

## Indexing and Hashing

Dr. Manas Khatua  
Assistant Professor  
Dept. of CSE  
IIT Jodhpur  
E-mail: [manaskhatua@iitj.ac.in](mailto:manaskhatua@iitj.ac.in)

# Why Indexing?



- Many queries **reference only a small proportion** of the records in a file
- It is **inefficient to read every tuple** in the relation to fetch the appropriate tuples
- Ideally, the system **should be able to locate these records directly**.
- To allow these forms of access, we **design additional structures** that we associate with files.
- **Indexing mechanisms** used to speed up access to desired data.
- **Search Key** – an attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form:

search-key	pointer
------------	---------
- Index files are typically much smaller than the original file
- **Two basic kinds of indices:**
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

# Index Evaluation Metrics

- **Access types:** The types of access,
  - 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:** The additional space occupied by an index structure

# Ordered Indices for Index-Sequential File

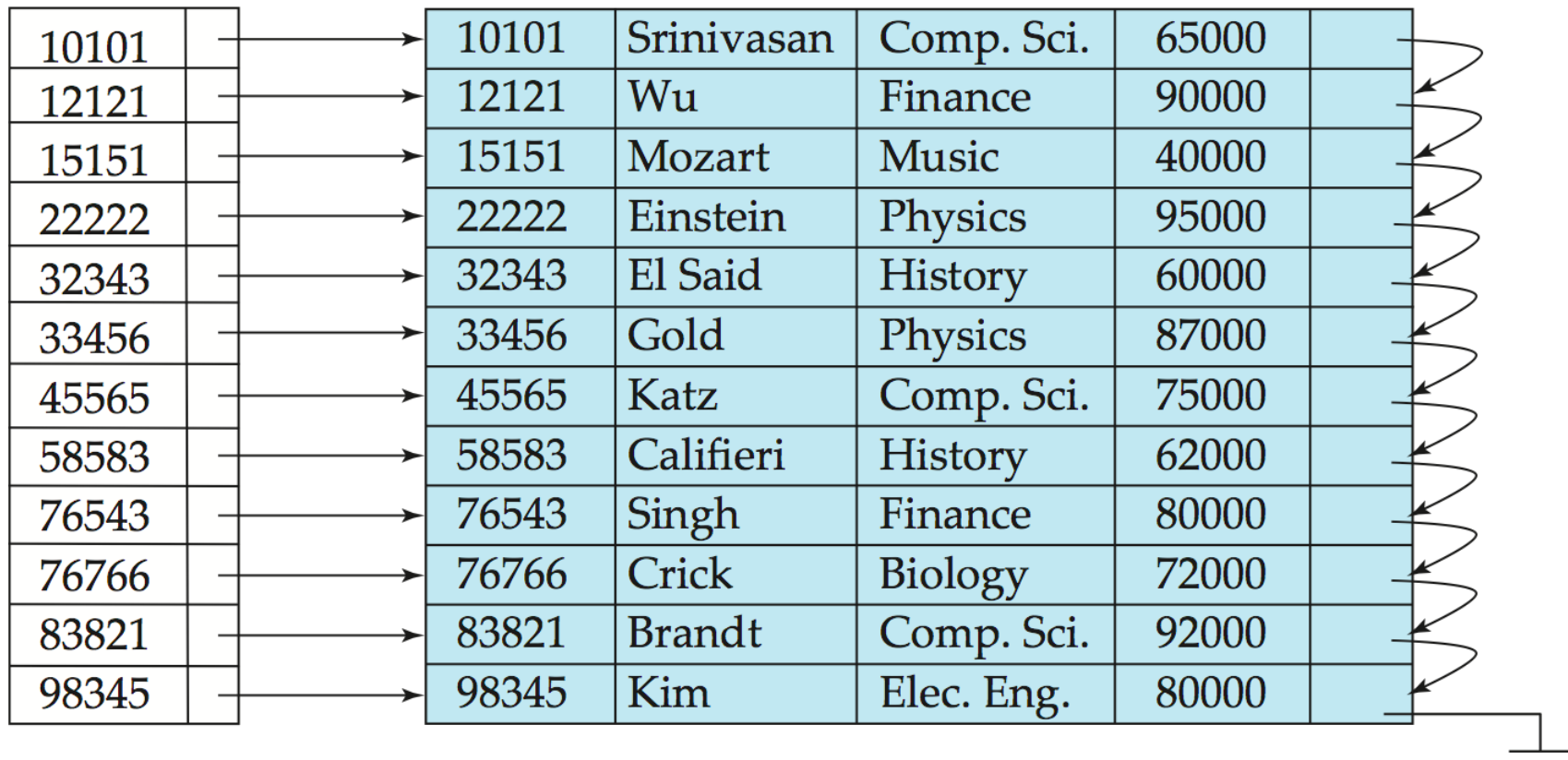
# Ordered Indices



- 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**.
- There are **two types of ordered indices**
  - Dense index
  - Sparse 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

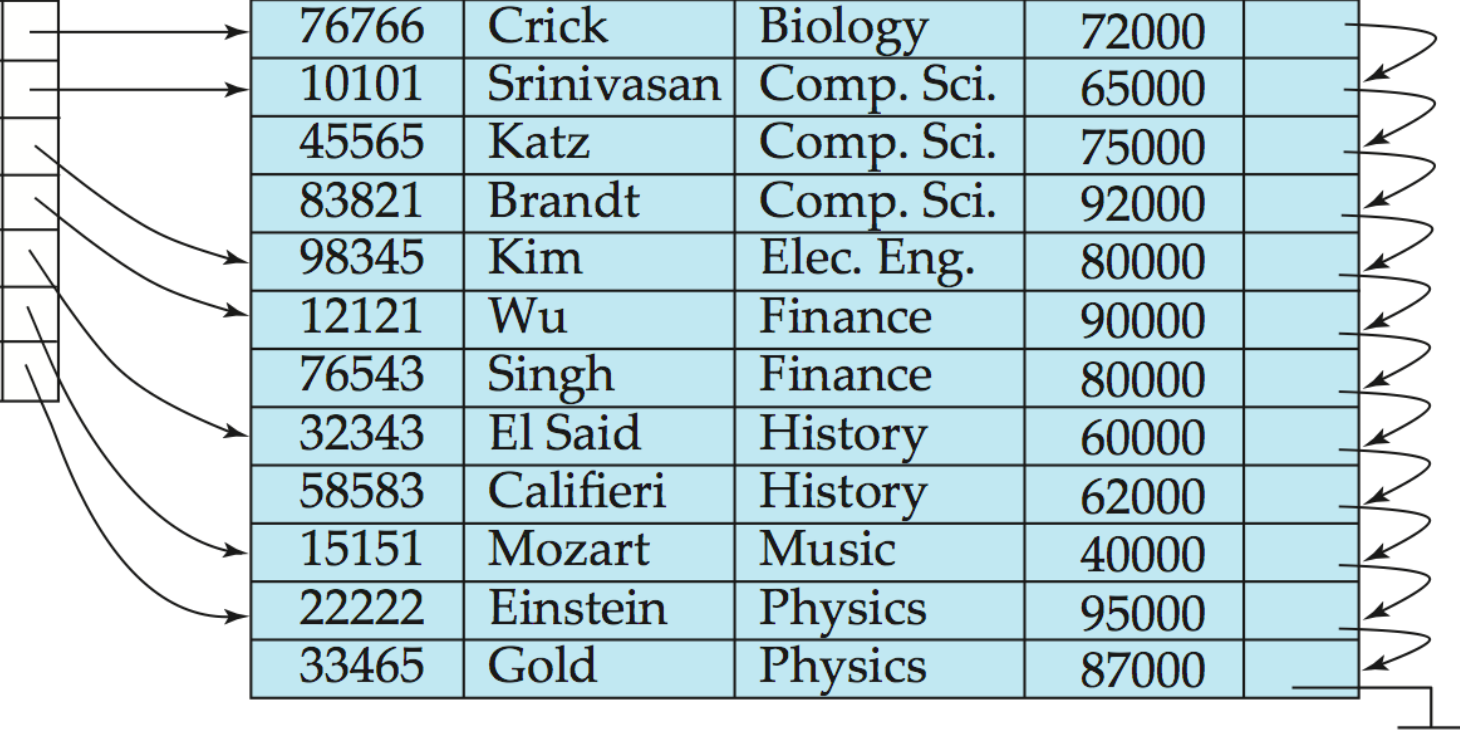


**Figure 11.2** Dense index.

# Cont...

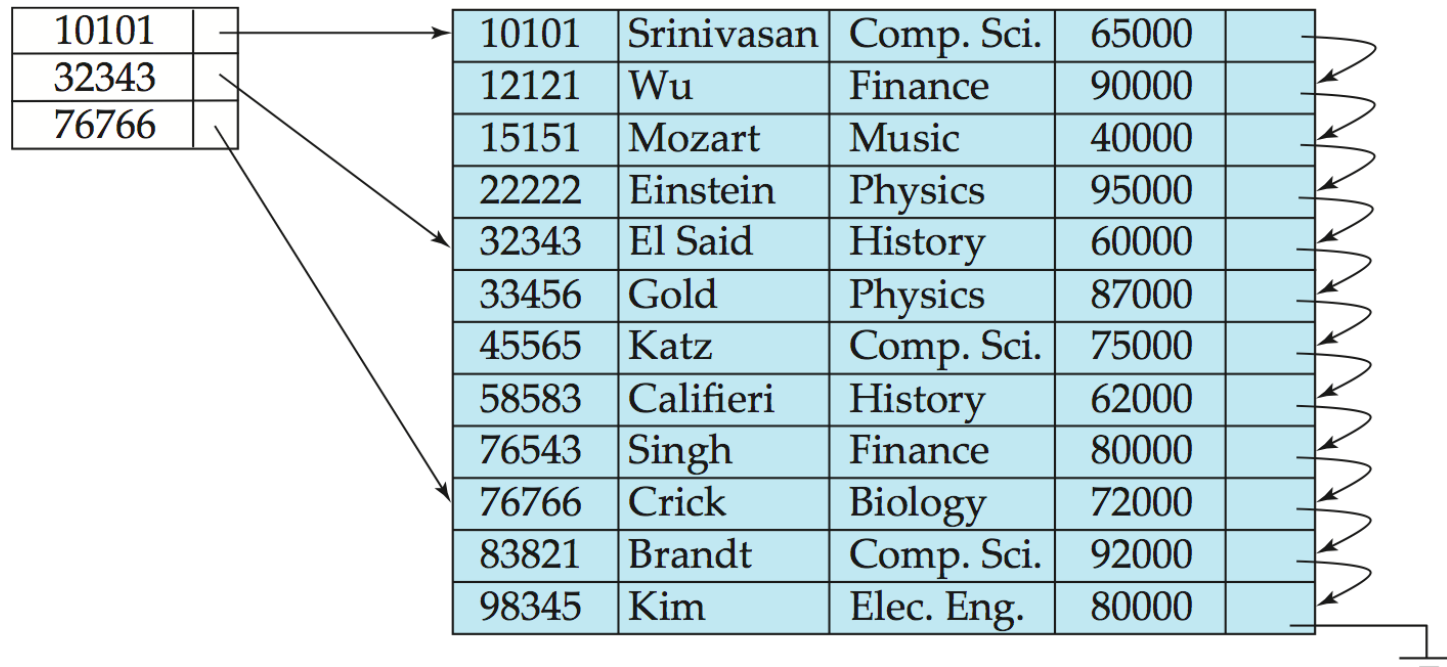
- Dense index on *dept\_name*, with *instructor* file sorted on *dept\_name*

Biology		76766	Crick	Biology	72000	
Comp. Sci.		10101	Srinivasan	Comp. Sci.	65000	
Elec. Eng.		45565	Katz	Comp. Sci.	75000	
Finance		83821	Brandt	Comp. Sci.	92000	
History		98345	Kim	Elec. Eng.	80000	
Music		12121	Wu	Finance	90000	
Physics		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	



# Sparse Index Files

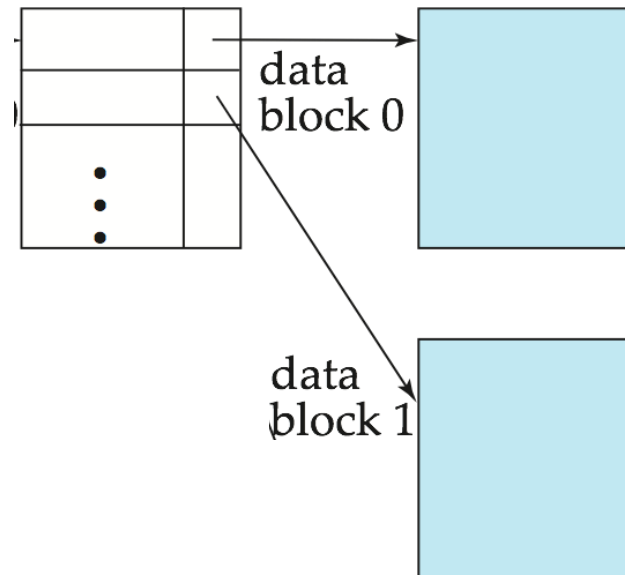
- **Sparse Index:** contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value  $K$  we:
  - Find index record with largest search-key value  $< K$
  - Search file sequentially starting at the record to which the index record points



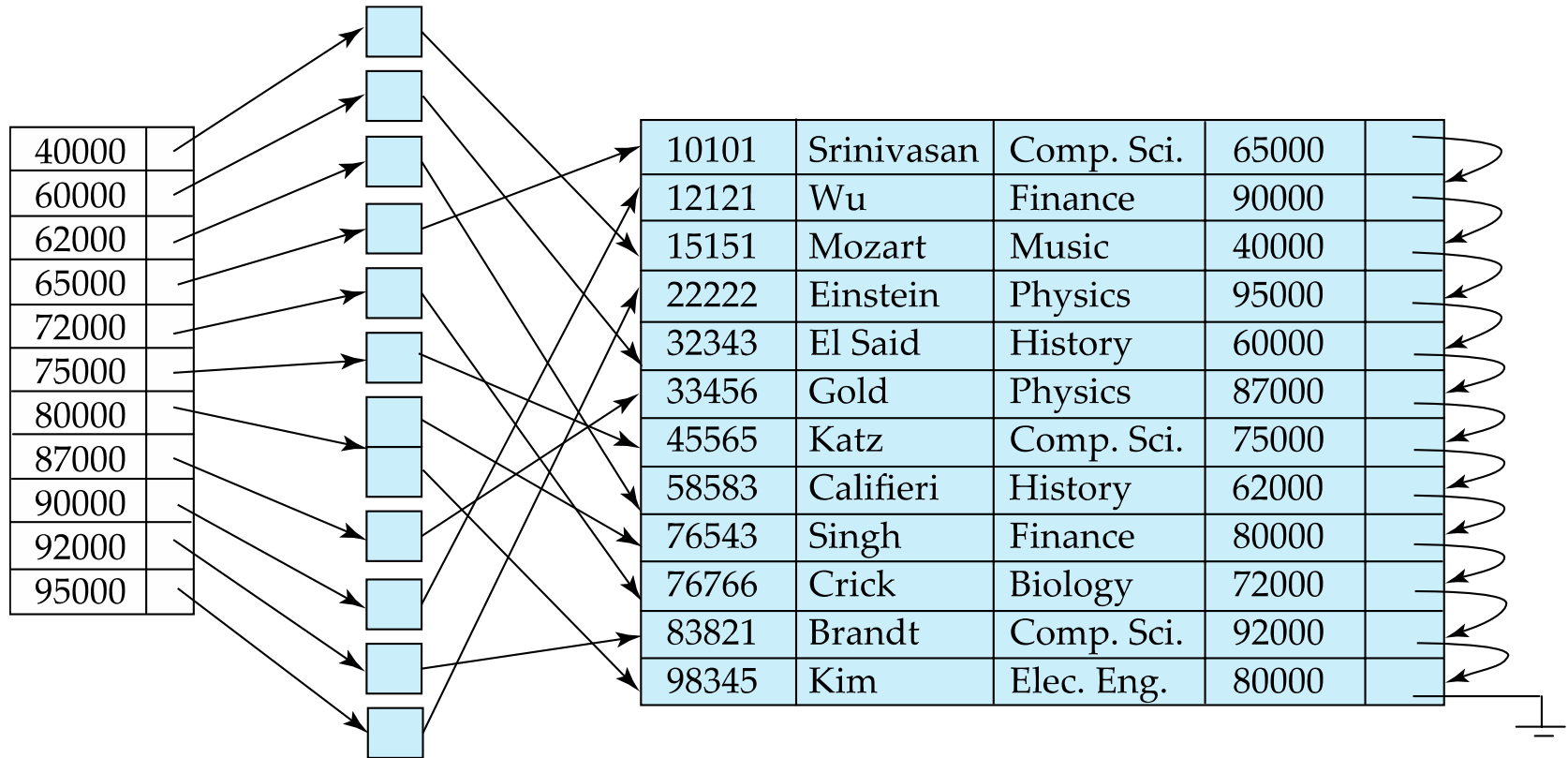


# Cont...

- 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



# Secondary Indices



## Secondary index on *salary* field of *instructor*

- Index record (based on salary) **points to a bucket** that **contains pointers** to all the actual records (w.r.t. search key) with that particular search-key value.
- Secondary indices **must 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 as
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access
- Secondary indices **improve the performance** of queries **that use keys other than the search key** of the clustering index (i.e. primary index)

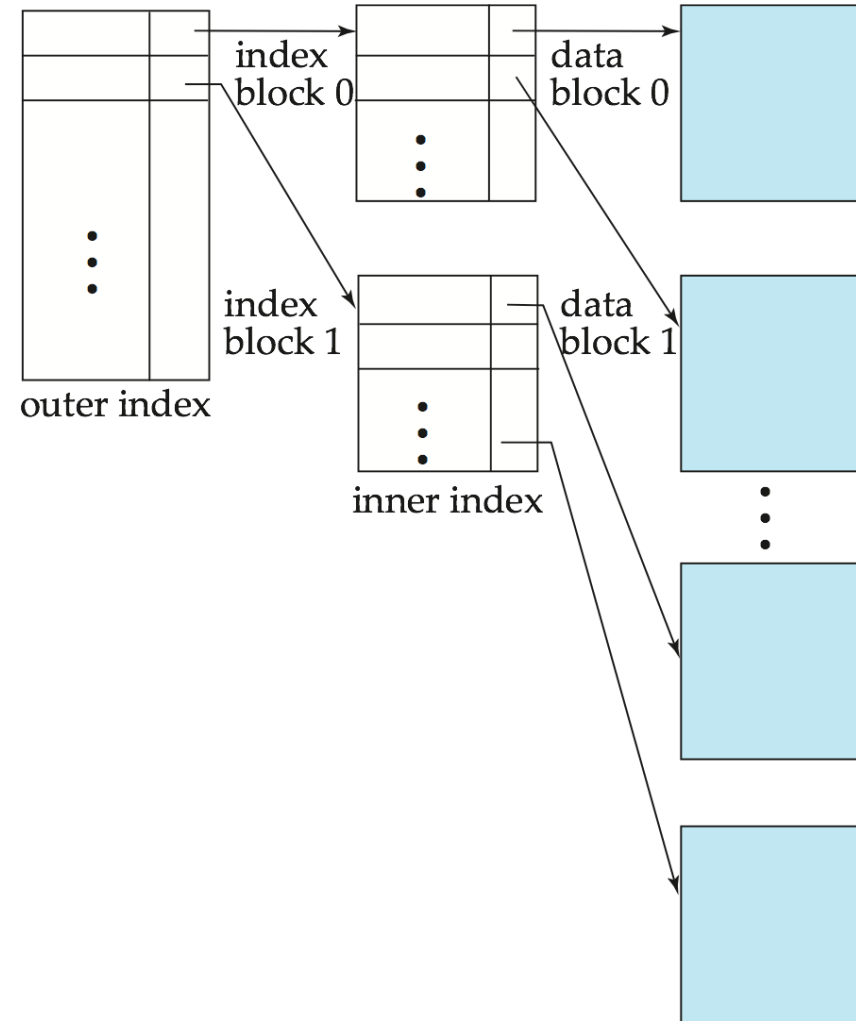
# Cont...



- Suppose we build a **dense index** on a relation with 10,00,00,000 tuples
- Let us assume that 100 index entries fit on a 4 KB block.
- Thus, our **index occupies 10,00,000 blocks = 4 GB !**
- When the **index is so large** that not all of it can be kept in memory, index blocks must be fetched from disk when required.
- The search for an entry in the index then requires several disk-block reads.
- **Solution:** Binary Search on Index
  - Binary search can be used on the index file to locate an entry
  - 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, we would be able to carry out only 7 index searches a second
  - **Note** that if **overflow blocks** have been used for inserting Index, binary search is not possible as the indices are not contiguous and are not in same block.
  - In this case, sequential search is used.

# Multilevel Index

- When 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.



# Index Update: Deletion

10101	
32343	
76766	

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

- 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 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

# **B<sup>+</sup>-Tree Indices for B<sup>+</sup>-Tree Index File**

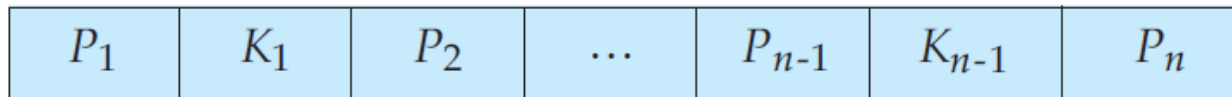


# B+-Tree Index Files

- Main **disadvantage** of the **index-sequential file** organization
  - Performance degrades as the file grows
    - both for index lookups and for sequential scans through the data
- This degradation can be remedied **by reorganization** of the file
  - But, frequent reorganizations are undesirable
- **Solution:** B<sup>+</sup>-tree index structure
  - B<sup>+</sup>-tree index takes the form of a **balanced tree**
  - Each nonleaf node in an **n-ary** B<sup>+</sup>-tree has between  $\lceil n/2 \rceil$  and  $n$  children
  - But, imposes **performance overhead** on insertion and deletion,
  - And, adds **space overhead**
  - **Advantage** that the cost of file re-organization is avoided.

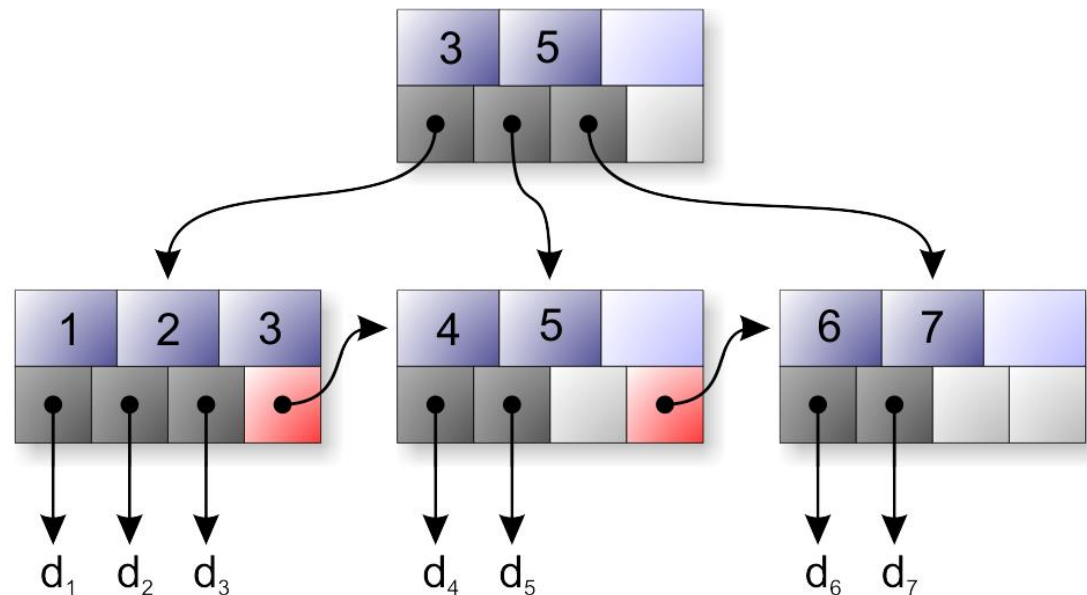
# Structure of a B<sup>+</sup>-Tree

- B<sup>+</sup>-tree index is a **multilevel index**, but it has a structure that **differs from** that of the **multilevel index-sequential file**.



**Figure 11.7** Typical node of a B<sup>+</sup>-tree.

- A simple B<sup>+</sup>-tree example linking the keys 1-7 to data values  $d_1$ - $d_7$ .
- Tree is **balanced**.
- Keys are **sorted**
- The **linked list** (at leaf level) allows **rapid in-order traversal**.
- This particular tree's **branching factor is 4**.



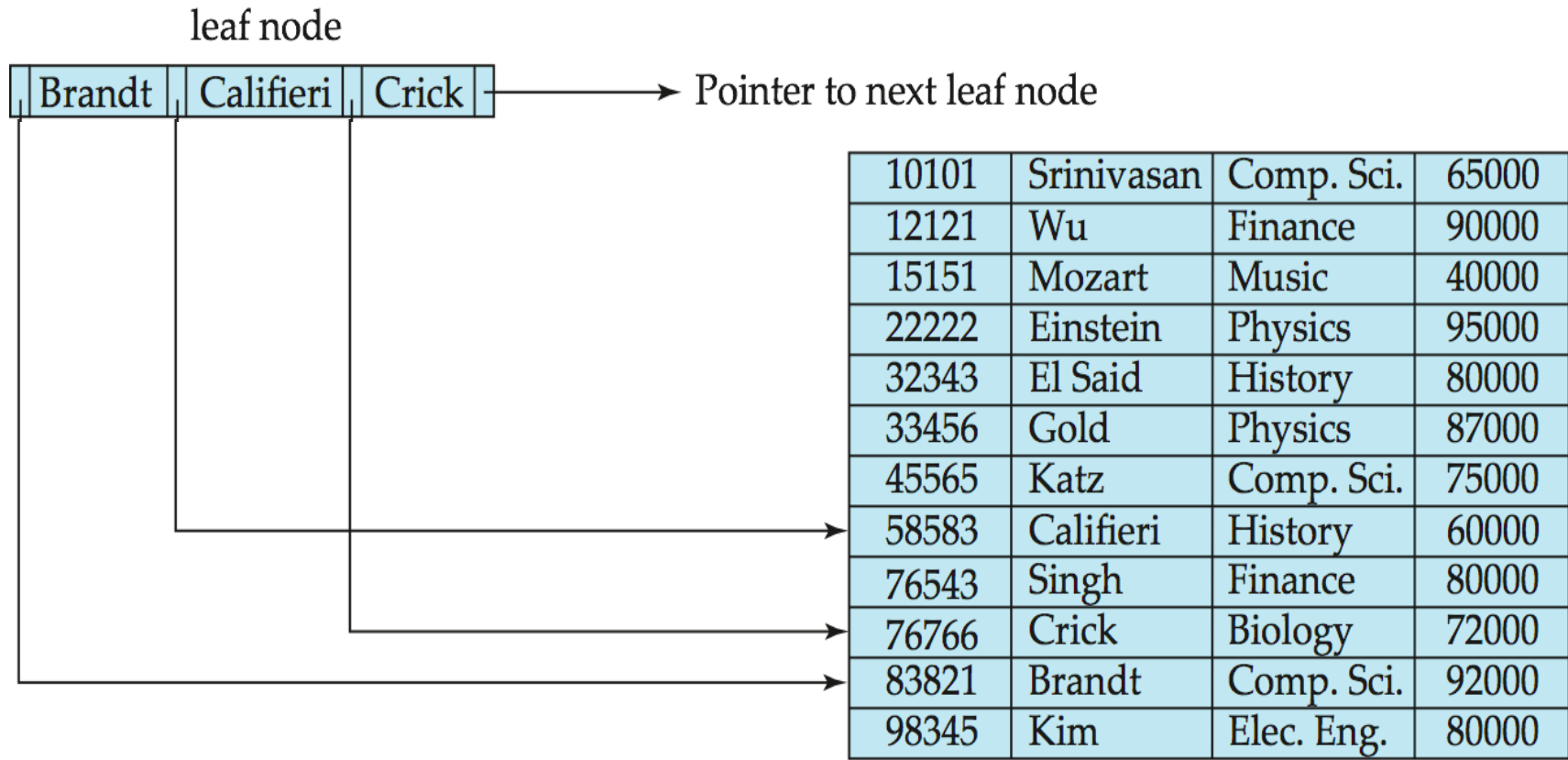
# Cont...



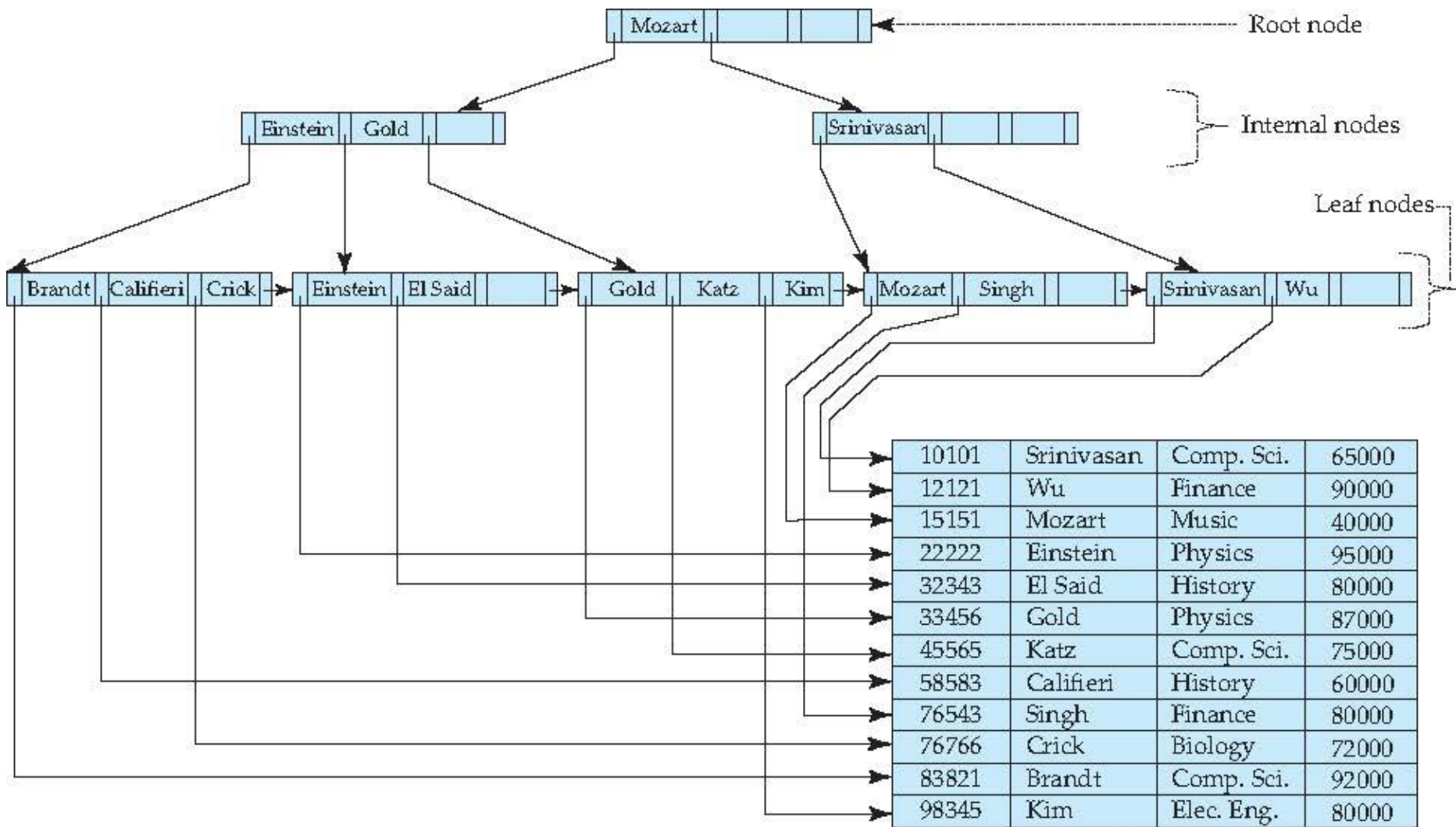
- **Leaf Node:**
  - Each leaf can hold **up to**  $(n-1)$  values.
  - Leaf nodes contain **as few as**  $\text{ceiling}[(n-1)/2]$  values.
  - The ranges of values in each leaf do not overlap, except if there are duplicate search-key values
- **Nonleaf Node**
  - form a multilevel (sparse) index on the leaf nodes
  - nonleaf nodes is the same as that for leaf nodes, **except that** all pointers are pointers to tree nodes
  - A nonleaf node may hold **up to**  $n$  pointers, and must hold **at least**  $\text{ceiling}[n/2]$  pointers
  - The number of pointers in a node is called the **fanout** of the node.
  - Nonleaf nodes are also referred to as internal nodes.
- **Root Node**
  - Unlike other nonleaf nodes, the root node can hold **fewer than**  $n/2$  pointers
  - however, it must hold **at least** two pointers, unless the tree consists of only one node.

# Cont...

- Let the **search key** is *name* of the *instructor* relation file;
- A leaf node is shown below



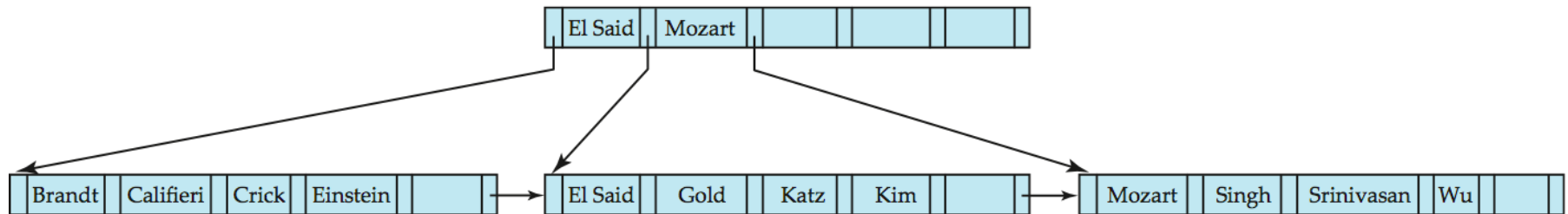
# B<sup>+</sup>-Tree for Instructor



B<sup>+</sup>-tree for *instructor* file ( $n = 4$ )

# Cont..

- The modified B<sup>+</sup>-Tree for Instructor **when n=6**



B<sup>+</sup>-tree for *instructor* file ( $n = 6$ )

- Observe that the **height of this tree is less** than that of the previous tree, which had  $n = 4$ .
- B<sup>+</sup>-trees are all balanced i.e., the length of every path from the root to a leaf node is the same
- the balance property ensures **good performance** for **lookup**, **insertion**, and **deletion**.

# Queries on B<sup>+</sup>-Tree

- we wish to find records with a search-key value of  $V$
- Starting with the root as the current node, the function repeats the following steps until a leaf node is reached.
- The current node is examined, looking for the smallest  $i$  such that search-key value  $K_i \geq V$ .
  - Suppose such value  $K_i$  is found; then
    - if  $K_i = V$ , the current node is set to the node pointed to by  $P_{i+1}$ ,
    - otherwise  $K_i > V$ , and the current node is set to the node pointed to by  $P_i$
  - If no such value  $K_i$  is found, then
    - clearly  $V > K_{m-1}$ , where  $P_m$  is the last nonnull pointer in the node.
    - the current node is set to that pointed to by  $P_m$ .
- The above procedure is repeated until a leaf node is reached.

# Cont...

- At the **leaf node**,
  - let  $K_i$  be the first **search-key**  $= V$ 
    - pointer  $P_i$  directs us to a record with search-key value  $K_i$ .
    - then returns the leaf node  $L$  and the index  $i$ .
  - If **not found**
    - no record with key value  $V$  exists in the relation
    - return failure status
  - If there is **at most one record** with a search key value  $V$ , corresponding pointer retrieve the record and is done
  - If there are **more than one matching record**, the remaining records also need to be fetched
  - How to find all other records with search-key value  $V$ ?
    - If node  $L$  contains at least one search-key value greater than  $V$ , then there are no more records matching  $V$ .
    - Otherwise, the next leaf, pointed to by  $P_n$  may contain further entries for  $V$ . The node pointed to by  $P_n$  must then be searched.



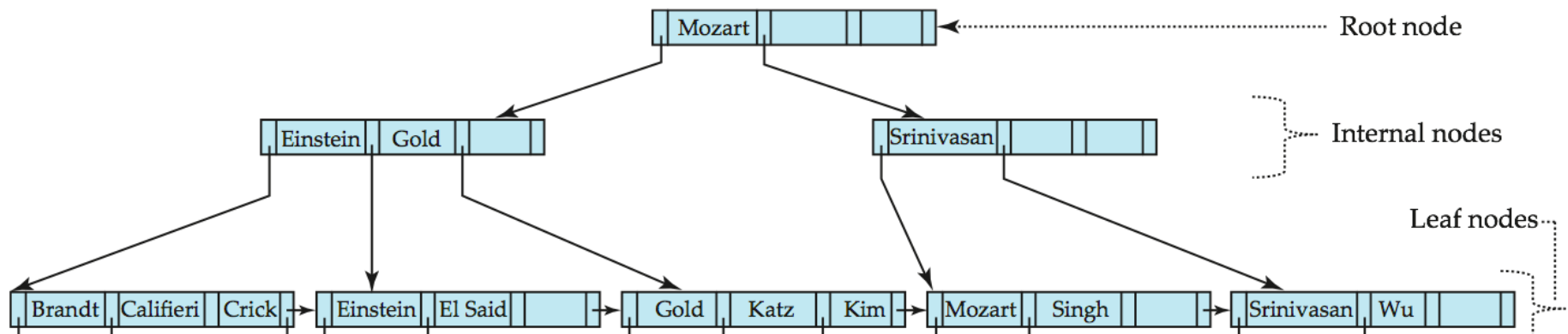
# B<sup>+</sup>-Tree v/s Balanced Binary Search Tree



- In **processing a query**, we **traverse a path** in the tree **from the root to some leaf node**.
- If there are  $N$  records in the file, the path is no longer than  $\lceil \log_{\lceil n/2 \rceil}(N) \rceil$ .
- In practice, only a few nodes need to be accessed.
- Typically, a node is made to be the same size as a disk block, which is typically 4 kilobytes
- With a search-key size of 32 bytes, and a disk-pointer size of 8 bytes,  $n$  is around 100 in a **B<sup>+</sup>-Tree**.
- With  $n = 100$ , if we have 1 million search-key values in the file, a lookup requires only  $\log_{50}(1,000,000) = 4$  nodes to be accessed.
- Thus, **at most 4 blocks** need to be read from disk for the lookup.
- In a **balanced binary tree**, the path for a lookup can be of length  $\lceil \log_2(N) \rceil$ .
- In the previous example, a balanced binary tree requires around **20 node accesses**.

# Insertion into B<sup>+</sup>-Tree

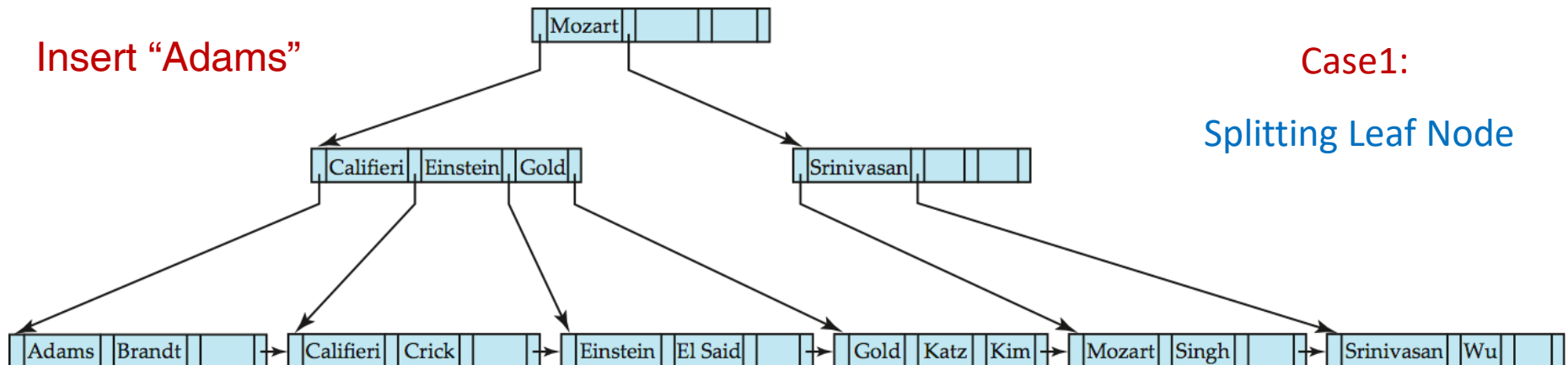
- We first **find the leaf node** in which the search-key value would appear
- We then **insert an entry** (that is, a search-key value and record pointer pair) in the leaf node, positioning it such that the **search-keys are still in order**.
- Maintain **n-ary** property using **node splitting**.



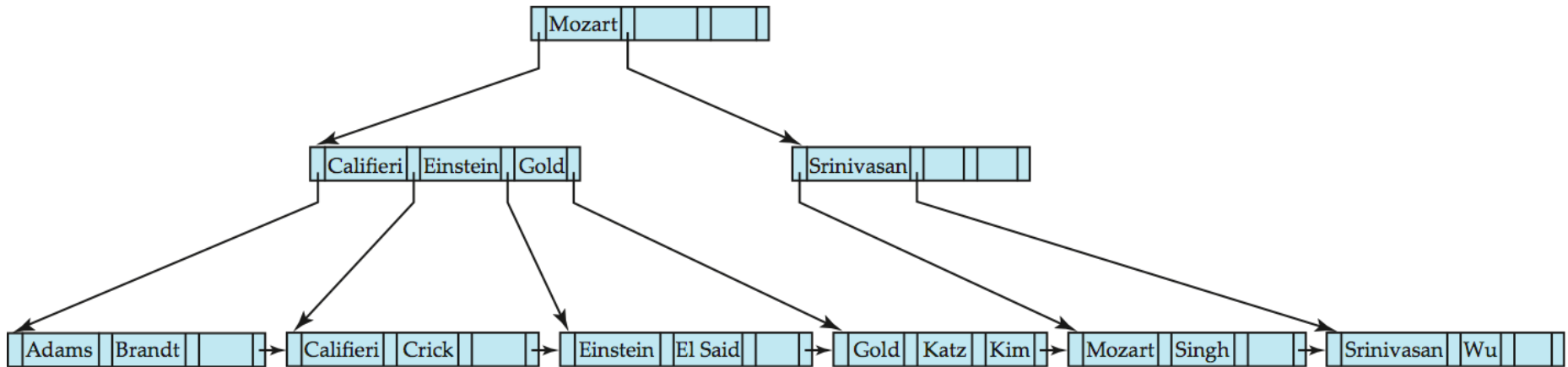
Insert "Adams"

Case1:

Splitting Leaf Node



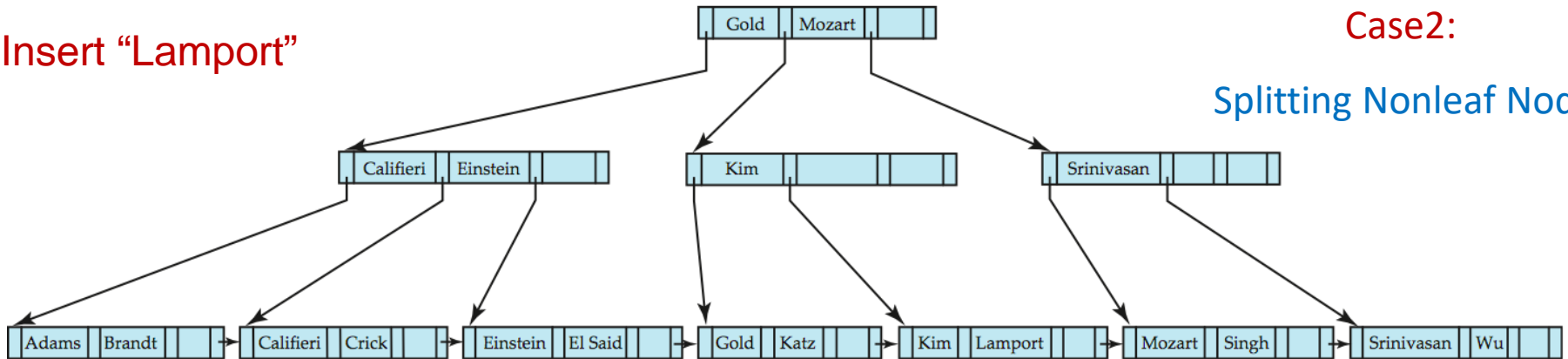
# Insertion into B<sup>+</sup>-Tree



Insert "Lamport"

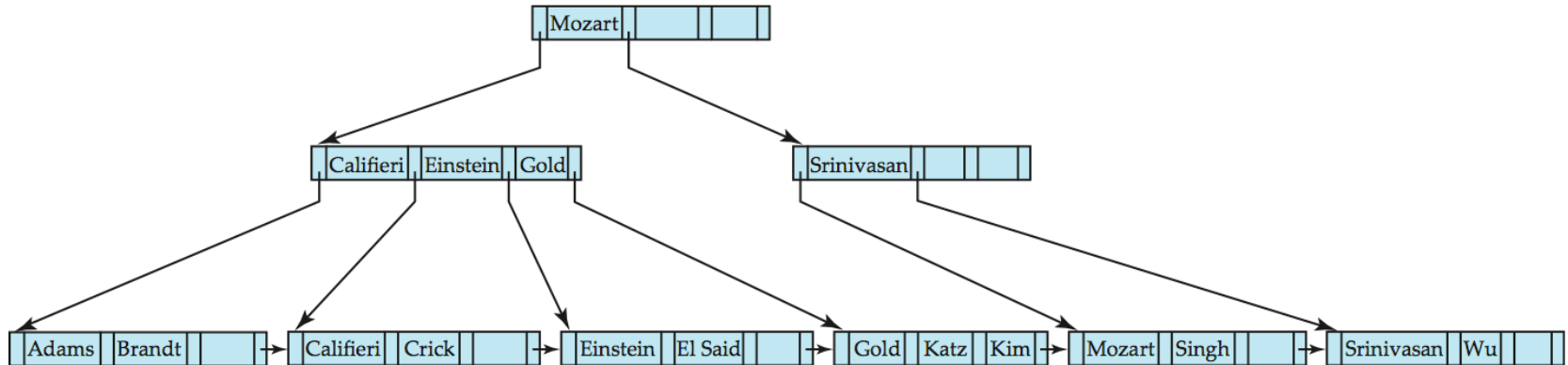
Case2:

Splitting Nonleaf Node

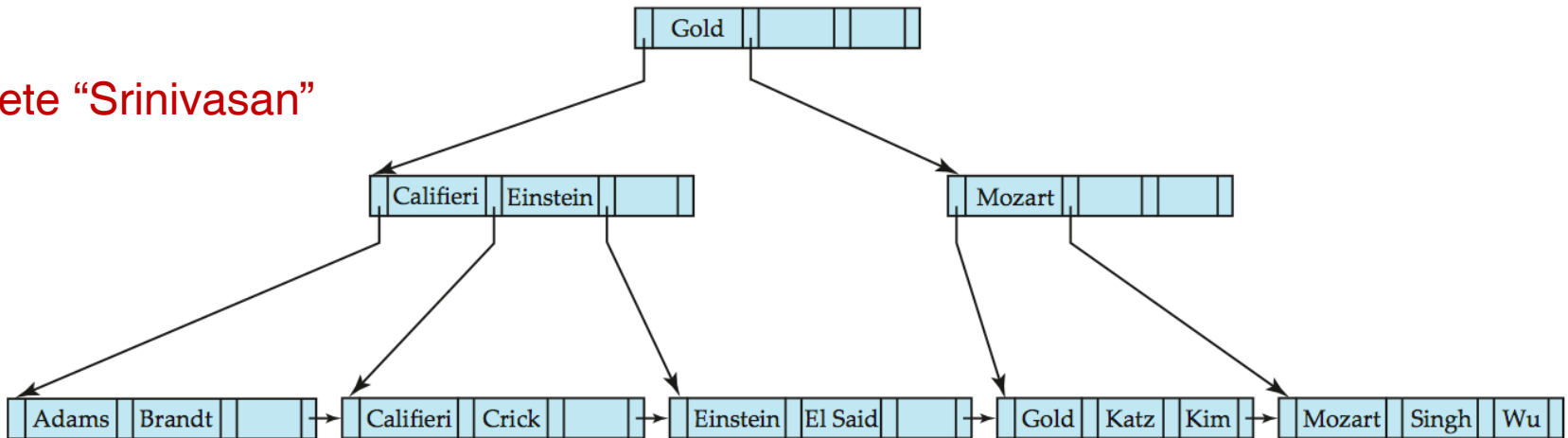


- When an overfull nonleaf node is split, the **child pointers are divided** among the original and the newly created nodes
- In the worst case, all nodes along the path to the root must be split. **If the root itself is split**, the entire tree becomes deeper.

# Deletion from B<sup>+</sup>-Tree



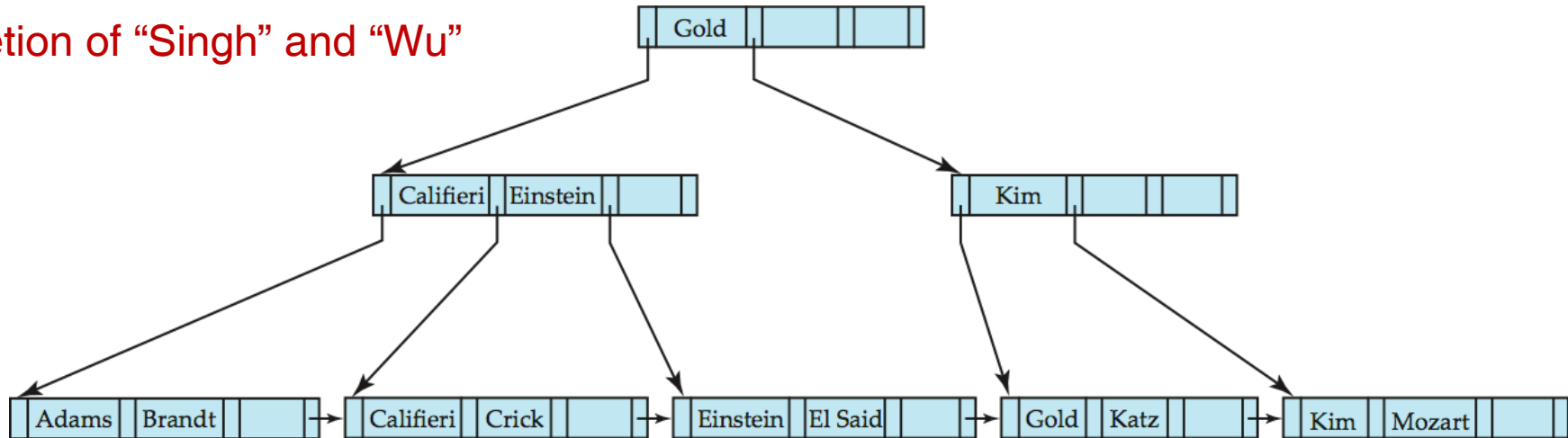
Delete “Srinivasan”



- Deleting “Srinivasan” causes **merging of under-full leaves** as leaf node must hold at least  $\text{ceil}(n-1/2)$  pointers. But “Wu” will have only one pointer.
- Borrow** from sibling node **is not possible** in this case.

# Deletion from B<sup>+</sup>-Tree

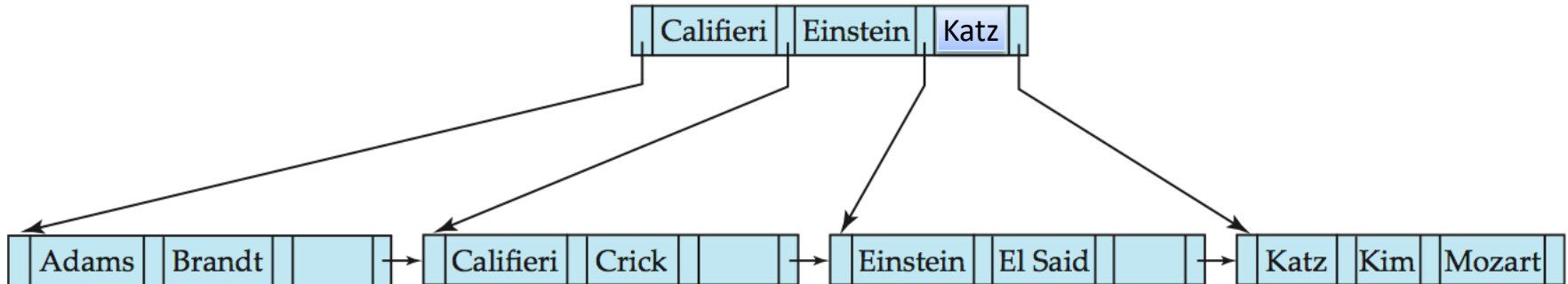
Deletion of “Singh” and “Wu”



- 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

# Deletion from B<sup>+</sup>-Tree

## Deletion of “Gold”



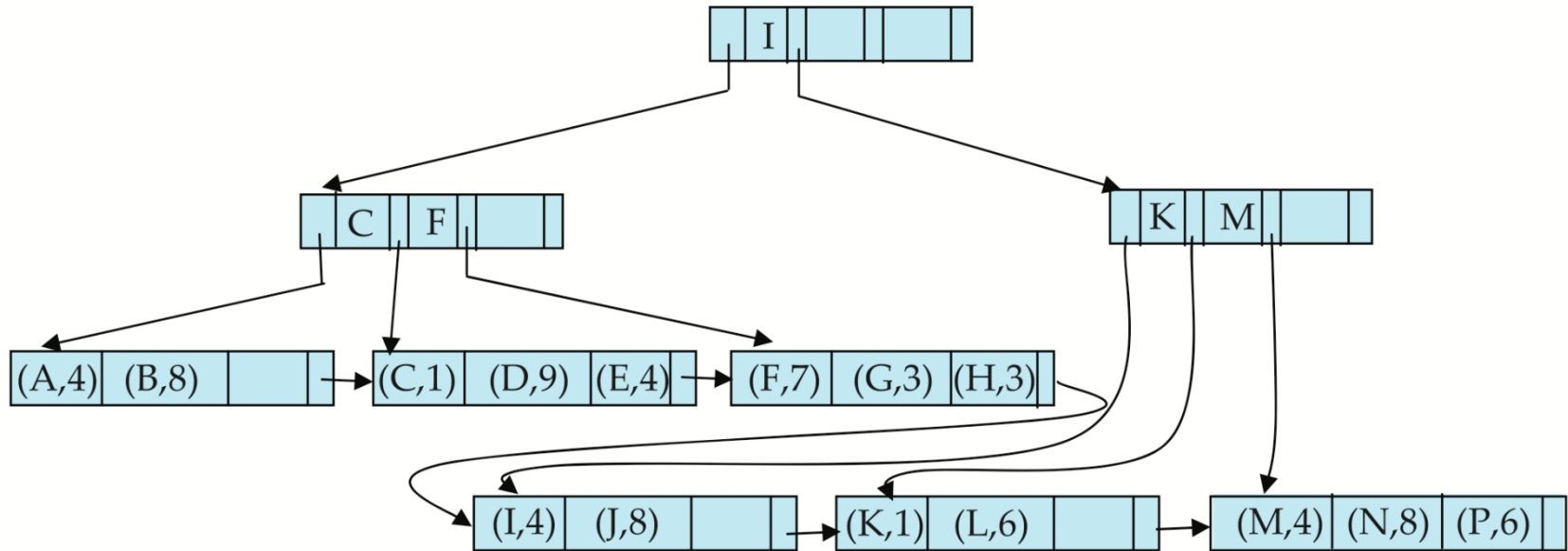
- Leaf Node with “Gold, Katz” became **underfull**, and was **merged with its sibling**
- Parent node becomes **underfull**, and is merged with its sibling
  - Value separating two nodes (at the parent) is **pulled down** when merging
- Root node then has only one child, and is deleted

# B<sup>+</sup>-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**.
- We solve the degradation of index lookups by using B<sup>+</sup>-tree indices on the file.
- We solve the degradation problem for storing the actual records by **using the leaf level** of the B<sup>+</sup>-tree **to organize the blocks containing the actual records**.
- We use the B<sup>+</sup>-tree structure **not only as an index**, but **also as an organizer for records** in a file.
- In a **B<sup>+</sup>-tree file organization**, the **leaf nodes of the tree store records**, instead of storing pointers to records.
- Since records are usually 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** of records from a **B<sup>+</sup>-tree file organization** are handled in the same way as insertion and deletion of entries in a B<sup>+</sup>-tree index.

# Cont...



Example of B<sup>+</sup>-tree File Organization

- Good **space utilization is important** since records use more space than pointers.



# Hash Indexing

## Hash File Organization

# Static Hashing



- The main **drawback** of **sequential file organization** is that
  - we must **access an index structure** to locate data,
  - or must **use binary search**,
  - 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**.
- We shall use the term **bucket** to denote a unit of storage that can store one or more records.
- A **bucket** is typically a disk **block**.
- Formally, let  $K$  denote the set of **all search-key values**, and let  $B$  denote the set of **all bucket addresses**.
- A **hash function**  $h$  is a function from  $K$  to  $B$ .

# Example

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

# Hash Function



- Hashing can be used for two different purposes:
  - **hash file organization**
    - obtain the address of the disk block containing a desired record directly by computing a function on the search-key value of the record.
  - **hash index organization**
    - organize the search keys, with their associated pointers, into a hash file structure.
- **Good hash function:** distributes search-key values to buckets effectively, which means
  - The distribution is **uniform**
    - the hash function assigns each bucket the same number of search-key values from the set of all possible search-key values.
  - The distribution is **random**
    - 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

# Example



- Case 1:
  - Let a hash function for the *instructor* file using the search key *dept\_name*.
  - Assume that we decide to have 26 buckets, and we define a hash function that maps names beginning with the *i*-th letter of the alphabet to the *i*-th bucket.
  - it fails to provide a uniform distribution, since we expect more names to begin with such letters as B and R than Q and X, for example. It is not random as well.
- Case 2:
  - suppose that we want a hash function on the search key *salary*.
  - we use a hash function that divides the values into 10 ranges, say \$30,000–\$40,000, \$40,001–\$50,000 and so on.
  - The distribution of search-key values is uniform (since each bucket has the same number of different salary values), but is not random (since salaries between \$60,001 and \$70,000 are far more common than the remaining), and the distribution of records is not uniform

# Handling of Bucket Overflows

- Hence, 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.
- If the mapped bucket does not have enough space, a **bucket overflow** occurs.
  - **Insufficient buckets**
    - The **number of buckets** ( $n_b$ ) must be chosen such that  $n_b > (n_r / f_r)$ ,
    - $n_r$  denotes the **total number of records that will be stored** and
    - $f_r$  denotes the **number of records that will fit in a bucket**.
  - **Skew**
    - Some buckets are assigned more records than are others, so a bucket may overflow even when other buckets still have space.
    - Reasons: (1) Multiple records may have the same search key. (2) The chosen hash function may result in nonuniform distribution of search keys.
- Despite allocation of a few more buckets than required, bucket overflow can still occur. We handle bucket overflow by using **overflow buckets**.
- All the overflow buckets of a given bucket are **chained together in a linked list**, called **overflow chaining**.

# Open and Closed Hashing

- **Closed Hashing:**
  - the set of buckets is **not fixed**;
  - allows **overflow chaining**
  - **Application:** in database systems; as deletion under open hashing is troublesome
- **Open hashing:**
  - the set of **buckets is fixed**;
  - there are **no overflow chains**;
  - One solution: **linear probing** - use the next bucket (in cyclic order) that has space
  - Other Solution: **Rehashing**
- **Application:** used to construct symbol tables for compilers and assemblers

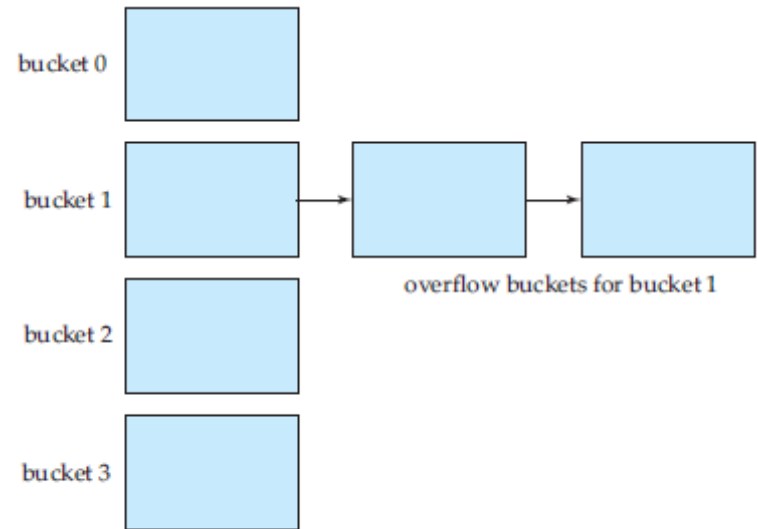
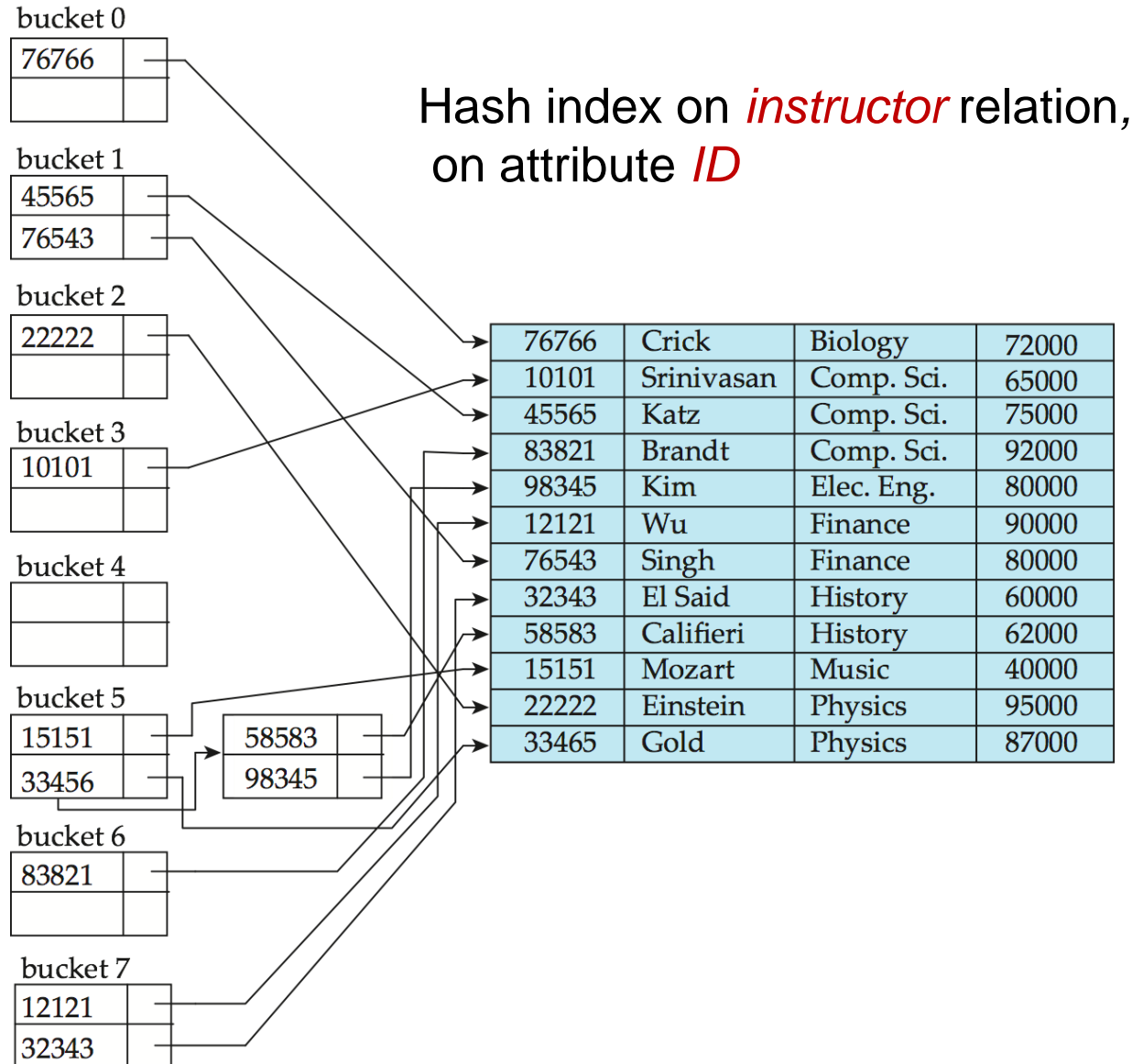


Figure 11.24 Overflow chaining in a hash structure.

# 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**



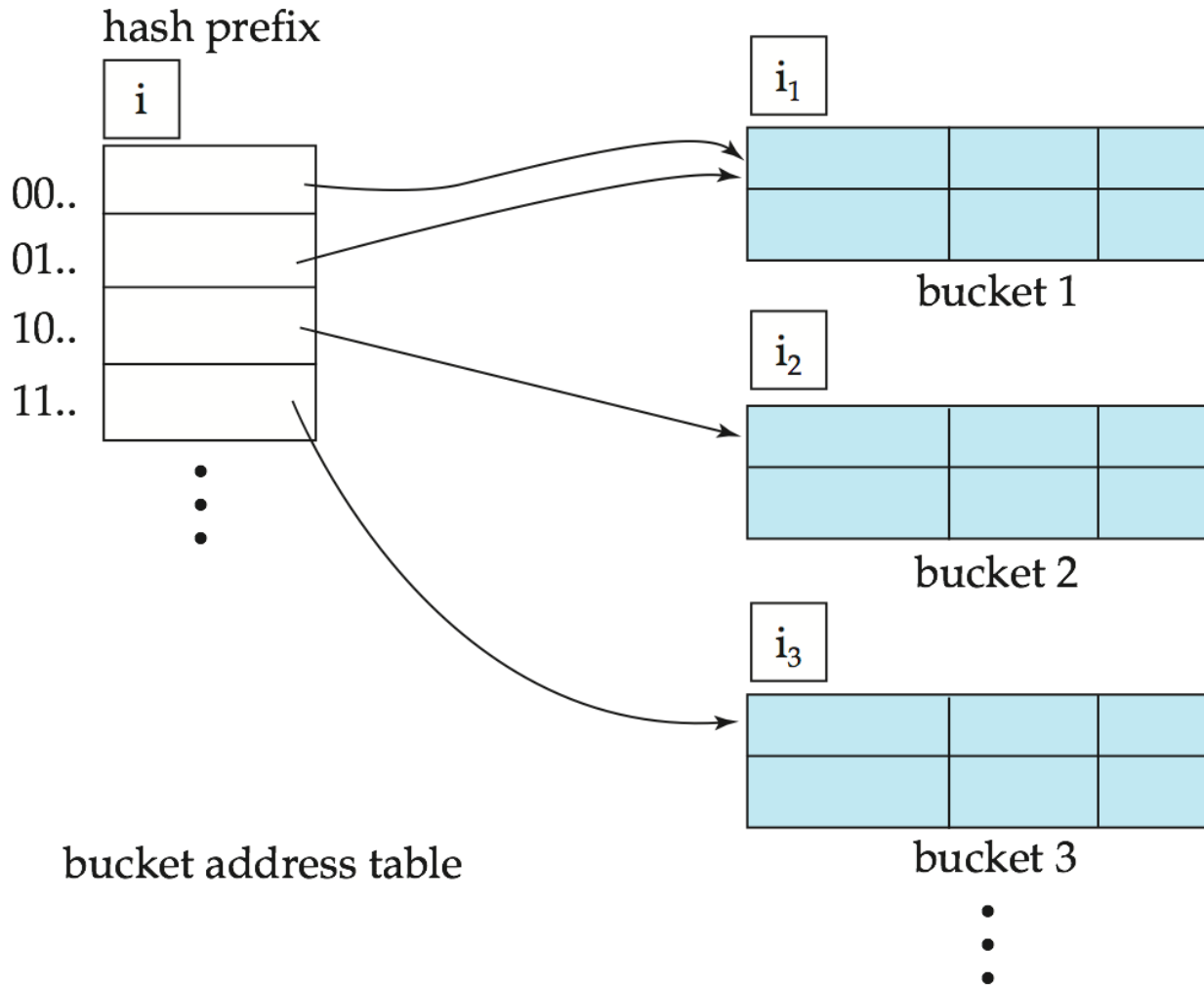
# Deficiencies of Static Hashing

- In static hashing,
  - We must **choose the hash function** when we implement the system, and it cannot be changed easily thereafter
  - Function  $h$  maps search-key values to a **fixed set of  $B$  of bucket** addresses.
  - However, 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 (i.e. **dynamic hashing**)

# Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- Two popular forms of dynamic hashing
  - Extendable hashing
  - Linear hashing
- **Extendable 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,  $0 \leq i \leq 32$ .
    - 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.

# Extendable Hash Structure



- In this structure,  $i_2 = i_3 = i$ ; whereas  $i_1 = i - 1$
- do not create a bucket for each hash value

# Example: for Extendable Hashing



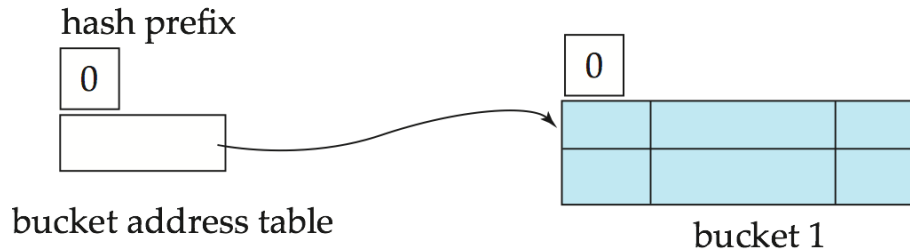
<i>dept_name</i>	$h(\text{dept\_name})$
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

Hash function for *dept\_name*.

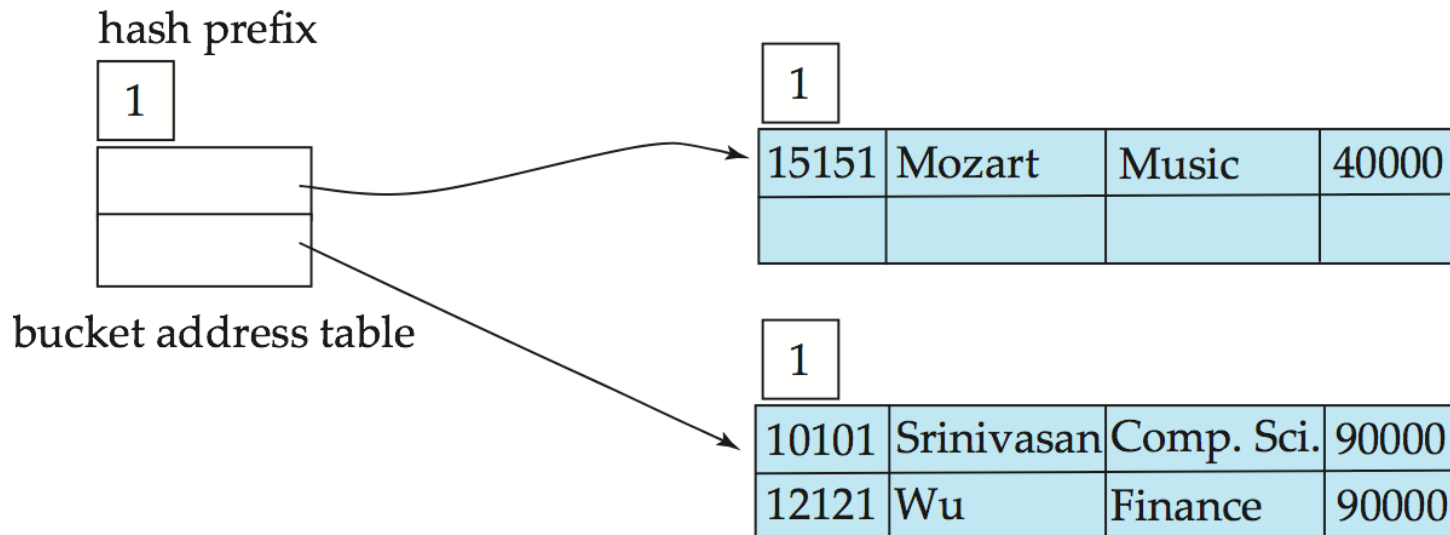
- Operations:
  - **Queries:** To locate the bucket containing search-key value  $K_i$ , the system takes the **first  $i$  high-order bits of  $h(K_i)$** , looks at the corresponding table entry for this bit string, and follows the bucket pointer in the table entry.
  - **Insertion**
  - **Deletion**

# Extendable Hashing: Insertion

- Initial Hash structure; Let the allowed bucket size = 2

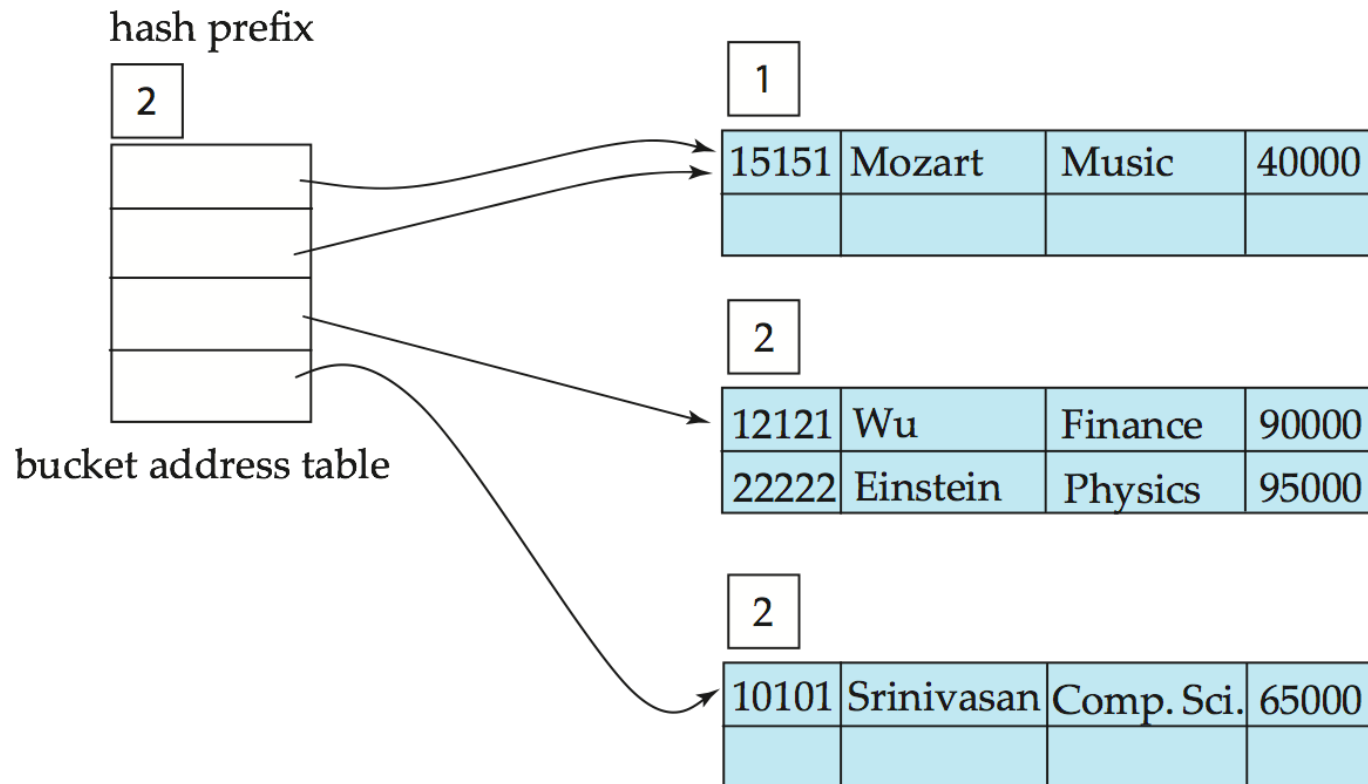


- Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records
- Splitting of record bucket is required



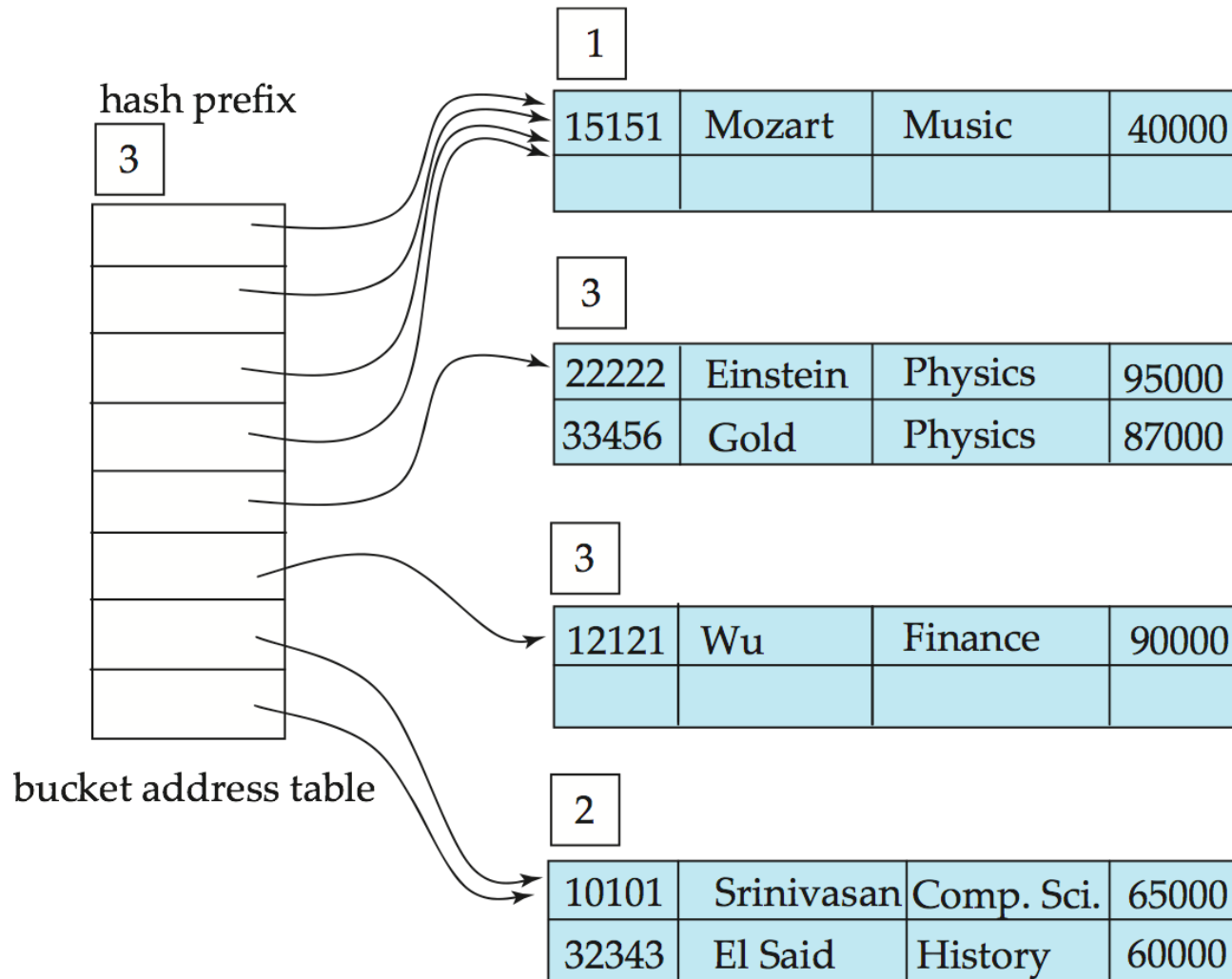
# Extendable Hashing: Insertion

- Hash structure after **insertion** of “Einstein” record
- Again, **Splitting** is required



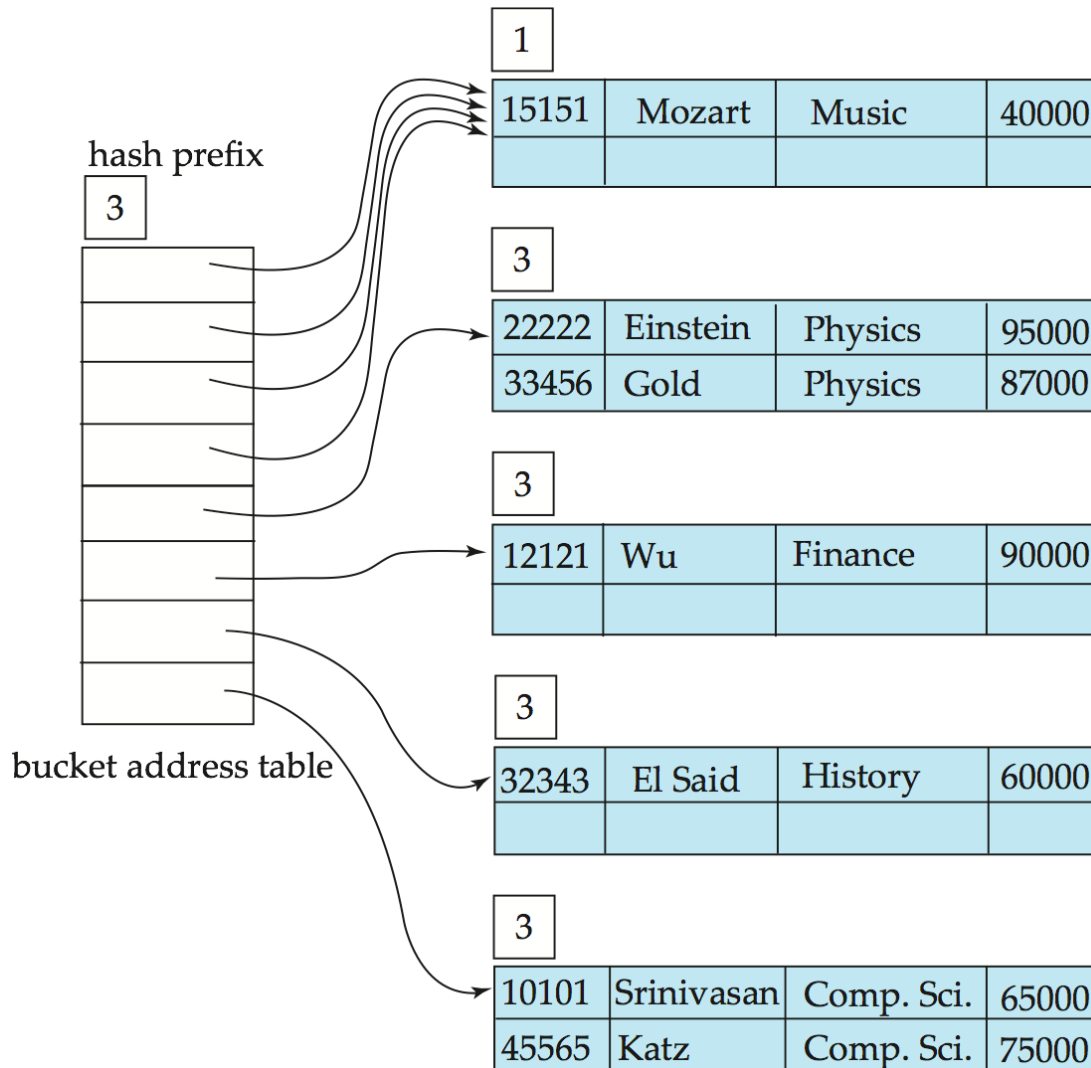
# Extendable Hashing: Insertion

- Hash structure after insertion of “Gold” and “El Said” records



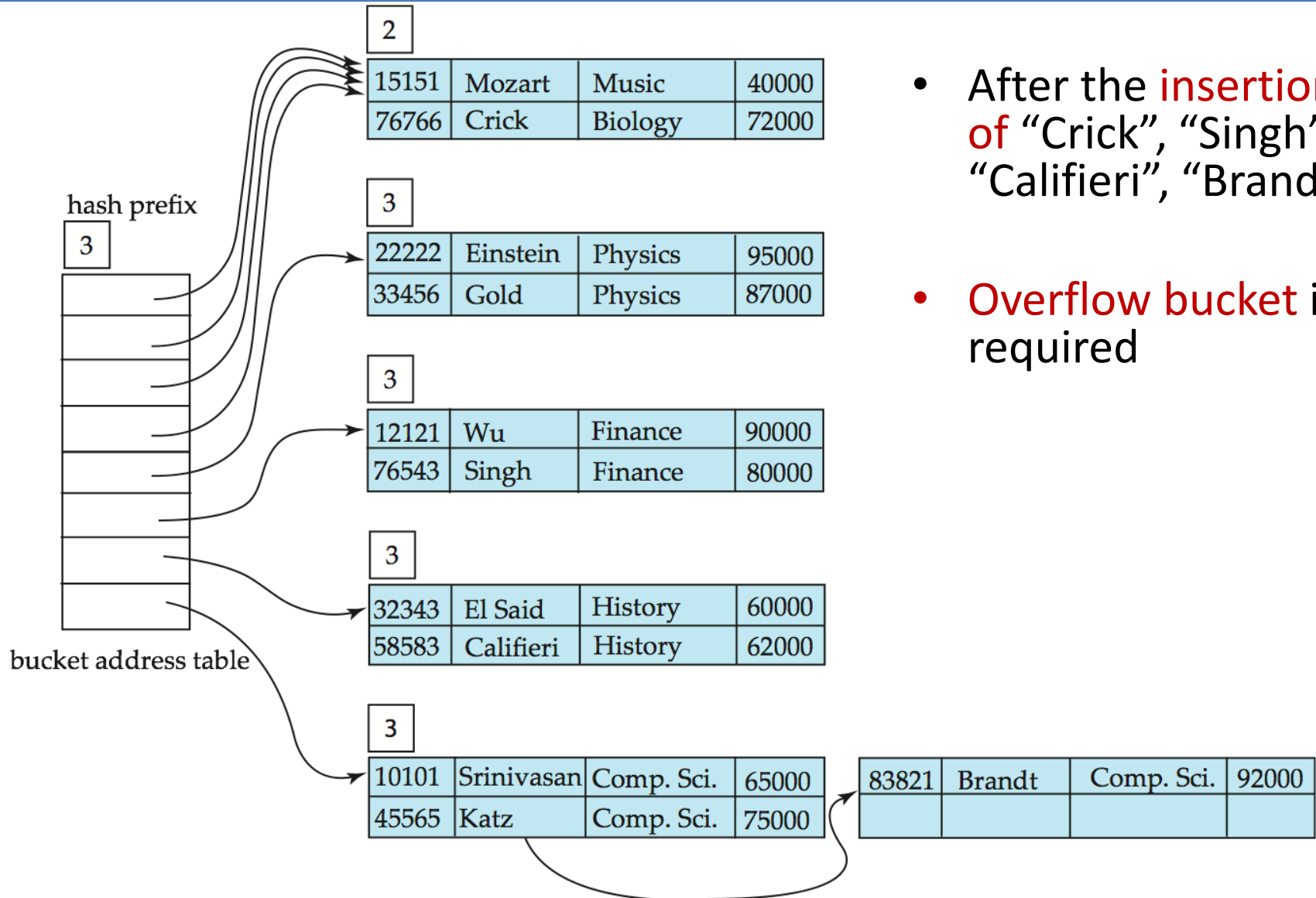
# Extendable Hashing: Insertion

- Hash structure after insertion of “Katz” record

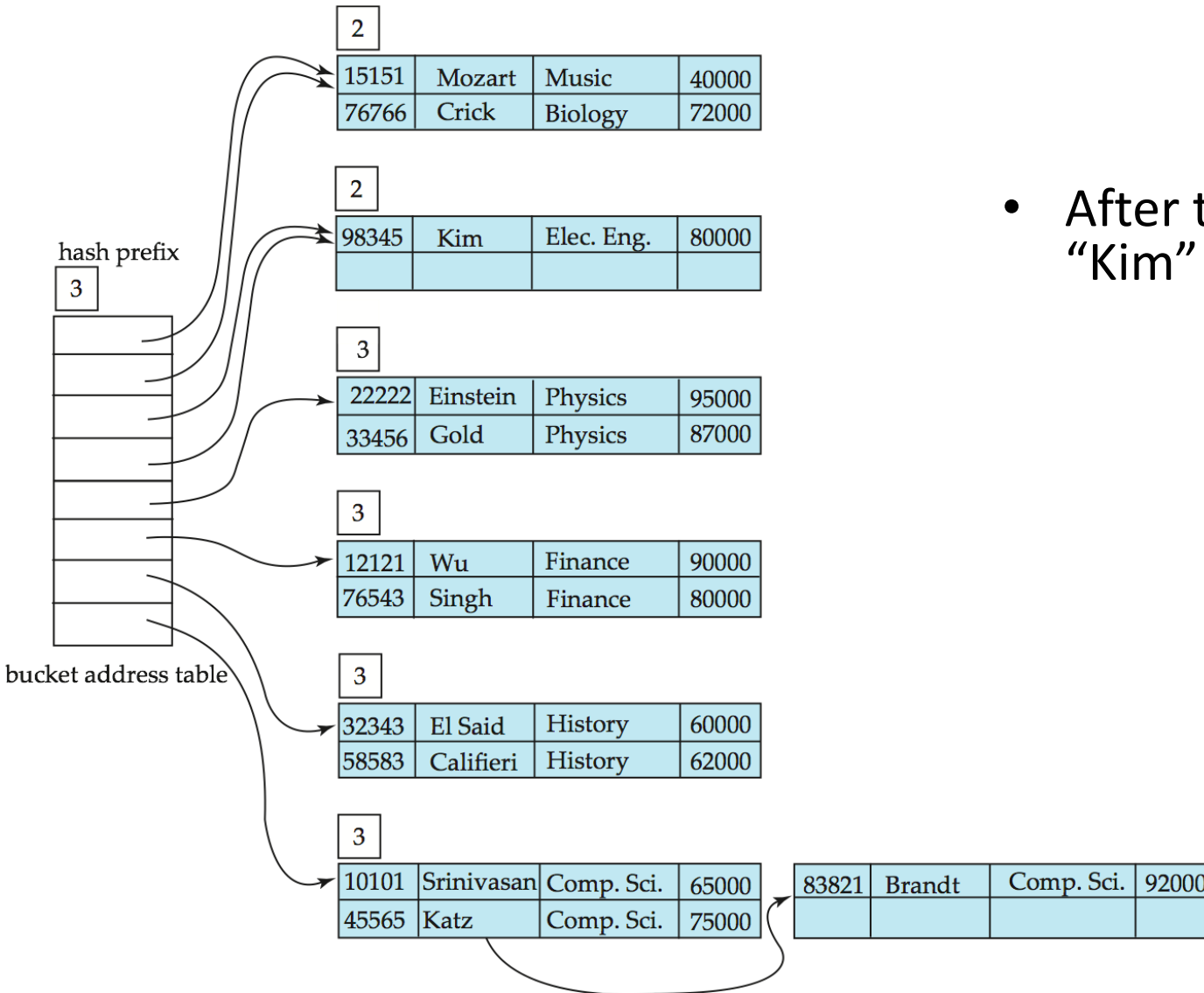




# Extendable Hashing: Insertion



# Extendable Hashing: Insertion



- After the insertion of “Kim”

# Extendable Hashing: Insertion Rule



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 **re-insert** (in  $j$  or  $z$ )
  - **re-compute** 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.
    - **re-compute** new bucket address table entry for  $K_j$   
Now  $i > i_j$  so use the first case above.

# Extendable Hashing: Deletion Rule



- 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

# Performance Analysis



- **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<sup>+</sup>-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an **expensive operation**
- In practice:
  - **Oracle** supports static hash organization, but not hash indices
  - **SQLServer** supports only B<sup>+</sup>-trees

# Thanks!