Chapter 11: Indexing and Hashing

Database System Concepts, 6th Ed.

Chapter 12: 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 Concept

- Many queries reference only a small proportion of the records in a file.
- For example,
  - "Find all instructors in the Physics department" or
  - "Find the total number of credits earned by the student with *ID 22201"*
- It is inefficient for the system to read every tuple in the *instructor* relation to check if the dept name value is "Physics".
- Likewise, it is inefficient to read the entire *student* relation just to find the one tuple for the *ID "22201".*
- Ideally, the system should be able to locate these records directly. To allow these forms of access, we design additional structures (index) that we associate with files.
- The index structure is much smaller than the original relation.

Basic Concept

- An index for a file in a database system works in much the same way as the index in the textbook.
- Database-system indices play the same role as book indices in libraries.
- For example, to retrieve a *student* record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate *student* record.
- There are two basic kinds of indices:
  - **Ordered indices.** Based on a sorted ordering of the values.
  - **Hash indices.** Based on a uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function, called a ***hash function***.
- Several techniques exist for both ordered indexing and hashing. No one technique is the best. Rather, each technique is best suited to particular database applications.

Index Evaluation Metrics

- Each technique must be evaluated on the basis of these factors:
- **Access types:** The types of access that are supported efficiently. Access types can include finding records with a specified attribute value (point query) and finding records whose attribute values fall in a specified range (range query).
- **Access time:** The time it takes to find a particular data item, or set of items, using the technique in question.
- **Insertion time:** The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
- **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
- **Space overhead:** The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

Basic Concepts

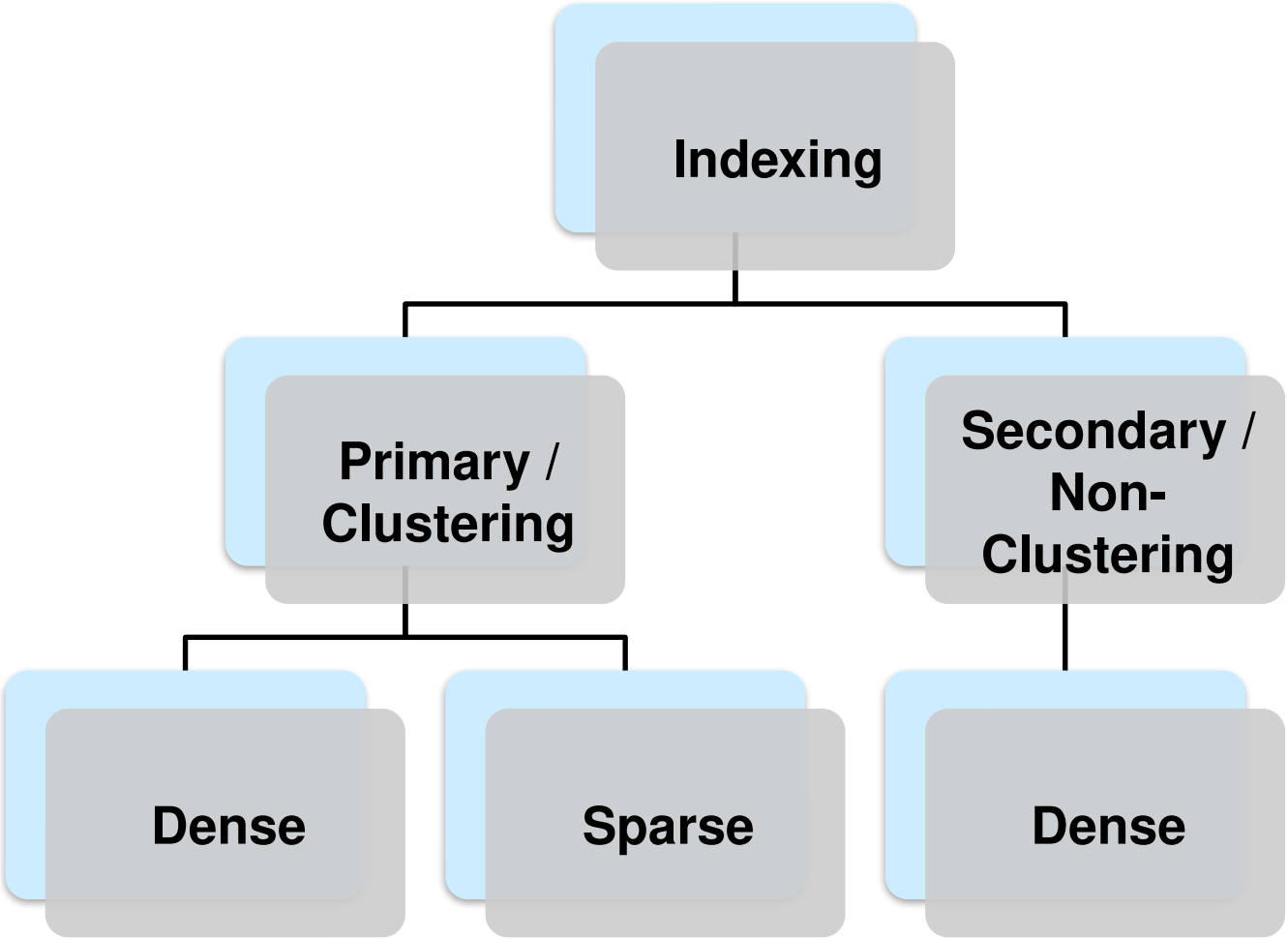
- Indexing mechanism is used to speed up access to desired data.
  - E.g., author catalog in library
- We often want to have more than one index for a file.
  - E.g., we may wish to search for a book by author, by subject, or by title.
- **Search Key** – An attribute or set of attributes used to look up records in a file.
- This definition of *key* differs from that used in *primary key*, *candidate key*, and *superkey*.
- Using this notion of a search key, if there are several indices on a file, there are several search keys.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

Ordered Indices

- To gain fast random access to records in a file, an index structure is used.
- Each index structure is associated with a particular search key.
- The records in the indexed file may themselves be stored in some sorted order.
- A file may have several indices, on different search keys.
- In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index:** If the file containing the records is sequentially ordered, a **Primary index** is an index whose search key also defines the sequential order of the file.
  - Also called **clustering index**
  - The search key of a clustering index is often the primary key, although that is **NOT** necessarily so.
- **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.

Classification of Indexing

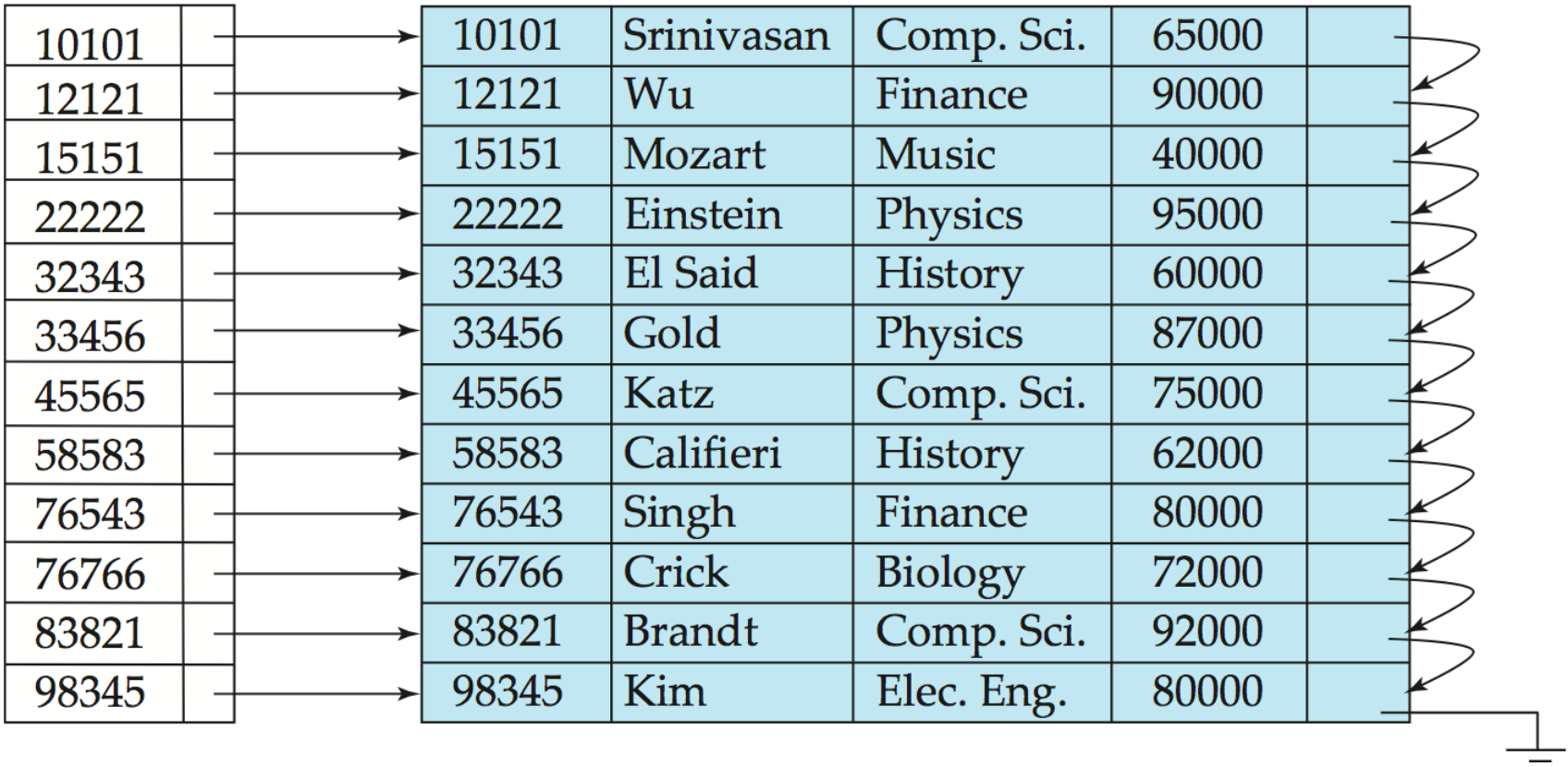


Dense and Sparse Indices

- An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value.
- The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.
- There are two types of ordered indices: 1. Dense Index, 2. Sparse Index
- **Dense index***:* In a dense index, an index entry appears for every search-key value in the file.
- In a dense clustering index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record, since, because the index is a clustering one, records are sorted on the same search key.
- In a dense non-clustering index, the index must store a list of pointers to all records with the same search-key value.

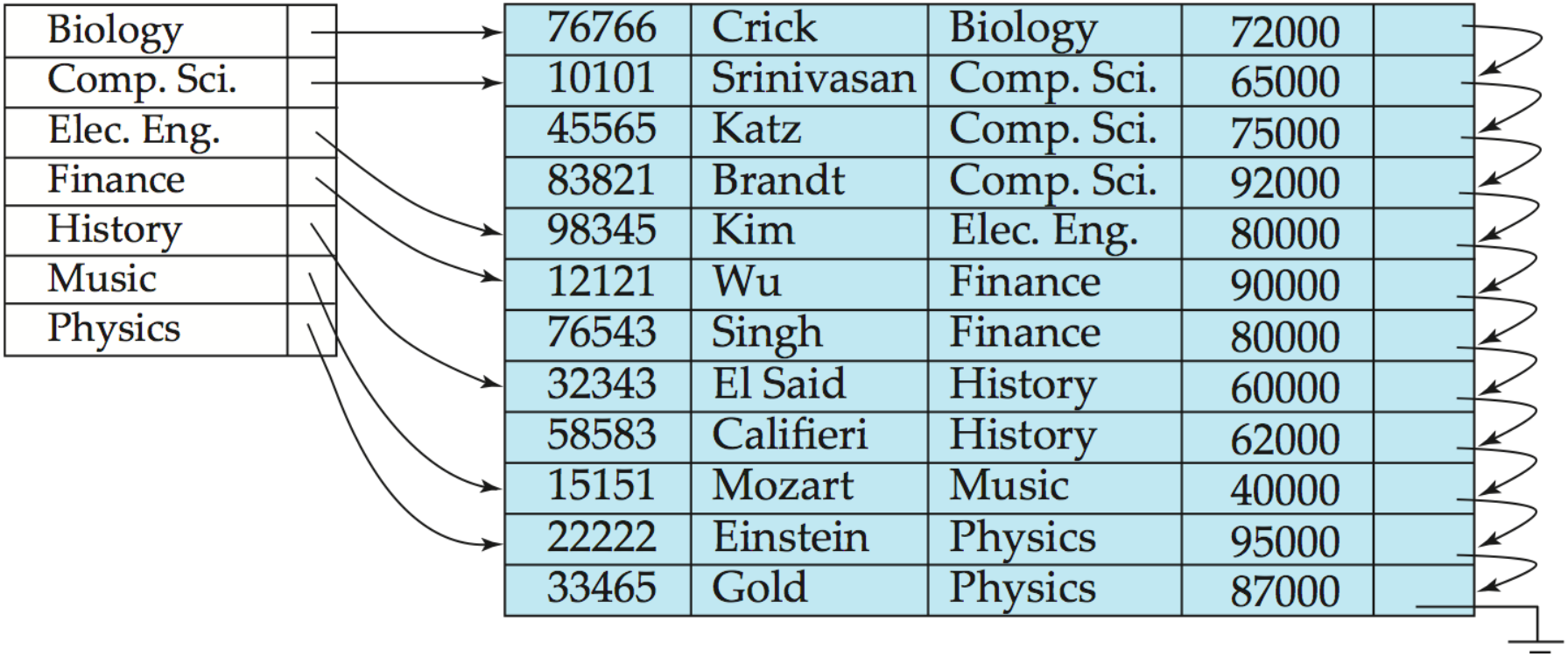
Dense Index Files

- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



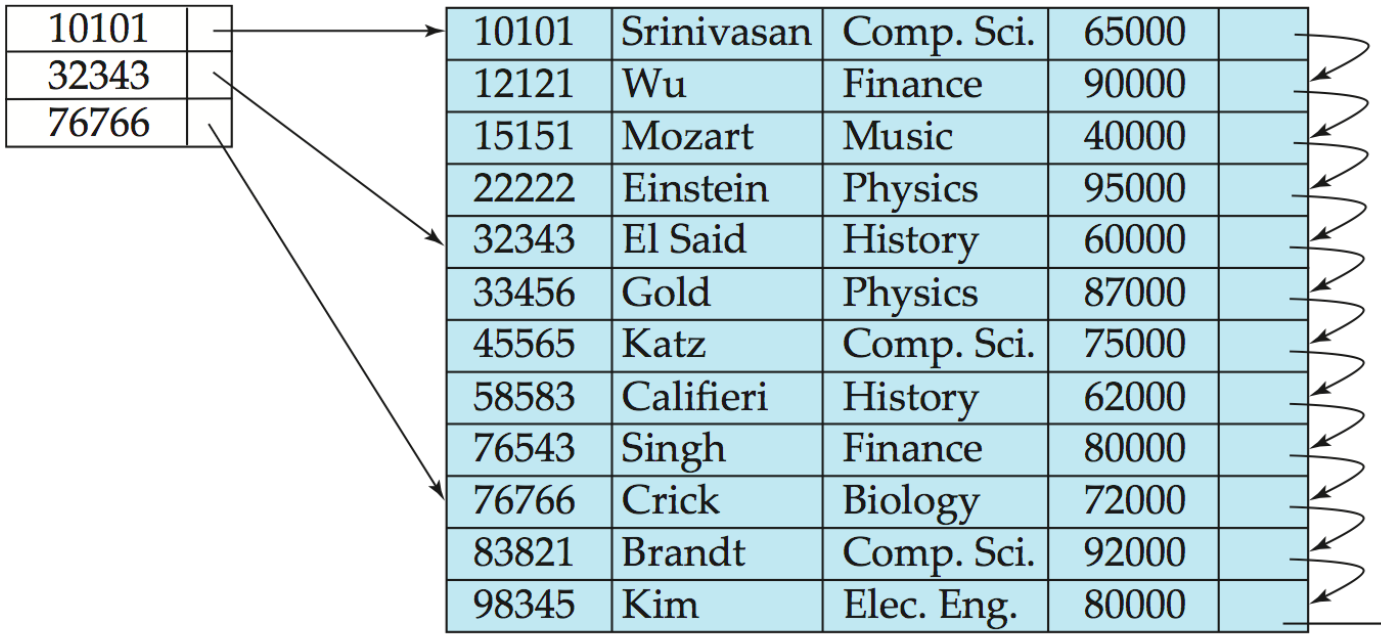
Dense Index Files (Cont.)

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



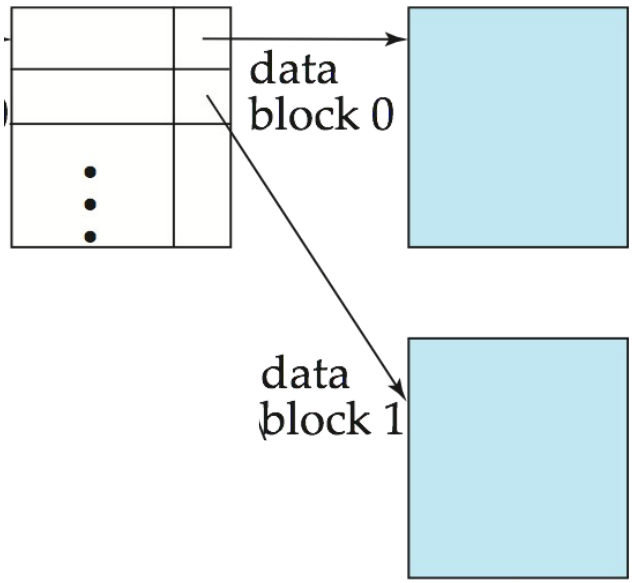
Sparse Index Files

- **Sparse Index**: contains index records for only some of the search-key values. Sparse indices can be used only if the index is a clustering index. Like dense index, each index entry contains a search-key value and a pointer to the first data record with that search-key value
- 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



Dense Vs Sparse Indices

- 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.
- The reason this design is a good trade-off is that the dominant cost in processing a database request is the time that it takes to bring a block from disk into main memory. Once the block has been brought in, the time to scan the entire block is negligible.
- Using this sparse index, we locate the block containing the record that we are seeking. Thus, unless the record is on an overflow block, we minimize block accesses while keeping the size of the index (and thus our space overhead) as small as possible.



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

Secondary Indices

- Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file.
- A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file.
- If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.
- A secondary index on a candidate key looks just like a dense clustering index, except that the records pointed to by successive values in the index are not stored sequentially.
- In general, however, secondary indices may have a different structure from clustering indices.

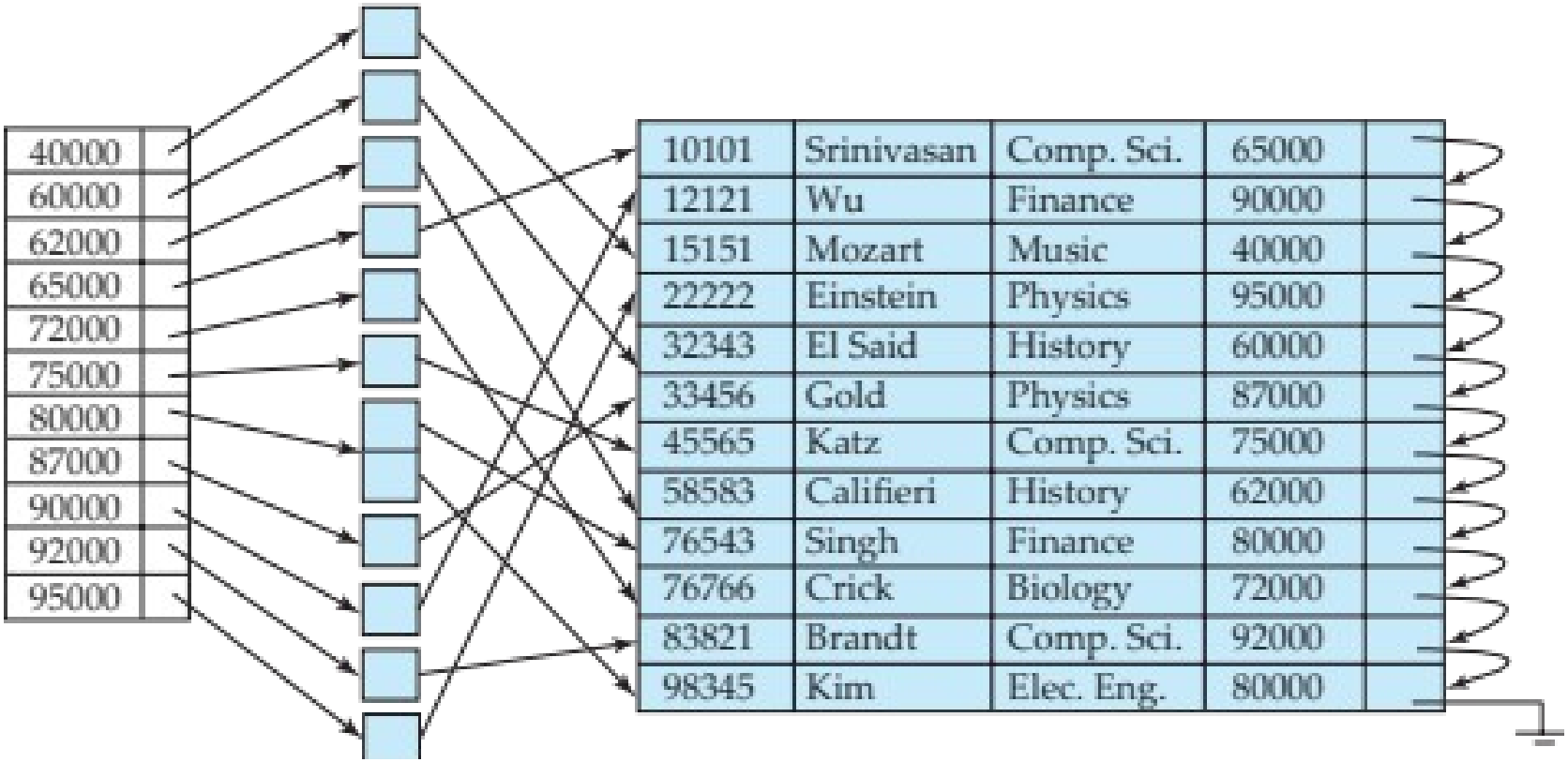
Secondary Indices are always Dense

- If the search key of a clustering index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file.
- In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value.
- The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.
- We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file.

Secondary Indices

- A sequential scan in clustering index order is efficient because records in the file are stored physically(?) in the same order as the index order.
- Because secondary-key order and physical-key order differ, if we attempt to scan the file sequentially in secondary-key order, the reading of each record is likely to require the reading of a new block from disk, which is very slow.
- Secondary indices improve the performance of queries that use keys other than the search key of the clustering index. However, they impose a significant overhead on modification of the database.
- The designer of a database decides which secondary indices are desirable on the basis of an estimate of the relative frequency of queries and modifications.

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, Why?

- Suppose a dense index is built on a relation with 1,000,000 tuples. Index entries are smaller than data records, so let us assume that 100 index entries fit on a 4 kilobyte block. Thus, our index occupies 10,000 blocks. If the relation instead had 100,000,000 tuples, the index would instead occupy 1,000,000 blocks, or 4 gigabytes of space. Such large indices are stored as sequential files on disk.
- If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required.
- Again main memory is also required for a number of other tasks. The search for an entry in the index then requires several disk-block reads.

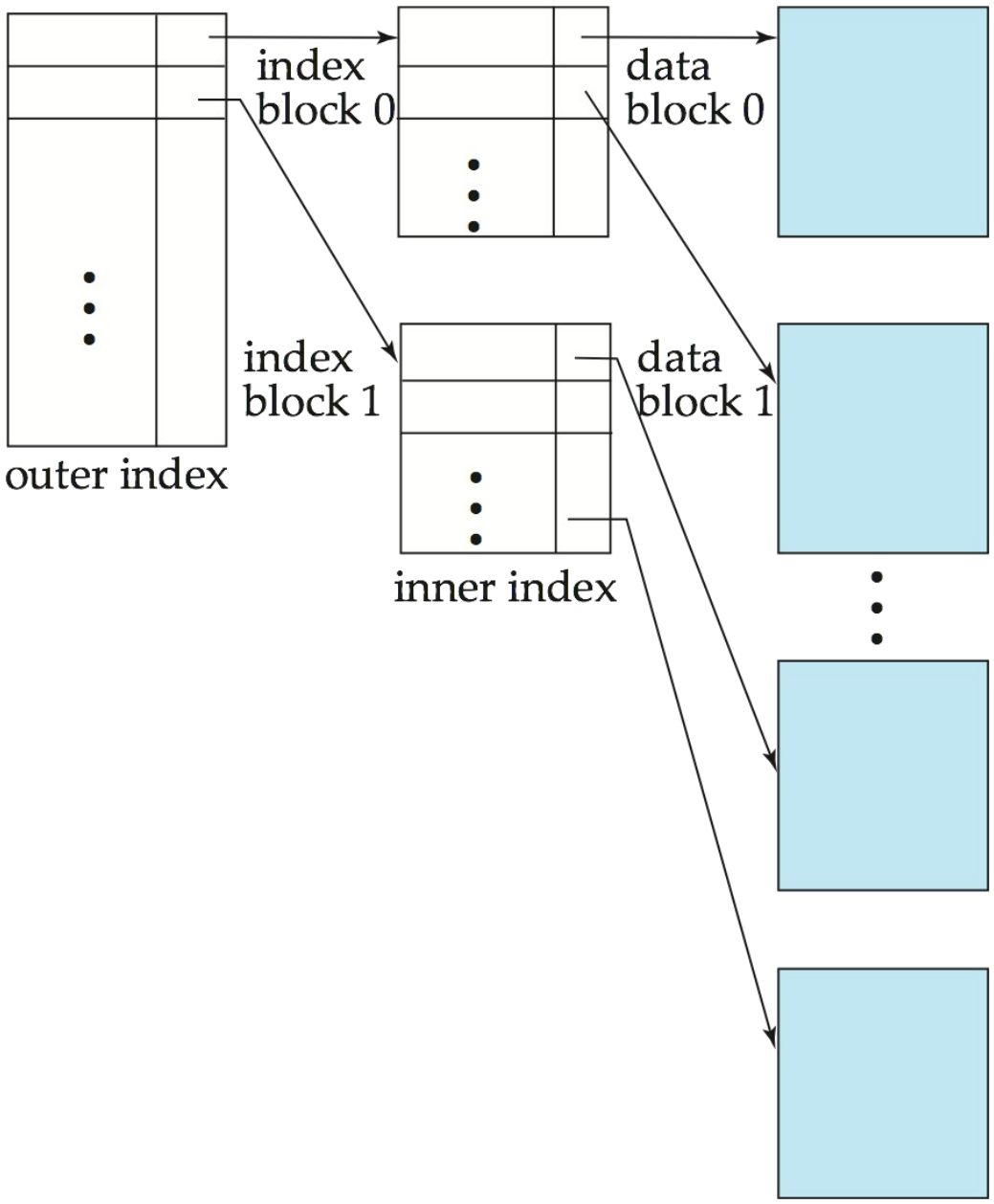
Multilevel Index, Why?

- Binary search can be used on the index file to locate an entry, but the search still has a large cost. If the index would occupy b blocks, binary search requires as many as $\log_2(b)$ blocks to be read.
- For a 10,000-block index, binary search requires 14 block reads.
- On a disk system where a block read takes on average 10 milliseconds, the index search will take 140 milliseconds. This may not seem much, but we would be able to carry out only seven (7) index searches a second, whereas a more efficient search mechanism would let us carry out far more searches per second.
- But, if overflow blocks have been used, binary search is only possible on non-overflow blocks and actual cost may be higher.
- A sequential search requires b sequential block reads, which may take even longer.
- Thus, the process of searching a large index may be costly.

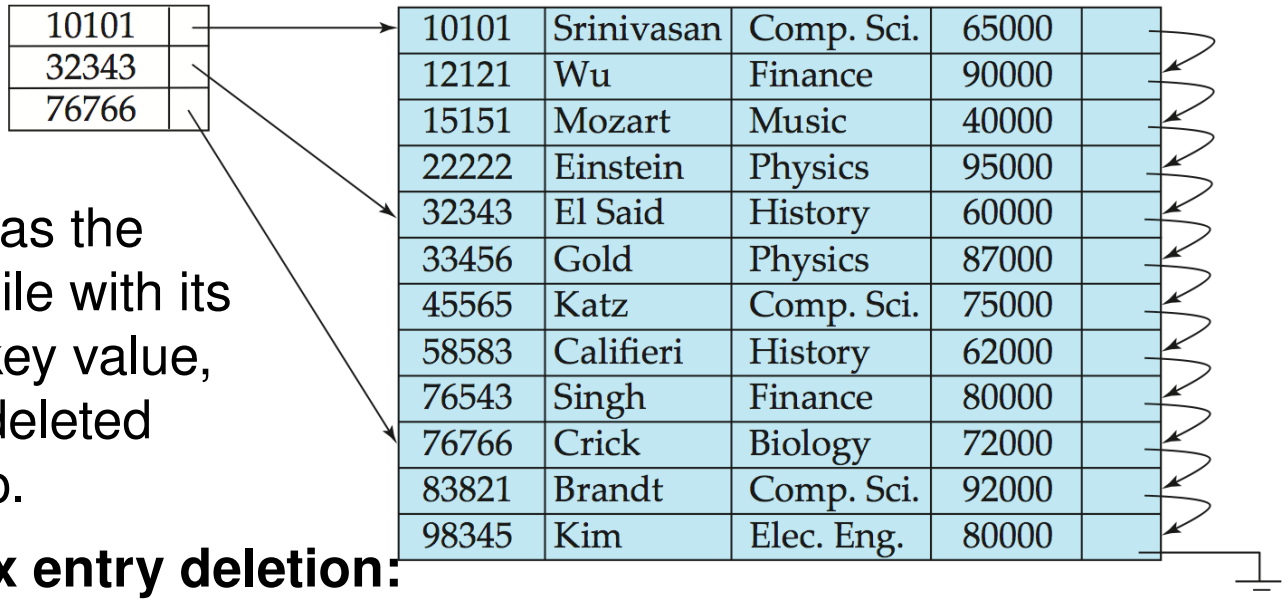
Multilevel Index

- 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
- In earlier example, an inner index with 10,000 blocks would require 10,000 entries in the outer index, which would occupy just 100 blocks. If we assume that the outer index is already in main memory, we would read only one index block for a search using a multilevel index, rather than the 14 blocks we read with binary search.
- Thus, we can perform 14 times as many index searches per second
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on. Indices with two or more levels are called **multilevel indices.** Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search.
- Indices at all levels must be updated on insertion or deletion from the file

Multilevel Index (Cont.)



Index Update: Deletion (Book)



- 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 (Book)

- **Single-level index insertion:**
  - Perform a lookup using the search-key value that appears 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

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 is 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** *salary* = 80000
  - May fetch many records that satisfy the first but not the second condition

Images of Indices

A composite index contains more than one column in a specific order. In this case, Last Name comes first.

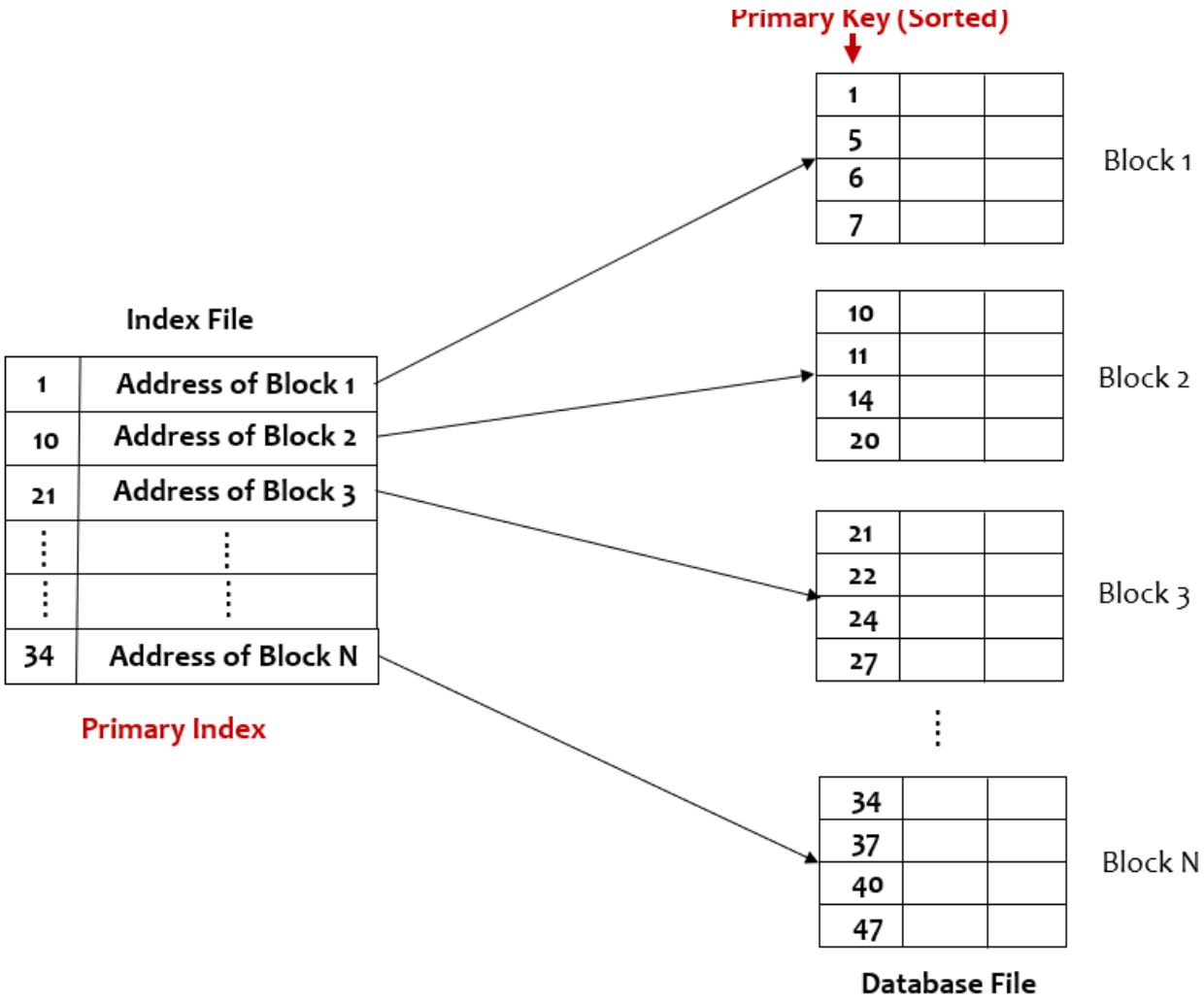
Composite Index

Allan, Ethen
Allan, John
Cooper, Stephen
Cooper, Thomas
Faust, Liz
Greenburg, Dale
Greenburg, Mike
Greenburg, Simon
-
Wu, Ellen

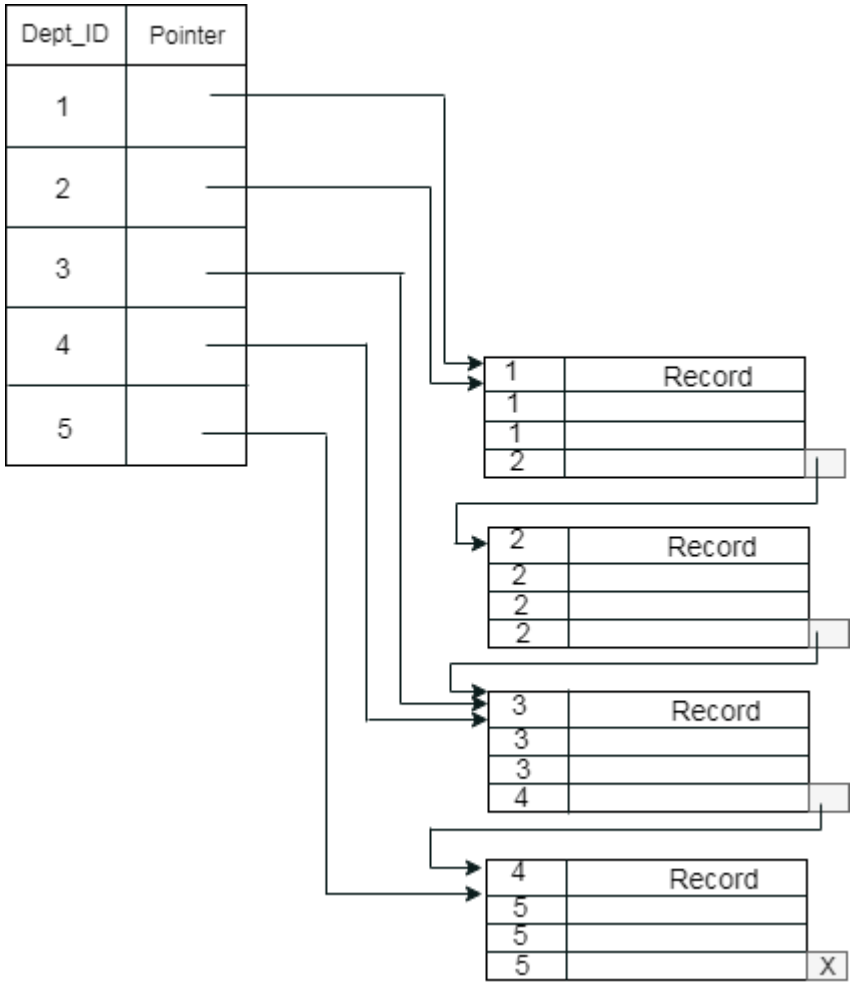
Table

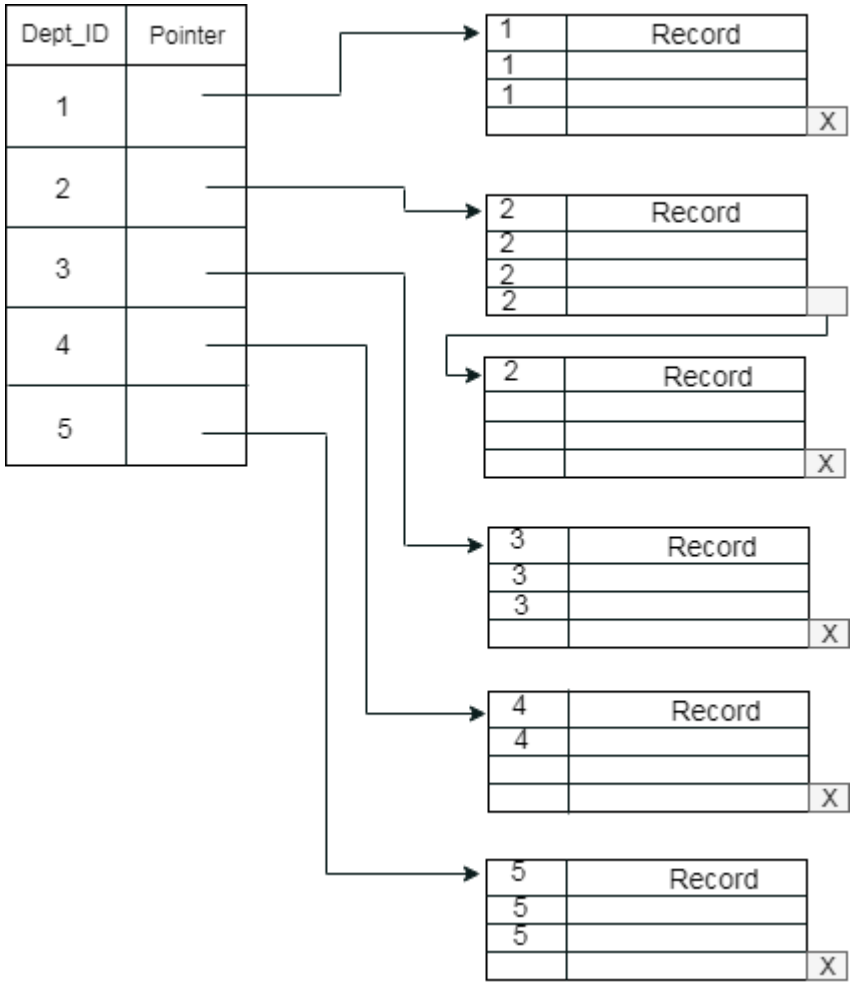
1	Dale	Greenburg
2	Ellen	Wu
3	Ethen	Allan
4	John	Allan
5	Liz	Faust
6	Mike	Greenburg
7	Simon	Greenburg
8	Stephen	Cooper
-	Thomas	Cooper
26	-	-

Images of Indices



Images of Indices





Images of Indices

```
SELECT * FROM
SCOUNTER WHERE
AIRPORT = 'LCY'.
```

Binary search

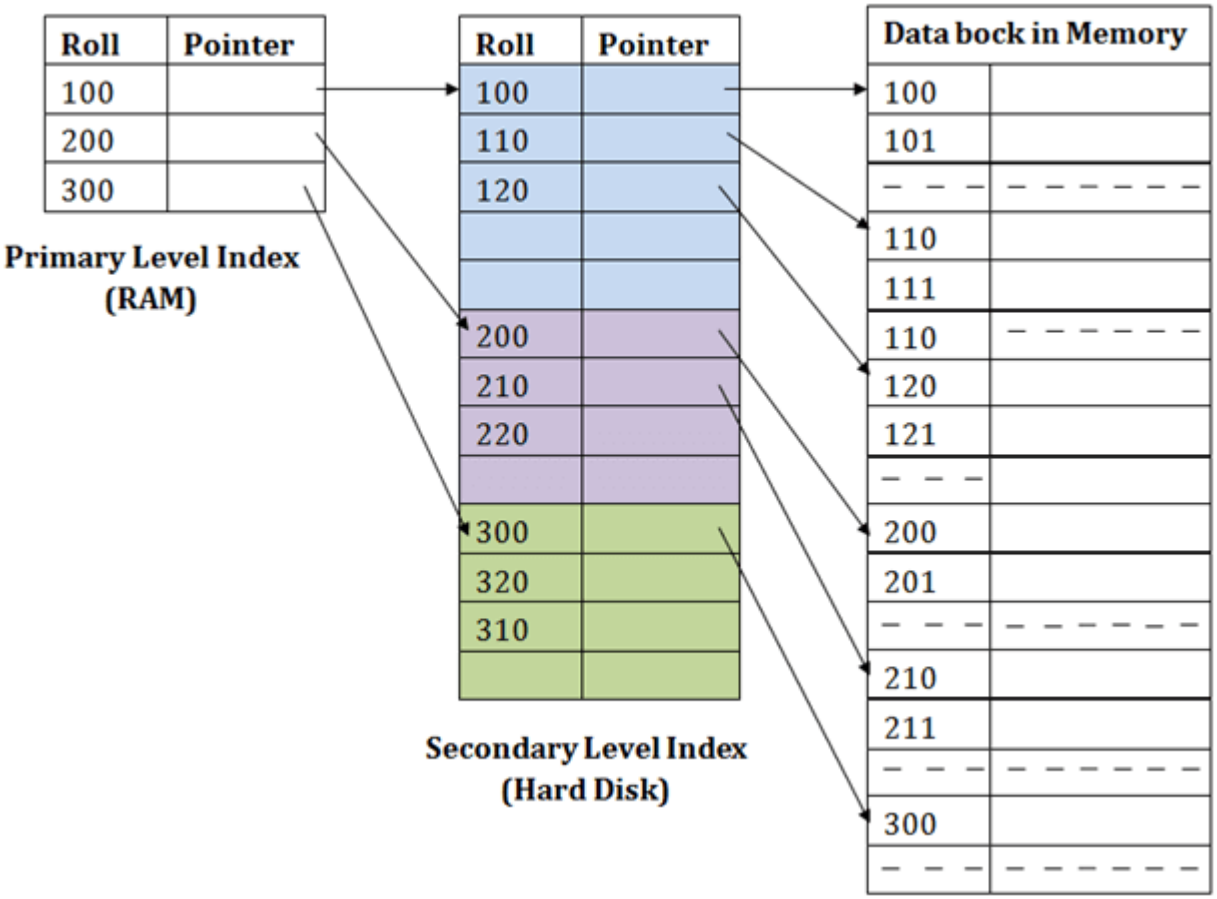
AIRPORT	P
ACA	1
ACE	2
BER	3
BER	6
DEN	7
FRA	8
HAM	14
LCY	4
LCY	9
LGW	10
LHR	5
LHR	11
MUC	12
RTM	13

Secondary index on AIRPORT

Table SCOUNTER

MANDT	CARRID	COUNTNUM	AIRPORT
001	LH	00000005	ACA
001	BA	00000004	ACE
001	UA	00000001	BER
001	LH	00000002	LCY
001	BA	00000003	LHR
001	LH	00000007	BER
001	AA	00000001	DEN
001	LH	00000003	FRA
001	BA	00000001	LCY
001	LH	00000001	LGW
001	LH	00000004	LHR
001	BA	00000002	MUC
001	LH	00000006	RTM
001	LH	00000008	HAM

Images of Indices



B+-Tree Index Files

B+-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, both for index lookups and for sequential scans through the data, since many overflow blocks get created.
  - Periodic reorganization of entire file is required, but frequent reorganizations are undesirable
- 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.

B+-Tree Index Files

- (Minor) disadvantage of B+-trees:
  - imposes performance overhead on insertion and deletion, and adds space overhead.
  - The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided.
  - Furthermore, since nodes may be as many as half empty (if they have the minimum number of children), there is some wasted space.
  - This space overhead, too, is acceptable given performance benefits of the B+-tree structure.
- Advantages of B+-trees outweigh disadvantages
  - B+-trees are used extensively

Example of B+-Tree

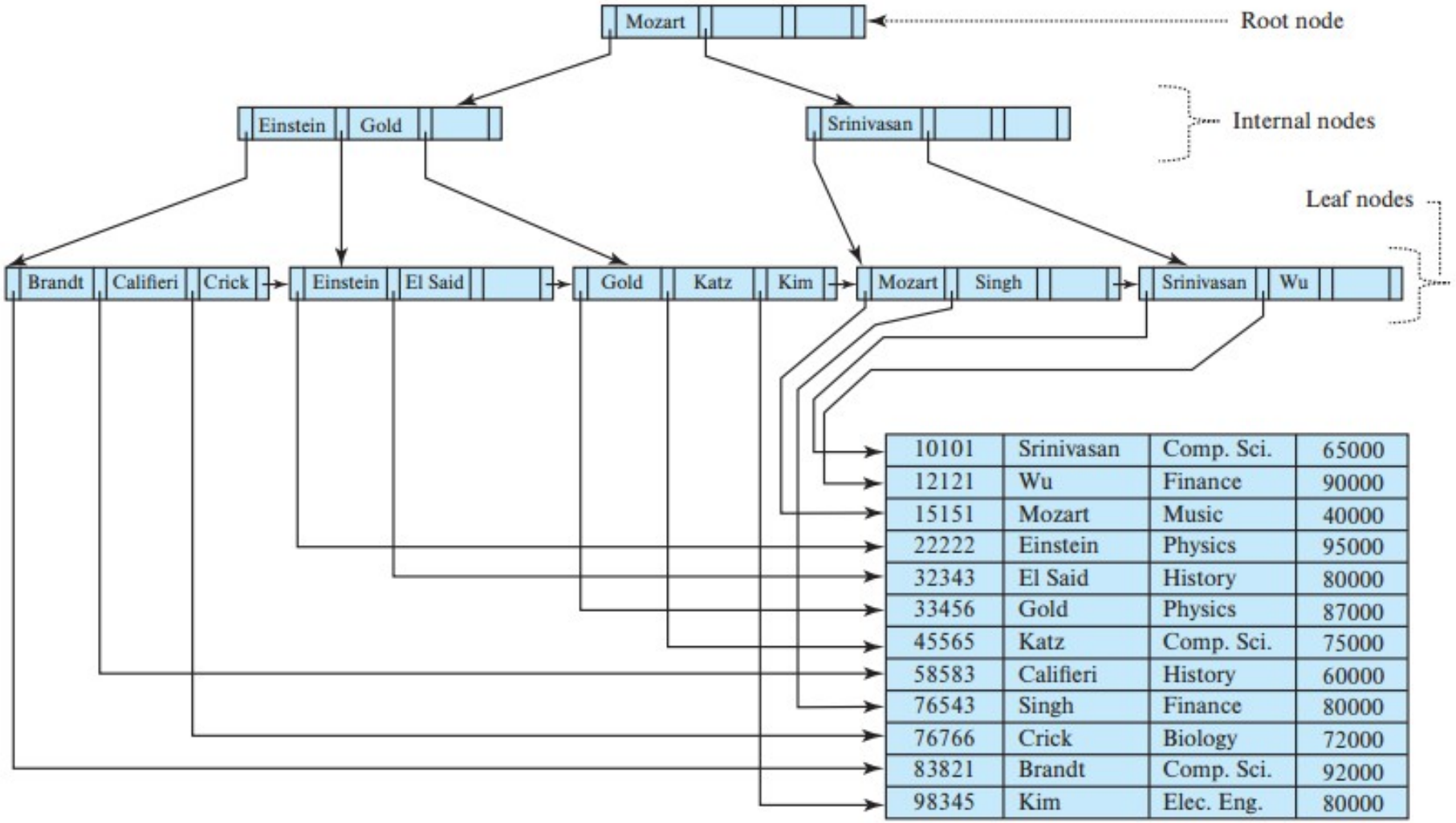


Figure 14.9 B+-tree for *instructor* file ($n = 4$).

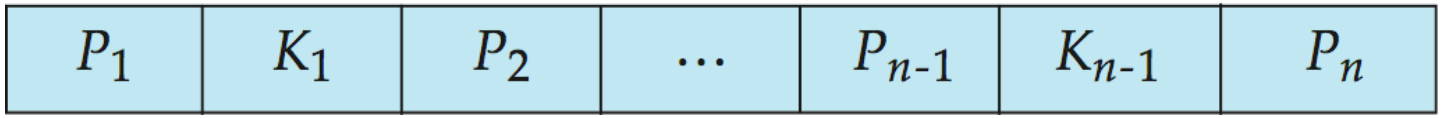
B+-Tree Index Files (Cont.)

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

- A B+-tree index takes the form of a **balanced tree** in which every path from the root to a leaf of the tree is of the same length.
- Each node that is not a root or a leaf has between $n/2$ and n children, where n (no. of Pointers) is fixed for a particular tree.
- A leaf node has between $(n–1)/2$ and $n–1$ values. With $n = 4$, B+-tree, each leaf must contain at least 2 values, and at most 3 values.
- The ranges of values in each leaf do not overlap, except if there are duplicate search-key values, in which case a value may be present in more than one leaf.
- 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

- A B+-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file.
- Typical node



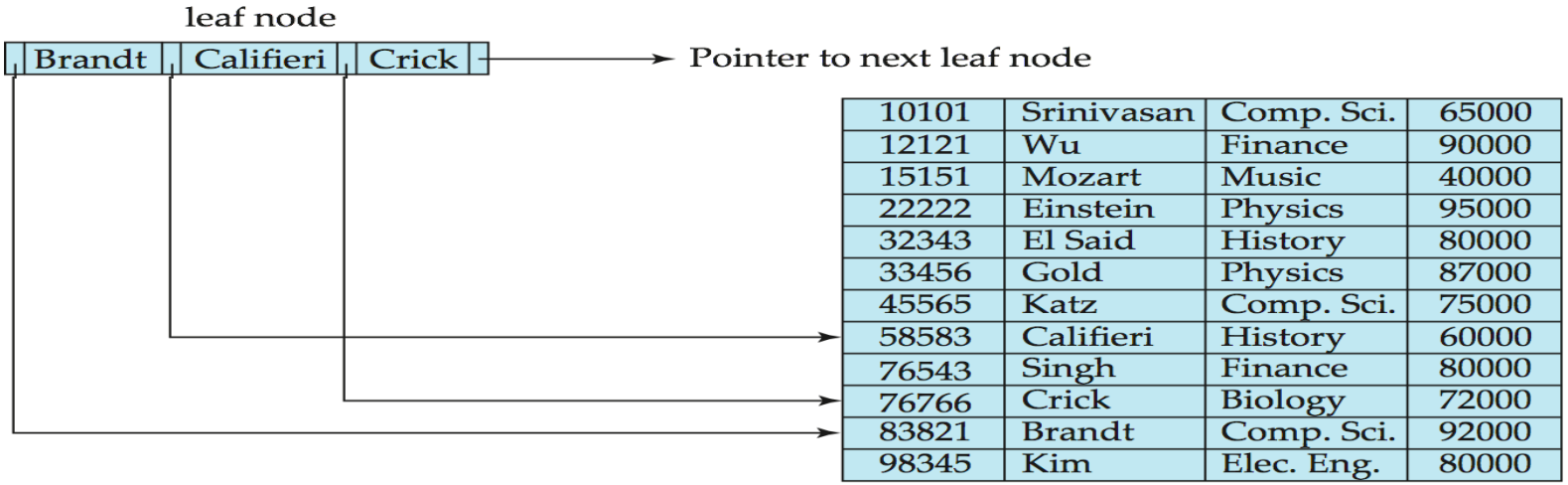
  - It contains up to *n – 1* search-key values and *n* pointers
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

Leaf Nodes 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
- Usually used as a dense index where every search-key value must appear in some leaf node.
- Since there is a linear order on the leaves based on the search-key values, P_n is used to **chain together** the leaf nodes in search-key order. This ordering allows for efficient sequential processing.



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

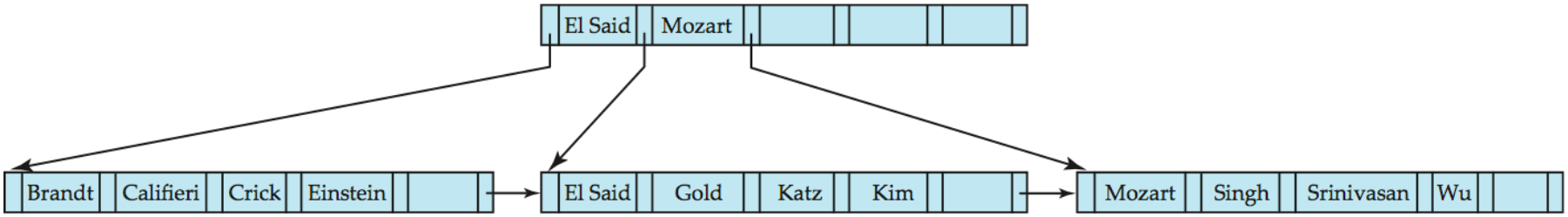
Non-Leaf Nodes in B+-Trees

- Nonleaf nodes form a multi-level sparse index on the leaf nodes. Nonleaf nodes are also referred as **internal nodes**.
- The structure is the same as that for leaf nodes, except that all pointers are pointers to tree nodes.
- A non-leaf node may hold up to n pointers, and must hold at least $(n/2$ pointers. The number of pointers in a node is called the **fanout** of the node
- For a non-leaf node with n pointers:

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

  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For 2 i $n-1$, 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

Example of B+-tree



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

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

Example of B+-tree

Node Type	Children Type	Min Number of Children	Max Number of Children	Example $b = 7$	Example $b = 100$
Root Node (when it is the only node in the tree)	Records	0	$b - 1$	0–6	1–99
Root Node	Internal Nodes or Leaf Nodes	2	b	2–7	2–100
Internal Node	Internal Nodes or Leaf Nodes	$\lceil b/2 \rceil$	b	4–7	50–100
Leaf Node	Records	$\lceil b/2 \rceil$	b	4–7	50–100

b = no. of pointers

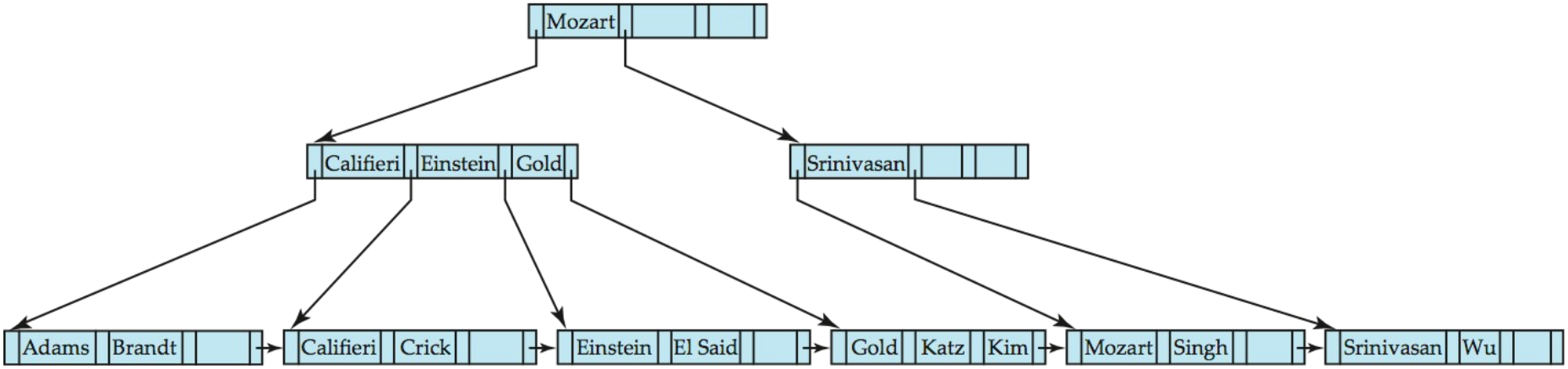
Observations about B+-trees

- 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* n/2 values
  - Next level has at least 2* n/2 * n/2 values
  - .. etc.
  - If there are *K* search-key values in the file, the tree height is no more than $\log_{n/2}(K)$
  - 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. $V \quad K_i$.
     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 $K_1 \quad K_2 \quad K_3 \quad \ldots \quad K_{n-1}$
  - Search-keys in the subtree to which P_i points
    - are $\quad K_{i,}$, but not necessarily $< K_{i,}$
    - To see why, suppose same search key value V is present in two leaf node L_i and L_{i+1}. Then in parent node K_i must be equal to V

Queries on B+-Trees (Cont.)

- In processing a query, we traverse a path in the tree from the root to some leaf node. If there are K search-key values in the file, the height of the tree is no more than $\log_{n/2}(K)$.
- In practice, only a few nodes need to be accessed. A node is generally the same size as a disk block, typically 4 kilobytes.
- With a search-key size of 12 bytes, and a disk-pointer size of 8 bytes, n is around 200. Even with a more conservative estimate of 32 bytes for the search-key size, n is around 100.
  - 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.
- The root node of the tree is usually heavily accessed and is likely to be in the buffer, so typically only three or fewer blocks need to be read from disk.

Diff. Bet B+-Tree and in-memory Tree

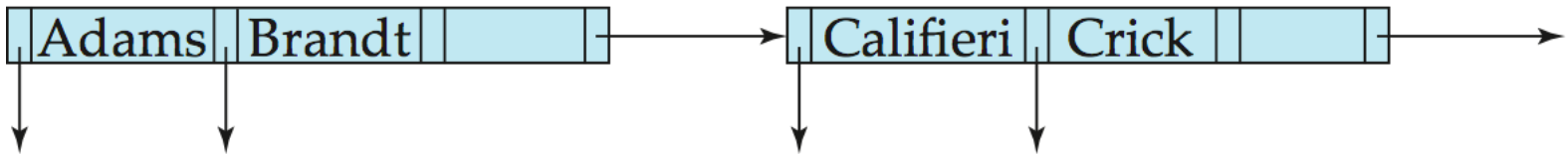
- An important difference between B+-tree structures and in-memory tree structures, such as binary trees, is the size of a node, and as a result, the height of the tree.
- In a binary tree, each node is small, and has at most two pointers. In a B+-tree, each node is large—typically a disk block—and a node can have a large number of pointers. Thus, B+-trees tend to be **fat and short**, unlike **thin and tall** binary trees.
- In a balanced binary tree, the path for a lookup can be of length $\log_2(K)$ where N is the number of records in the file being indexed.
- With 1 million search key values — around 20 nodes are accessed in a lookup, if each node were on a different disk block.
- The difference is significant, since each block read could require a disk arm seek, and a block read together with the disk arm seek takes about 10 milliseconds on a typical disk.

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.

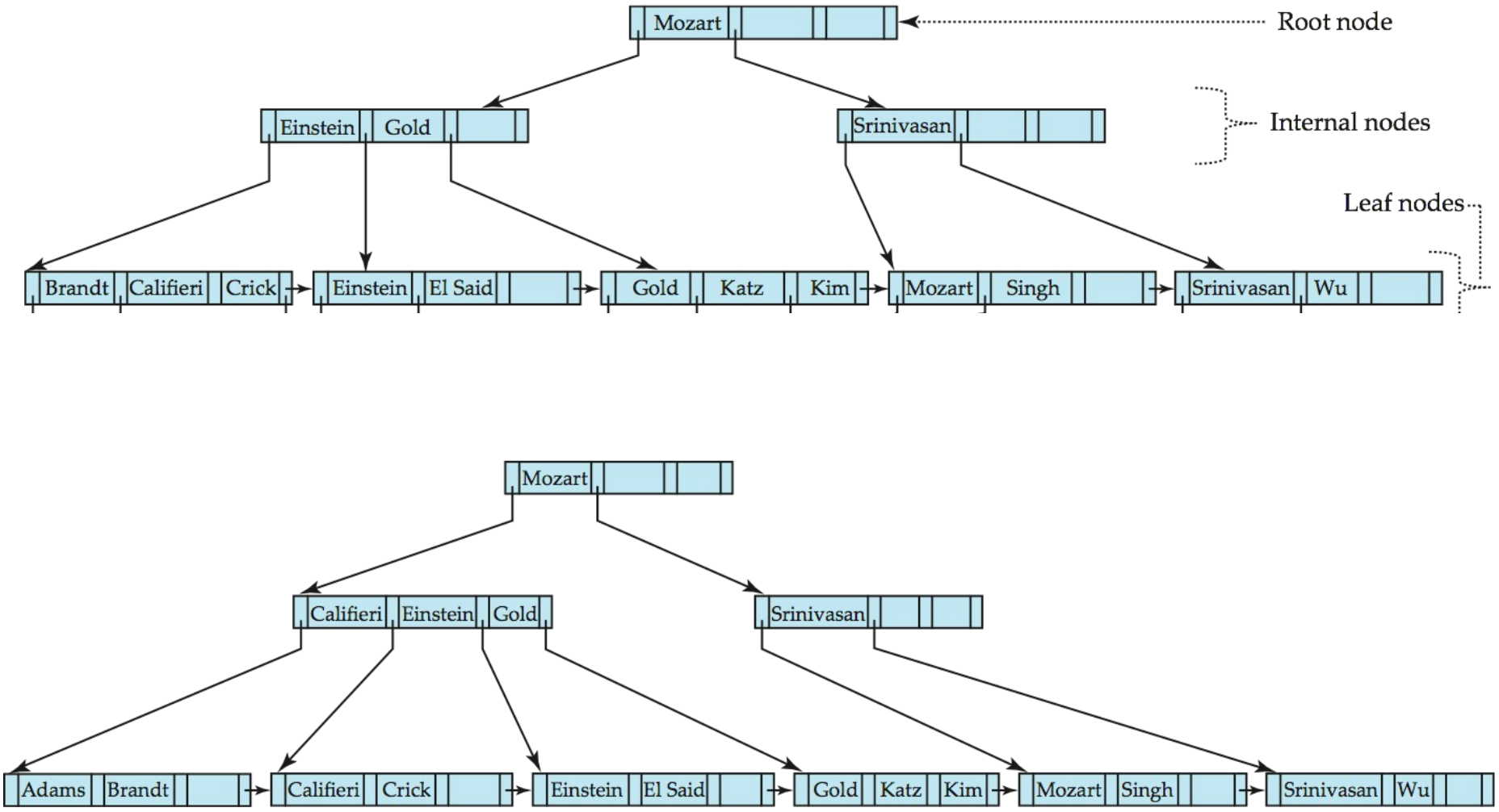
Updates on B+-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first *n*/2 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.



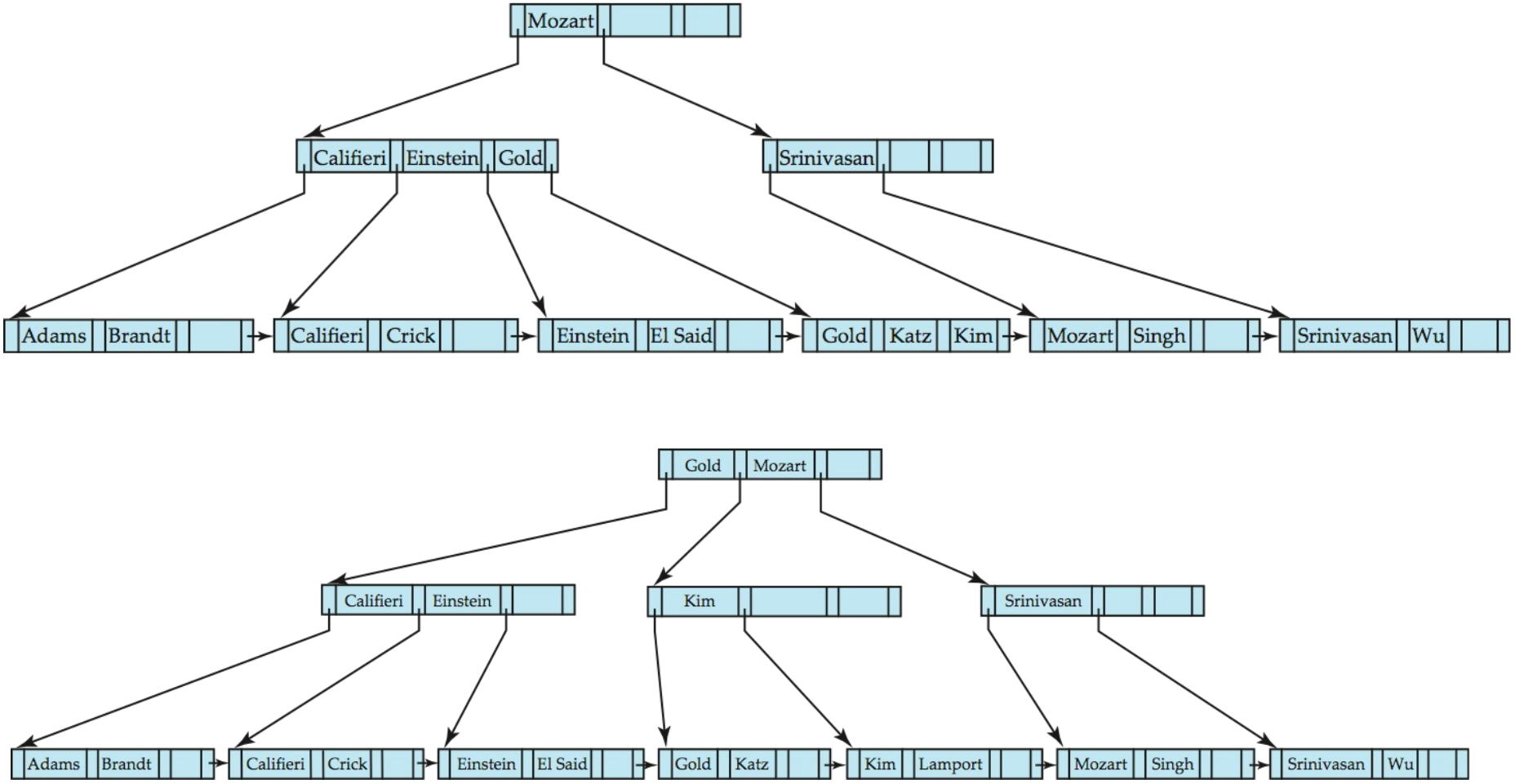
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

B+-Tree Insertion



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

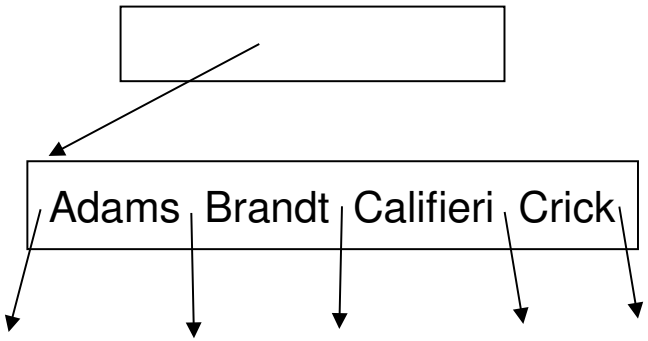
B+-Tree Insertion

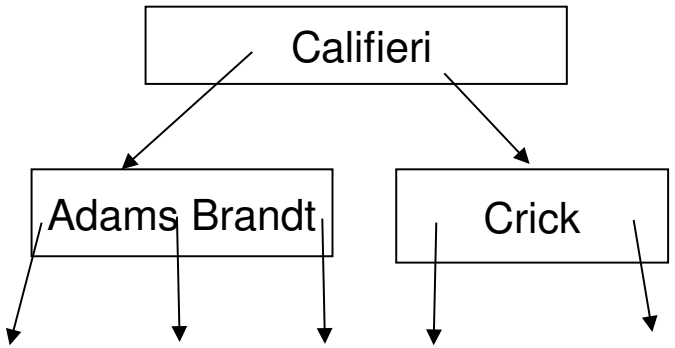


B+-Tree before and after insertion of “Lamport”

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 $P_1, K_1, \ldots, K_{n/2-1}, P_{n/2}$ from M back into node N
  - Copy $P_{n/2+1}, K_{n/2+1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{n/2}, N')$ into parent N
- **Read pseudocode in book!**





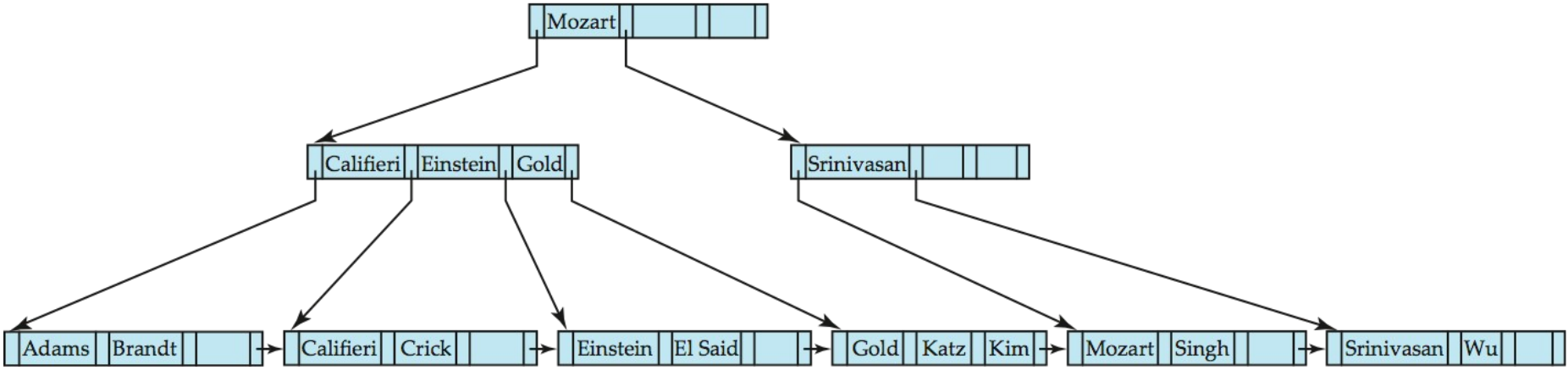
Updates on B+-Trees: Deletion

- 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.

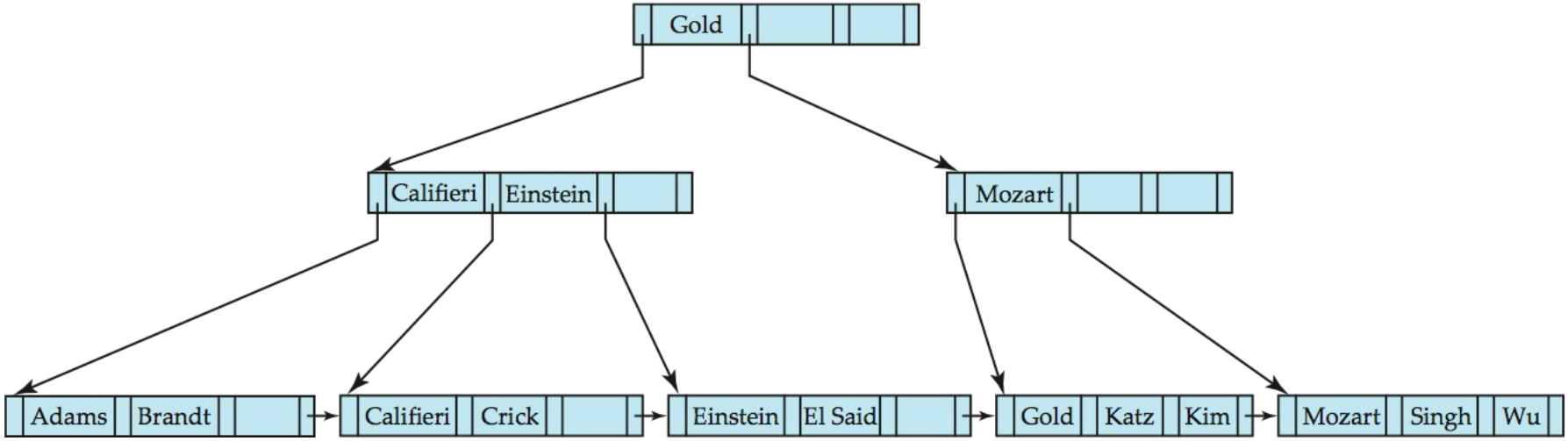
Updates on B+-Trees: Deletion

- 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 $n/2$ 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.

Examples of B+-Tree Deletion

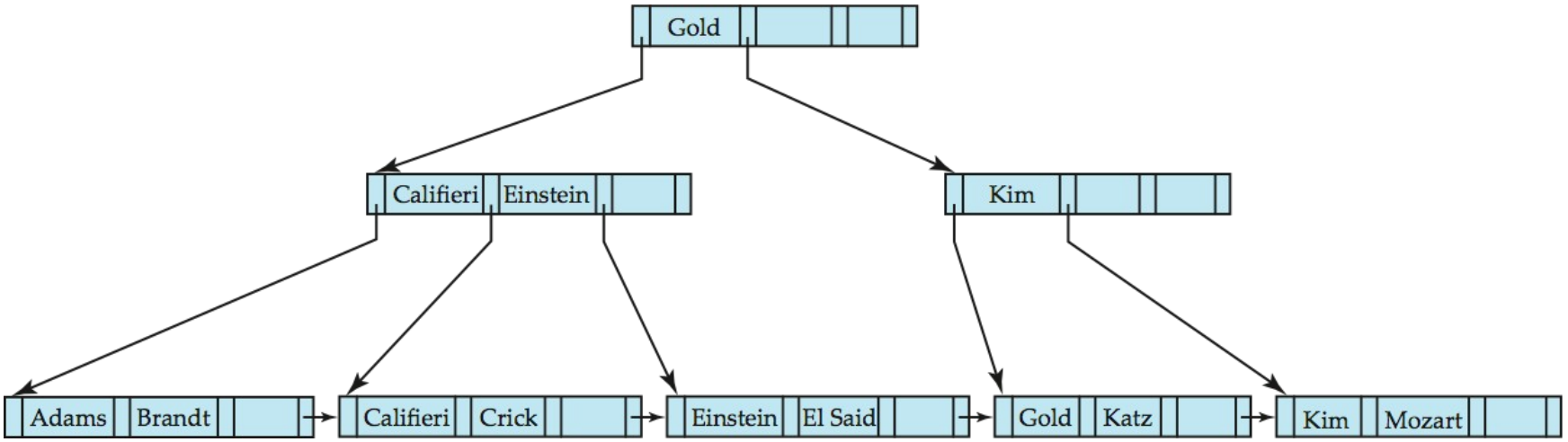


Before and after deleting “Srinivasan”



- Deleting “Srinivasan” causes merging of under-full leaves

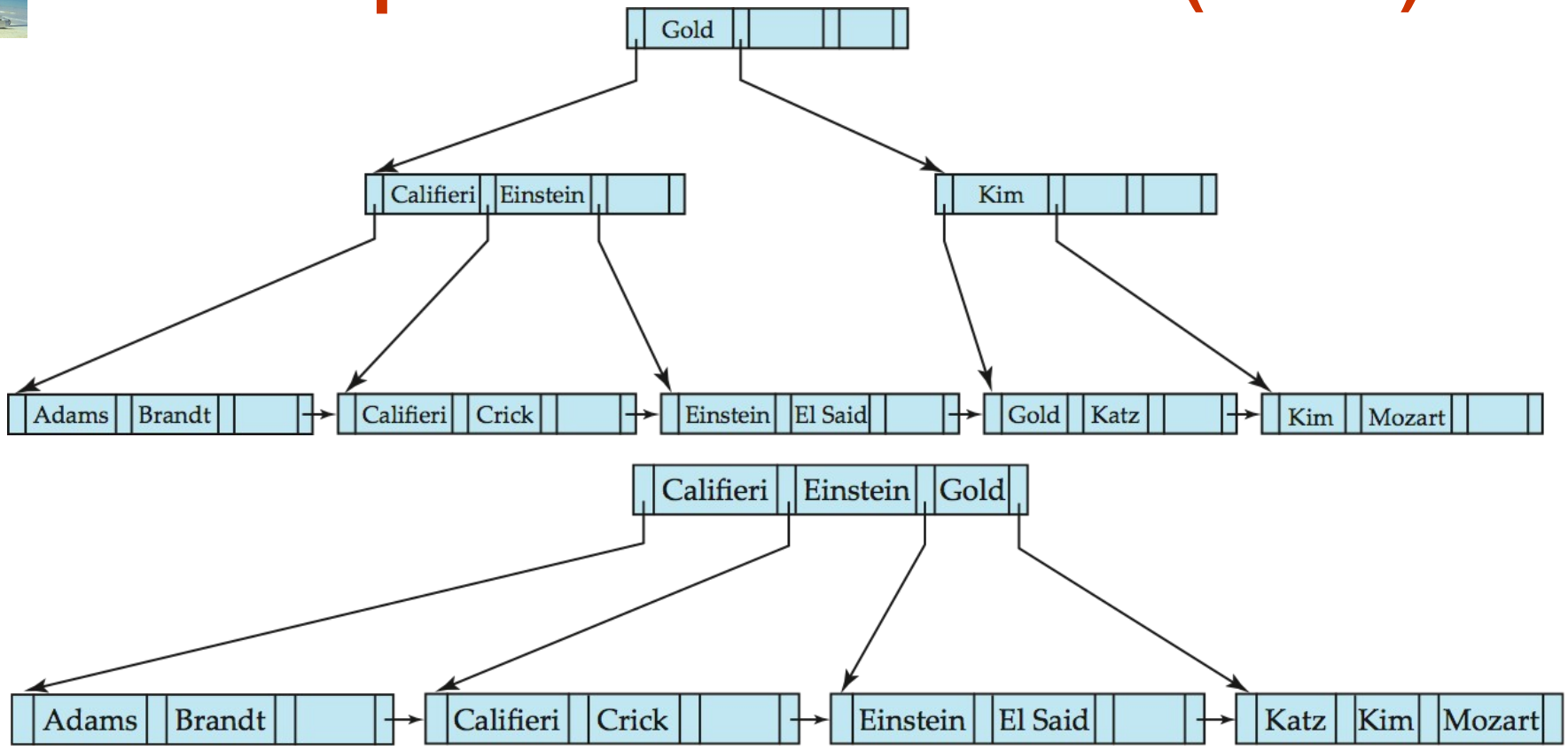
Examples of B+-tree Deletion (Cont.)



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
- B+ Tree simulator:
- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- https://dichchankinh.com/~galles/visualization/BPlusTree.html
- https://goneill.co.nz/btree-demo.php

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

Non-Unique Search Keys

- If a relation can have more than one record containing the same search key value, it is said to be a **nonunique search key**.
- One problem is in the efficiency of record deletion.
- Suppose a particular search-key value occurs a large number of times, and one of the records with that search key is to be deleted. The deletion may have to search through a number of entries, potentially across multiple leaf nodes, to find the entry corresponding to the particular record being deleted.
- A simple solution, used by most database systems, is to make search keys unique by creating a composite search key containing the original search key and another attribute, which together are unique across all records.
- The extra attribute can be a record-id, which is a pointer to the record, or any other attribute (primary key) whose value is unique among all records with the same search key value. The extra attribute is called a **uniquifier** attribute.

Non-Unique Search Keys

- When a record is to be deleted, the composite search-key value is computed from the record, and then used to look up the index. Since the value is unique, the corresponding leaf level entry can be found with a single traversal from root to leaf, with no further accesses at the leaf level. As a result, record deletion can be done efficiently.
- A search with the original search-key attribute simply ignores the value of the uniquifier attribute when comparing search-key values.
- Alternatives to scheme is to store each key value only once in the tree, and to keep a bucket (or list) of record pointers with a search-key value, to handle nonunique search keys.
- **Advantage:** This approach is more space efficient since it stores the key value only once; however, it creates several complications when B+-trees are implemented.

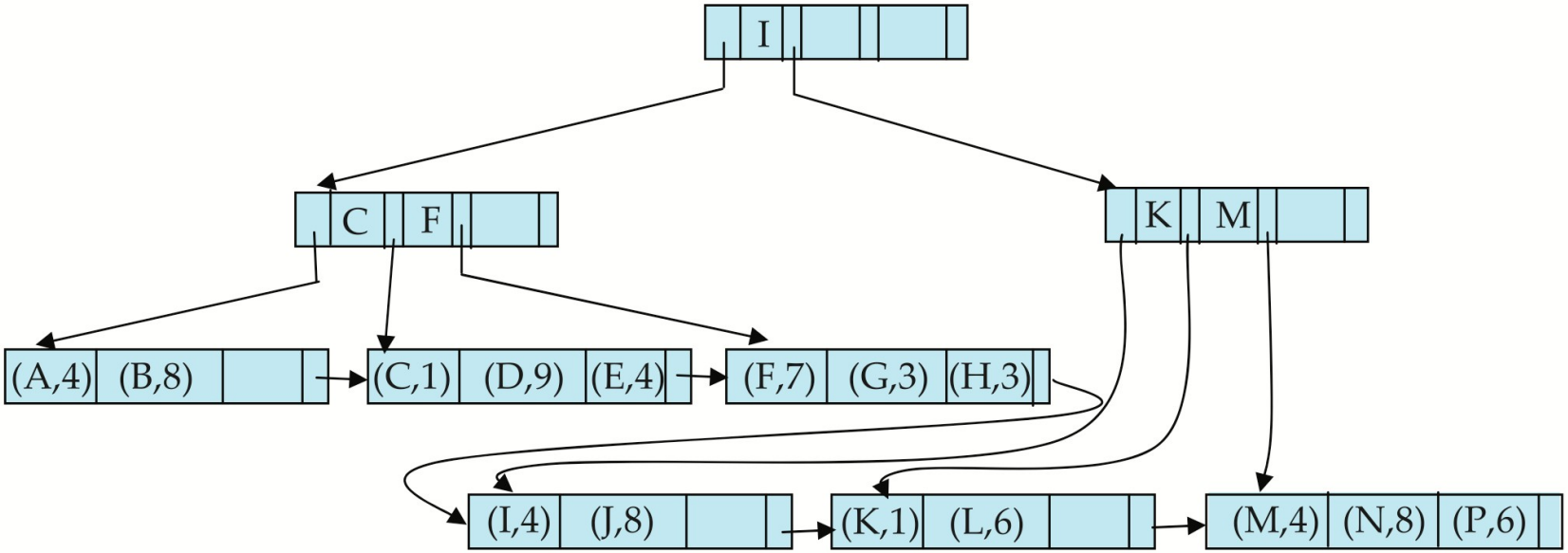
Non-Unique Search Keys

- **Disadvantages:**
- If the buckets are kept in the leaf node, extra code is needed to deal with variable-size buckets, and to deal with buckets that grow larger than the size of the leaf node.
- If the buckets are stored in separate blocks, an extra I/O operation may be required to fetch records.
- In addition to these problems, the bucket approach also has the problem of inefficiency for record deletion if a search-key value occurs a large number of times.
- Due to the inefficiency of deletion, as well as other complications due to duplicate search keys, B+-tree implementations in most database systems only handle unique search keys, and they automatically add record-ids or other attributes to make nonunique search keys unique.

B+-Tree File Organization

- The main drawback of index-sequential file organization is the degradation of performance as the file grows: With growth, an increasing percentage of index entries and actual records become out of order and are stored in overflow blocks.
- Degradation of index lookup problem is solved by using B+-Tree indices on the file.
- The B+-tree structure is used not only as an index, but also as an organizer for records in a file.
- The leaf nodes in a B+-tree file organization store records, instead of pointers. 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.
- However, leaf nodes are still required to be at least half full
- 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

- In B+-tree for file organization, good space utilization is important since records use more space than key and pointers.
- To improve space utilization, involve **more** sibling nodes in redistribution during splits and merges for both leaf nodes and nonleaf nodes
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

B+-Tree File Organization

- **During insertion:** if a node is full the system attempts to redistribute some of its entries to one of the adjacent nodes, to make space for a new entry.
- If the adjacent nodes are themselves full, the system splits the node, and splits the entries evenly among one of the adjacent nodes and the two nodes that it obtained by splitting the original node.
- Since the three nodes together contain one more record than can fit in two nodes, each node will be about two-thirds full.
- More precisely, each node will have at least $\lfloor 2n/3 \rfloor$ entries, where *n* is the maximum number of entries that the node can hold.
- **During deletion** of a record, if the occupancy of a node falls below $\lfloor 2n/3 \rfloor$, the system attempts to borrow an entry from one of the sibling nodes.
- If both sibling nodes have $\lfloor 2n/3 \rfloor$ records, instead of borrowing an entry, the system redistributes the entries in the node and in the two siblings evenly between two of the nodes, and deletes the third node, because the total number of entries is $3\lfloor 2n/3 \rfloor - 1$ which is less than *2n.*
- However, the cost of update becomes higher as more sibling nodes are involved in the redistribution.

Secondary Indices and Record Reallocation

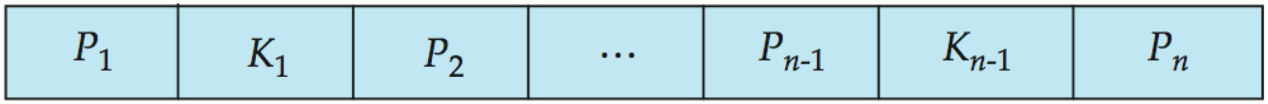
- In B+-tree file organization, may change the location of records even when the records have not been updated. As an example, when a leaf node is split , a number of records are moved to a new node.
- In such cases, all secondary indices that store pointers to the relocated records would have to be updated (even key-values have not changed)
- Each leaf node may contain a fairly large number of records, and each of them may be in different locations on each secondary index.
- Thus a leaf-node split may require tens or even hundreds of I/O operations to update all affected secondary indices, making it a very expensive operation.
- *Solution*: use primary-index search key instead of record pointer in secondary index
  - Extra traversal :use the secondary index to find the primary-index search-key values, and then the primary index to find the corresponding records.
    - Higher cost for queries, but node splits are cheap
    - Add record-id if primary-index search key is non-unique

Indexing Strings

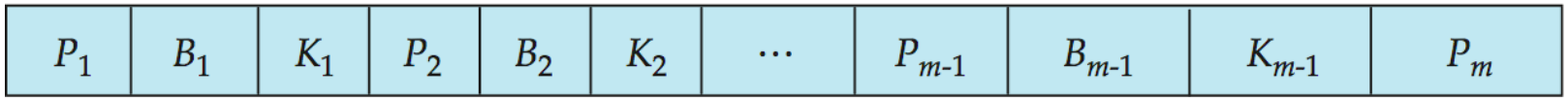
- Creating B+-tree indices on string-valued attributes raises 2 problems:
  - 1. strings can be of variable length
  - 2. strings can be long, leading to a low fanout and a correspondingly increased tree height.
- Thus space utilization as criterion for splitting, not number of pointers
- The fanout of nodes can be increased by using a technique called **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

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



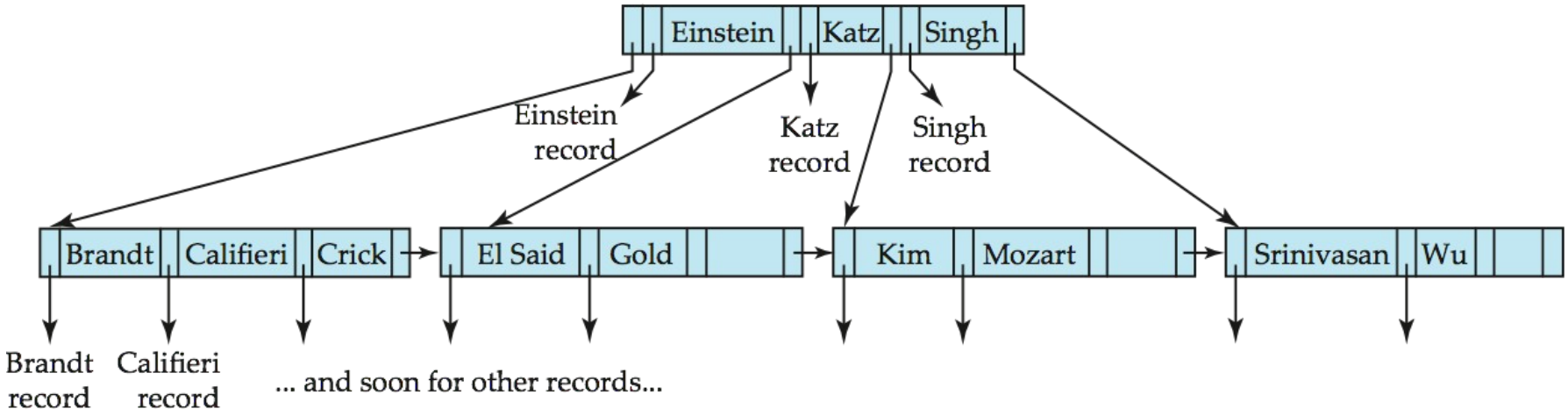
(a)



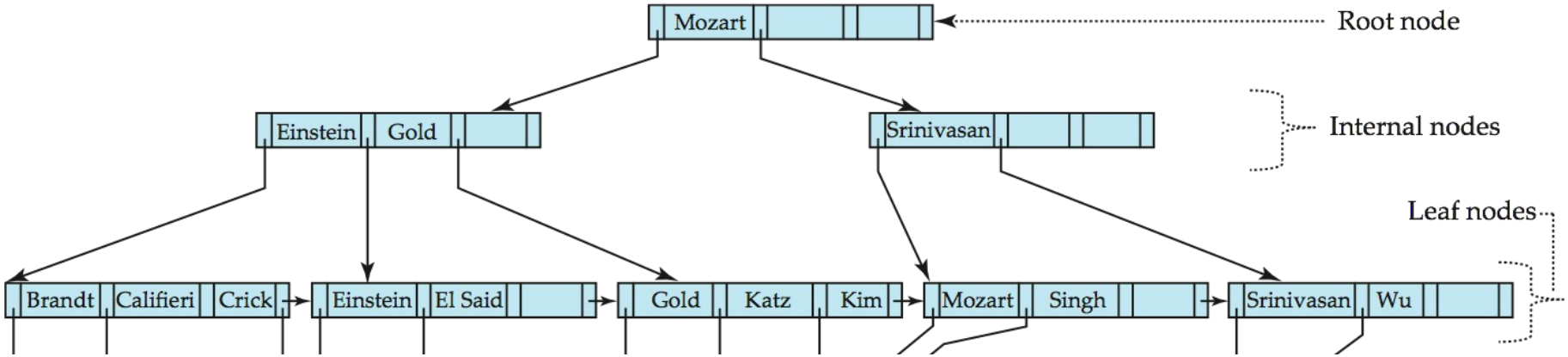
(b)

- Nonleaf node – pointers B_i 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.

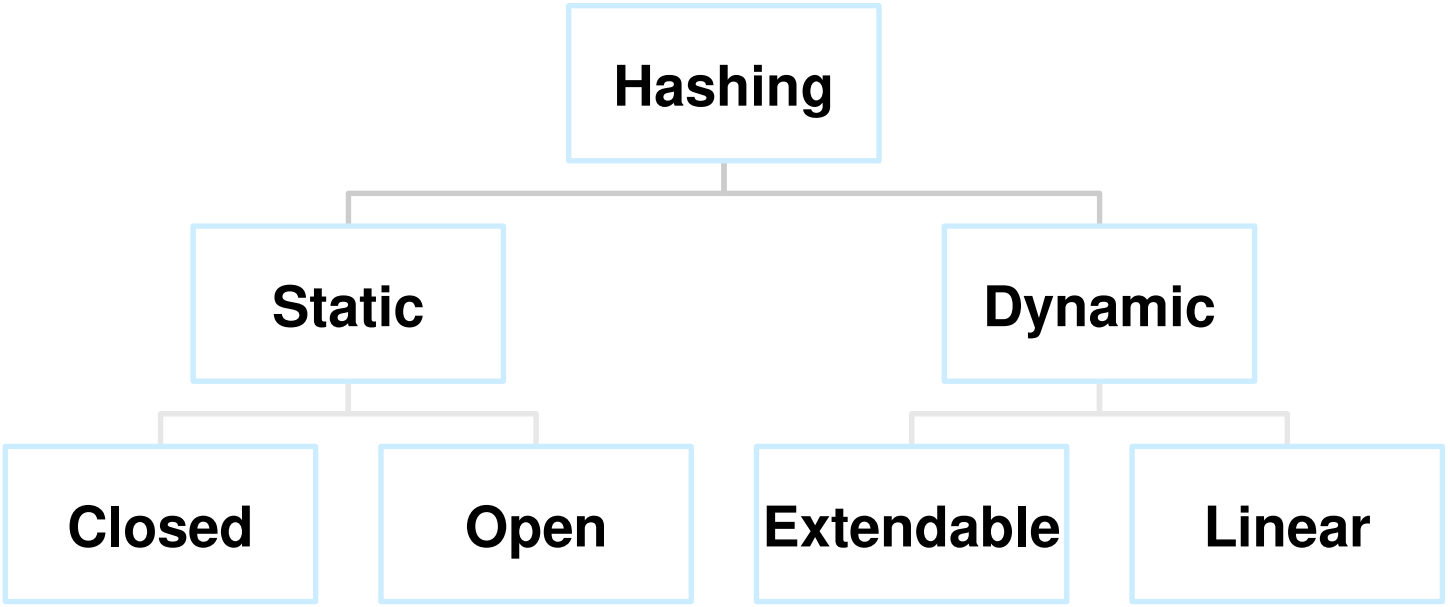
B+ Tree vs. B Tree

B + Tree	B Tree
Search keys can be repeated.	Search keys cannot be redundant.
Data is only saved on the leaf nodes.	Both leaf nodes and internal nodes can store data
Data stored on the leaf node makes the search more accurate and faster.	Searching is slow due to data stored on Leaf and internal nodes.
Deletion is not difficult as an element is only removed from a leaf node.	Deletion of elements is a complicated and time-consuming process.
Linked leaf nodes make the search efficient and quick.	You cannot link leaf nodes.

Hashing

Database System Concepts, 6th Ed.

Classification of Hashing



Static Hashing

- One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations.
- File organizations based on the technique of **hashing** allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices.
- A **bucket** is a unit of storage containing one or more records. A bucket is typically a disk block but could be chosen to be smaller or larger than 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.*

Static Hashing

- Hash function is used to locate records for access, insertion as well as deletion.
- To **insert** a record with search key K_i, we compute $h(K_i)$, which gives the address of the bucket for that record. Assuming there is space in the bucket, the record is stored in that bucket.
- To perform a **lookup** on a search-key value K_i , we simply compute $h(K_i)$, then search the bucket with that address.
- 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.
- Suppose that two search keys, K_5 and K_7, have the same hash value; that is, $h(K_5) = h(K_7)$. If we perform a lookup on K_5, the bucket $h(K_5)$ contains records with search-key values K_5 and records with search-key values K_7. Thus, we have to check the search-key value of every record in the bucket to verify that the record is one that we want.

Static Hashing

- **Deletion** is equally straightforward. If the search-key value of the record to be deleted is K_i, we compute $h(K_i)$, then search the corresponding bucket for that record, and delete the record from the bucket.
- Hashing can be used for two different purposes. In a **hash file organization**, we obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record.
- In a **hash index organization** we organize the search keys, with their associated pointers, into a hash file structure.

Example of Hash File Organization

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

- There are 8 buckets,
- The representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the representations of the characters modulo 8
  - 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 Function

- The worst possible hash function maps all search-key values to the same bucket. Such a function is undesirable because all the records have to be kept in the same bucket. A lookup has to examine every such record to find the one desired.
- An ideal hash function distributes the stored keys uniformly across all the buckets, so that every bucket has the same number of records.
- Since we do not know at design time precisely which search-key values will be stored in the file, we want to choose a hash function that assigns search-key values to buckets in such a way that the distribution has these two qualities:
- **Uniform**: The distribution is *uniform.* That is, the hash function assigns each bucket the same number of search-key values from the set of *all* possible search-key values.

Hash Function

- **Random:** The distribution is *random*. That is, in the average case, each bucket will have nearly the same number of values assigned to it, regardless of the actual distribution of search-key values.
- More precisely, the hash value will not be correlated to any externally visible ordering on the search-key values, such as alphabetic ordering or ordering by the length of the search keys; the hash function will appear to be random.
- Illustration: (form book)
- Example 1: Simple hash function, but fails to provide a uniform distribution.
- Example 2: The distribution of search-key values may be uniform, but is not random.

Hash Functions

- The distribution of records is not uniform—some buckets receive more records than others do. If the function has a random distribution, even if there are correlations in the search keys, the randomness of the distribution will make it very likely that all buckets will have roughly the same number of records, as long as each search key occurs in only a small fraction of the records.
- If a single search key occurs in a large fraction of the records, the bucket containing it is likely to have more records than other buckets, regardless of the hash function used.
- Typical hash functions perform computation on the internal binary representation of characters in 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.

Hash Functions

- Hash functions require careful design.
- A bad hash function may result in lookup taking time proportional to the number of search keys in the file.
- A well designed function gives an average-case lookup time that is a (small) constant, independent of the number of search keys in the file.

Handling of Bucket Overflows

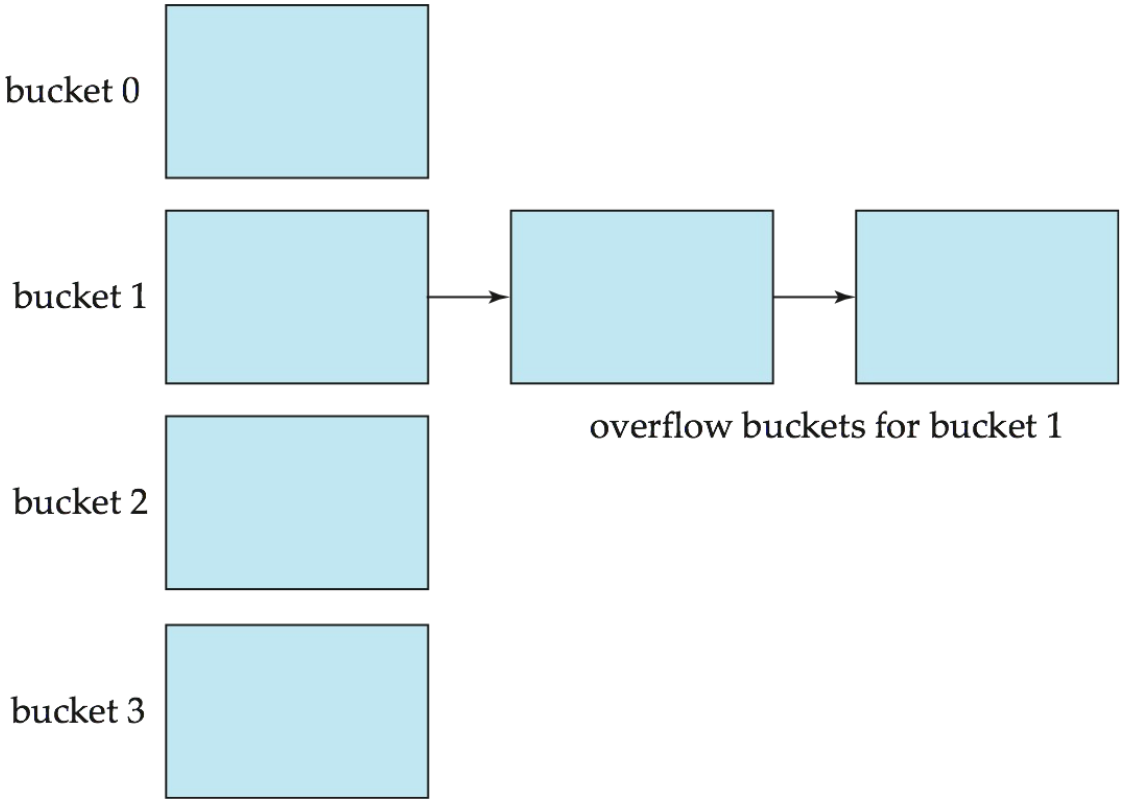
- Suppose, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space, a **bucket overflow** is said to occur.
- Bucket overflow can occur for several reasons:
- **Insufficient buckets** : The number of buckets, n_B, must be chosen such that $n_B > n_r / f_r$, where
  - n_r denotes the total number of records that will be stored and
  - f_r denotes the number of records that will fit in a bucket.
  - This designation, of course, assumes that the total number of records is known when the hash function is chosen.
- **Skew in distribution of records**: Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space. This situation is called bucket **skew**. Skew can occur for two reasons:
  - Multiple records have same search-key value
  - Chosen hash function produces non-uniform distribution of key values

Handling of Bucket Overflows

- So that the probability of bucket overflow is reduced, the number of buckets is chosen to be $(n_r / f_r) * (1 + d)$, where d is a **fudge factor**, typically around 0.2.
- Some space is wasted: About 20 percent of the space in the buckets will be empty. But the benefit is that the probability of overflow is reduced.
- Despite allocation of a few more buckets than required, bucket overflow can still occur.
- We handle bucket overflow by using **overflow buckets.** If a record must be inserted into a bucket *b,* and *b* is already full, the system provides an overflow bucket for *b,* and inserts the record into the overflow bucket. If the overflow bucket is also full, the system provides another overflow bucket, and so on.
- All the overflow buckets of a given bucket are chained together in a linked list. Overflow handling using such a linked list is called **overflow chaining.**

Handling of Bucket Overflows (Cont.)

- .



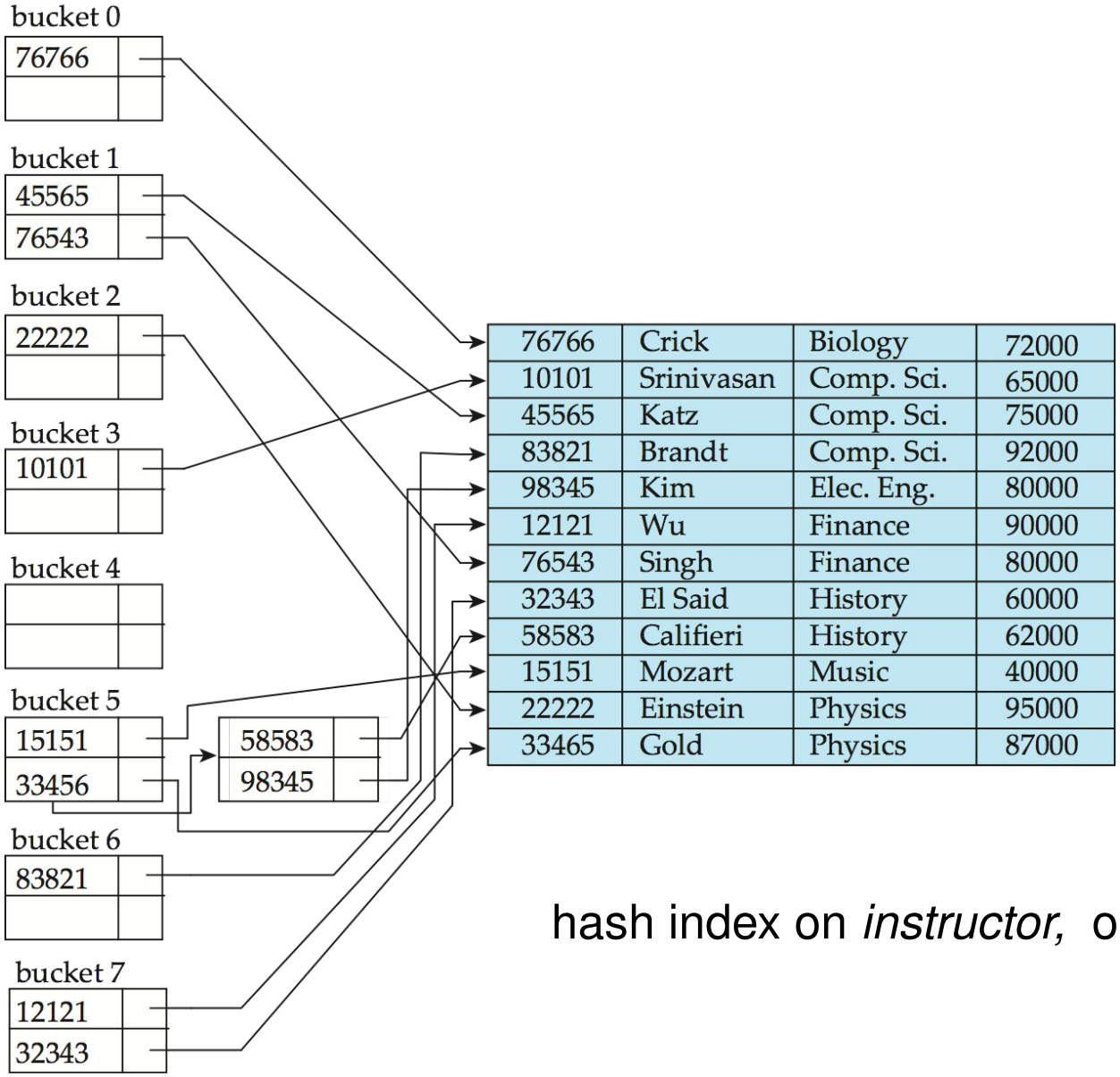
Closed and Open Hashing

- The form of hash structure of above form sometimes referred to as **closed hashing (**more commonly **closed addressing)**.
- Under an alternative approach, called **open hashing (**more commonly **open addressing)**, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets *B.*
- One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing.* Other policies, such as computing further hash functions, are also used.
- Open hashing has been used to construct symbol tables for compilers and assemblers, but closed hashing is preferable for database systems.
- The reason is that deletion under open hashing is troublesome. Usually, compilers and assemblers perform only lookup and insertion operations on their symbol tables.
- However, in a database system, it is important to be able to handle deletion as well as insertion. Thus, open hashing is of only minor importance in database implementation.

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.
- Apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).
- We use the term *hash index* to denote hash file structures as well as secondary hash indices.
- 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, since hash file organization provides the same direct access to records that indexing provides, we pretend that a file organized by hashing also has a clustering hash index on it.

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*

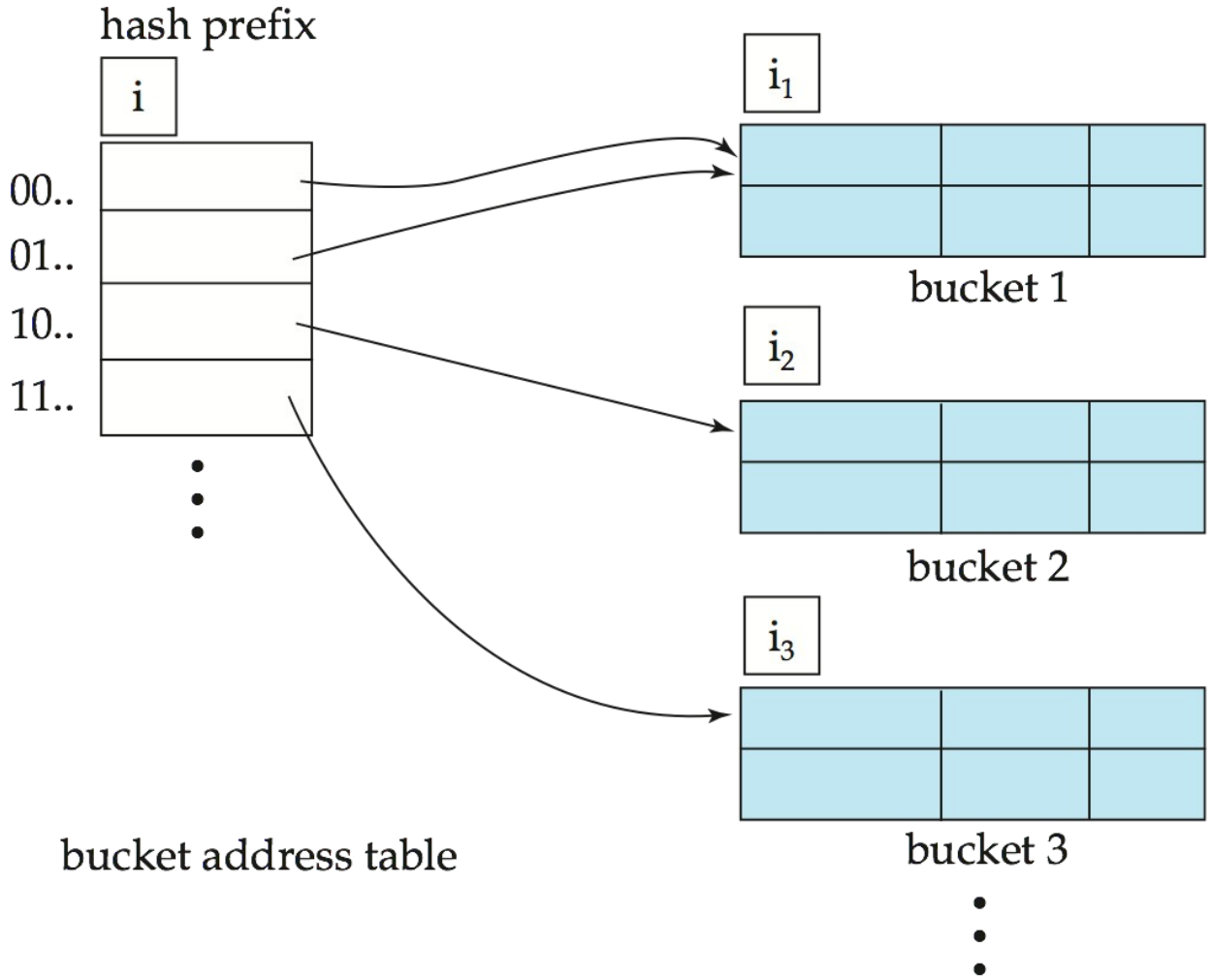
Deficiencies of Static Hashing

- 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
  - This reorganization is a massive, time-consuming operation. Furthermore, it is necessary to forbid access to the file during reorganization.
- Better solution: allow the number of buckets to be modified dynamically to accommodate the growth or shrinkage of the database.

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 binary 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 *i* 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

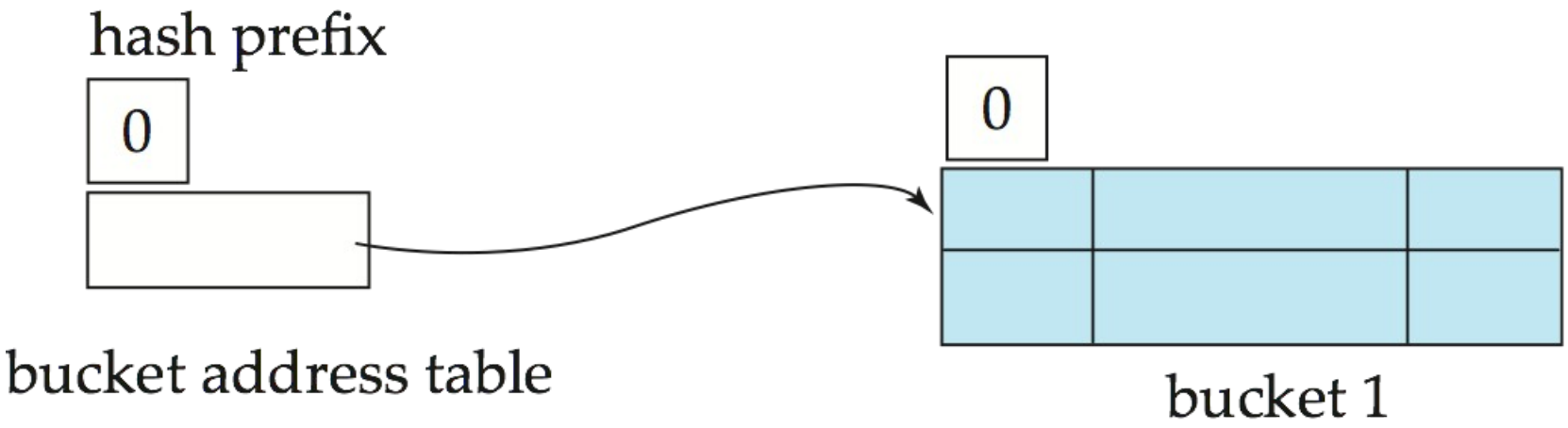
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same i_j –1 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

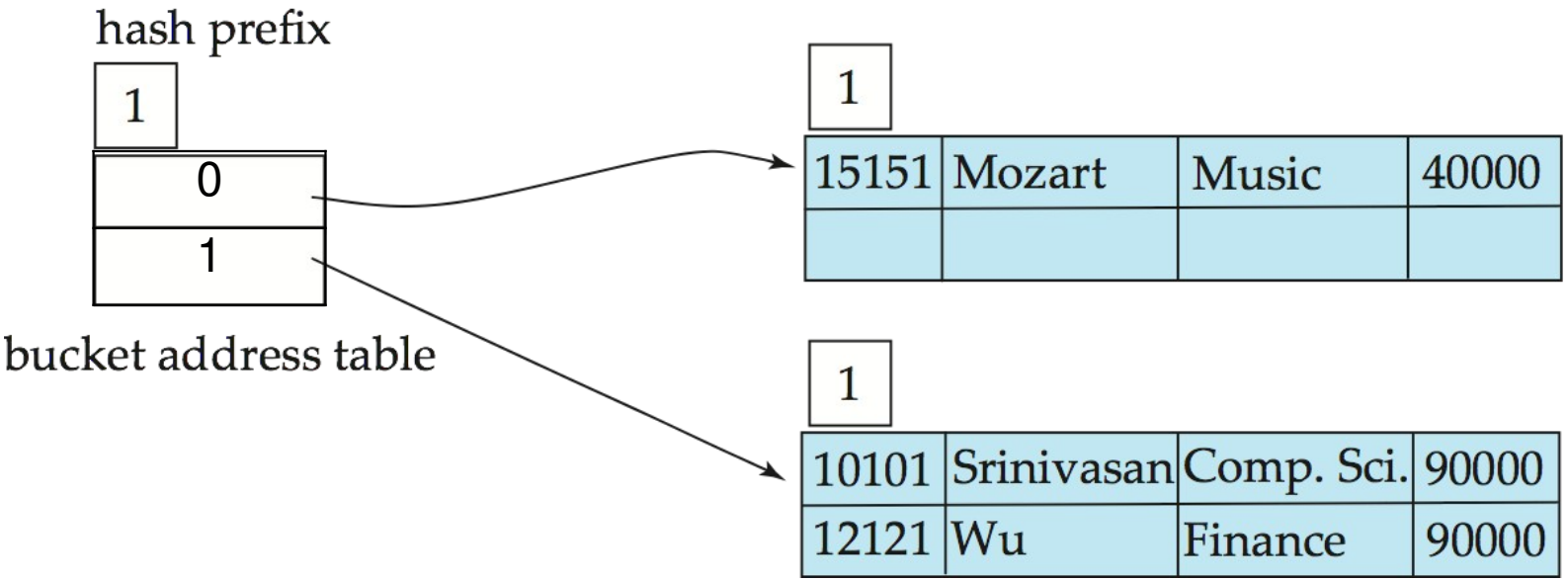
Example (Cont.)

- Initial Hash structure; bucket size = 2



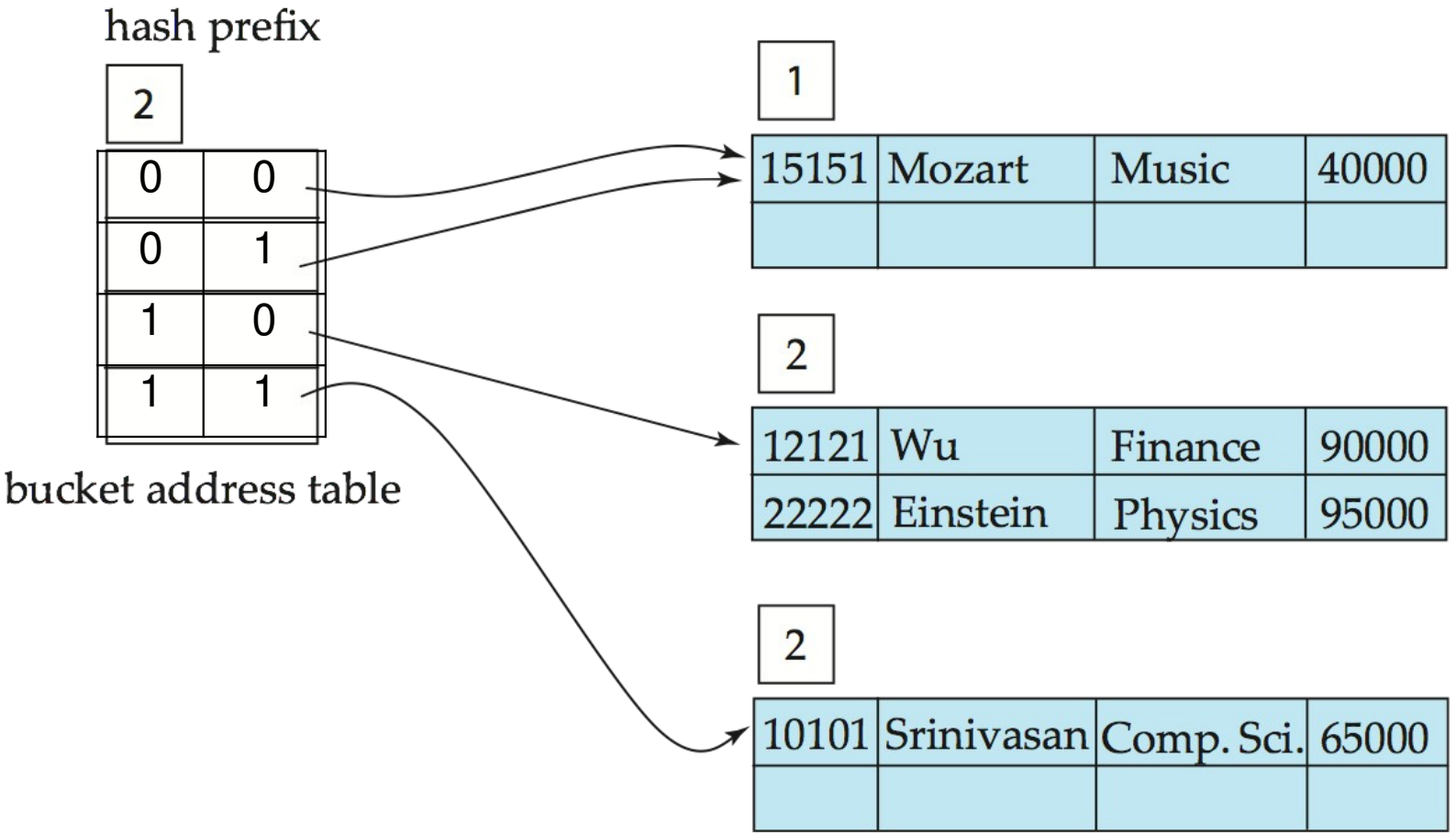
Example (Cont.)

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



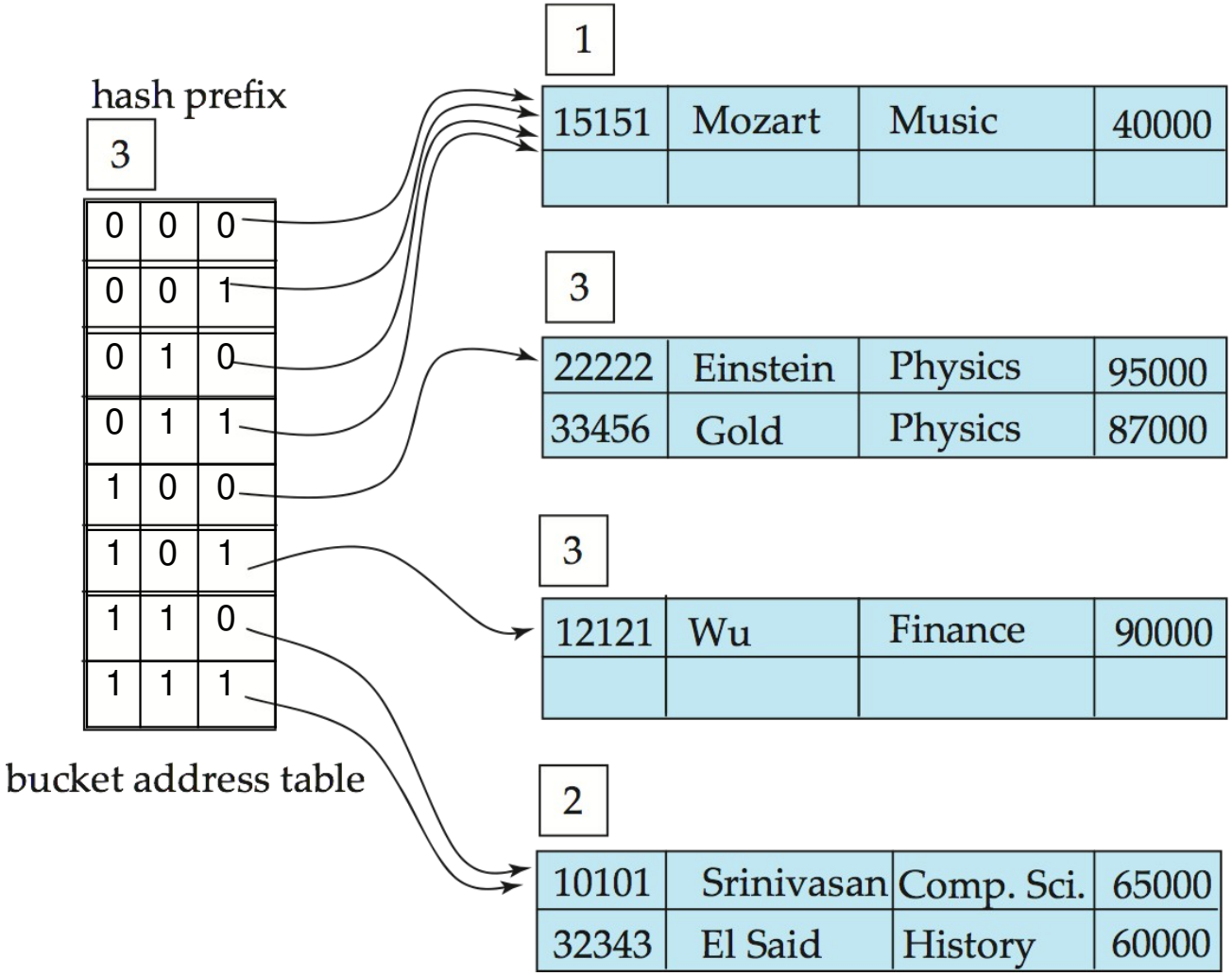
Example (Cont.)

- Hash structure after insertion of Einstein record



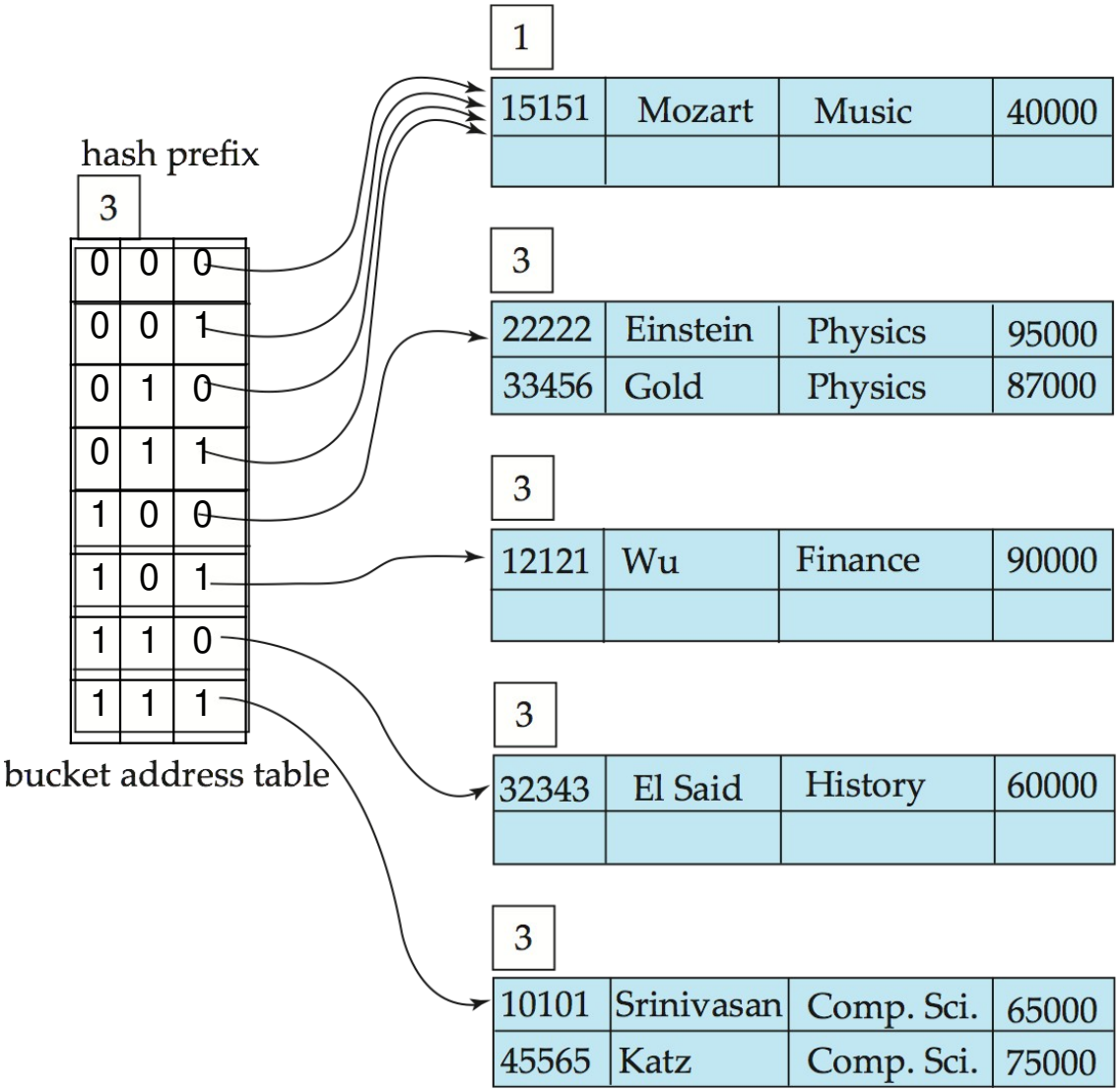
Example (Cont.)

- Hash structure after insertion of Gold and El Said records

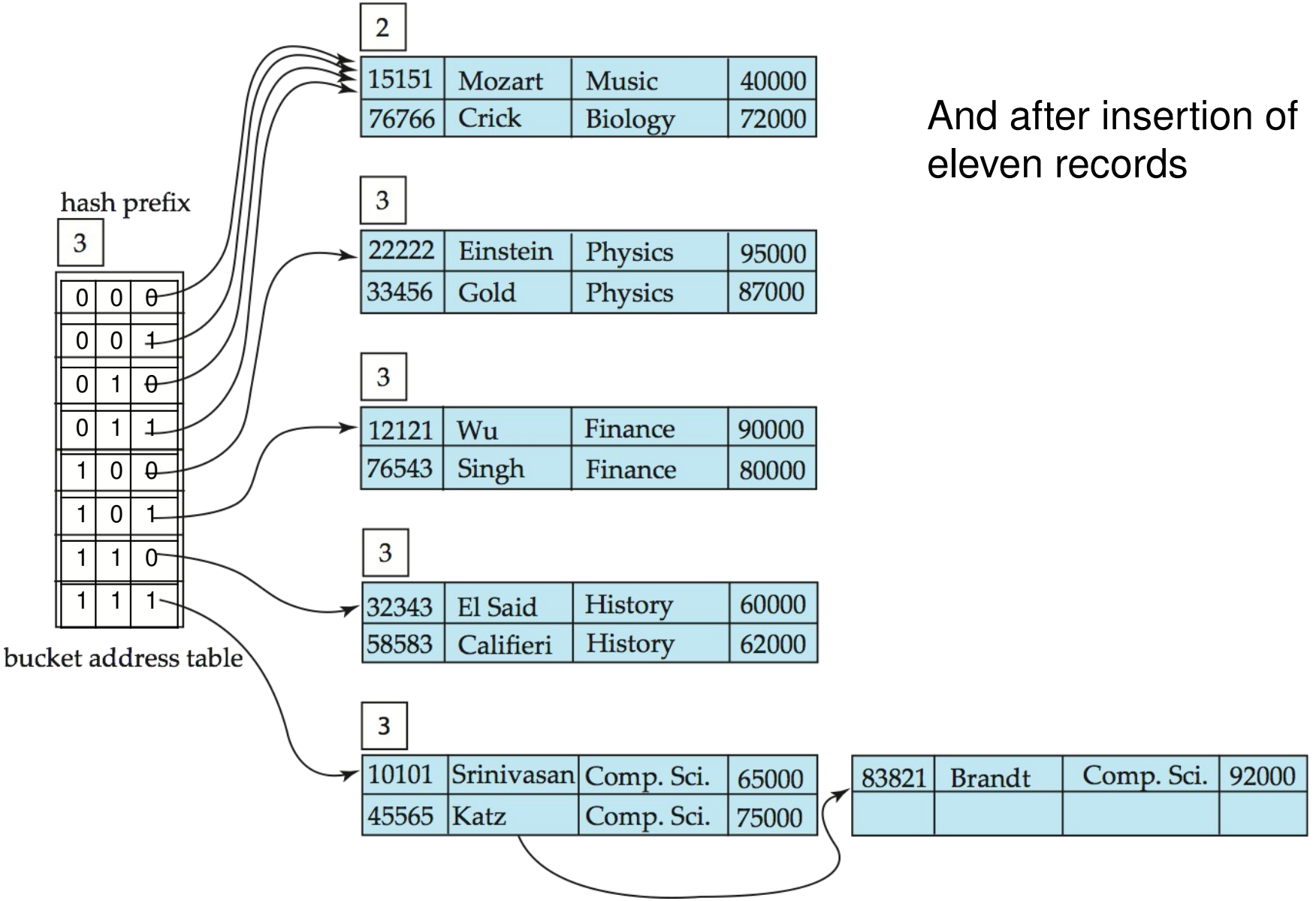


Example (Cont.)

- Hash structure after insertion of Katz record

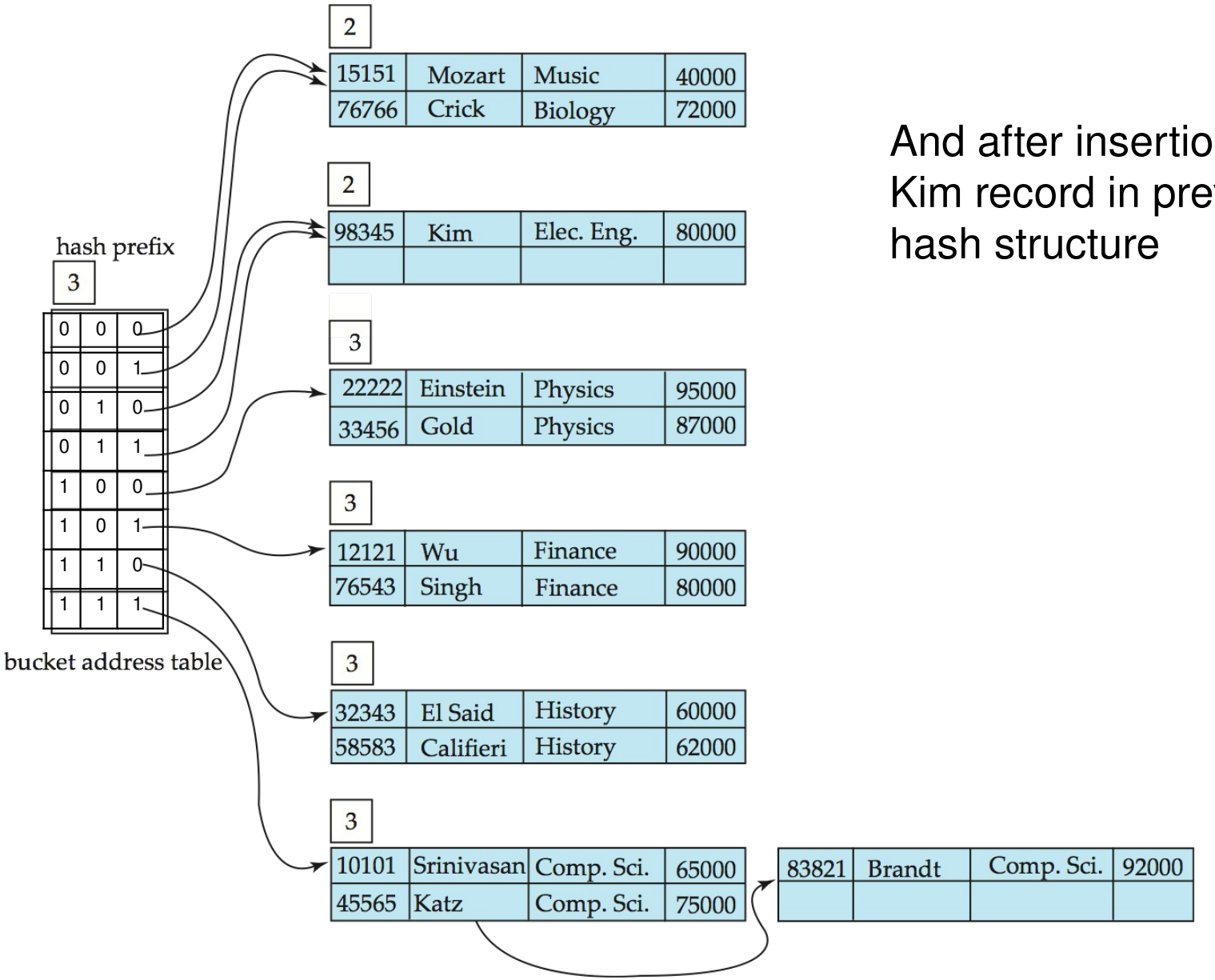


Example (Cont.)



And after insertion of eleven records

Example (Cont.)



And after insertion of Kim record in previous hash structure

Static Hashing vs. Dynamic Hashing

- Benefits of extendable hashing over static hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
  - Although the bucket address table incurs additional overhead, it contains one pointer for each hash value for the current prefix length. This table is thus small.
  - The main space saving of extendable hashing over other forms of hashing is that no buckets need to be reserved for future growth; rather, buckets can be allocated dynamically.
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record, since the system must access the bucket address table before accessing the bucket itself (minor effect on performance)

Static Hashing vs. Dynamic Hashing

- 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

- Thus, extendable hashing appears to be a highly attractive technique, provided the added complexity involved in its implementation is accepted.
- 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

- Files of records can be organized as
  - ordered files by using index-sequential organization or
  - B+-tree organizations or
  - by using hashing or
  - as heap files, where the records are not ordered in any particular way.
- Each scheme has advantages in certain situations. Most database systems support B+-trees and may additionally support some form of hash file organization or hash indices.
- To make a choice of file organization and indexing techniques for a relation, the implementer or the database designer must consider the following issues:
  - Is the cost of periodic reorganization of the index or hash organization acceptable?

Comparison of Ordered Indexing and Hashing

- What is the relative frequency of insertion and deletion?
- Is it desirable to optimize average access time at the expense of increasing the worst-case access time?
- What types of queries are users likely to pose?

- The fourth issue, the expected type of query, is critical to the choice of ordered indexing or hashing.
- If most queries are point queries:

$$\textbf{select } A_1, A_2, \ldots, A_n \\ \textbf{from } r \\ \textbf{where } A_i = c;$$

- Hashing scheme is preferable here.
  - An ordered-index lookup requires time proportional to the log of the number of values in r for A_i .
  - In a hash structure, however, the average lookup time is a constant independent of the size of the database.

Comparison of Ordered Indexing and Hashing

- In indexing, the worst-case lookup time is proportional to the log of the number of values in *r* for A_i .
- By contrast, for hashing, the worst-case lookup time is proportional to the number of values in *r* for A_i .

- **Choice:** the worst-case lookup time is unlikely to occur with hashing, and hashing is preferable in this case.
- If most queries are range queries:

```
select A1, A2, ..., An
from r
where Ai ≤ c2 and Ai ≥ c1;
```

- Ordered-index techniques are preferable to hashing.
  - Using index stucture, perform a lookup on value c_1. Once the bucket is found for value c_1, follow the pointer chain in the index to read the next bucket in order, and continue in this manner until c_2 is reached.

Comparison of Ordered Indexing and Hashing

- By contrast, using hash structure, perform a lookup on c_1 and can locate the corresponding bucket—but it is not easy, in general, to determine the next bucket that must be examined.
- Since values are scattered randomly by the good hash function, the values in the specified range are likely to be scattered across many or all of the buckets.
- Therefore, it has to read all the buckets to find the required search keys.

- **Choice:** Index methods are preferable where a range of values is specified in the query
- 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

Index Definition in SQL

- Create an index

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

  E.g.: **create index** *b-index* **on** *Instructor(dept_name)*
- Use **create unique index** to indirectly specify and enforce the condition that the search 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