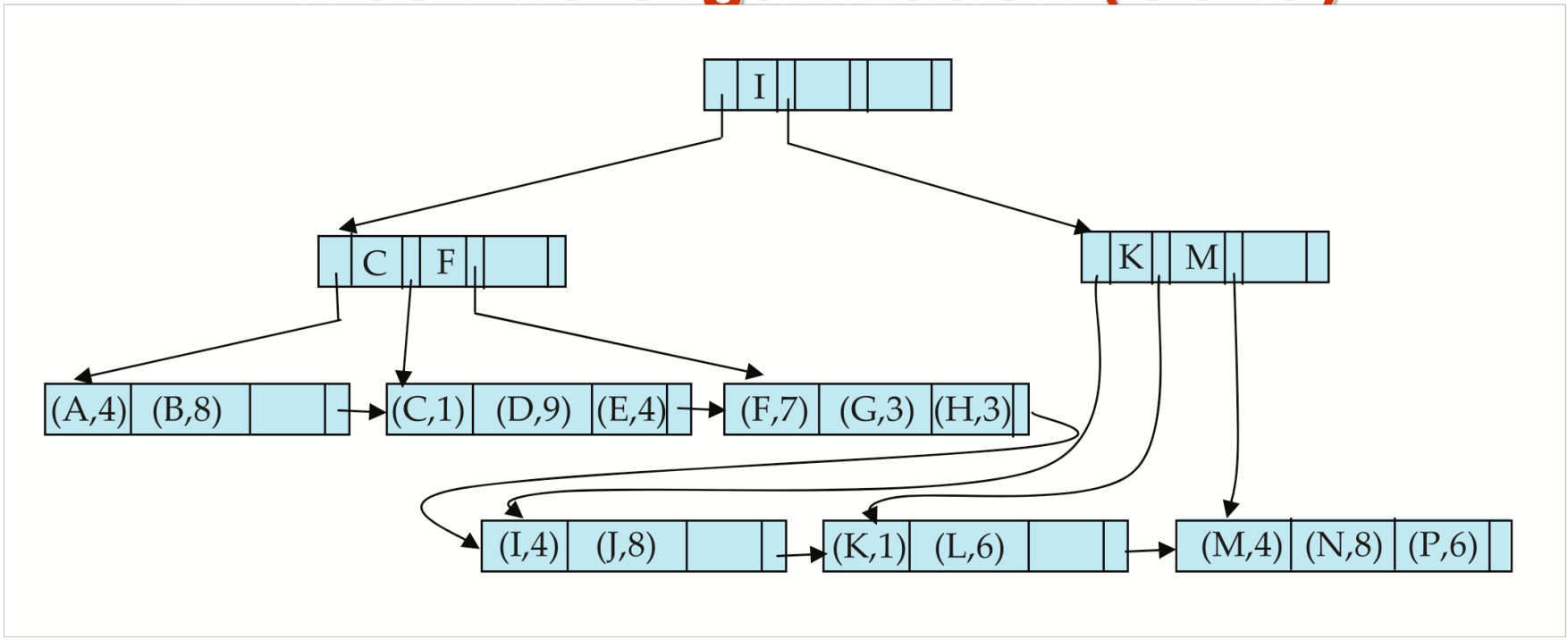
The leaf nodes in a B^+-tree file organization store records, instead of pointers.

Leaf nodes are still required to be half full

> Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.

Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

B+-Tree File Organization (Cont.)



Example of B+-tree File Organization

Good space utilization important since records use more space than pointers.

To improve space utilization, involve more sibling nodes in redistribution during splits and merges

Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

Other Issues in Indexing

Record relocation and secondary indices

If a record moves, all secondary indices that store record pointers have to be updated

Node splits in B^+-tree file organizations become very expensive

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, but node splits are cheap
- Add record-id if primary-index search key is non-unique

Indexing Strings

Variable length strings as keys

Variable fanout

Use space utilization as criterion for splitting, not number of pointers

Prefix compression

Key values at internal nodes can be prefixes of full key

- Keep enough characters to distinguish entries in the subtrees separated by the key value
  - E.g. “Silas” and “Silberschatz” can be separated by “Silb”

Keys in leaf node can be compressed by sharing common prefixes

Bulk Loading and Bottom-Up Build

Inserting entries one-at-a-time into a B^+-tree requires ≥ 1 IO per entry

assuming leaf level does not fit in memory

can be very inefficient for loading a large number of entries at a time (**bulk loading**)

Efficient alternative 1:

sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)

insert in sorted order

- insertion will go to existing page (or cause a split)
- much improved IO performance, but most leaf nodes half full

Efficient alternative 2: **Bottom-up B^+-tree construction**

As before sort entries

And then create tree layer-by-layer, starting with leaf level

- details as an exercise

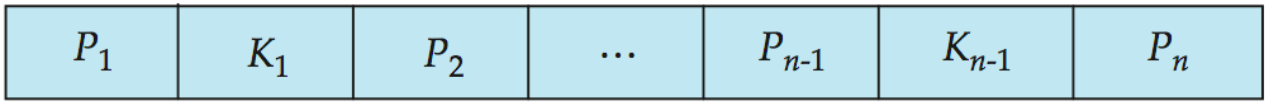
Implemented as part of bulk-load utility by most database systems

B-Tree Index Files

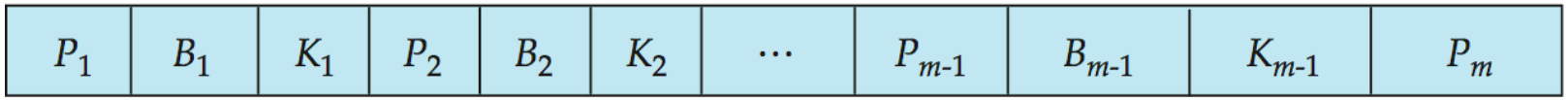
Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.

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.

Generalized B-tree leaf node



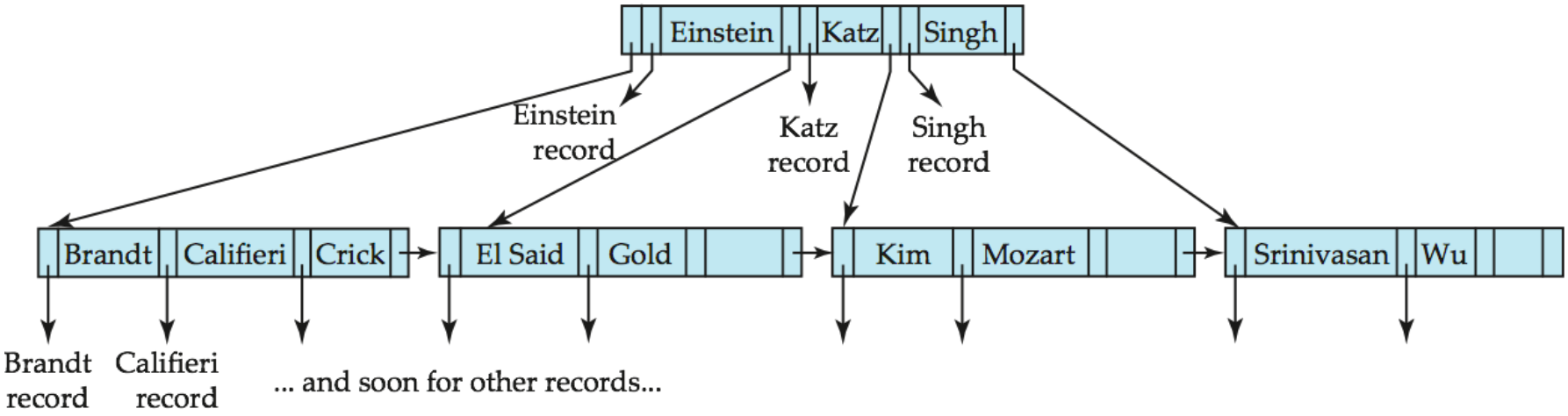
(a)



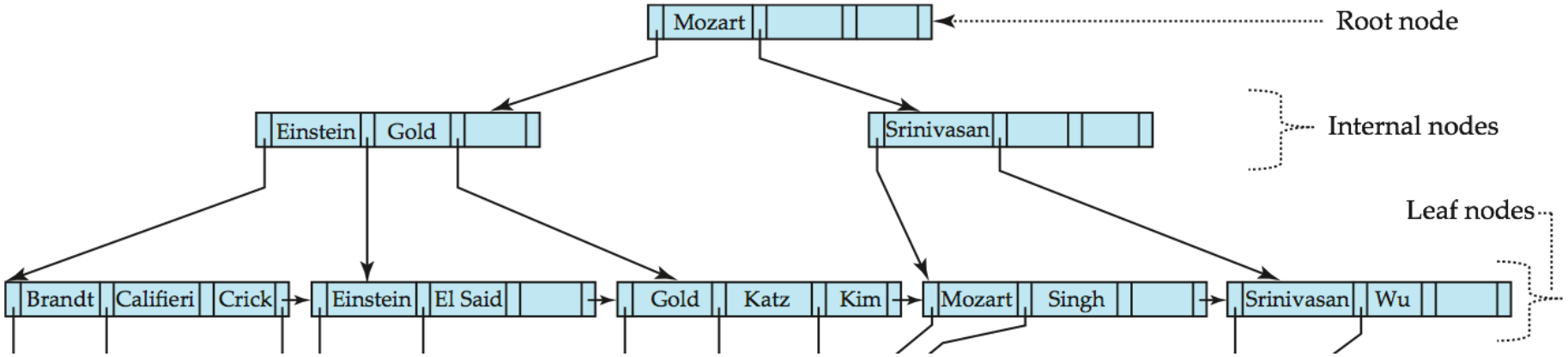
(b)

Nonleaf node – pointers Bi are the bucket or file record pointers.

B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



B-Tree Index Files (Cont.)

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.

Typically, advantages of B-Trees do not out weigh disadvantages.

Multiple-Key Access

Use multiple indices for certain types of queries.

Example:

select *ID*

from *instructor*

where *dept_name* = “Finance” **and** *salary* = 80000

Possible strategies for processing query using indices on single attributes:

1. Use index on *dept_name* to find instructors with department name Finance; test *salary* = *80000*
2. Use index on *salary* to find instructors with a salary of $80000; test *dept_name* = “Finance”.
3. Use *dept_name* index to find pointers to all records pertaining to the “Finance” department. Similarly use index on *salary*. Take intersection of both sets of pointers obtained.

Indices on Multiple Keys

Composite search keys are search keys containing more than one attribute

E.g. (*dept_name, salary*)

Lexicographic ordering: $(a_1, a_2) < (b_1, b_2)$ if either

$a_1 < b_1$, or

$a_1 = b_1$ and $a_2 < b_2$

Indices on Multiple Attributes

Suppose we have an index on combined search-key
(*dept_name, salary*).

With the **where** clause
where *dept_name* = “Finance” **and** *salary* = 80000
the index on (*dept_name, salary*) can be used to fetch only records that satisfy both conditions.

Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.

Can also efficiently handle
where *dept_name* = “Finance” **and** *salary* < 80000

But cannot efficiently handle
where *dept_name* < “Finance” **and** *balance* = 80000

May fetch many records that satisfy the first but not the second condition

Other Features

Covering indices

Add extra attributes to index so (some) queries can avoid fetching the actual records

- Particularly useful for secondary indices
  - Why?

Can store extra attributes only at leaf

Hashing

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

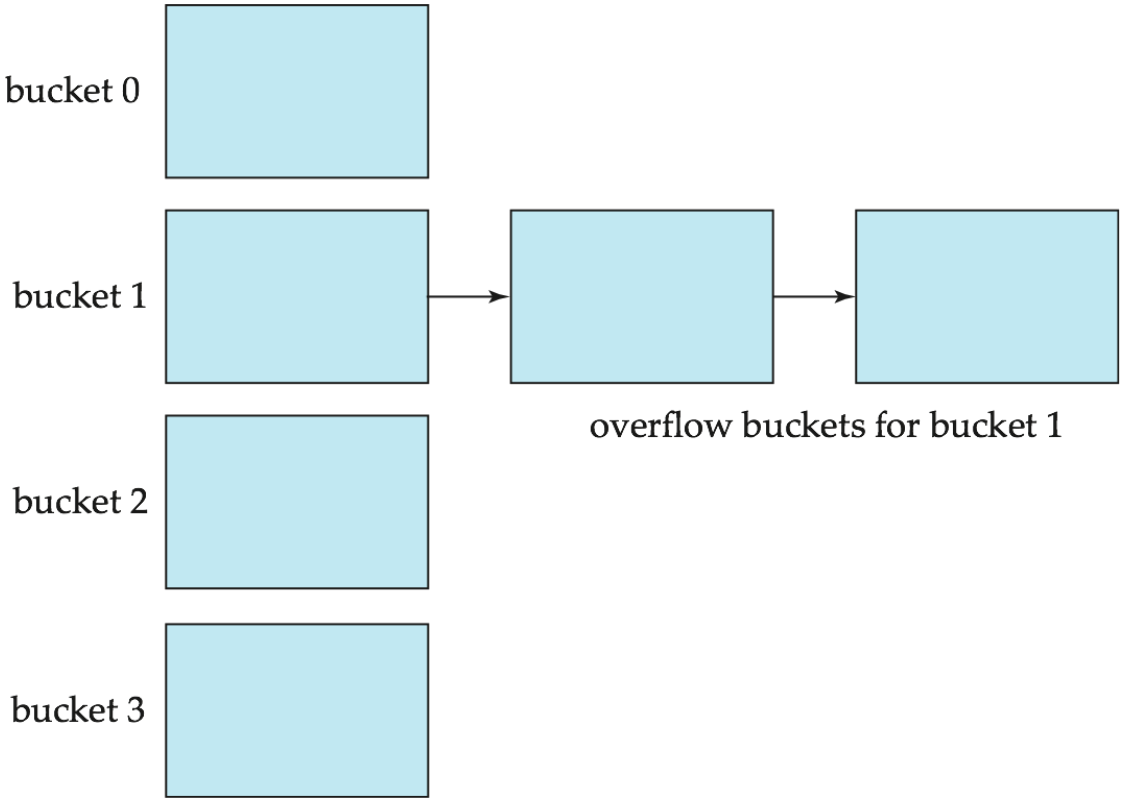
Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets***.

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 addressing**.

An alternative, called **open addressing**, which does not use overflow buckets, is not suitable for database applications.



Hash Indices

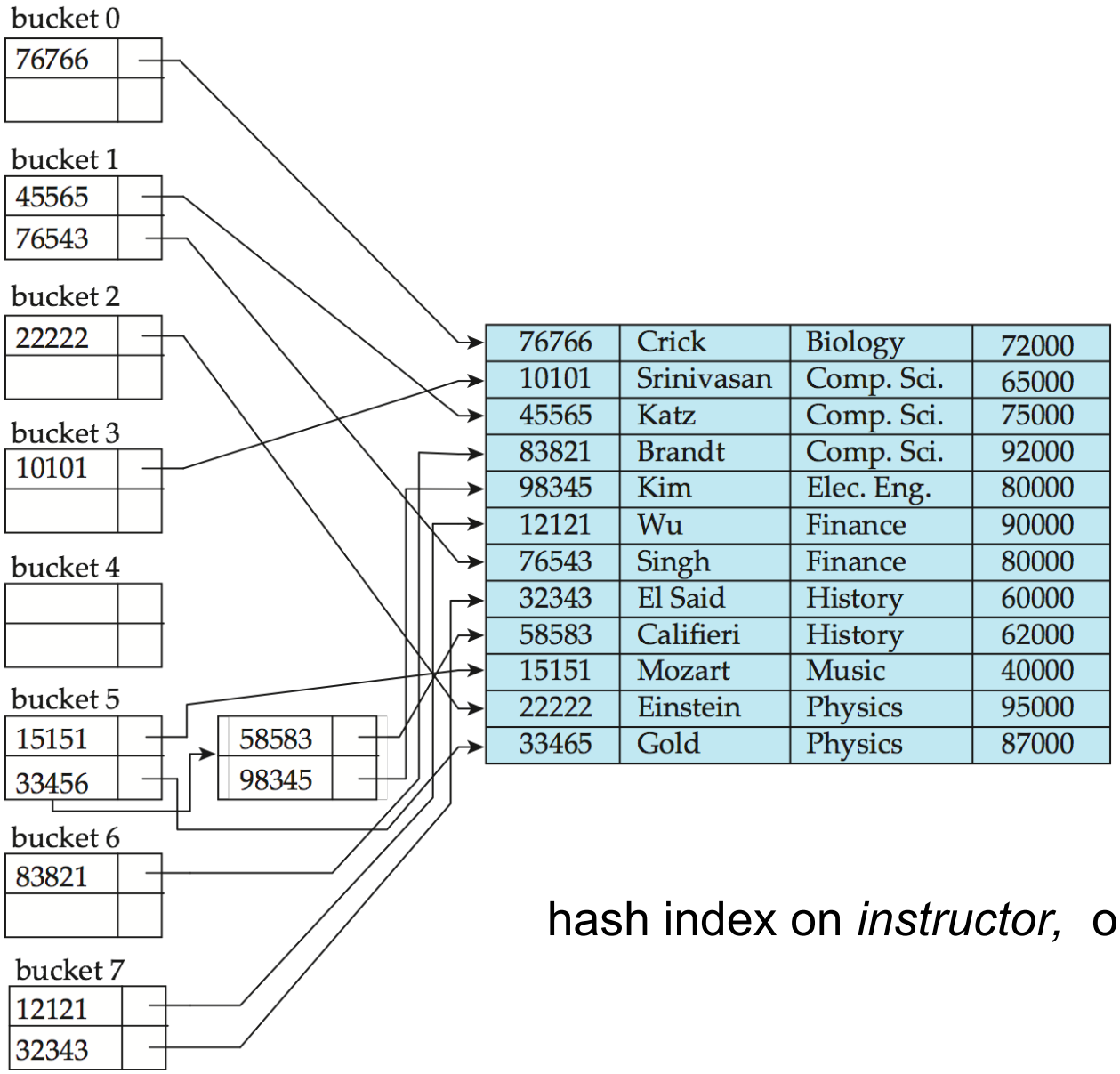
Hashing can be used not only for file organization, but also for index-structure creation.

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.

Dynamic Hashing

Good for database that grows and shrinks in size

Allows the hash function to be modified dynamically

Extendable hashing – one form of dynamic hashing

Hash function generates values over a large range — typically *b*-bit integers, with $b = 32$.

At any time use only a prefix of the hash function to index into a table of bucket addresses.

Let the length of the prefix be *i* bits, $0 \le i \le 32$.

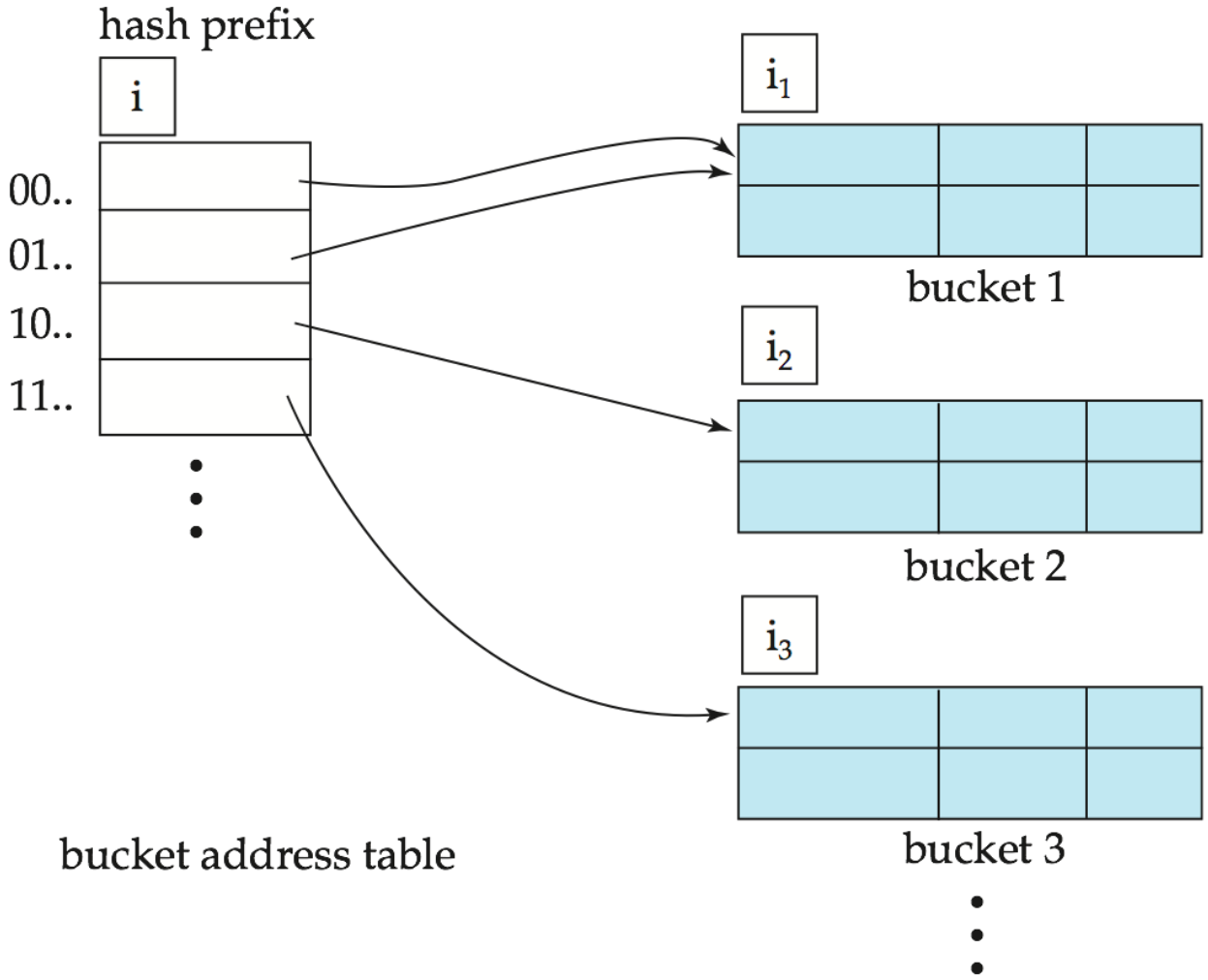
- Bucket address table size = 2^{i}. Initially $i = 0$
- Value of *i* grows and shrinks as the size of the database grows and shrinks.

Multiple entries in the bucket address table may point to a bucket (why?)

Thus, actual number of buckets is $< 2^{i}$

- 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)

Use of Extendable Hash Structure

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)

Insertion in Extendable Hash Structure (Cont)

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.

Deletion in Extendable Hash Structure

To delete a key value,

locate it 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 (can coalesce only with a "*buddy*" bucket having same value of i_j and same i_j –1 prefix, if it is present)

Decreasing bucket address table size is also possible

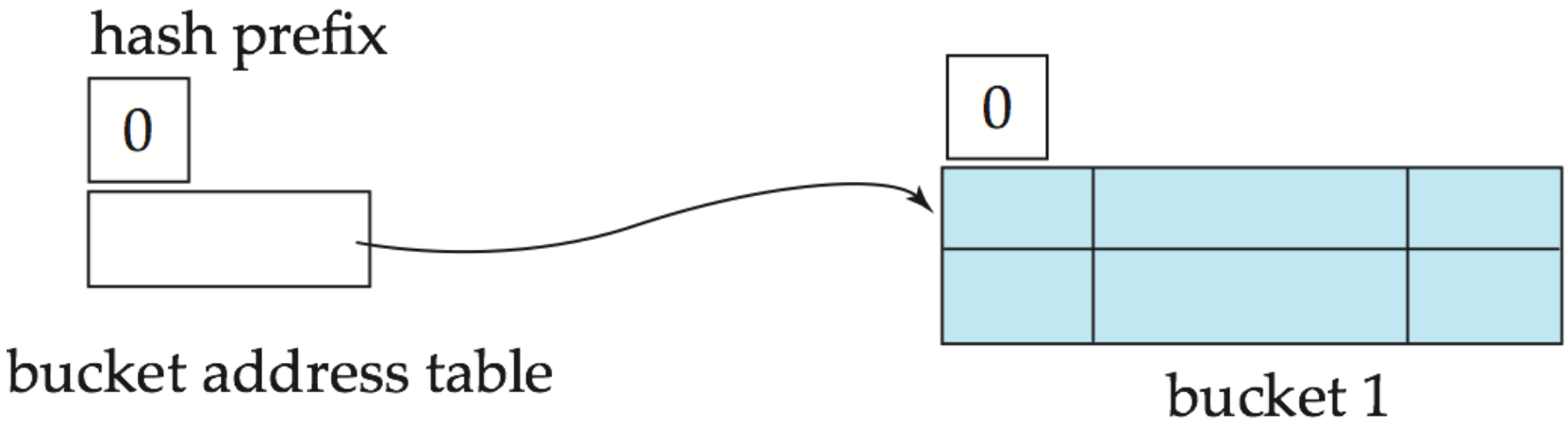
- Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Use of Extendable Hash Structure: Example

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

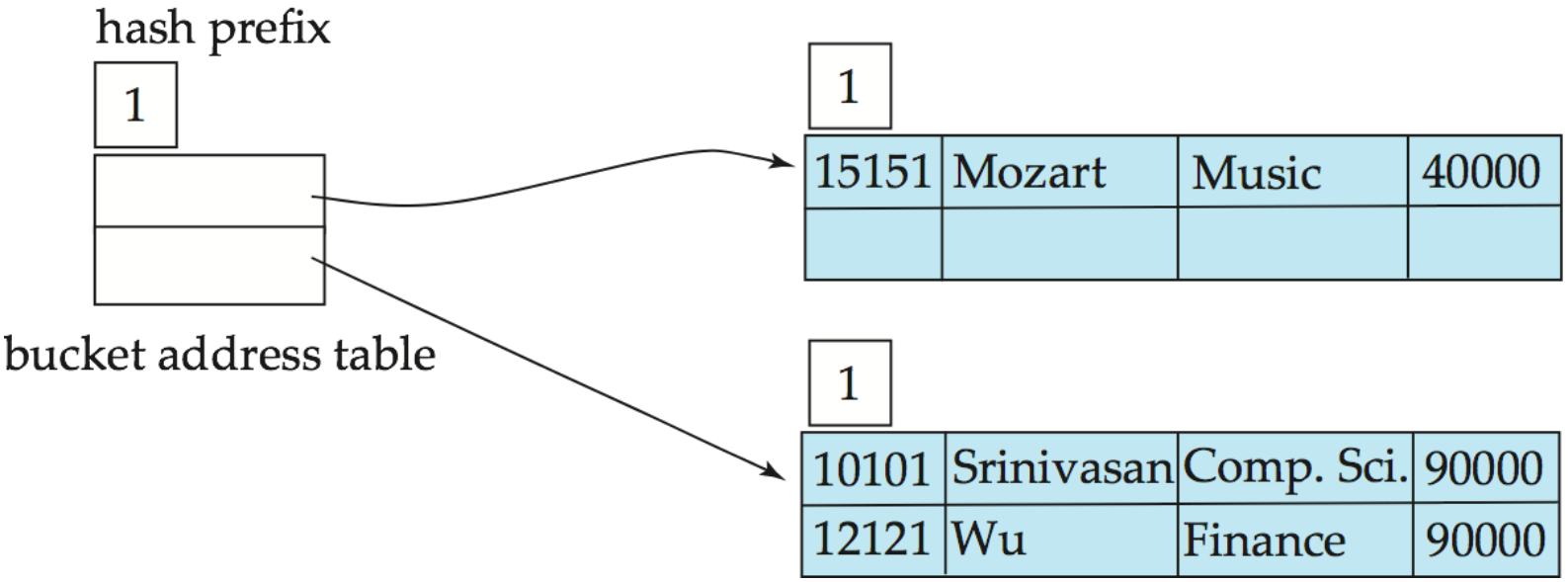
Example (Cont.)

Initial Hash structure; bucket size = 2



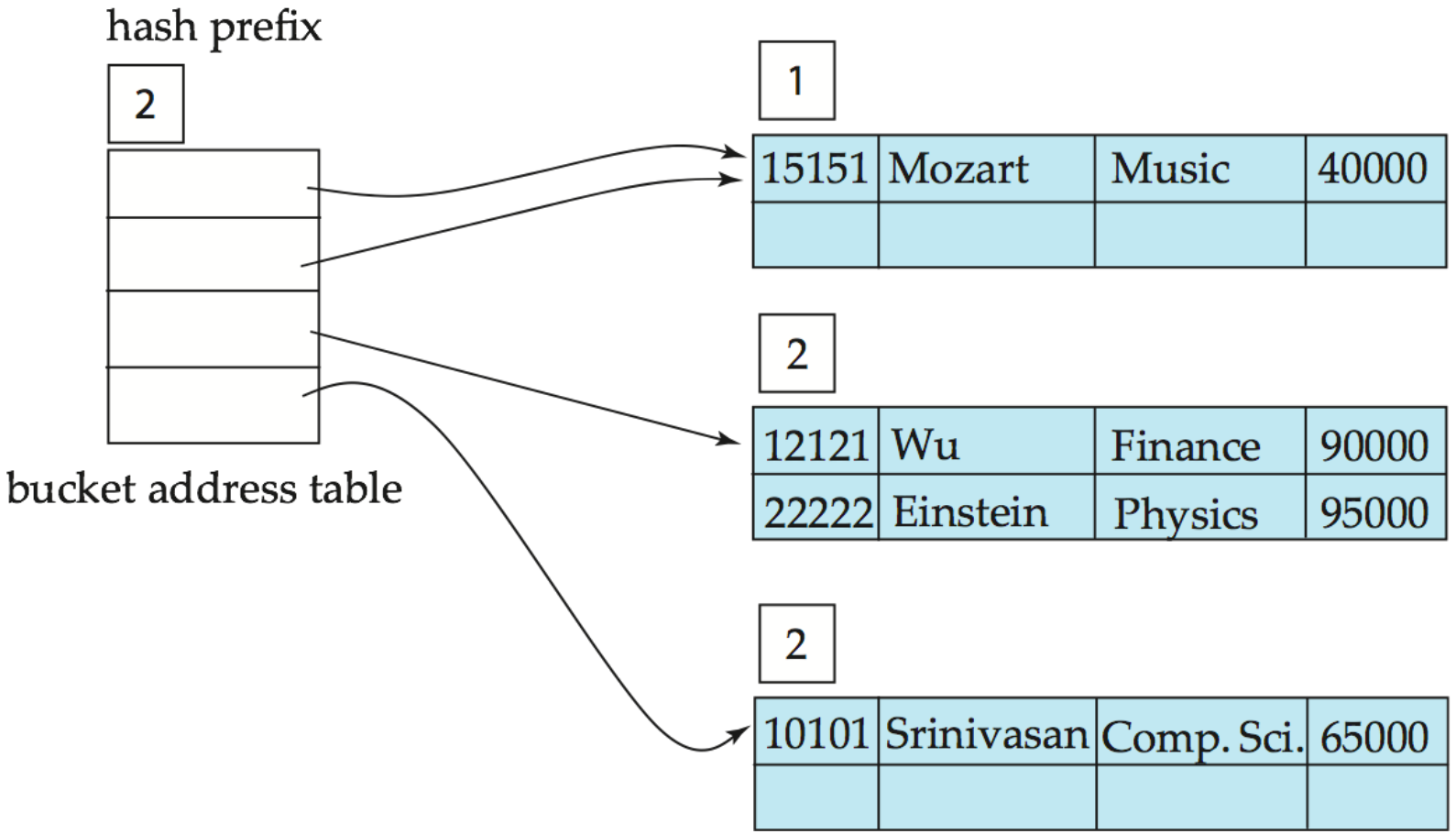
Example (Cont.)

Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records



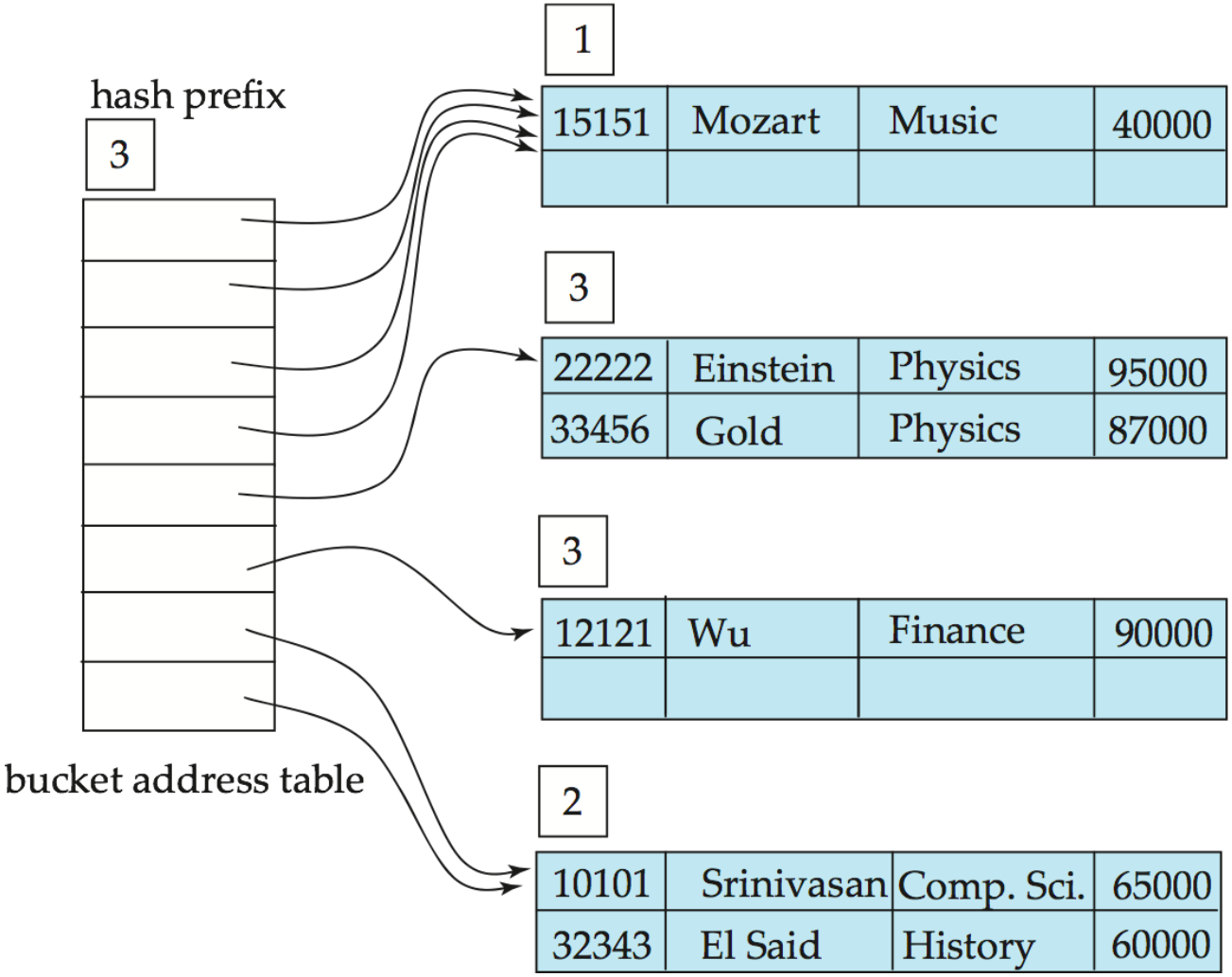
Example (Cont.)

Hash structure after insertion of Einstein record



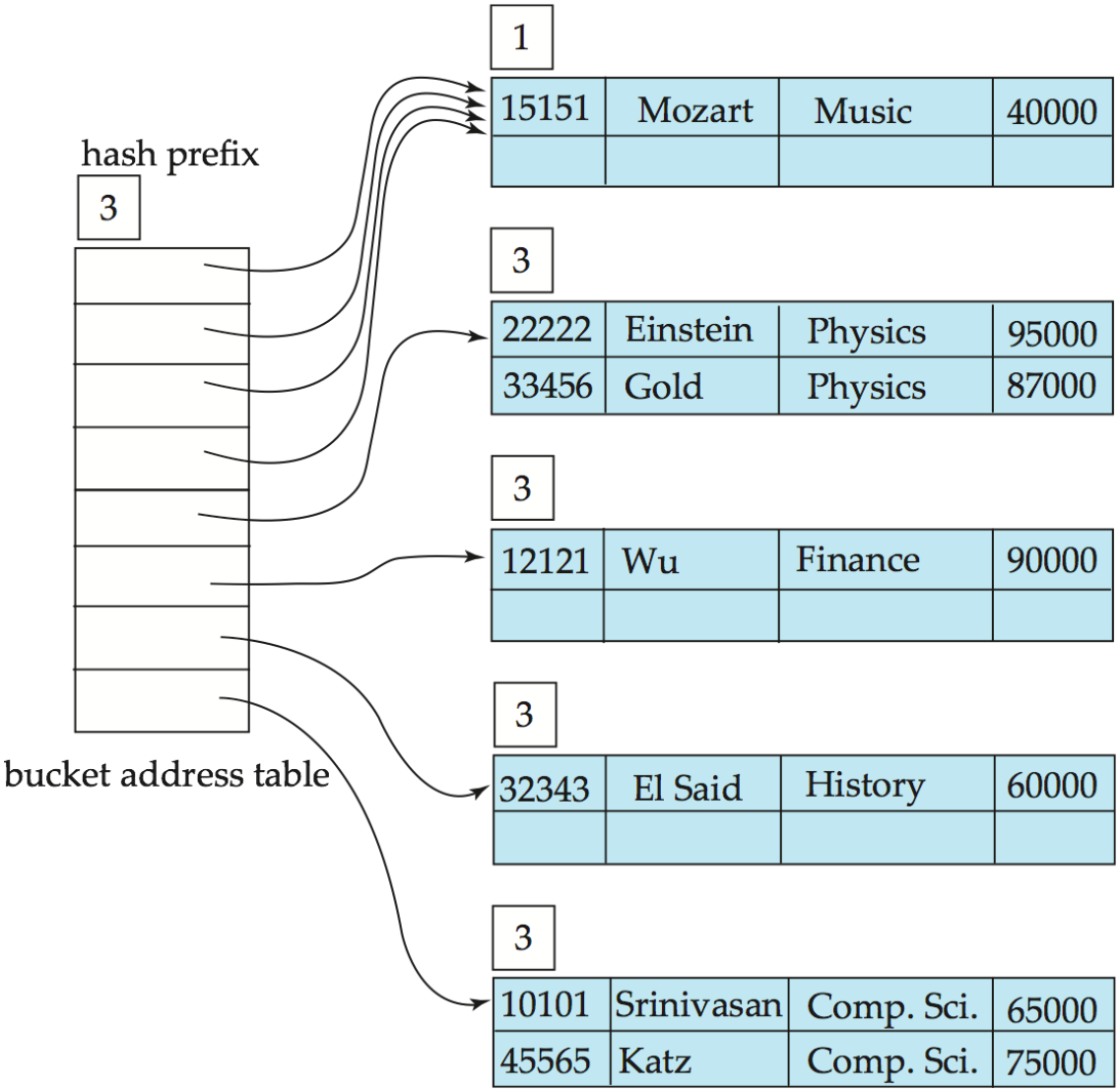
Example (Cont.)

Hash structure after insertion of Gold and El Said records

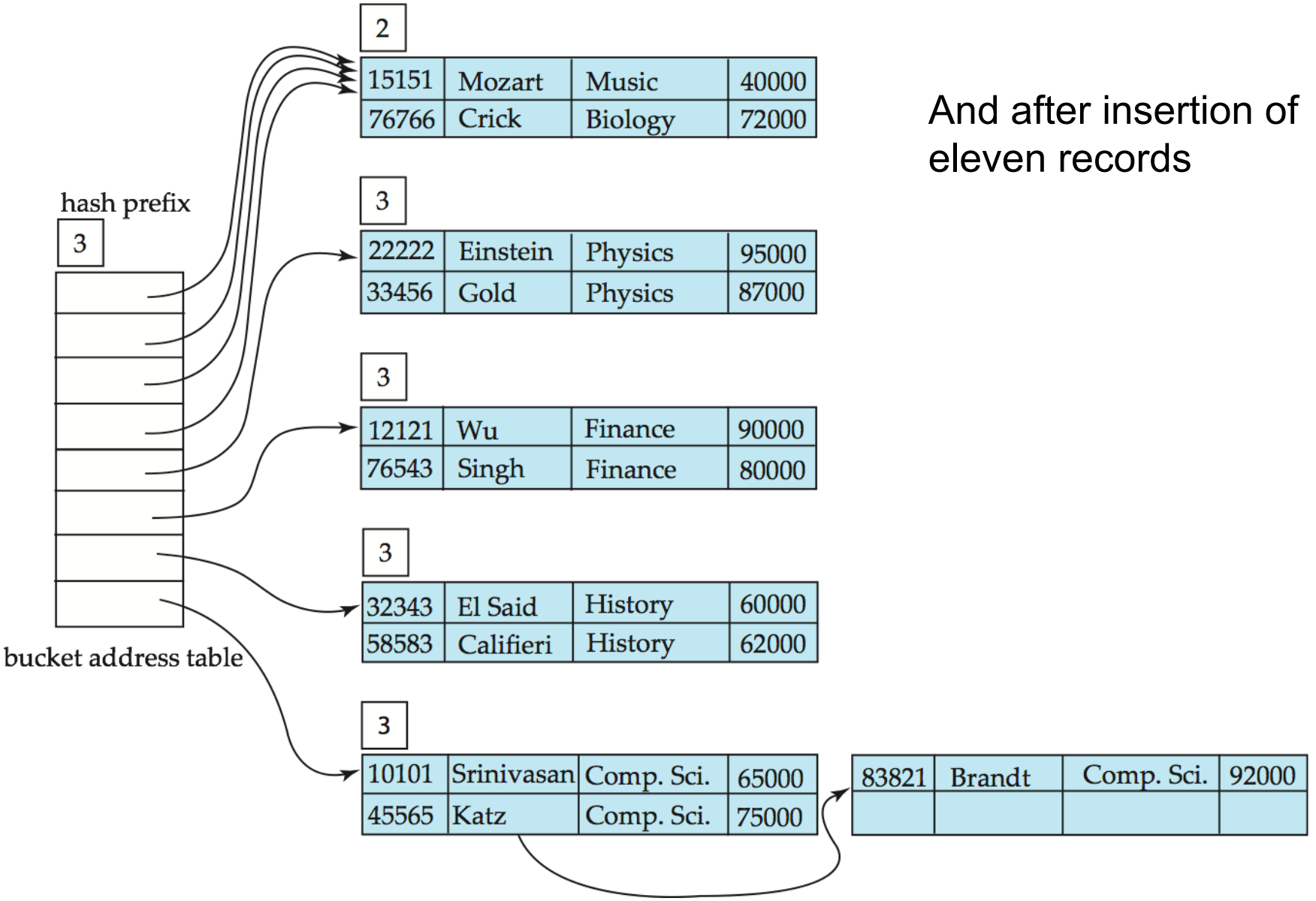


Example (Cont.)

Hash structure after insertion of Katz record

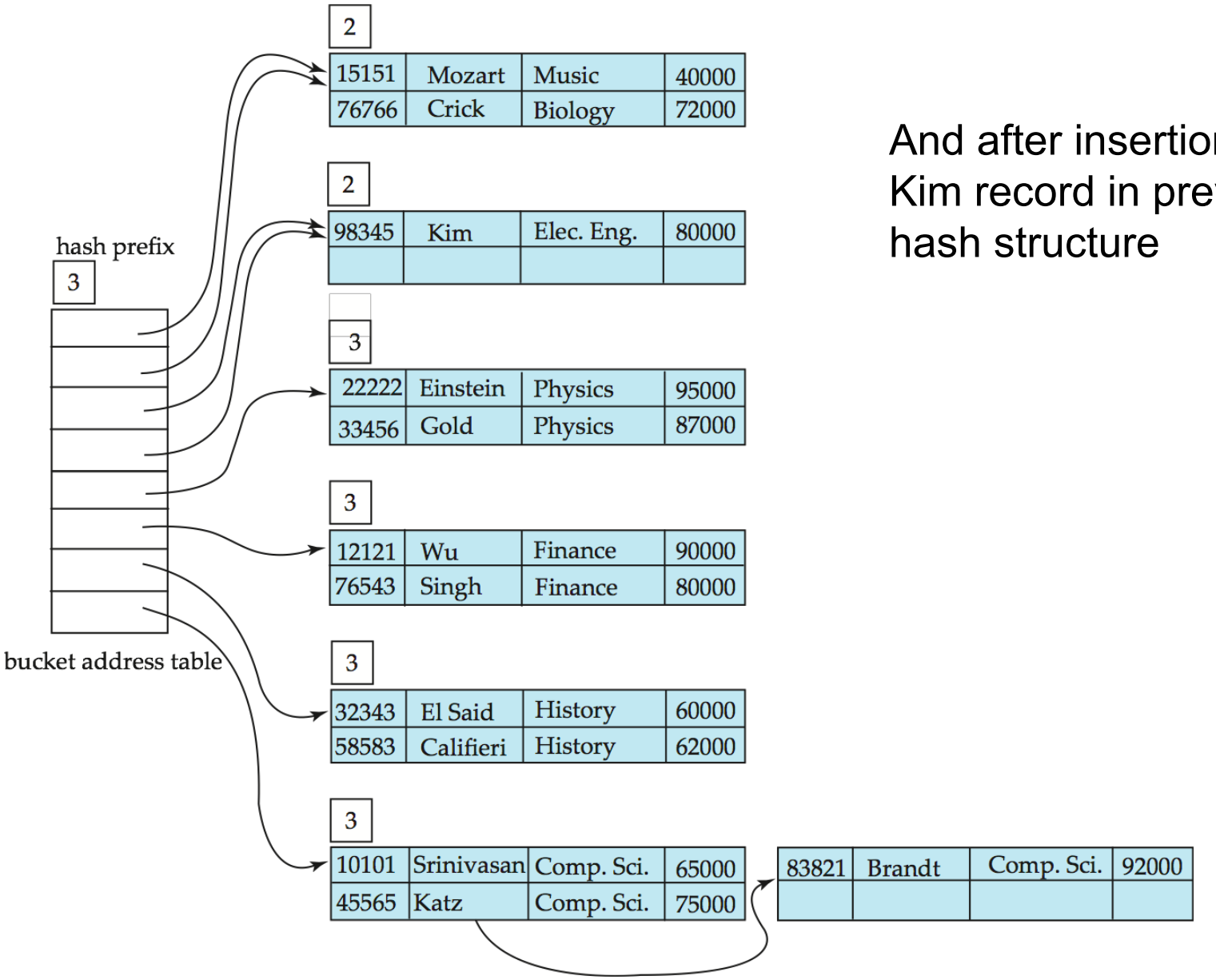


Example (Cont.)



And after insertion of eleven records

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
  - Solution: B^+-tree structure to locate desired record in bucket address table
- Changing size of bucket address table is an expensive operation

Linear hashing is an alternative mechanism

- Allows incremental growth of its directory (equivalent to bucket address table)
- At the cost of more bucket overflows

Comparison of Ordered Indexing and 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

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

Bitmap Indices

Bitmap indices are a special type of index designed for efficient querying on multiple keys

Records in a relation are assumed to be numbered sequentially from, say, 0

Given a number *n* it must be easy to retrieve record *n*

- Particularly easy if records are of fixed size

Applicable on attributes that take on a relatively small number of distinct values

E.g. gender, country, state, …

E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)

A bitmap is simply an array of bits

Bitmap Indices (Cont.)

In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute

Bitmap has as many bits as records

In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	*ID*	*gender*	*income_level*
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

Bitmap indices are useful for queries on multiple attributes

not particularly useful for single attribute queries

Queries are answered using bitmap operations

Intersection (and)

Union (or)

Complementation (not)

Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap

E.g. 100110 AND 110011 = 100010

100110 OR 110011 = 110111

NOT 100110 = 011001

Males with income level L1: 10010 AND 10100 = 10000

- Can then retrieve required tuples.
- Counting number of matching tuples is even faster

Bitmap Indices (Cont.)

Bitmap indices generally very small compared with relation size

E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.

- If number of distinct attribute values is 8, bitmap is only 1% of relation size

Deletion needs to be handled properly

Existence bitmap to note if there is a valid record at a record location

Needed for complementation

- not(*A*=*v*): *(NOT bitmap-A-v) AND ExistenceBitmap*

Should keep bitmaps for all values, even null value

To correctly handle SQL null semantics for NOT(*A*=*v*):

- intersect above result with (NOT *bitmap-A-Null*)

Efficient Implementation of Bitmap Operations

Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once

E.g. 1-million-bit maps can be and-ed with just 31,250 instruction

Counting number of 1s can be done fast by a trick:

Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation

- Can use pairs of bytes to speed up further at a higher memory cost

Add up the retrieved counts

Bitmaps can be used instead of Tuple-ID lists at leaf levels of B^+-trees, for values that have a large number of matching records

Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits

Above technique merges benefits of bitmap and B^+-tree indices

Index Definition in SQL

Create an index

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

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

Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.

Not really required if SQL **unique** integrity constraint is supported

To drop an index

drop index <index-name>

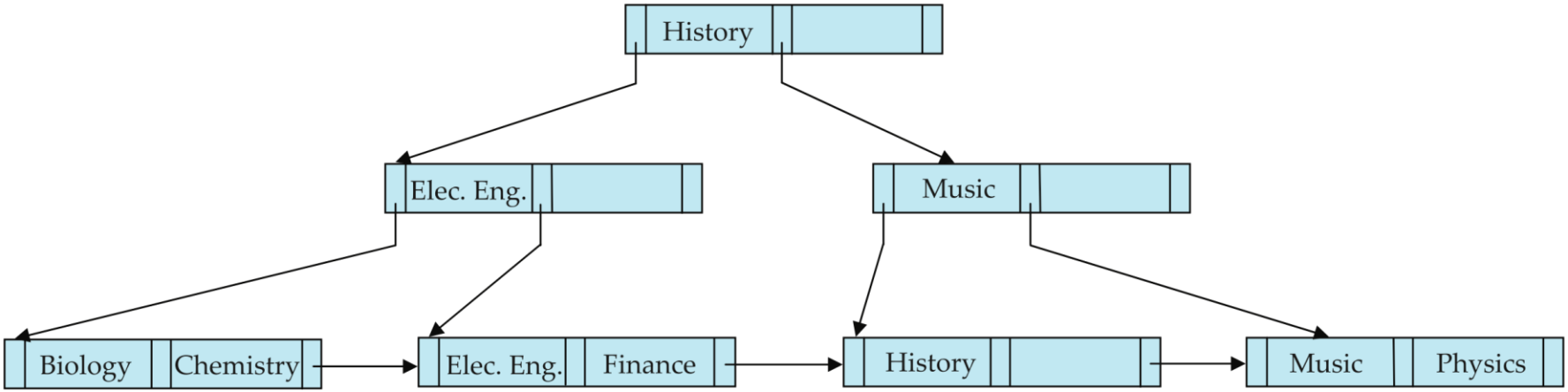
Most database systems allow specification of type of index, and clustering.

End of Chapter

Figure 11.01

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	

Figure 11.15


History
Elec. Eng.
Music
Biology
Chemistry
Elec. Eng.
Finance
History
Music
Physics

Partitioned Hashing

Hash values are split into segments that depend on each attribute of the search-key.

(A_1, A_2, . . . , A_n) for n attribute search-key

Example: n = 2, for *customer,* search-key being (*customer-street, customer-city*)

search-key value	*hash value*
(Main, Harrison)	101 111
(Main, Brooklyn)	101 001
(Park, Palo Alto)	010 010
(Spring, Brooklyn)	001 001
(Alma, Palo Alto)	110 010

To answer equality query on single attribute, need to look up multiple buckets. Similar in effect to grid files.

Grid Files

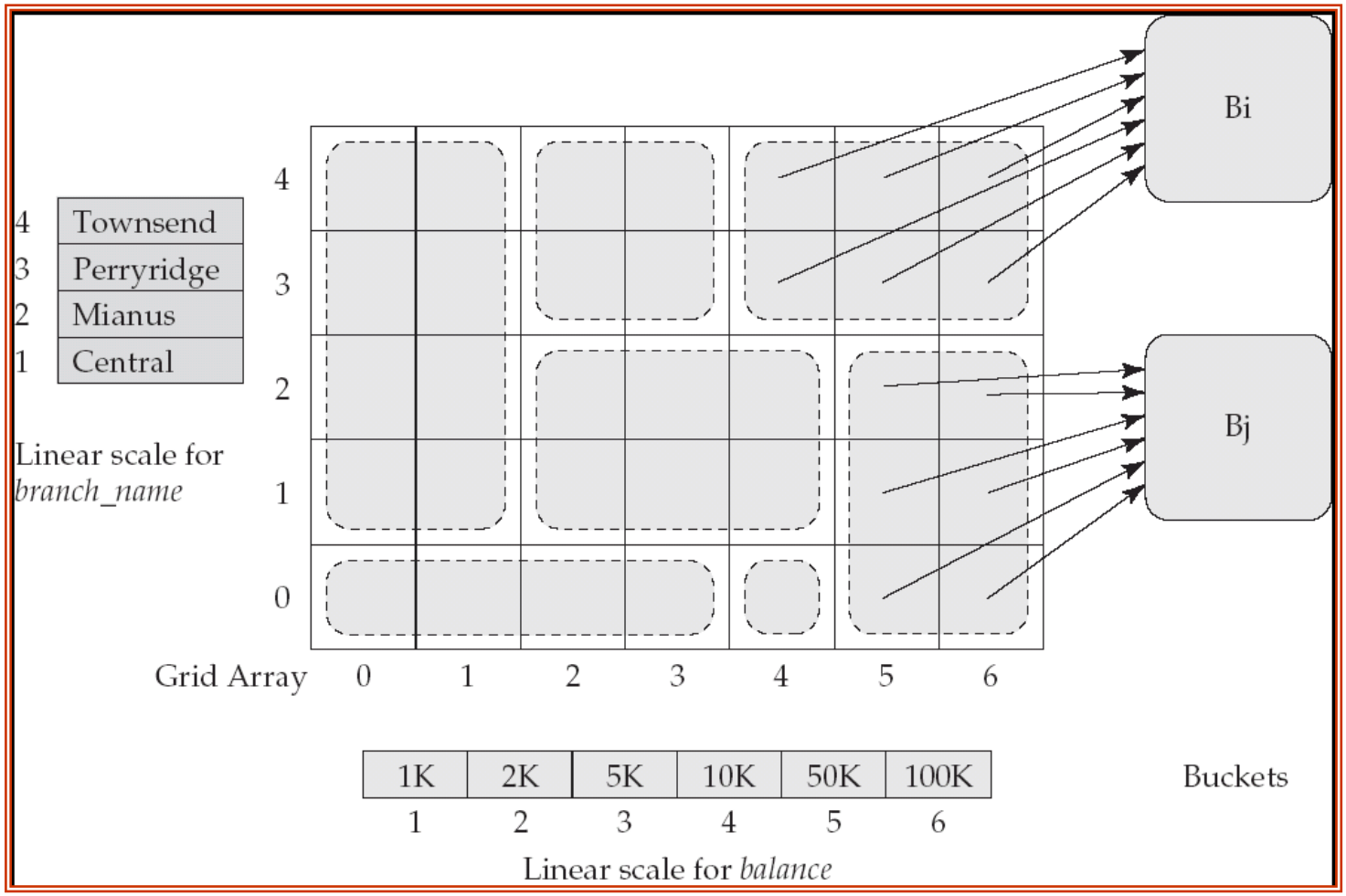
Structure used to speed the processing of general multiple search-key queries involving one or more comparison operators.

The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes.

Multiple cells of grid array can point to same bucket

To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer

Example Grid File for *account*



Queries on a Grid File

A grid file on two attributes A and B can handle queries of all following forms with reasonable efficiency

$(a_1 \leq A \leq a_2)$

$(b_1 \leq B \leq b_2)$

$(a_1 \leq A \leq a_2 \ \wedge \ b_1 \leq B \leq b_2)$,.

E.g., to answer $(a_1 \leq A \leq a_2 \ \wedge \ b_1 \leq B \leq b_2)$, use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells.

Grid Files (Cont.)

During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it.

- Idea similar to extendable hashing, but on multiple dimensions
- If only one cell points to it, either an overflow bucket must be created or the grid size must be increased

Linear scales must be chosen to uniformly distribute records across cells.

- Otherwise there will be too many overflow buckets.

Periodic re-organization to increase grid size will help.

- But reorganization can be very expensive.

Space overhead of grid array can be high.

R-trees (Chapter 23) are an alternative