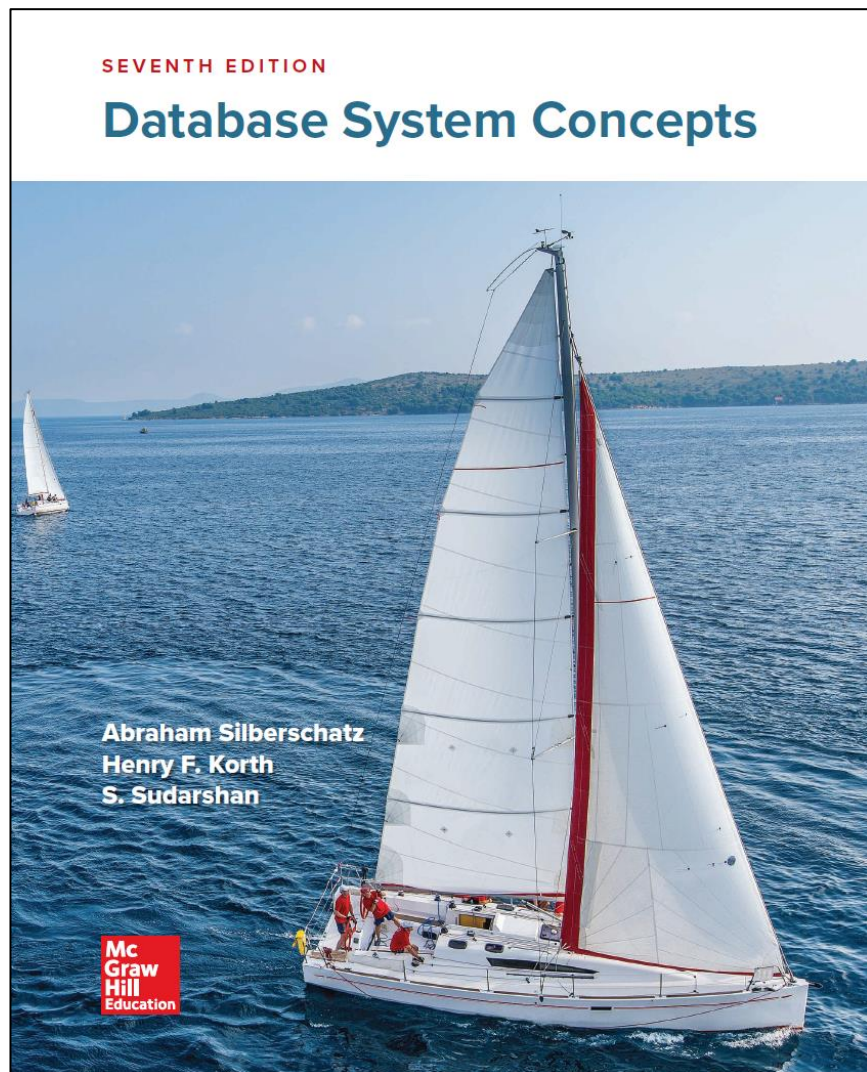


Data Administration in Information Systems

Indexing

Indexing



x Contents

PART FIVE ■ STORAGE MANAGEMENT AND INDEXING

Chapter 12 Physical Storage Systems

12.1 Overview of Physical Storage Media	559	12.6 Disk-Block Access	577
12.2 Storage Interfaces	562	12.7 Summary	580
12.3 Magnetic Disks	563	Exercises	582
12.4 Flash Memory	567	Further Reading	584
12.5 RAID	570		

Chapter 13 Data Storage Structures

13.1 Database Storage Architecture	587	13.7 Storage Organization in Main-Memory	
13.2 File Organization	588	Databases	615
13.3 Organization of Records in Files	595	13.8 Summary	617
13.4 Data-Dictionary Storage	602	Exercises	619
13.5 Database Buffer	604	Further Reading	621
13.6 Column-Oriented Storage	611		

Chapter 14 Indexing

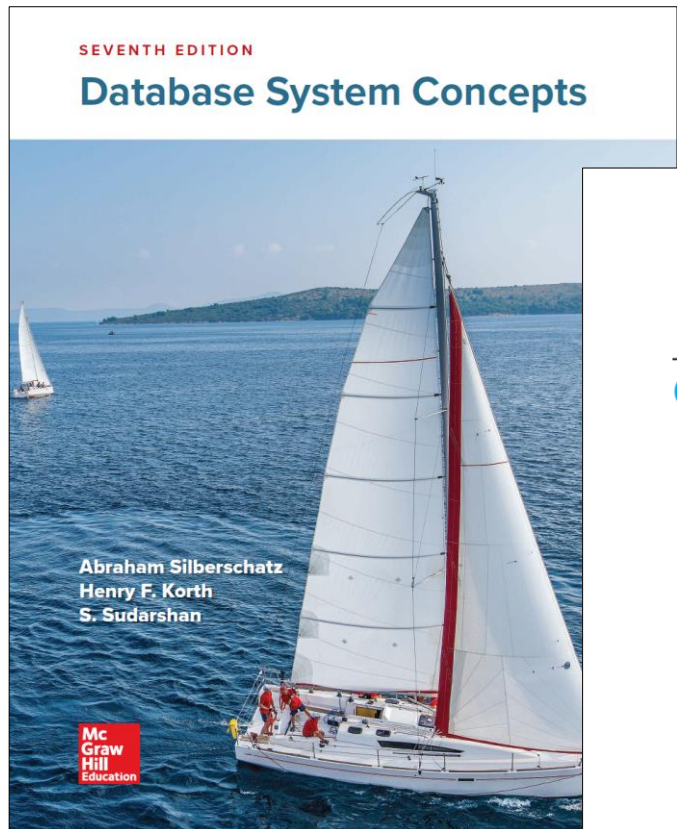
14.1 Basic Concepts	623	14.8 Write-Optimized Index Structures	665
14.2 Ordered Indices	625	14.9 Bitmap Indices	670
14.3 B*-Tree Index Files	634	14.10 Indexing of Spatial and Temporal Data	672
14.4 B*-Tree Extensions	650	14.11 Summary	677
14.5 Hash Indices	658	Exercises	679
14.6 Multiple-Key Access	661	Further Reading	683
14.7 Creation of Indices	664		

PART SIX ■ QUERY PROCESSING AND OPTIMIZATION

Chapter 15 Query Processing

15.1 Overview	689	15.7 Evaluation of Expressions	724
15.2 Measures of Query Cost	692	15.8 Query Processing in Memory	731
15.3 Selection Operation	695	15.9 Summary	734
15.4 Sorting	701	Exercises	736
15.5 Join Operation	704	Further Reading	740
15.6 Other Operations	719		

Indexing



Contents

Chapter 1 Introduction

- | | | | |
|----------------------------------|----|--|----|
| 1.1 Database-System Applications | 1 | 1.7 Database and Application Architect | |
| 1.2 Purpose of Database Systems | 5 | 1.8 Database Users and Administrators | |
| 1.3 View of Data | 8 | 1.9 History of Database Systems | 25 |
| 1.4 Database Languages | 13 | 1.10 Summary | 29 |
| 1.5 Database Design | 17 | Exercises | 31 |
| 1.6 Database Engine | 18 | Further Reading | 33 |

PART ONE ■ RELATIONAL LANGUAGES

Chapter 2 Introduction to the Relational Model

- | | | | |
|---------------------------------------|----|----------------------------|----|
| 2.1 Structure of Relational Databases | 37 | 2.6 The Relational Algebra | 48 |
| 2.2 Database Schema | 41 | 2.7 Summary | 58 |
| 2.3 Keys | 43 | Exercises | 60 |
| 2.4 Schema Diagrams | 46 | Further Reading | 63 |
| 2.5 Relational Query Languages | 47 | | |

Chapter 3 Introduction to SQL

- | | | | |
|--|----|----------------------------------|-----|
| 3.1 Overview of the SQL Query Language | 65 | 3.7 Aggregate Functions | 91 |
| 3.2 SQL Data Definition | 66 | 3.8 Nested Subqueries | 98 |
| 3.3 Basic Structure of SQL Queries | 71 | 3.9 Modification of the Database | 108 |
| 3.4 Additional Basic Operations | 79 | 3.10 Summary | 114 |
| 3.5 Set Operations | 85 | Exercises | 115 |
| 3.6 Null Values | 89 | Further Reading | 124 |

Index

- aborted transactions, 805–807, 819–820
- abstraction, 2, 9–12, 15
- acceptors, 1148, 1152
- accessing data. *See also* security from application programs, 16–17
- concurrent-access anomalies, 7
- difficulties in, 6
- indices for, 19
- recovery systems and, 910–912
- types of access, 15
- access paths, 695
- access time
- indices and, 624, 627–628
- query processing and, 692
- storage and, 561, 566, 567, 578
- access types, 624
- account nonces, 1271
- ACID properties. *See* atomicity; consistency; durability; isolation
- Active Server Page (ASP), 405
- active transactions, 806
- ActiveX DataObjects (ADO), 1239
- adaptive lock granularity, 969–970
- add constraint, 146
- ADO (ActiveX DataObjects), 1239
- ADO.NET, 184, 1239
- Advanced Encryption Standard (AES), 448, 449
- advanced SQL, 183–231
- accessing from programming languages, 183–198
- aggregate features, 219–231
- embedded, 197–198
- functions and procedures, 198–206
- JDBC and, 184–193
- ODBC and, 194–197
- Python and, 193–194
- triggers and, 206–213
- advertisement data, 469
- AES (Advanced Encryption Standard), 448, 449
- after triggers, 210
- aggregate functions, 91–96
- basic, 91–92
- with Boolean values, 96
- defined, 91
- with grouping, 92–95
- having clause, 95–96
- with null values, 96
- aggregation
- defined, 277
- entity-relationship (E-R) model and, 276–277
- intraoperation parallelism and, 1049
- on multidimensional data, 527–532
- partial, 1049
- pivoting and, 226–227, 530
- query optimization and, 764
- query processing and, 723
- ranking and, 219–223
- representation of, 279
- rollup and cube, 227–231
- skew and, 1049–1050
- of transactions, 1278
- view maintenance and, 781–782
- windowing and, 223–226
- aggregation operation, 57
- aggregation switch, 977
- airlines, database applications for, 3
- Ajax, 423–426, 1015
- algebraic operations. *See* relational algebra
- aliases, 81, 336, 1242
- all construct, 100
- alter table, 71, 146
- alter trigger, 210
- alter type, 159
- Amidahl's law, 974
- American National Standards Institute (ANSI), 65, 1237
- analysis pass, 944
- analytics. *See* data analytics and connective, 74
- and operation, 89–90
- anonymity, 1252, 1253, 1258, 1259
- ANSI (American National Standards Institute), 65, 1237
- anticipatory standards, 1237
- anti-join operation, 108, 776

Basic Concepts

- Indexing mechanisms used to speed up access to desired data
 - e.g., book catalog in library
- **Search Key** – attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

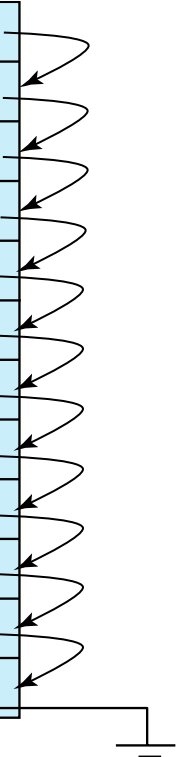
search-key	pointer
------------	---------

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

Index Evaluation Metrics

- Access types supported efficiently, for example:
 - records with a specified value in the attribute
 - records with an attribute value falling in a specified range of values
- Access time
- Insertion time
- Deletion time
- Space overhead

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	

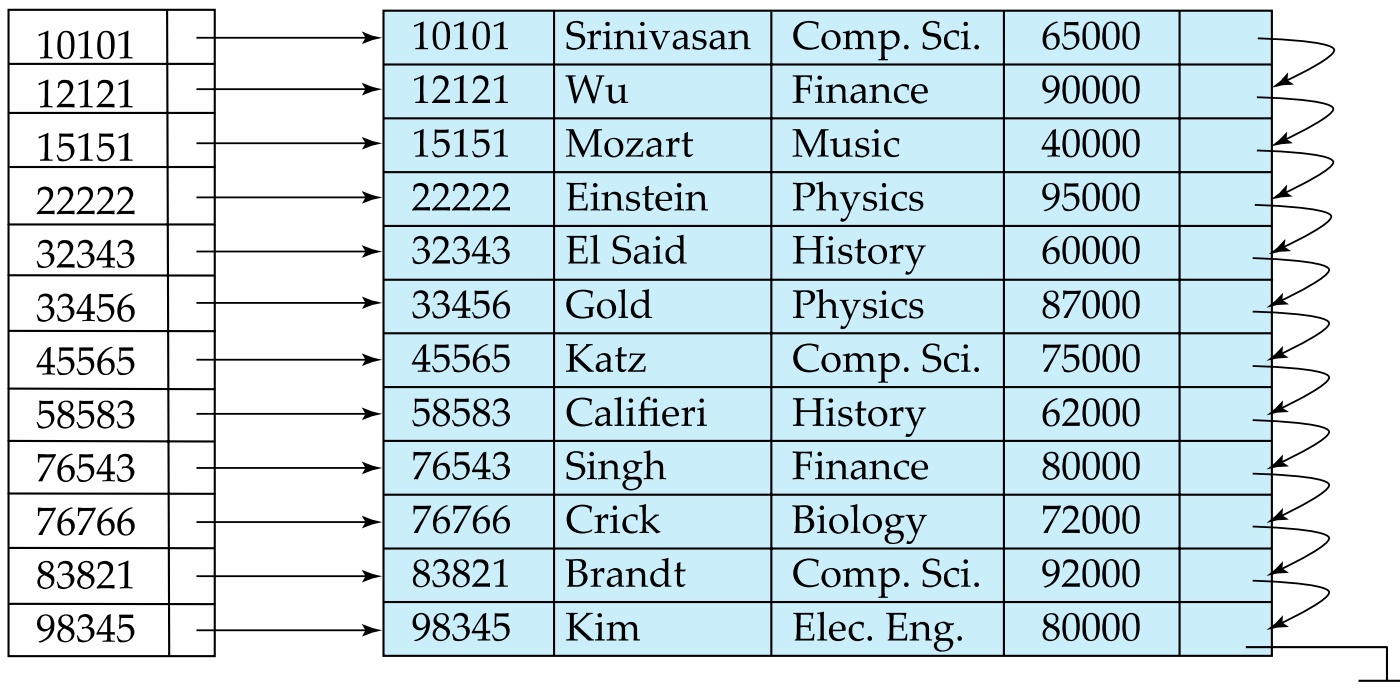


Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value
- **Clustered index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file
 - also called **primary index**
 - the search key of a clustered index is usually (but not necessarily) the primary key.
- **Non-clustered index**: an index whose search key specifies an order different from the sequential order of the file
 - also called **secondary index**
- **Index-sequential file**: sequential file ordered on a search key, with a clustering index on the search key

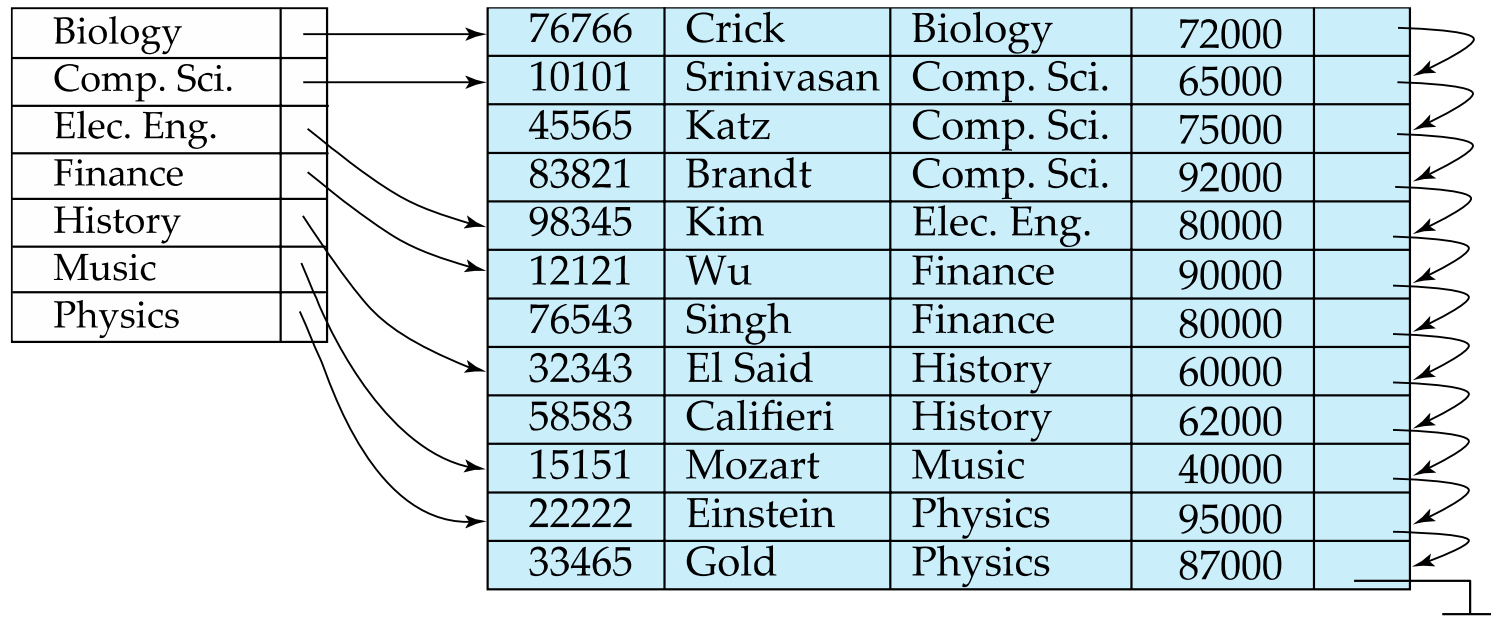
Dense Index Files

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



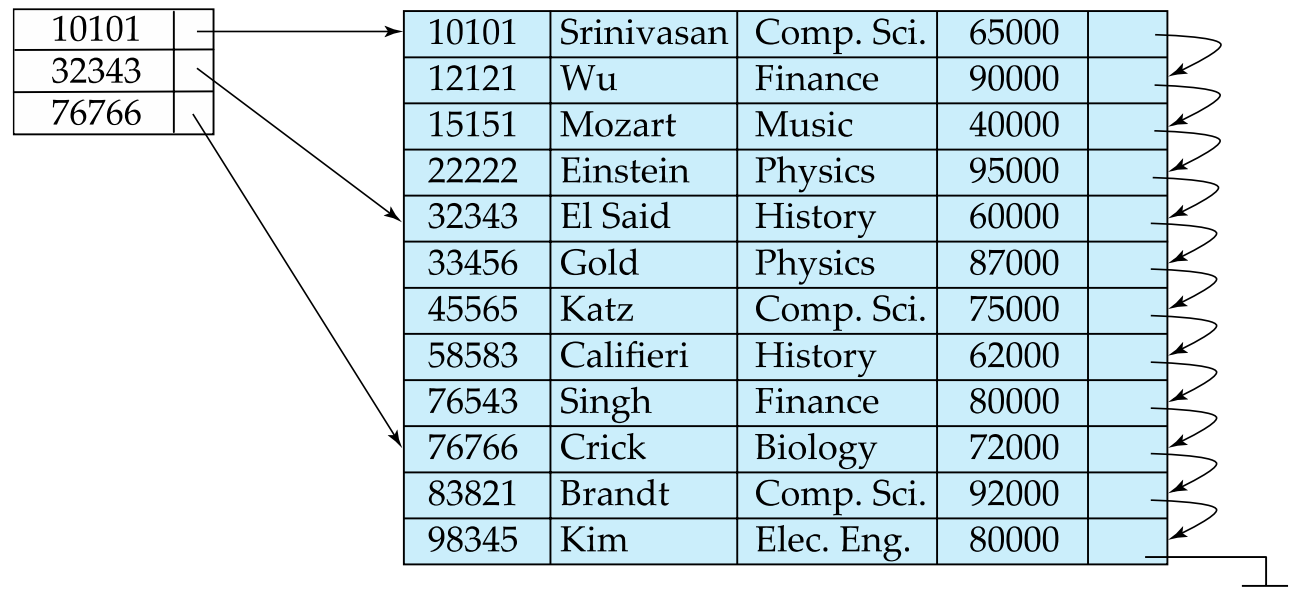
Dense Index Files (Cont.)

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



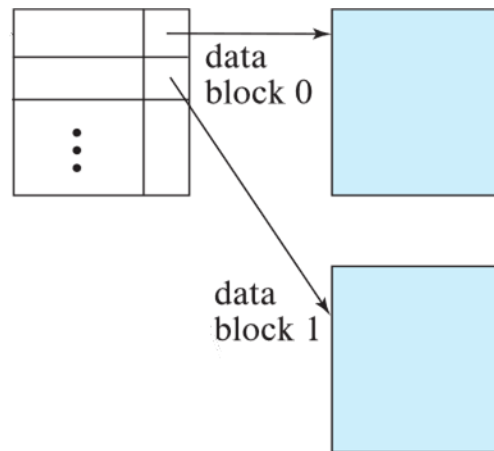
Sparse Index Files

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



Sparse Index Files (Cont.)

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



- for non-clustered index: sparse index on top of dense index (multilevel index)

Clustered vs. Non-clustered Indices

- Indices offer substantial benefits when searching for records
- But note: indices imposes overhead on database modification
 - when a record is inserted or deleted, every index on the relation must be updated
 - when a record is updated, any index on an updated attribute must be updated

Index Update: Insertion

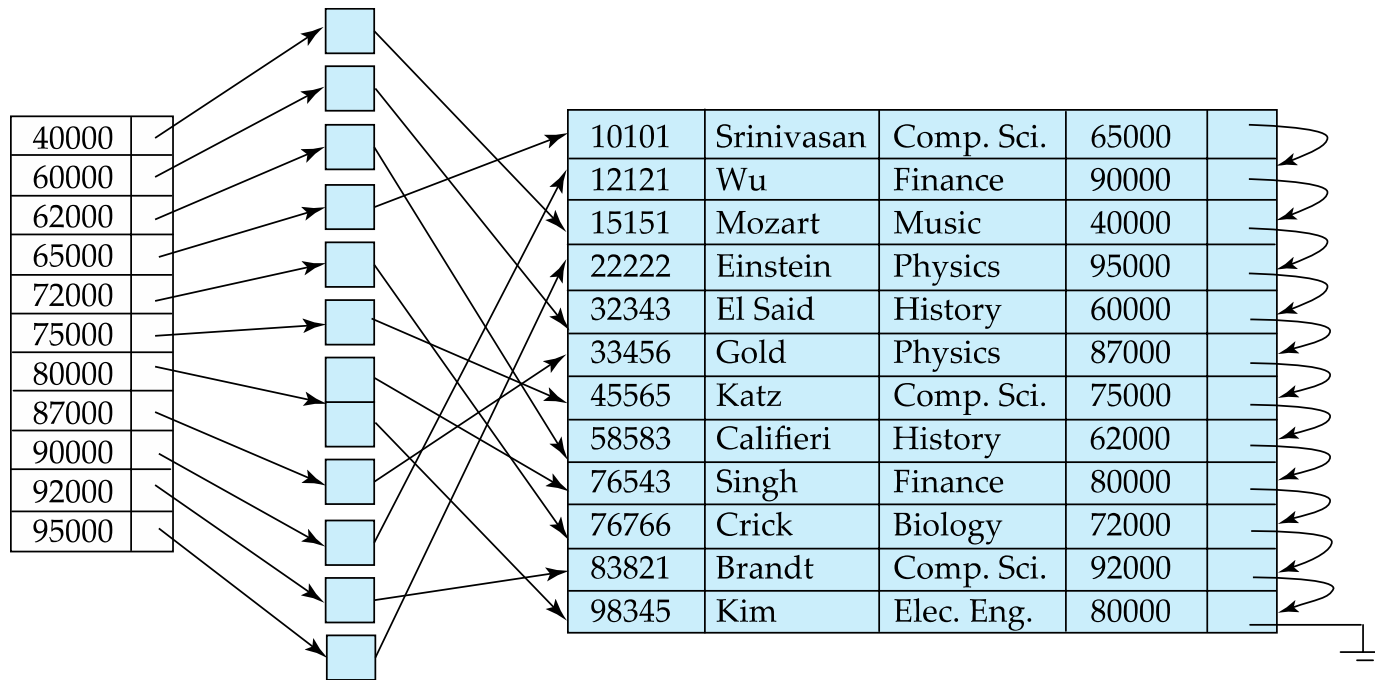
- Perform a lookup using the search-key value of the record to be inserted
- **Dense indices** – if the search-key value does not appear in the index, insert it
 - Indices are maintained as sequential files
 - Need to create space for new entry, overflow blocks may be required
- **Sparse indices** – 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

Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also
- **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

Non-clustered Indices Example

- Non-clustered 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
- Non-clustered indices must be dense

Clustered vs. Non-clustered Indices

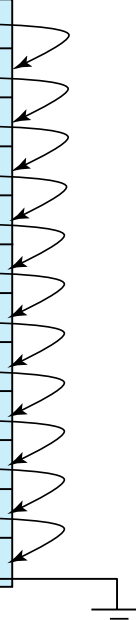
- Sequential scan using clustering index is efficient
- But note: a sequential scan using a non-clustered index is expensive on magnetic disk
 - each record access may fetch a new block from disk
 - each block fetch on magnetic disk requires about 5 to 10 milliseconds

Indices on Multiple Keys

- **Composite search key**

- e.g., index on *instructor* relation on attributes (*dept_name*, *salary*)
- values are sorted lexicographically
 - e.g. (Comp. Sci., 65000) < (Comp. Sci., 75000) and (Comp. Sci., 75000) < (Music, 40000)
- can query on just *dept_name*, or on (*dept_name*, *salary*)

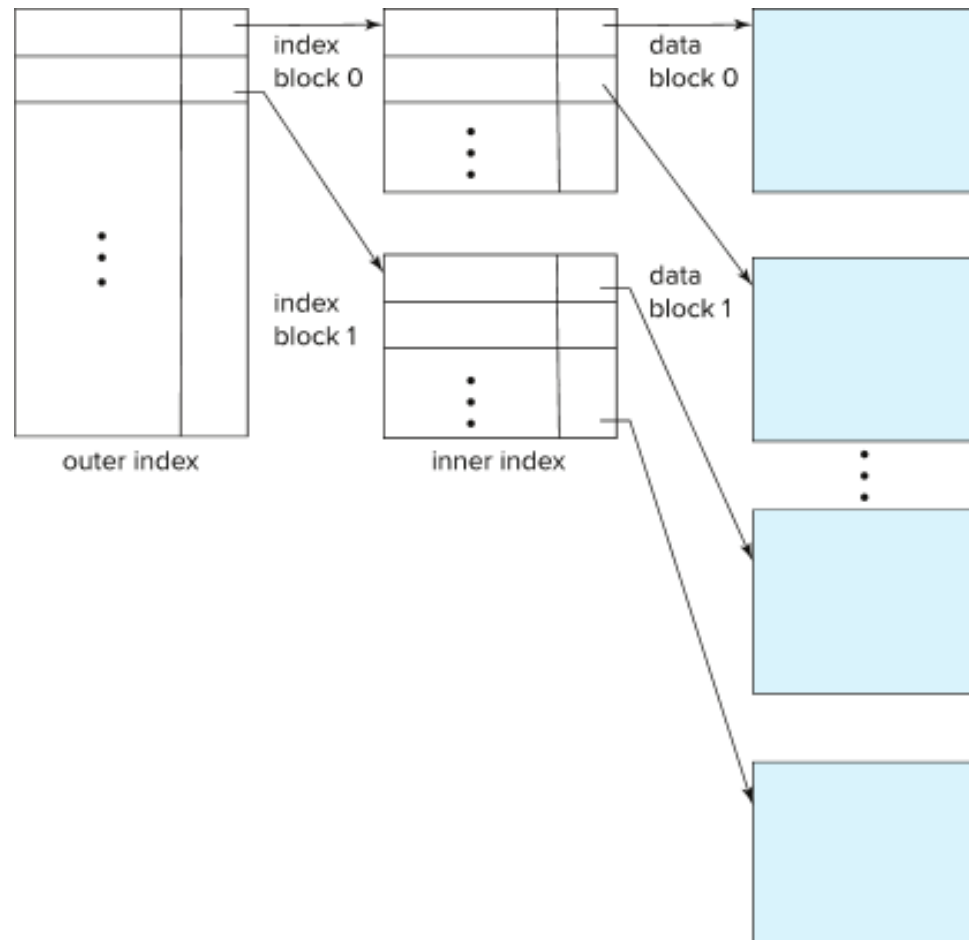
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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



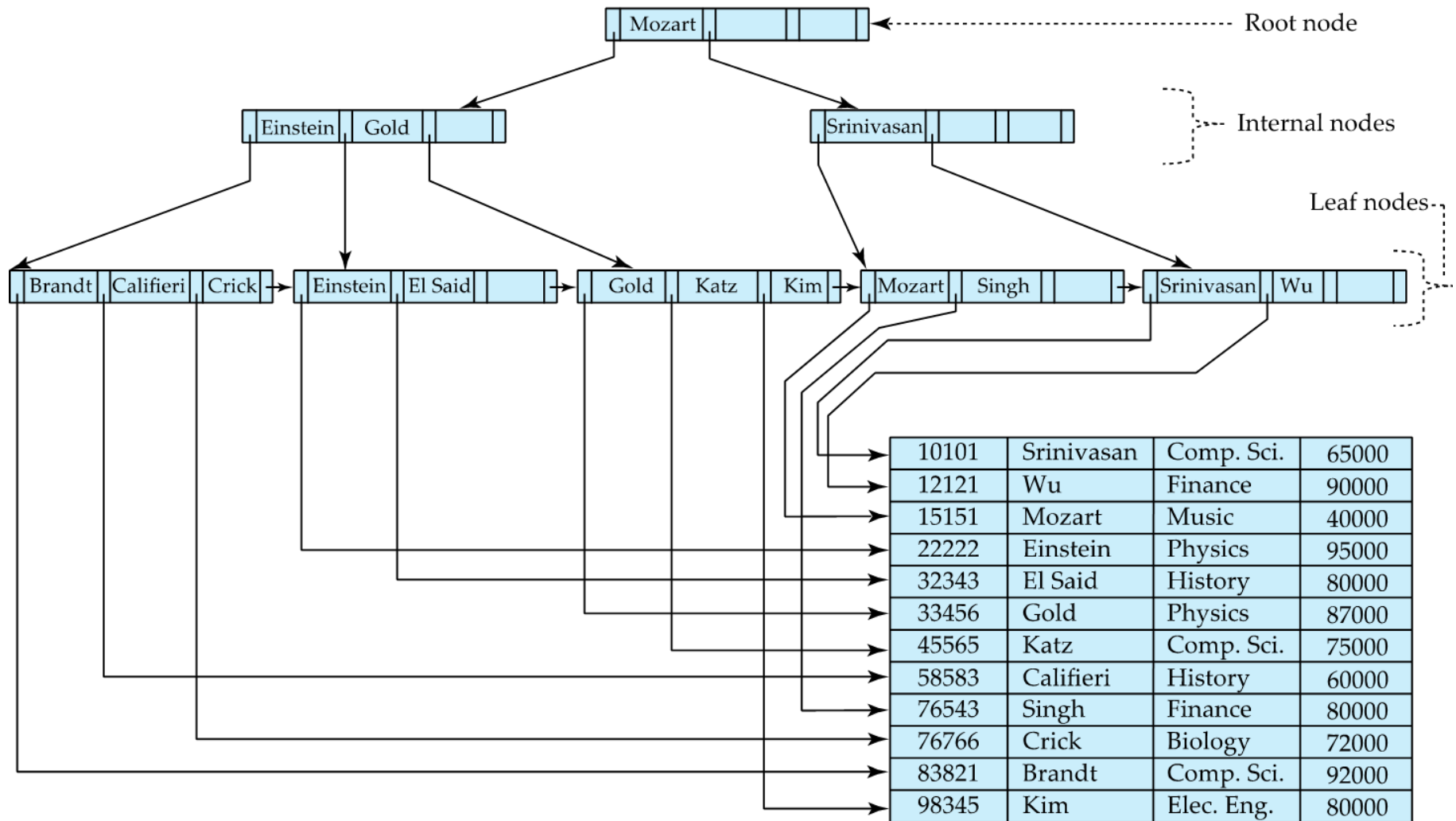
Multilevel Index

- If index does not fit in memory, access becomes expensive
- Solution: treat index kept on disk as a sequential file and construct a sparse index on it
 - outer index – a sparse index of the basic index
 - inner index – the basic index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file

Multilevel Index (Cont.)



Example of B⁺-Tree

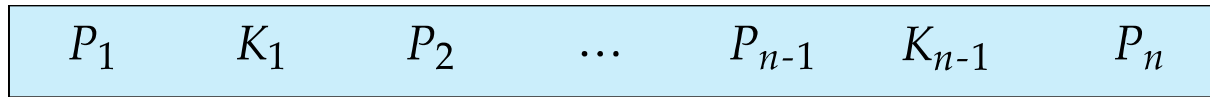


B⁺-Tree Indices

- A B⁺-tree is a rooted tree satisfying the following properties:
 - All paths from root to leaf are of the same length.
 - Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
 - A leaf node has between $\lceil (n - 1)/2 \rceil$ and $(n - 1)$ values.
 - Special cases:
 - If the root is not a leaf, it has at least 2 children.
 - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n - 1)$ values.

B⁺-Tree Node Structure

- Typical node



- 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

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

Leaf Nodes in B⁺-Trees

- Properties of a leaf node:
 - P_i ($i = 1, 2, \dots, n - 1$) points to a file record with search-key value K_i
 - If L_i and L_j are leaf nodes (with $i < j$) then L_i 's search-key values are $\leq L_j$'s search-key values
 - P_n points to next leaf node in search-key order

leaf node

Brandt	Califieri	Crick
--------	-----------	-------

➤ Pointer to next leaf node

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

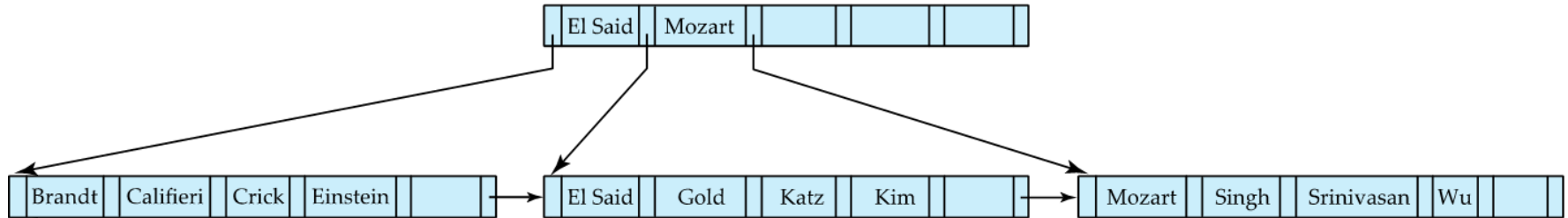
Non-Leaf Nodes in B⁺-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes.
- For a non-leaf node with n pointers:
 - All the search-keys in the subtree to which P_1 points are $< K_1$
 - For $2 \leq i \leq n - 1$ all the search-keys in the subtree to which P_i points have values $< K_i$ and $\geq K_{i-1}$
 - All the search-keys in the subtree to which P_n points have values $\geq K_{n-1}$
 - General structure:



Example of B⁺-tree

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



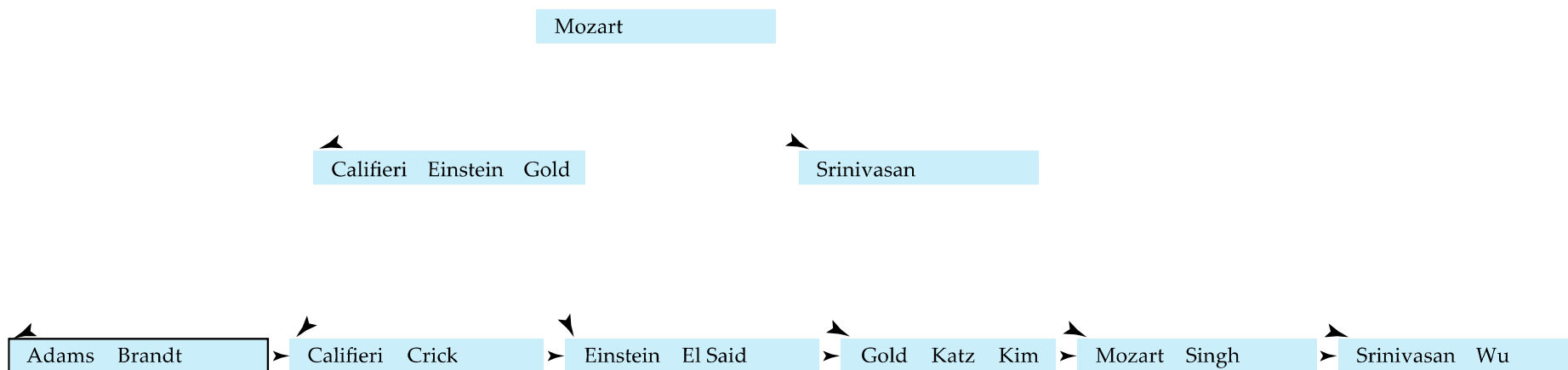
- Leaf nodes must have between 3 and 5 values ($\lceil (n - 1)/2 \rceil$ and $n - 1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil n/2 \rceil$ and n with $n = 6$).
- Root must have at least 2 children.

Observations about B⁺-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The B⁺-tree contains a relatively small number of levels
 - level below root has at least $2 \times \lceil n/2 \rceil$ values
 - next level has at least $2 \times \lceil n/2 \rceil \times \lceil n/2 \rceil$ values
 - etc.
- If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
 - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

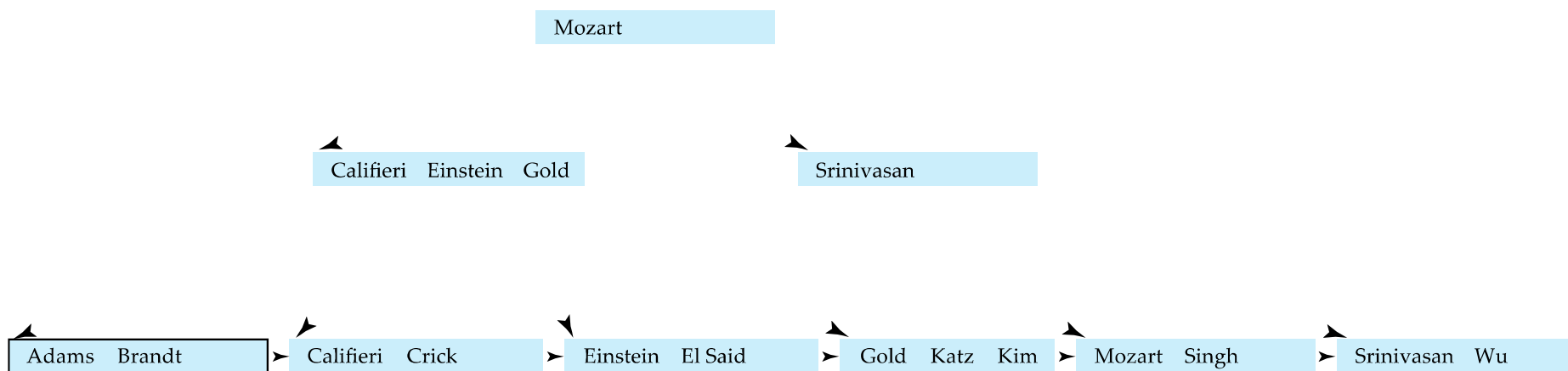
Queries on B⁺-Trees

- Start at the root node and look for value V
 - If there is a K_i such that $V = K_i$ then follow P_{i+1}
 - If there is a K_i such that $V < K_i$ then follow P_i
 - If there is no such K_i then follow last pointer
- Repeat the same procedure on every non-leaf node
- Once a leaf node is reached:
 - If there is a K_i such that $V = K_i$ then follow P_i
 - If there is no such K_i then value V is not found



Queries on B⁺-Trees (Cont.)

- **Range queries** find all records with search key values in a range $[V_{min}, V_{max}]$
 - Search for V_{min} until reaching a leaf node
 - note: the leaf node may or may not contain V_{min}
 - Run through the leaf nodes sequentially
 - Retrieve records from every pointer P_i such that $V_{min} \leq K_i \leq V_{max}$
 - Stop on the first K_i such that $V_{max} < K_i$ (or go until the end)



Queries on B⁺-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
 - and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n = 100$
 - at most $\log_{50}(1,000,000) = 4$ nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
 - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

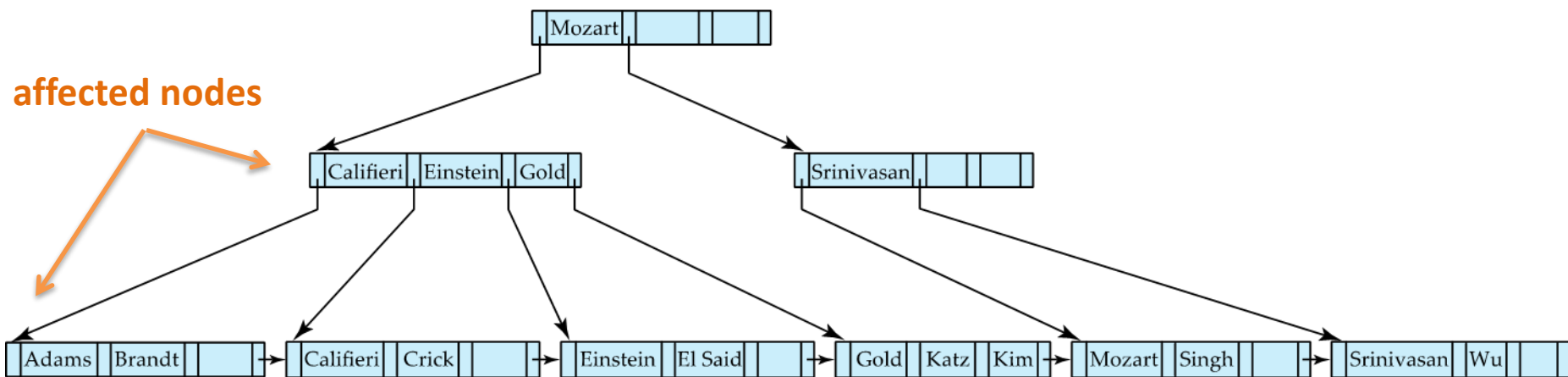
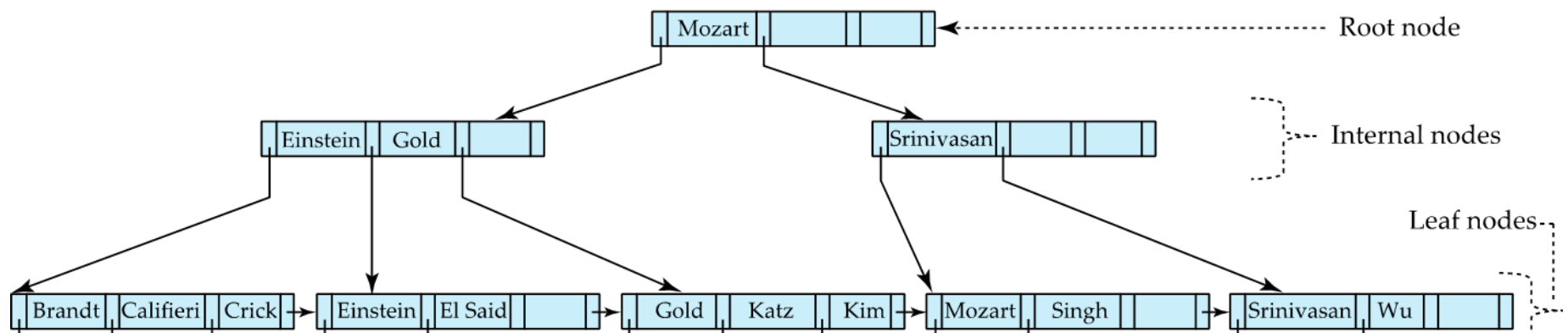
Updates on B⁺-Trees: Insertion

- Assume record already added to the file
 - Let P_r be pointer to the record
 - Let K_r be the search key value of the record
- Find the leaf node in which the search-key value would appear
 - If there is room in the leaf node, insert (P_r, K_r) pair in the leaf node
 - Otherwise, split the node (along with the new (P_r, K_r) entry) as discussed in the next slide, and propagate updates to parent nodes.

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

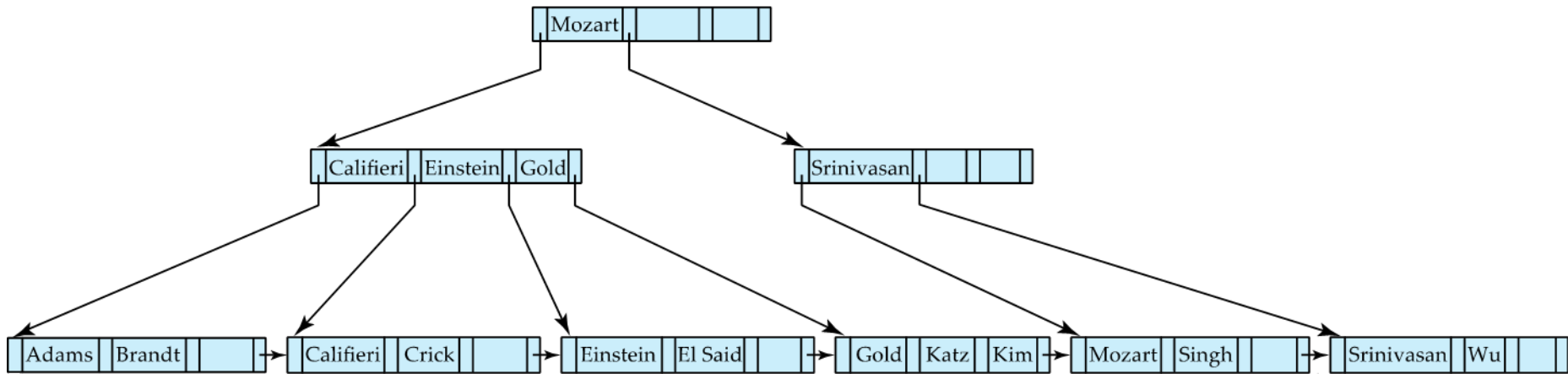
- Splitting a leaf node:
 - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
 - let the new node be p and let k be the least key value in p . Insert (k, p) in the parent of the node being split.
 - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
 - In the worst case the root node may be split increasing the height of the tree by 1.

B⁺-Tree Insertion

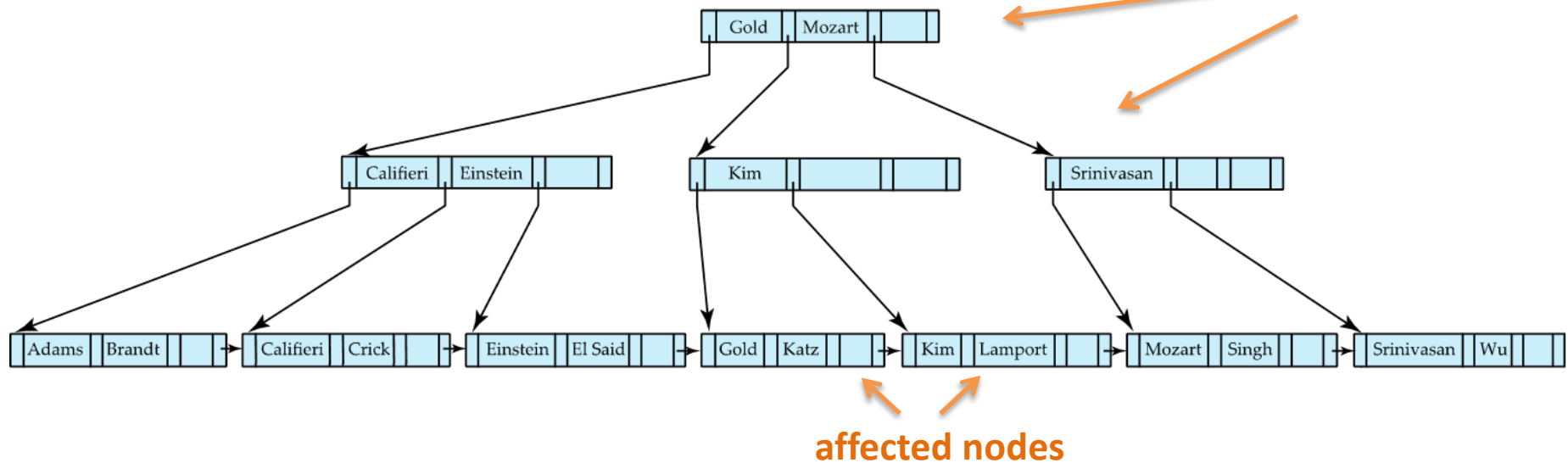


B⁺-Tree before and after insertion of "Adams"

B⁺-Tree Insertion (Cont.)



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



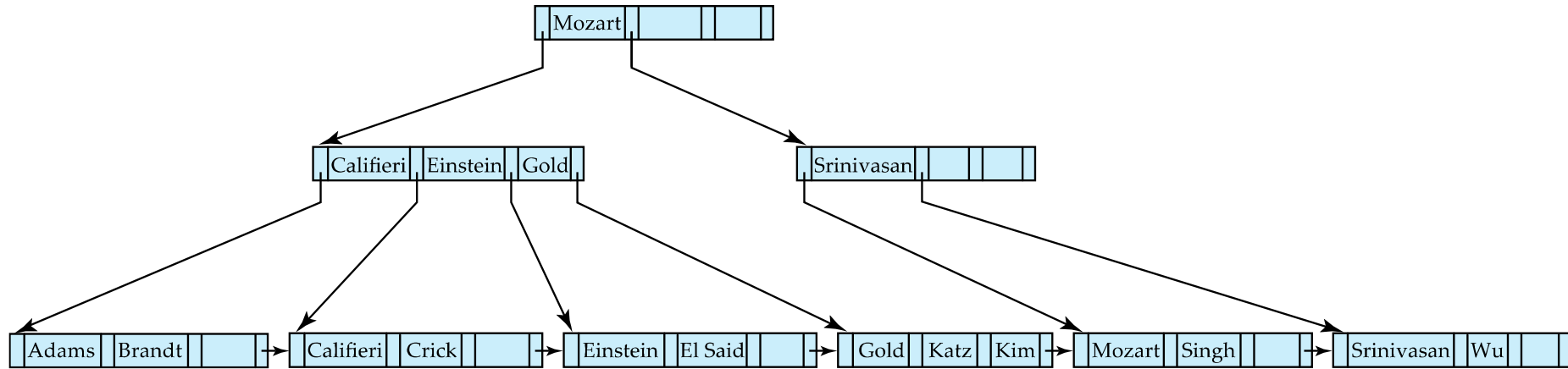
Updates on B⁺-Trees: Deletion

- Assume record already deleted from file.
 - Let P_r be pointer to the record
 - Let K_r be the search key value of the record
- Remove (P_r, K_r) from the leaf node
- 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.

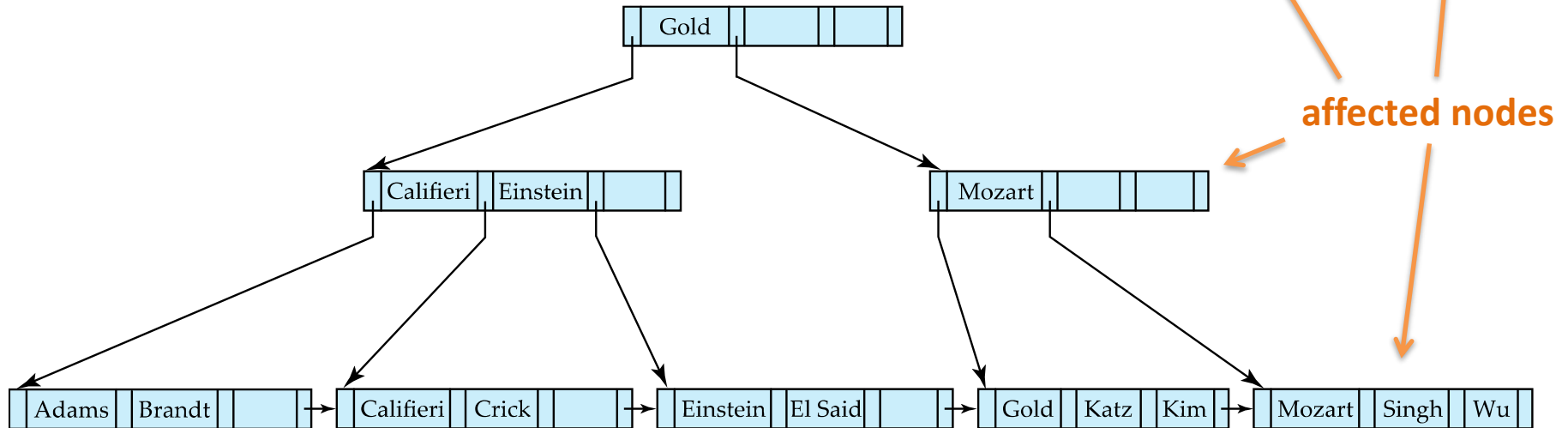
Updates on B⁺-Trees: Deletion (Cont.)

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
 - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
 - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Examples of B⁺-Tree Deletion

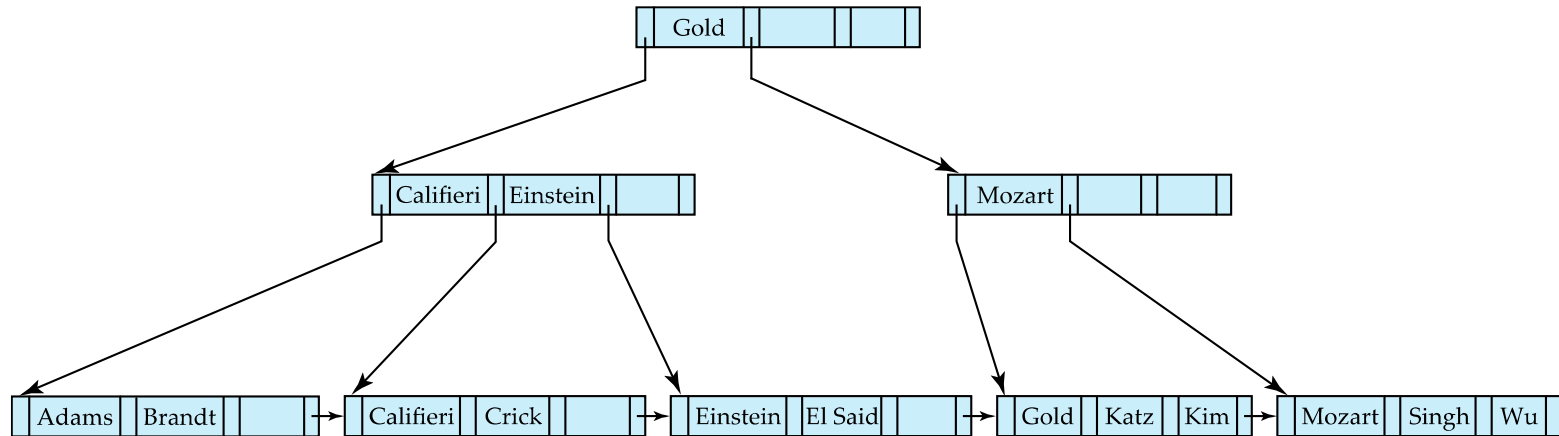


Before and after deleting "Srinivasan"

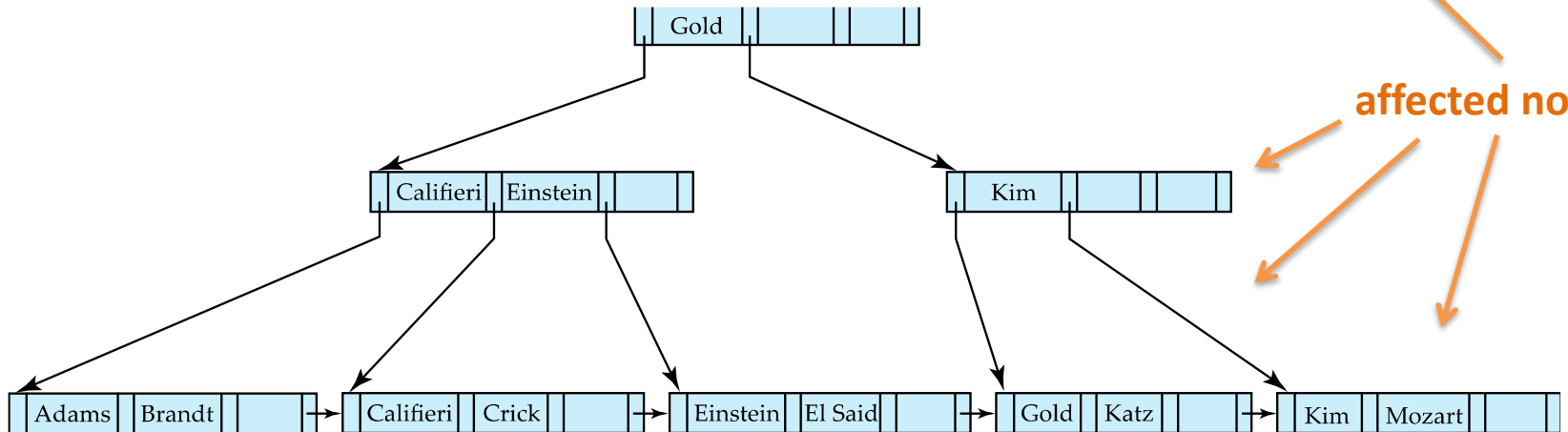


- Deleting "Srinivasan" causes **merging** in leaves, and **redistribution** in non-leaves

Examples of B⁺-Tree Deletion (Cont.)

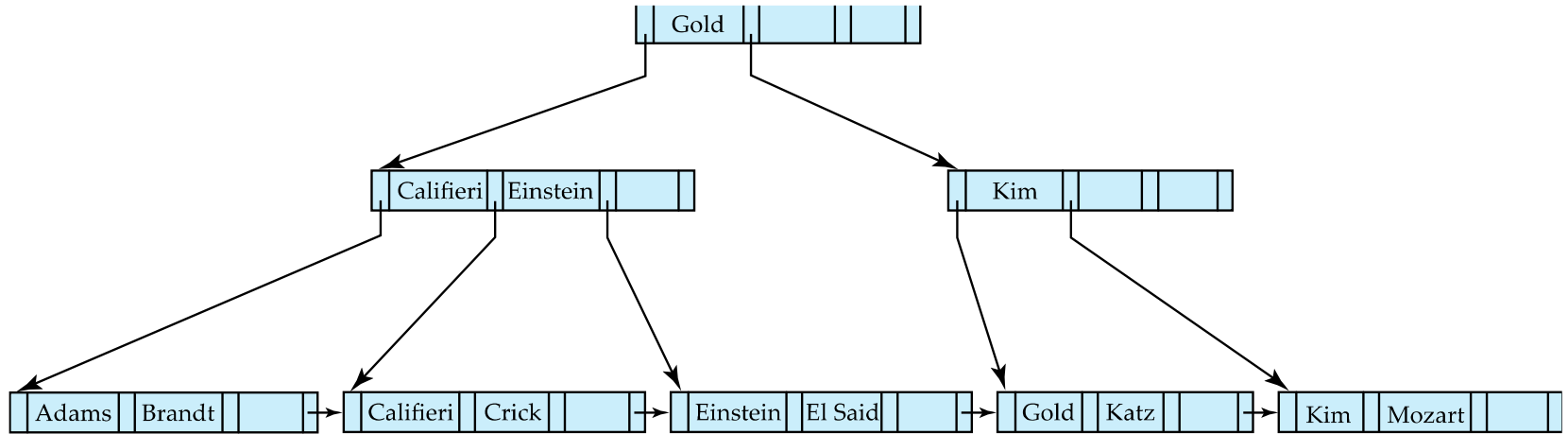


Before and after deleting "Singh" and "Wu"

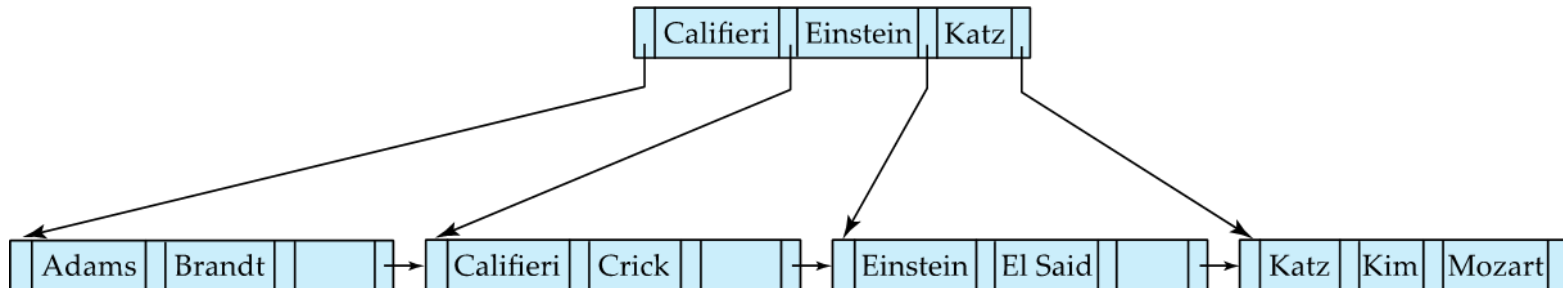


- Deleting Singh and Wu causes **redistribution** in leaves, and update in non-leaf
- Search-key value in the parent changes as a result

Examples of B⁺-Tree Deletion (Cont.)



Before and after deletion of “Gold”



- Deleting "Gold" causes **merging** in leaves, update and **merging** in non-leaves
- Root node then has only one child, and is deleted

Complexity of Updates

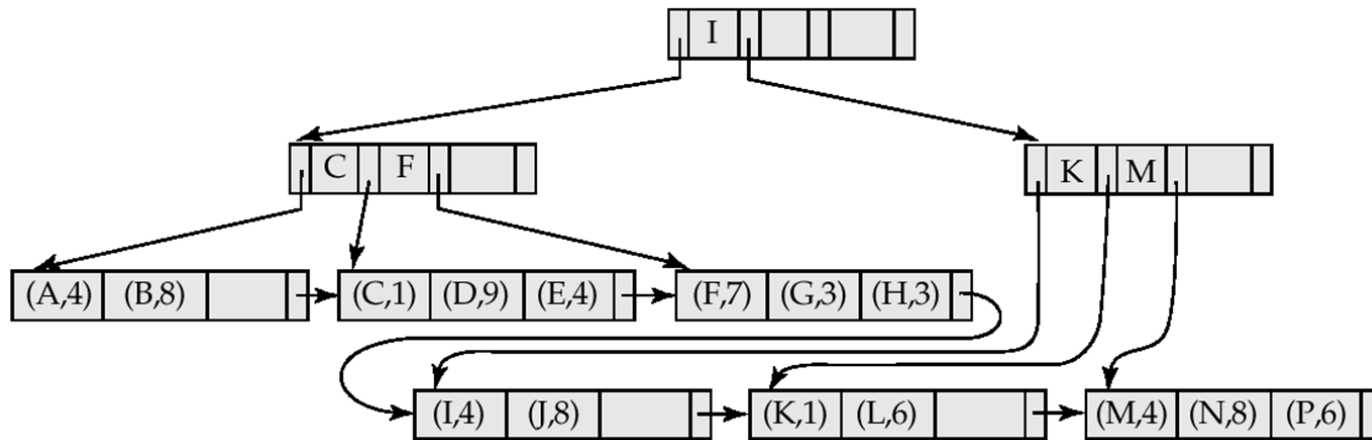
- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
 - With K entries and maximum fanout of n , worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
 - Internal nodes tend to be in buffer
 - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
 - 2/3rds with random, 1/2 with insertion in sorted order

B⁺-Tree File Organization

- B⁺-Tree File Organization:
 - Leaf nodes in a B⁺-tree file organization store records, instead of pointers
 - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
 - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a non-leaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B⁺-tree index.

B⁺-Tree File Organization (Cont.)

- Example of B+-tree File Organization



- Good space utilization important since records use more space than pointers.
- Use more sibling nodes in redistribution during splits and merges
 - e.g. redistributing $2n$ entries across 3 nodes gives a space utilization of at least $\lfloor 2n/3 \rfloor$ instead of $\lfloor n/2 \rfloor$

Static Hashing

- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
 - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B .
- Hash function is used to locate entries for access, insertion as well as deletion.
- Entries with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate an entry.
- In a **hash index**, buckets store entries with pointers to records
- In a **hash file-organization** buckets store records

Example of Hash File Organization

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

- There are 10 buckets.
- The binary representation of each character is assumed to be an integer.
- The hash function returns the sum of the binary representations of the characters modulo 10
 - e.g. $h(\text{Music}) = 1$ $h(\text{History}) = 2$
 $h(\text{Physics}) = 3$ $h(\text{Elec. Eng.}) = 3$

Example of Hash File Organization (Cont.)

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

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

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

bucket 7

Hash Functions

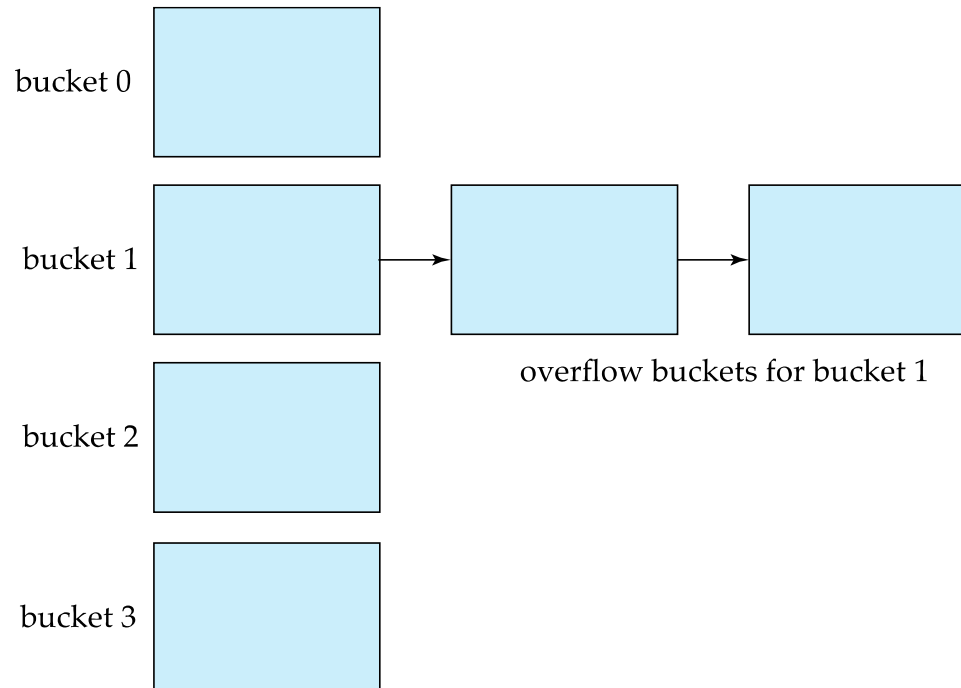
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
 - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.

Handling of Bucket Overflows

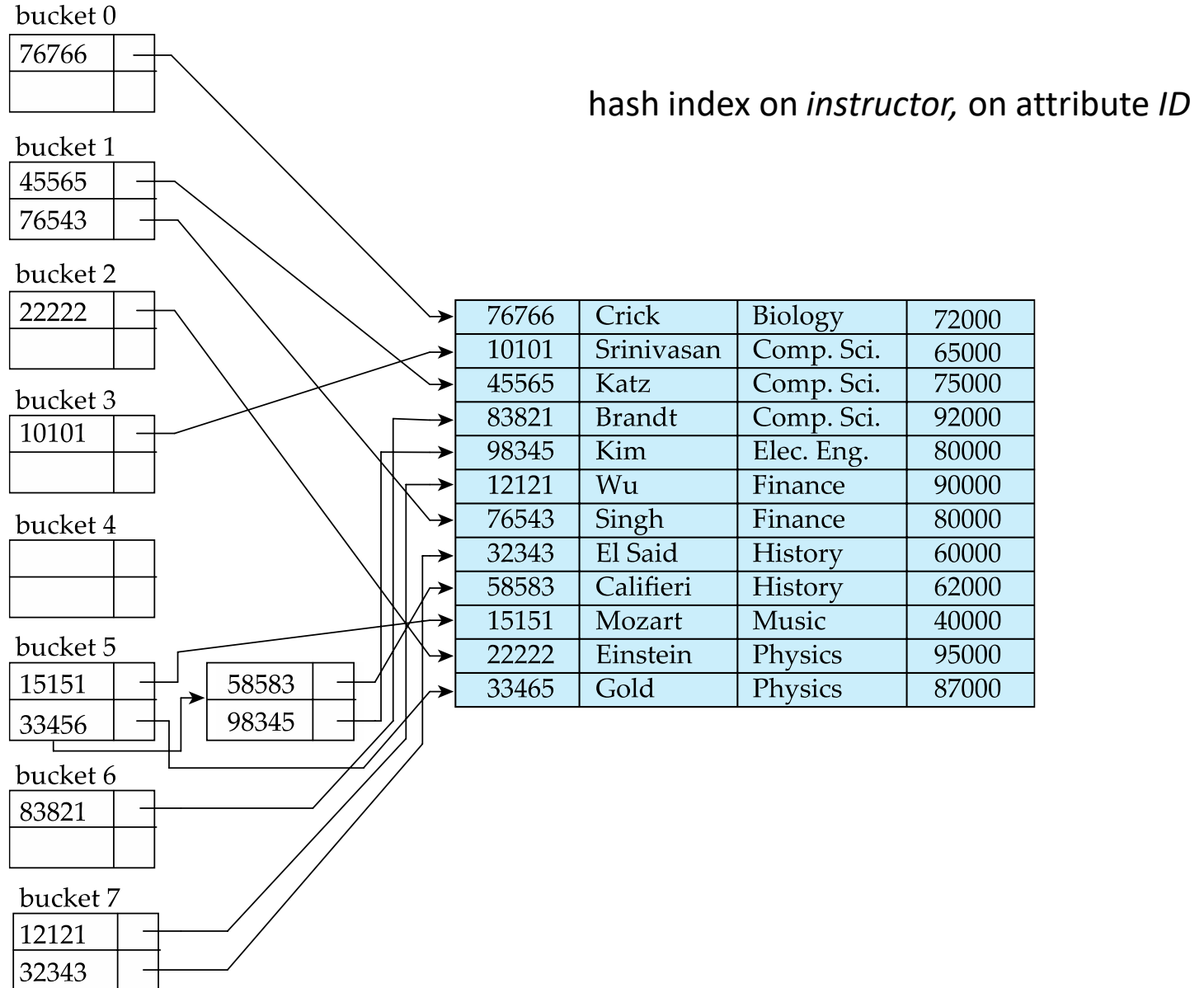
- Bucket overflow can occur because of
 - Insufficient buckets
 - Skew in distribution of records. This can occur due to two reasons:
 - multiple records have same search-key value
 - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.

Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.



Example of Hash Index



Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
 - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
 - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
 - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
 - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

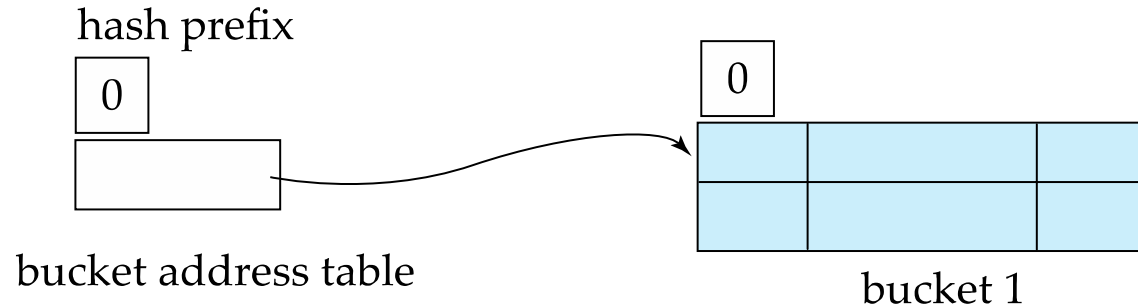
- **Extendable hashing** – one form of dynamic hashing
 - Hash function generates values over a large range — typically b -bit integers, with $b = 32$.
 - At any time use only a prefix of the hash function to index into a table of bucket addresses.
 - Let the length of the prefix be i bits, $0 \leq i \leq 32$.
 - 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
 - Thus, actual number of buckets is $< 2^i$
 - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

Use of Extendable Hash Structure: Example

<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

Example (Cont.)

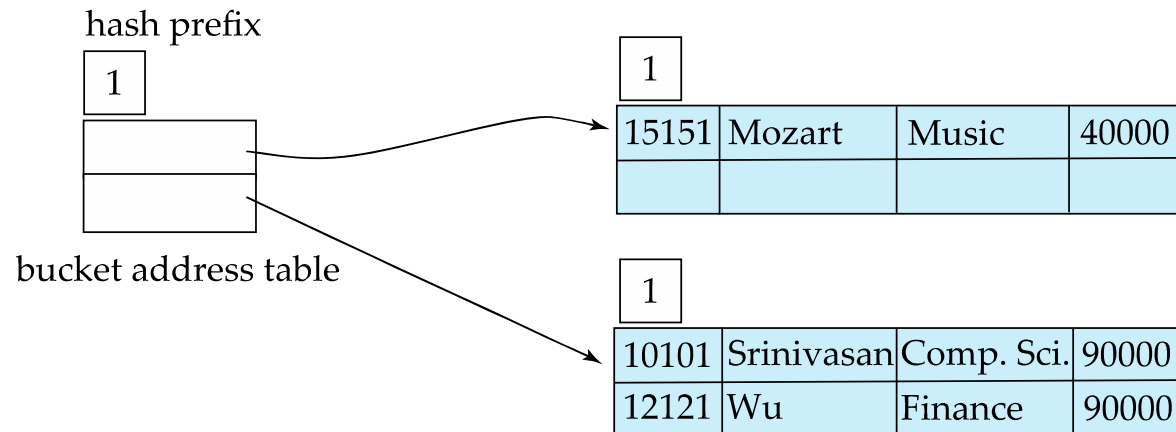
- Initial hash structure, no records yet; using 0-bit prefix



<i>dept_name</i>	<i>h(dept_name)</i>
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.)

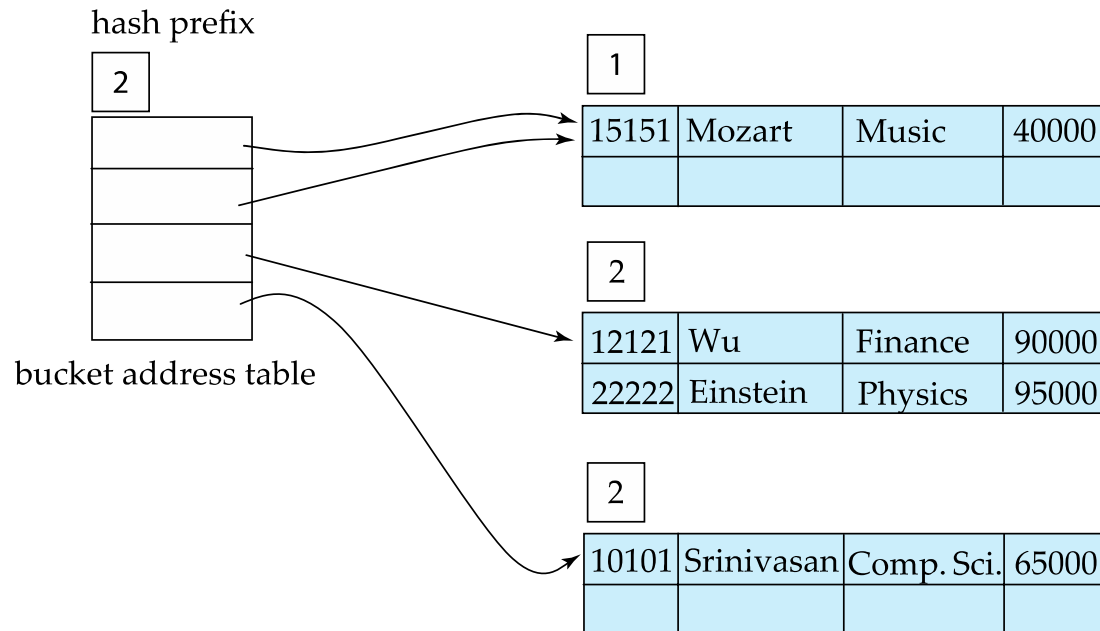
- Hash structure after insertion of 3 records; using 1-bit prefix



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.)

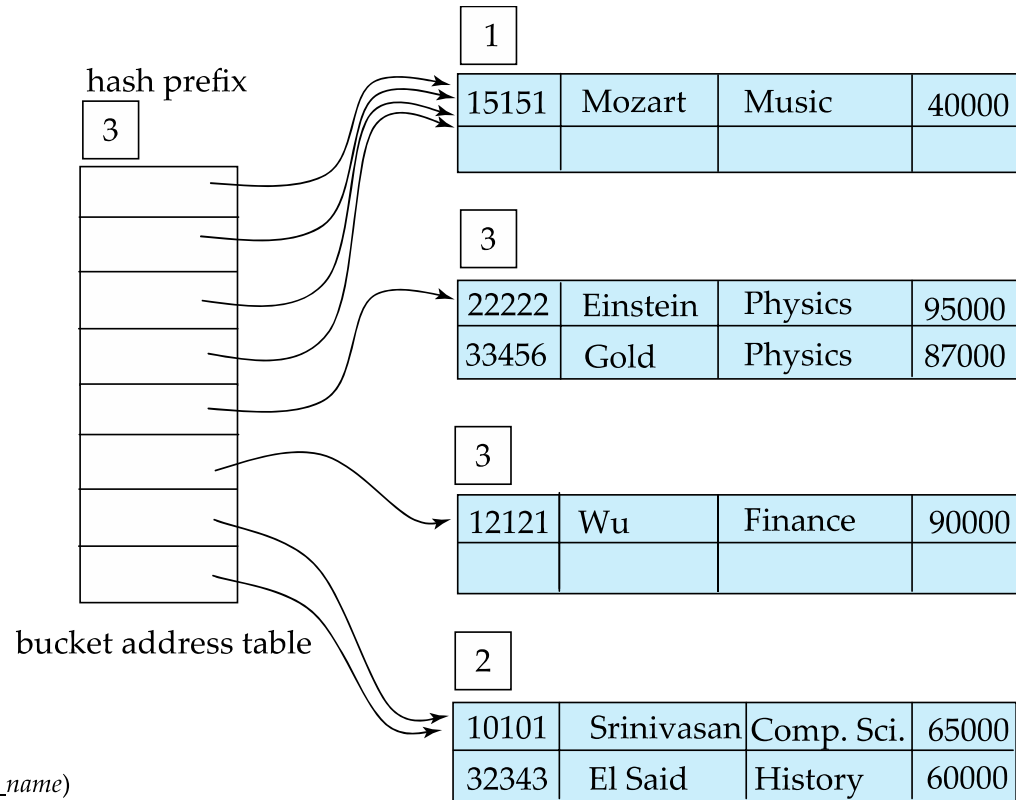
- Hash structure after insertion of 4 records; using 2-bit prefix



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.)

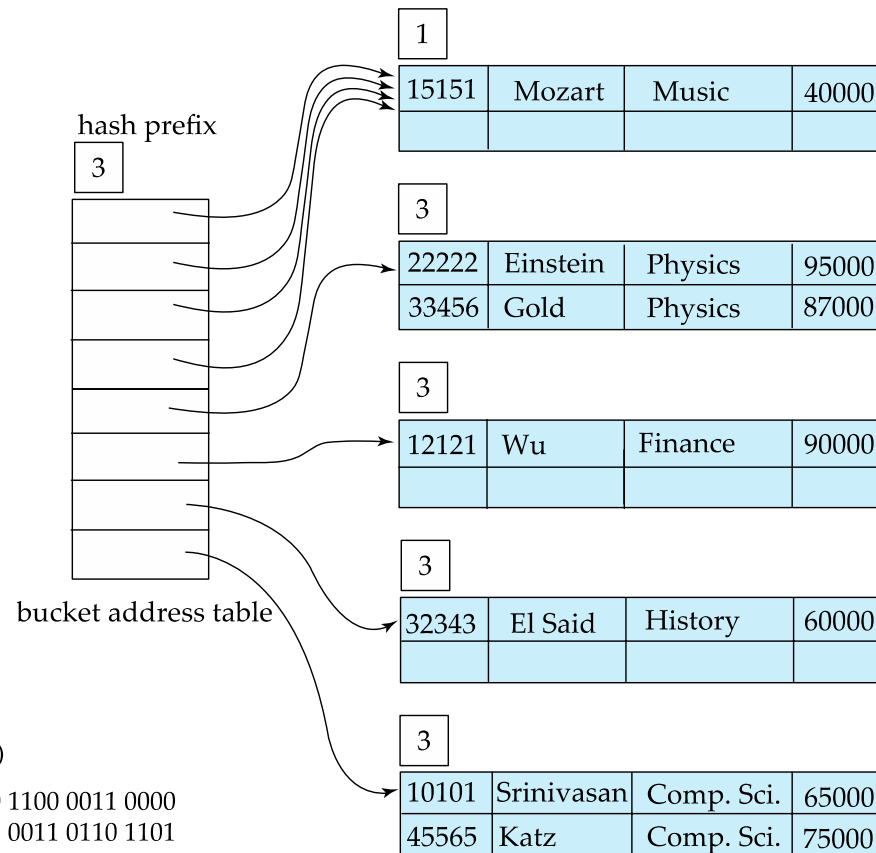
- Hash structure after insertion of 6 records; using 3-bit prefix



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.)

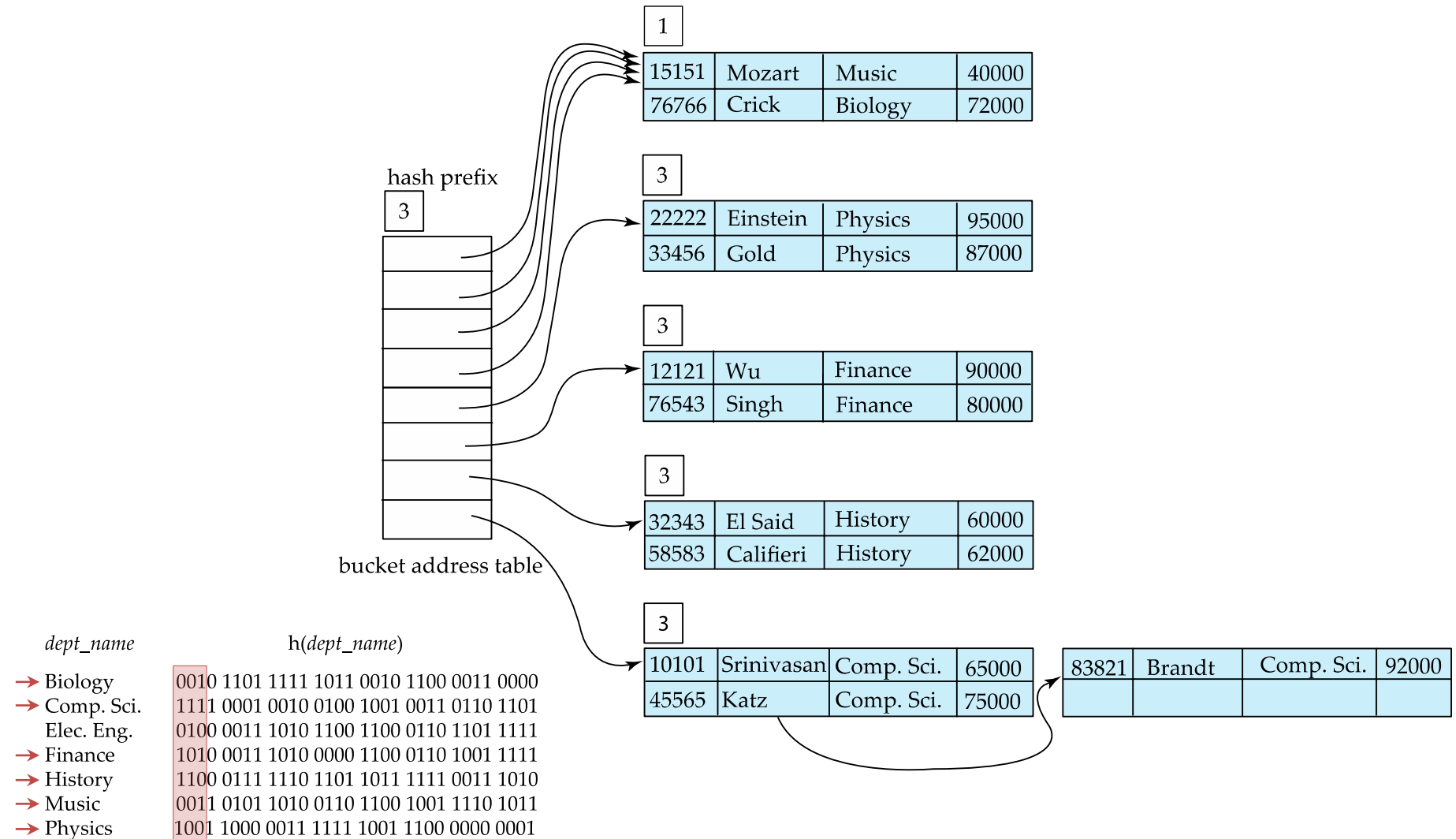
- Hash structure after insertion of 7 records; using 3-bit prefix



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

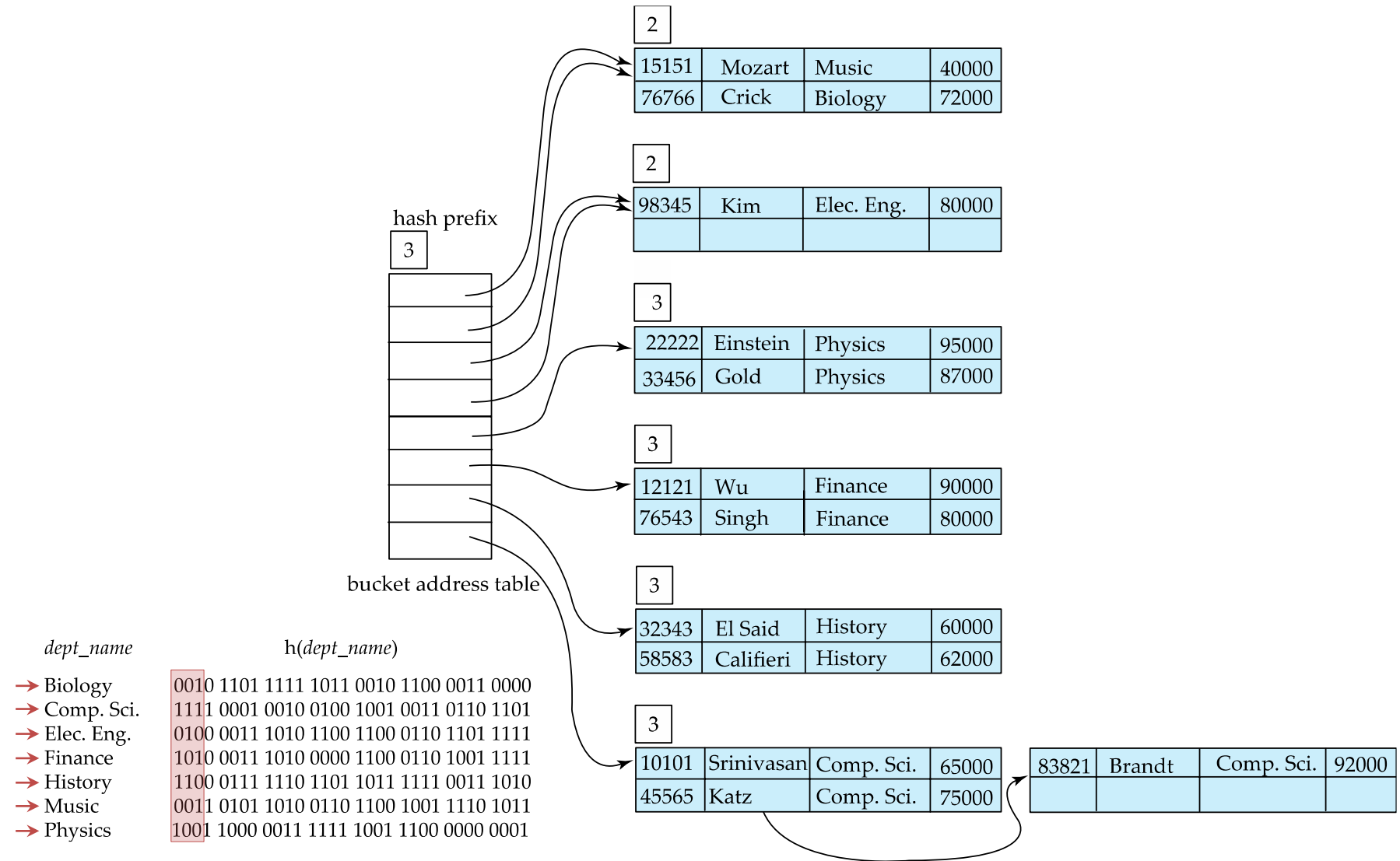
Example (Cont.)

- Hash structure after insertion of 11 records; using 3-bit prefix

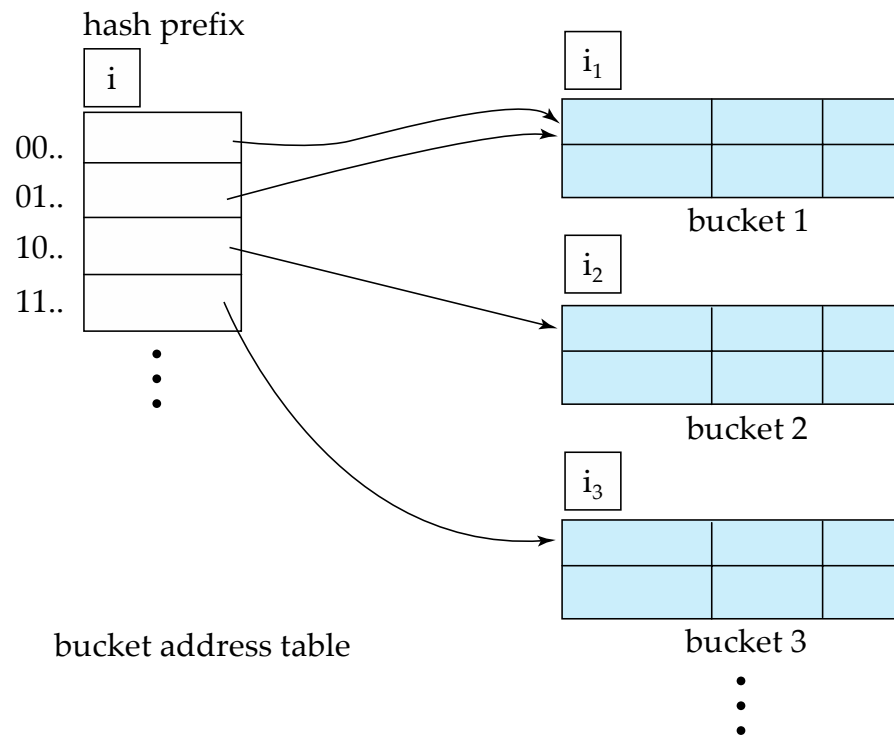


Example (Cont.)

- Hash structure after insertion of 12 records; using 3-bit prefix



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
 - Overflow buckets used instead in some cases

Insertion in Extendable Hash Structure

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

Comparison of Ordered Indexing and Hashing

- Expected type of queries:
 - Hashing is generally better at retrieving records having a specified value of the key.
 - If range queries are common, ordered indices are preferred
- In practice:
 - PostgreSQL supports hash indices, but discourages its use
 - Oracle supports static hash organization, but not hash indices
 - SQLServer supports B⁺-trees; hash indexes in memory only
 - Hash-indices are extensively used in-memory but not used much on disk

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

Covering Indices

- **Covering index**
 - Include extra attributes in the index so some queries can avoid fetching the actual records
 - e.g. include *salary* in index on *dept_name*
 - Store extra attributes only at leaf
 - keep size of search key, fanout of non-leaf nodes, height of the index
- Particularly useful for non-clustered indices
 - Since all desired information is included in the index, avoids random access to the table

Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
 - Given a number n it must be easy to retrieve record n
 - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
 - e.g., gender, country, state, ...
 - e.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
 - Bitmap has as many bits as records
 - In a bitmap for value v , the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise
- Example

record number	<table><tr><th><i>ID</i></th><th><i>gender</i></th><th><i>income_level</i></th></tr><tr><td>76766</td><td>m</td><td>L1</td></tr><tr><td>1</td><td>22222</td><td>f</td><td>L2</td></tr><tr><td>2</td><td>12121</td><td>f</td><td>L1</td></tr><tr><td>3</td><td>15151</td><td>m</td><td>L4</td></tr><tr><td>4</td><td>58583</td><td>f</td><td>L3</td></tr></table>			<i>ID</i>	<i>gender</i>	<i>income_level</i>	76766	m	L1	1	22222	f	L2	2	12121	f	L1	3	15151	m	L4	4	58583	f	L3	Bitmaps for <i>gender</i>		Bitmaps for <i>income_level</i>	
				<i>ID</i>	<i>gender</i>	<i>income_level</i>																							
76766	m	L1																											
1	22222	f	L2																										
2	12121	f	L1																										
3	15151	m	L4																										
4	58583	f	L3																										
			m	10010																									
			f	01101	L1	10100																							
					L2	01000																							
					L3	00001																							
					L4	00010																							
					L5	00000																							

Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
 - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
 - Intersection (and)
 - Union (or)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
 - e.g., $100110 \text{ AND } 110011 = 100010$
 $100110 \text{ OR } 110011 = 110111$
 $\text{NOT } 100110 = 011001$
 - gender m with income level $L1$: $10010 \text{ AND } 10100 = 10000$
 - Can then retrieve required tuples.
 - Counting number of matching tuples is even faster