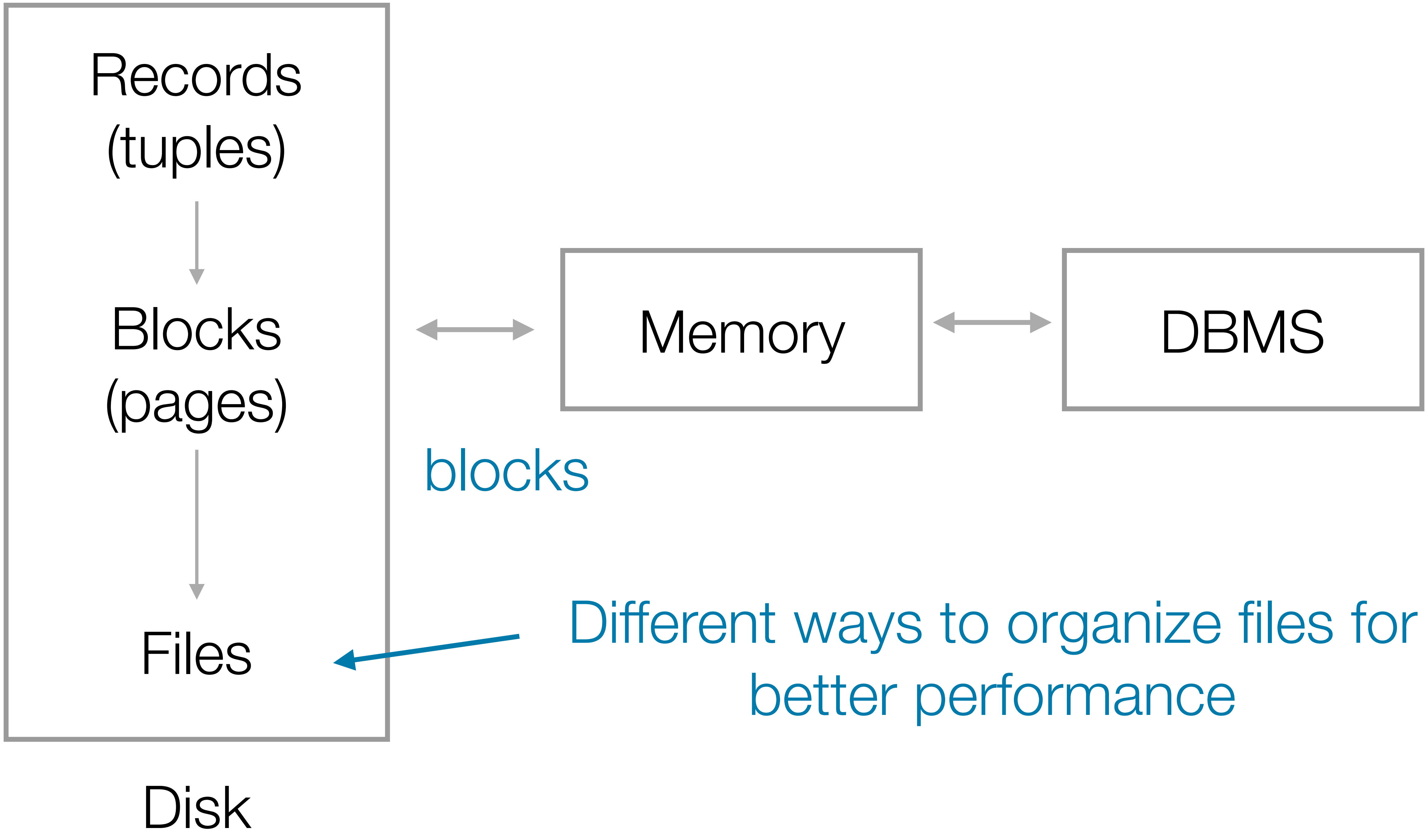
Indexing

CS 377: Database Systems

Review: Data Store Overview



Today and Next Lecture

1. Index Overview
2. Hashing Index
3. B+ Tree

Index: Motivation

- Suppose we want to search for employees of a specific age in our company database with a relation:
  SELECT * from Employee where age = 25;
  - Simple scan: O(N) — inefficient to read all tuples to find one
- Idea: Sort the records by age and we know how to do this fast (several efficient algorithms such as merge sort, heapsort, etc.)
  - Binary search: $O(\log_2 N)$

Index: Motivation

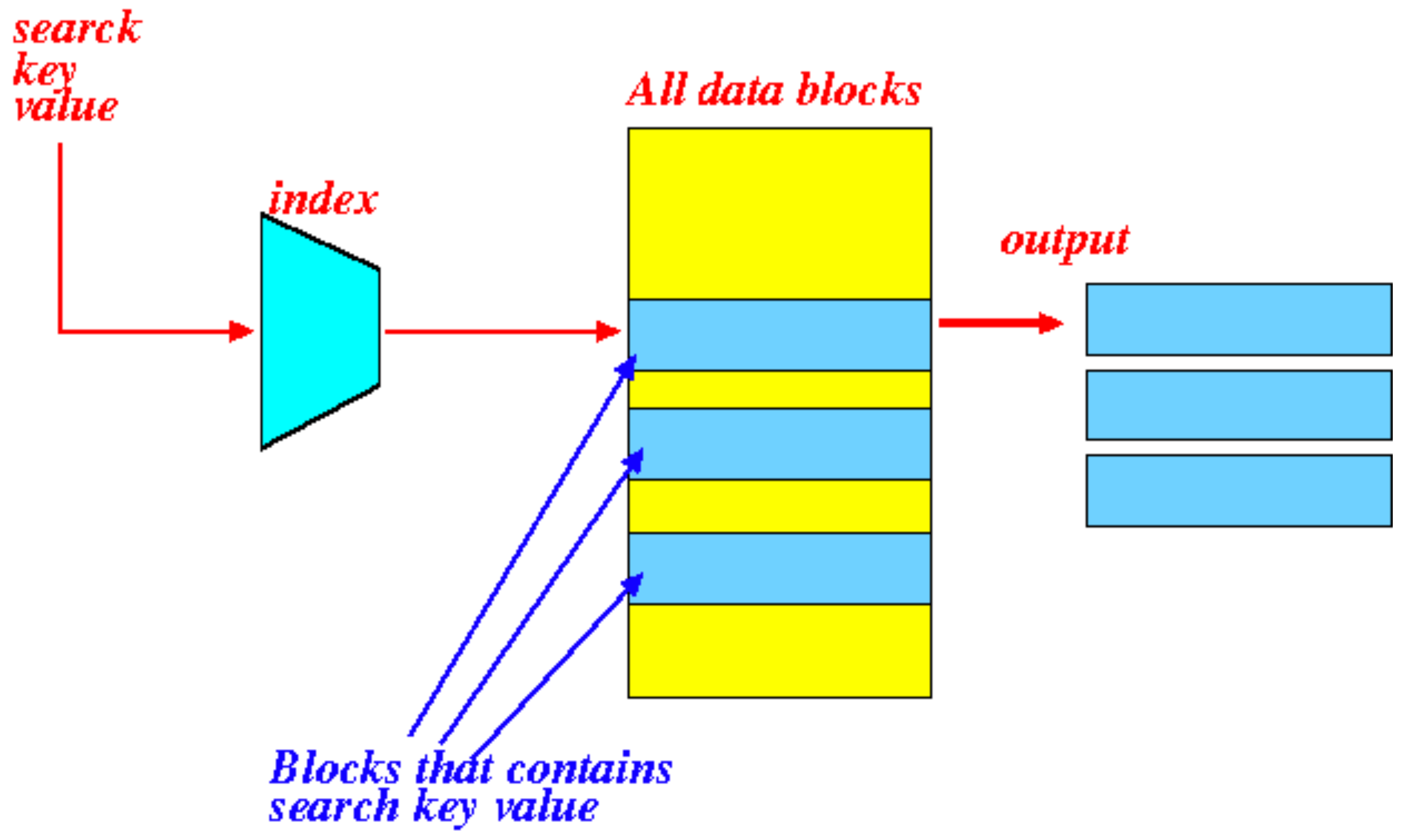
- What if we want to be able to search quickly over multiple attributes (e.g., not just age)?
  - Idea: Keep multiple copies of the records, each sorted by one attribute set — very expensive from a storage perspective
- Are there better techniques that allow better tradeoffs between storage and query speed?

Indexes

- Data structures that organize records via trees or hashing
  - Speed up search for a subset of records based on values in a certain field (search key)
  - Any subset of the fields of the relation can be the search field
  - Search key need not be the same as the key!
- Contains a collection of data entries (each entry with sufficient information to locate the records)

Index Effect

Index maps the search key value to the list of blocks that contains the search key value



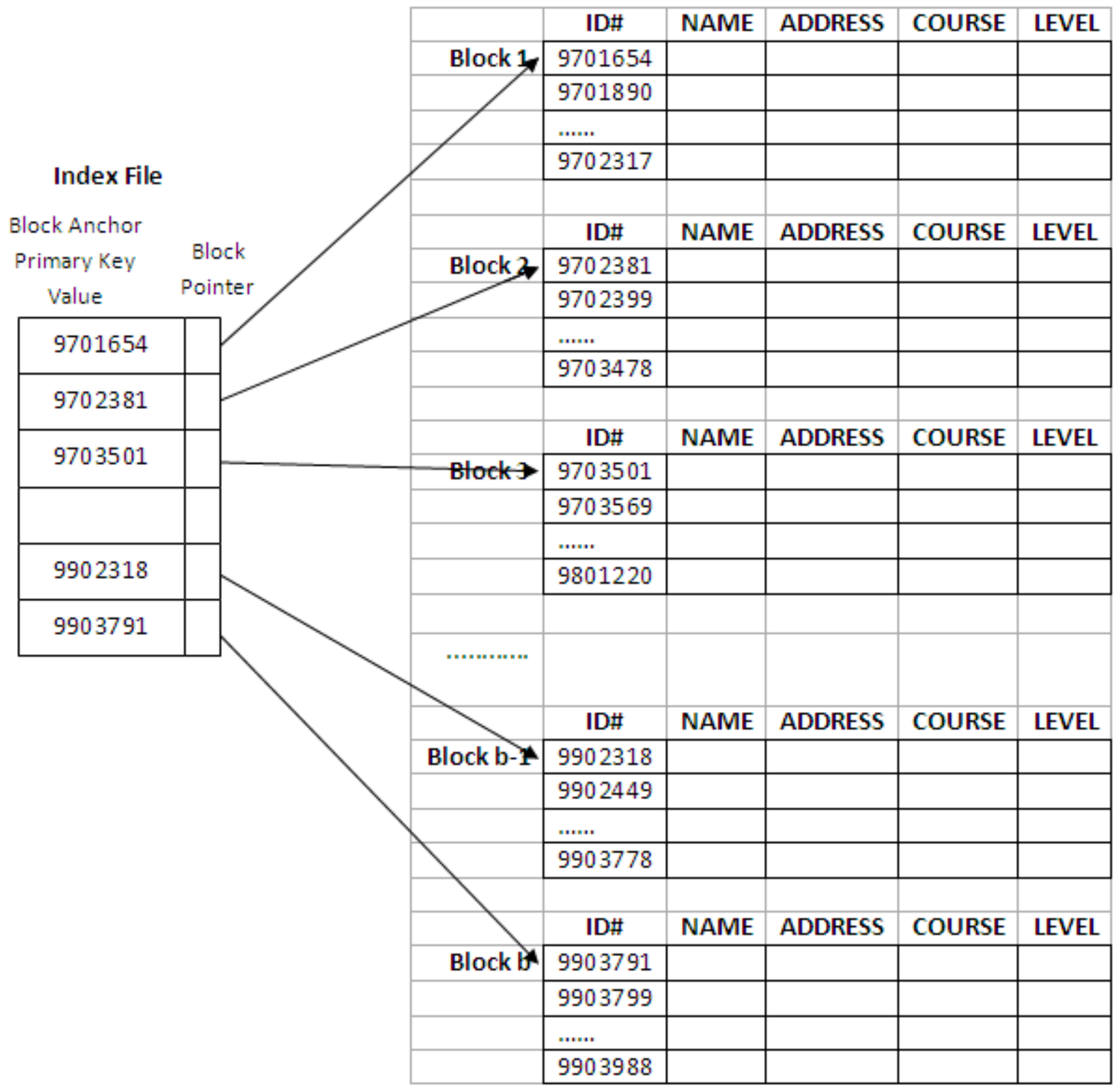
http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/3-index/intro.html

Index File

- Stores records in the following format:
  Search Key | Block Ptr

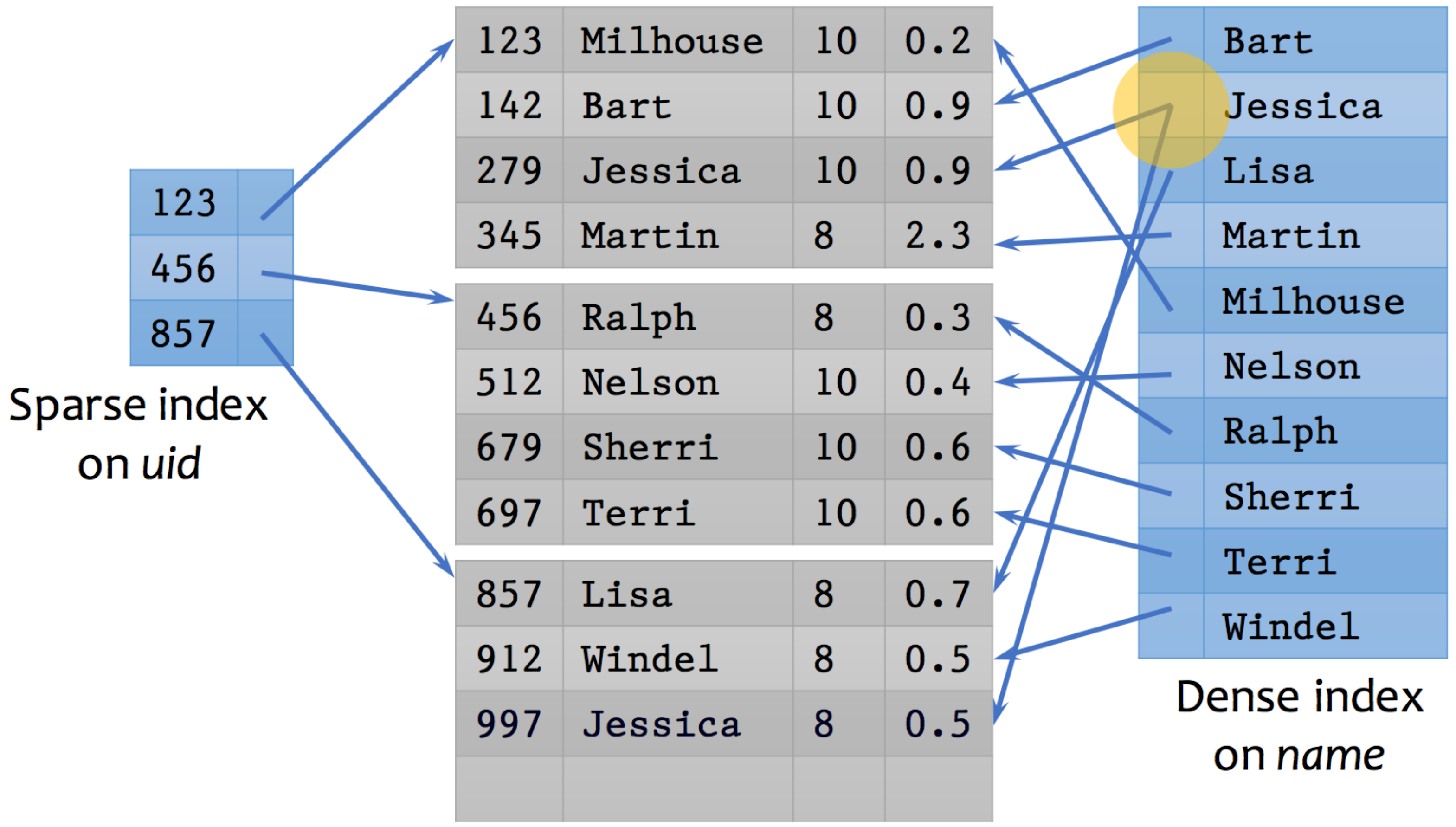
- Size of index file is much smaller than size of a data file

- Allows you to locate the block that contains the record quickly



Dense vs Sparse Index

- Dense: one index entry for each search key value
- Sparse: index entries only for some of the search values



123	
456	
857	

123	Milhouse	10	0.2
142	Bart	10	0.9
279	Jessica	10	0.9
345	Martin	8	2.3
456	Ralph	8	0.3
512	Nelson	10	0.4
679	Sherri	10	0.6
697	Terri	10	0.6
857	Lisa	8	0.7
912	Windel	8	0.5
997	Jessica	8	0.5

	Bart
	Jessica
	Lisa
	Martin
	Milhouse
	Nelson
	Ralph
	Sherri
	Terri
	Windel

Dense vs Sparse Index

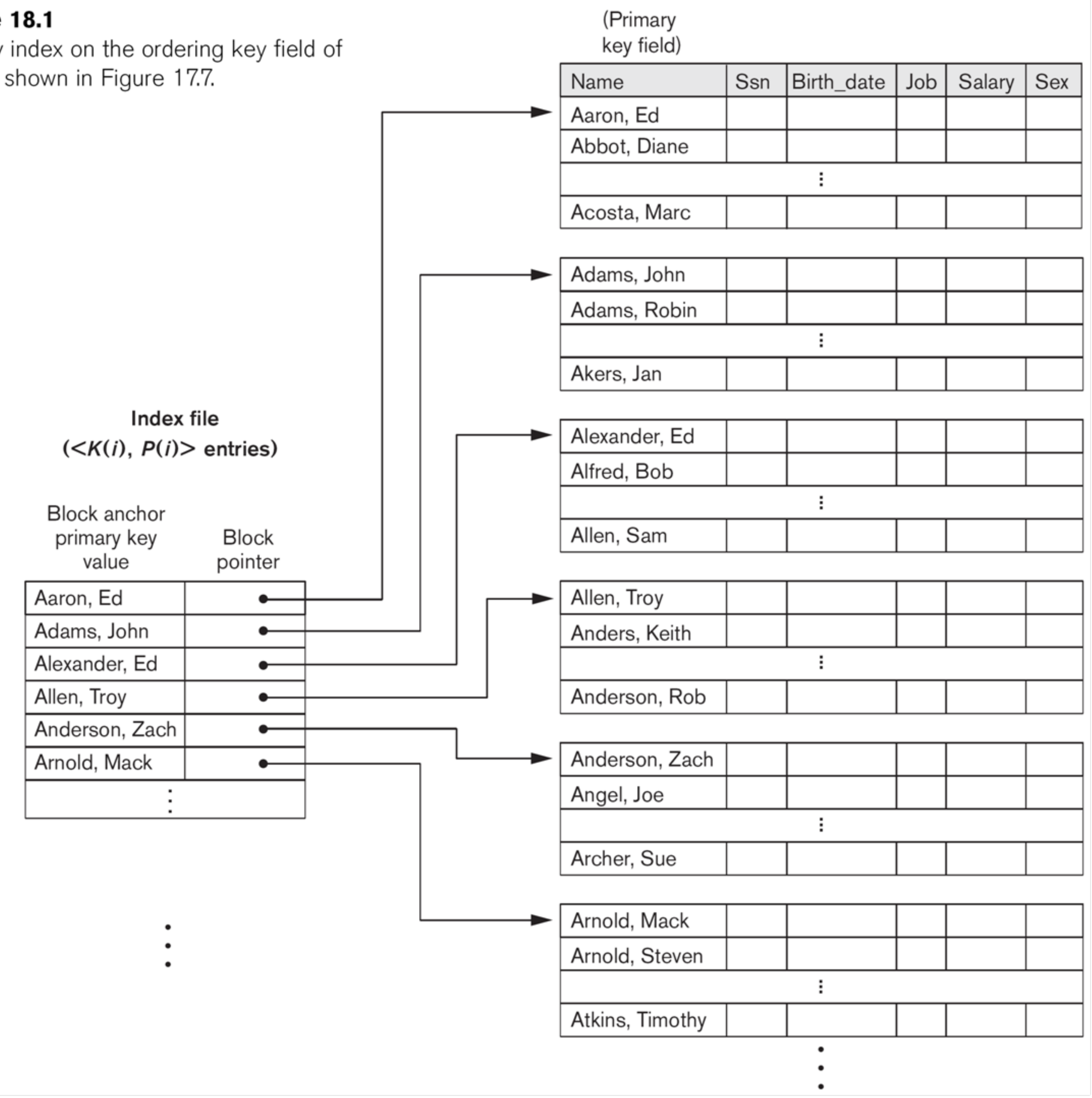
- Sparse:
  - Less index space
  - Potentially more varied time to find a record within a block
  - Easier update process
  - Records must be clustered
- Dense:
  - Can directly tell if a record exists without accessing file
  - More index space

Primary Index

- Created for the primary key of a table
  - Usually ordered index whose search key is the sort key for the sequential file
- Typically sparse index
  - Binary search on the index file requires fewer block accesses than on data file

Example: Primary Index

Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



Clustering Index

- Created on an ordered non-key field (not unique), known as a cluster field
- One index entry for each distinct value of the field
- Index entry points to the first data block that contains records with that field value
- Insertion and deletion are relatively straightforward

Example: Clustering Index

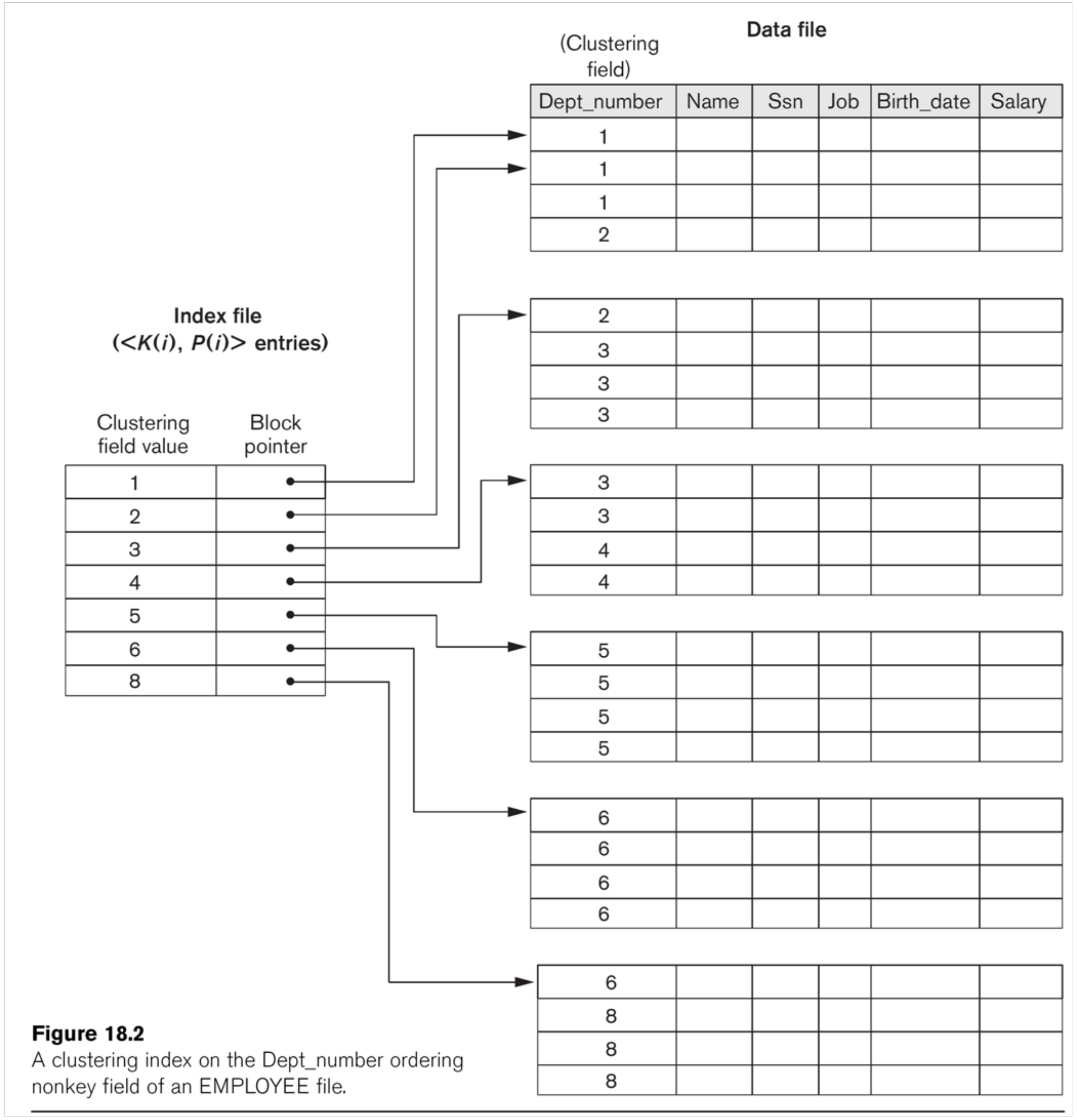
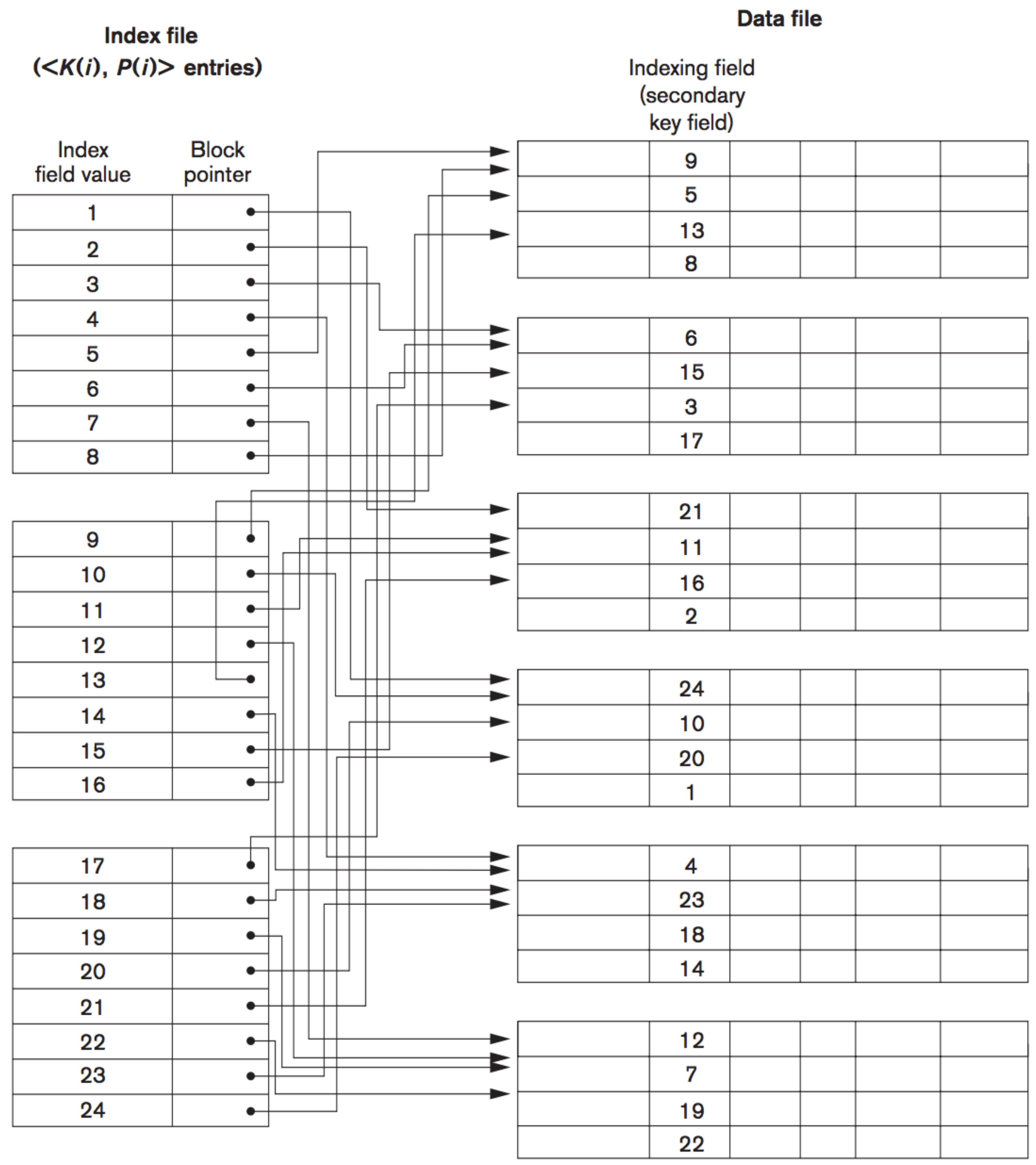


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Secondary Index

- An ordered index whose search key is not the sort key for the sequential file
    - Unique non-ordering key field results in a dense index with an entry for each record
    - Not unique fields results in an entry for each distinct value (nondense index)
- Multiple secondary indexes for the same file
- Requires more space and longer search time

Example: Secondary Index (Dense)

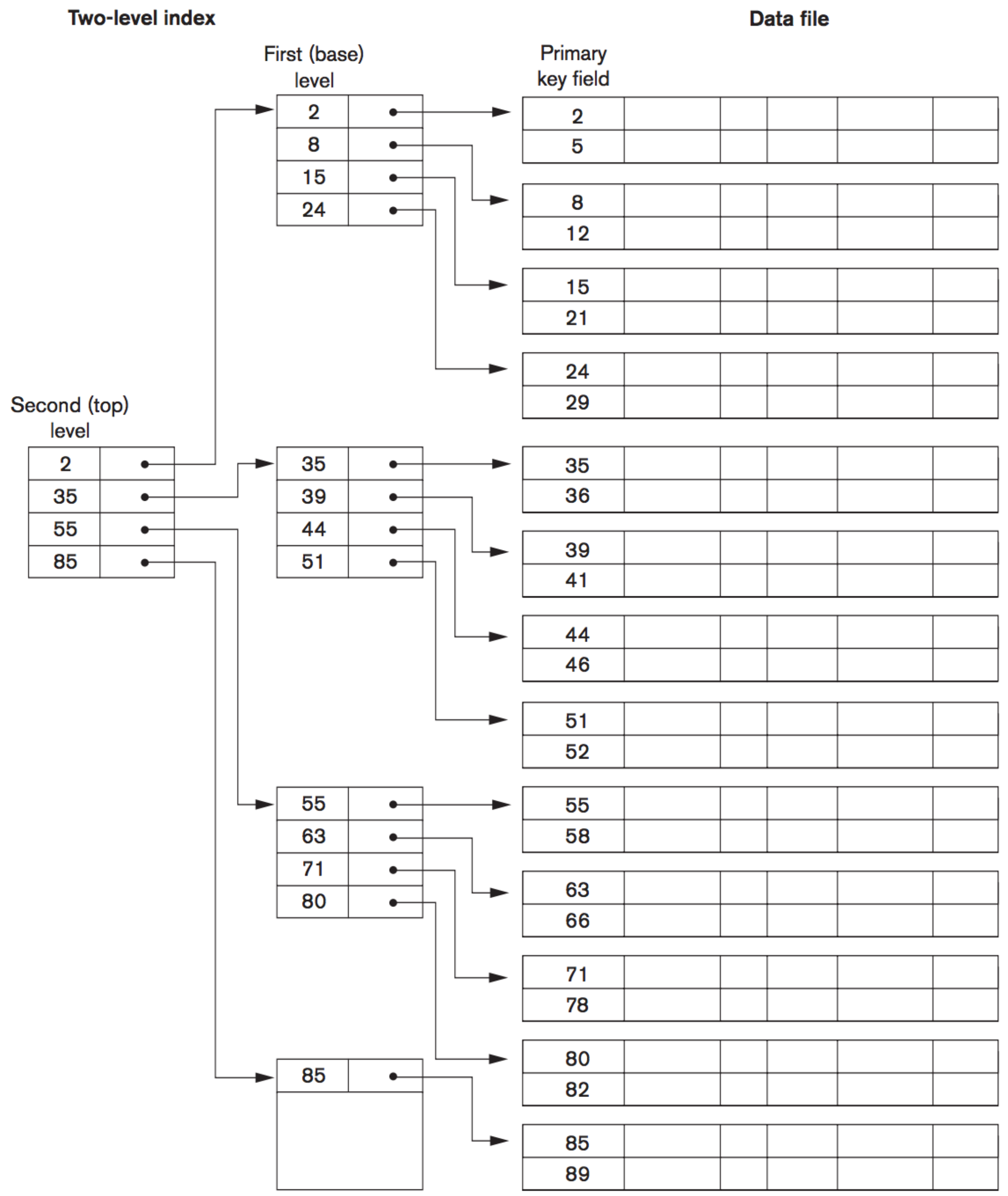


Summary of Types of Indexes

	Search key used for ordering of file	Search key not used for ordering of file
Search key is key of relation	Primary index — sparse	Secondary index (key) — dense
Search key is not key of relation	Clustering index — sparse	Secondary index (nonkey) — dense or sparse

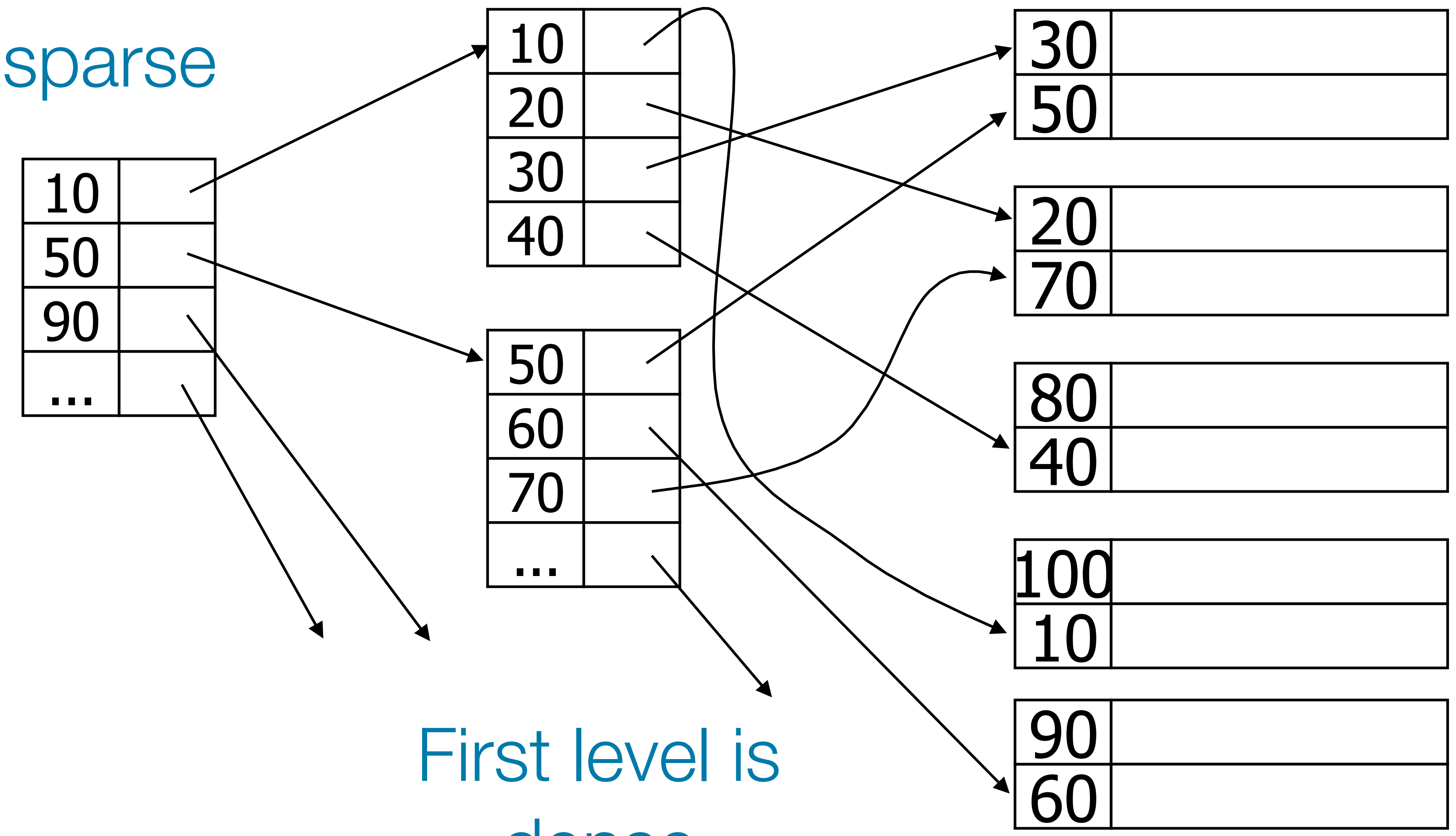
Multi-level Index

- What if index can't fit in memory?
- What if we want faster search than $\log_2(n)$?
- Solution: An index file is also a data file — create an index on the index file



Example: Multi-level Index

Second level is sparse



First level is dense

unordered file (according to search key)

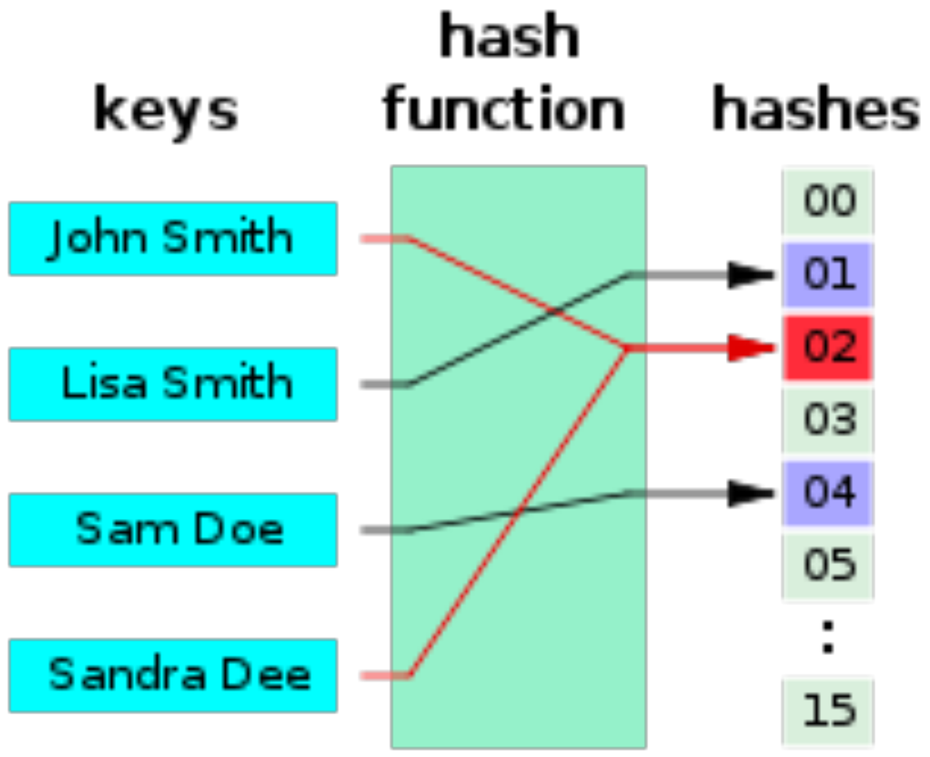
Hash Index

Hash Function

- A hash function, h, is a function which transforms a search key from a set K, into an index in a table of size n
  h: K —> {0, 1, …, n-2, n-1}

- Bucket is a location (slot) in the bucket array (or the hash table)

- Different search keys can be hashed into the same bucket



Hash Function: Properties

- Minimize collisions (different hash keys should hash to different values whenever possible)
- Uniform — each bucket is assigned the same number of search key values from the set of all possible values
- Random — each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file
- Be easy and quick to compute

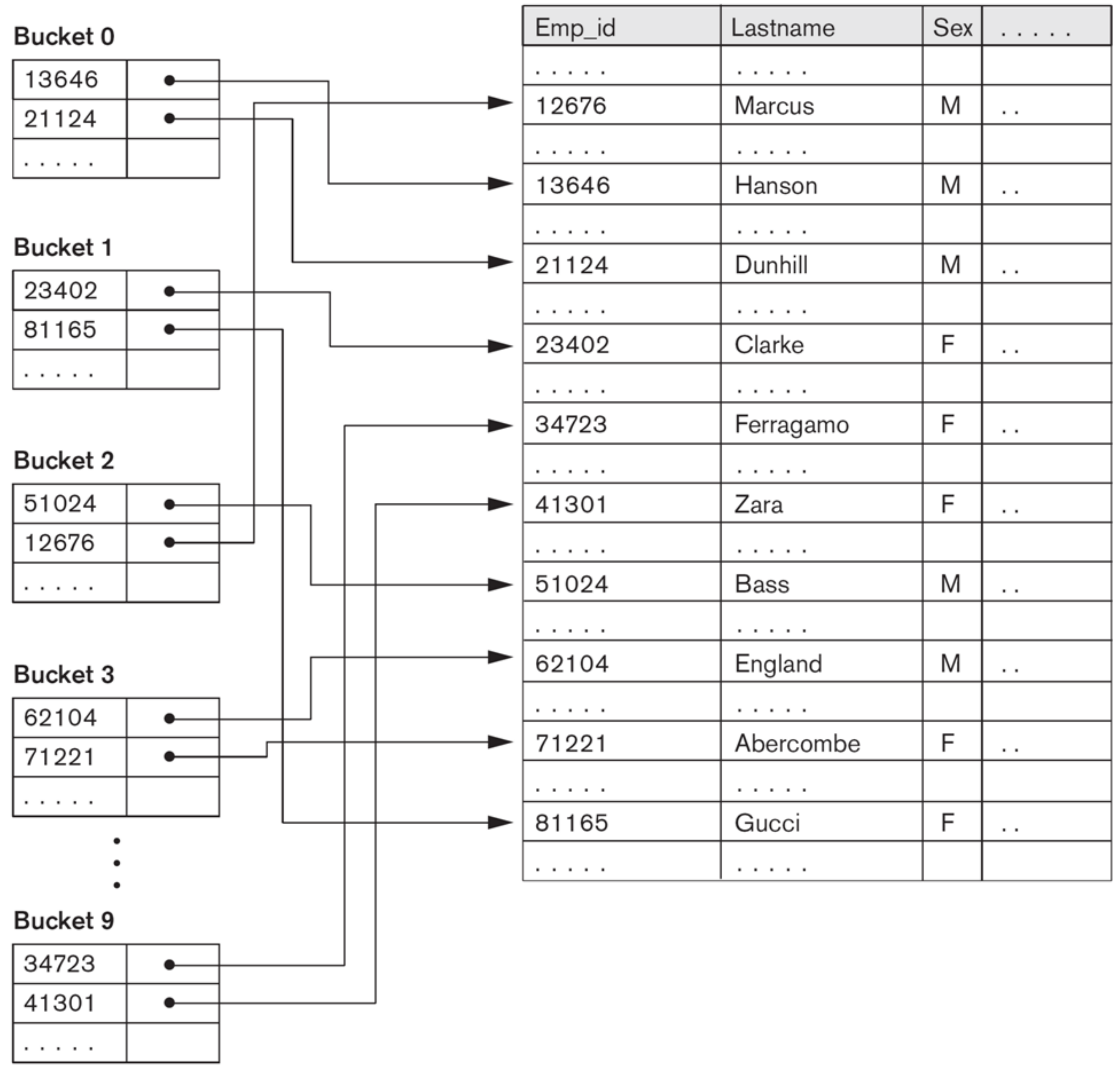
Hash Function: Usage

- The mark of a computer scientist is their belief in hashing
    - Possible to insert, delete, and lookup items in a large set in O(1) time per operation
- Widely used in a variety of applications
    - Cryptography, table or database lookup, caches for large datasets, finding duplicate records, finding similar “items”, etc.

Hash Index

- Hash function, h, distributes all search-key values to a collection of buckets
- Each bucket contains a primary page plus overflow pages
  - Buckets contain data entries
  - Entire bucket has to be searched sequentially to locate a record (since different search-key values may be mapped to same bucket)

Example: Hash Index



Emp_id	Lastname	Sex	. . . . .
. . . . .	. . . . .		
12676	Marcus	M	. .
. . . . .	. . . . .		
13646	Hanson	M	. .
. . . . .	. . . . .		
21124	Dunhill	M	. .
. . . . .	. . . . .		
23402	Clarke	F	. .
. . . . .	. . . . .		
34723	Ferragamo	F	. .
. . . . .	. . . . .		
41301	Zara	F	. .
. . . . .	. . . . .		
51024	Bass	M	. .
. . . . .	. . . . .		
62104	England	M	. .
. . . . .	. . . . .		
71221	Abercombe	F	. .
. . . . .	. . . . .		
81165	Gucci	F	. .
. . . . .	. . . . .		

Figure 18.15
Hash-based indexing.

Bucket Overflows

- Causes of bucket overflows
    - Insufficient buckets
    - Skew in distribution of records
        - Multiple records with the same search-key value
        - Hash function produces non-uniform distribution
- Overflow chaining links the buckets together to handle when a certain bucket has a large number of entries

Example: Overflow Hash

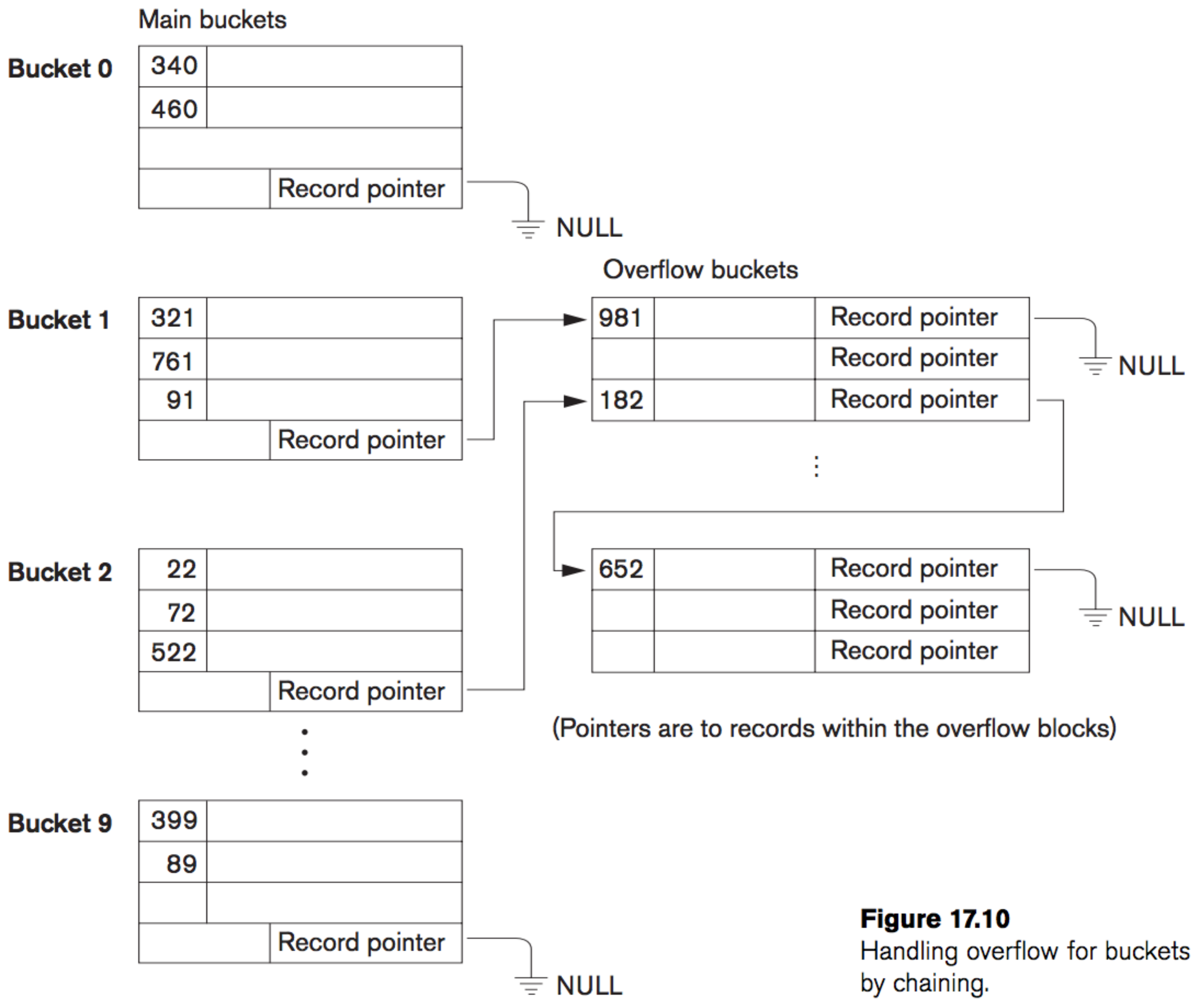
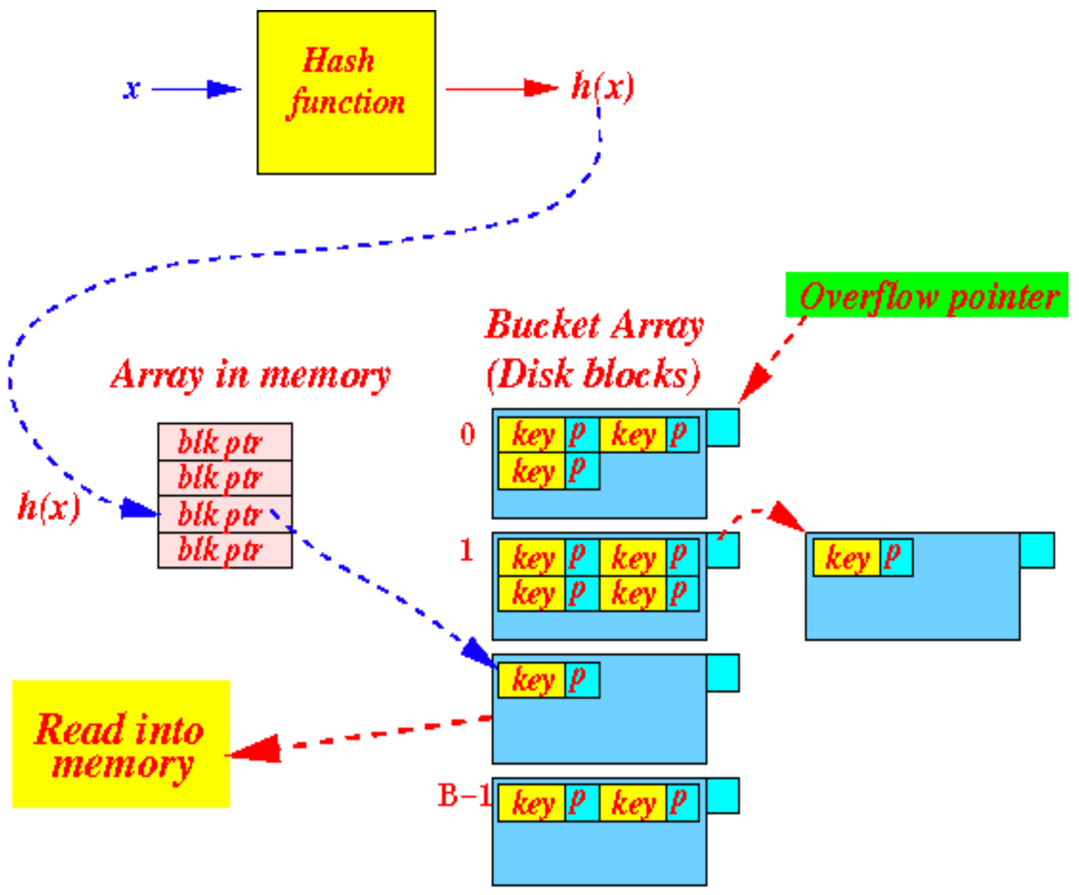


Figure 17.10
Handling overflow for buckets by chaining.

Hash Index Query

- Compute hash value h(x)
- Read the disk block pointed to by the block pointer h(x) into memory
- Linear search the bucket for x, ptr(x)
- Use ptr(x) to access x on disk



http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/3-index/hashing.html

Hash Index Insert/Deletion

- Hash the new item h(x)
- Find the hash bucket for the item
- Add/delete item from hash bucket
    - If there is insufficient space, allocate an overflow bucket and then add it to the overflow bucket

Static Hashing Issues

Databases grow and shrink with time, while static hashing assumes a fixed set of B bucket addresses

- If initial number of buckets is too small, performance will degrade due to too much overflow
- If space allocated for anticipated growth or database shrinks, buckets will be underutilized and space will be wasted

Fixing Static Hashing

- One solution: periodic re-organization of the file with new hash functions
    - Expensive — rehash all keys into a new table!
    - Disrupts normal operations
- Another (better) solution: dynamic hashing techniques that allow size of the hash table to change with relative low cost

Extendible Hashing (Fagin, 1979)

Main idea:

- Directory of pointers to the buckets
- Double the number of buckets by splitting just the bucket that overflowed
- Directory is much smaller than file, so doubling it is cheaper

Extendible Hashing Structure

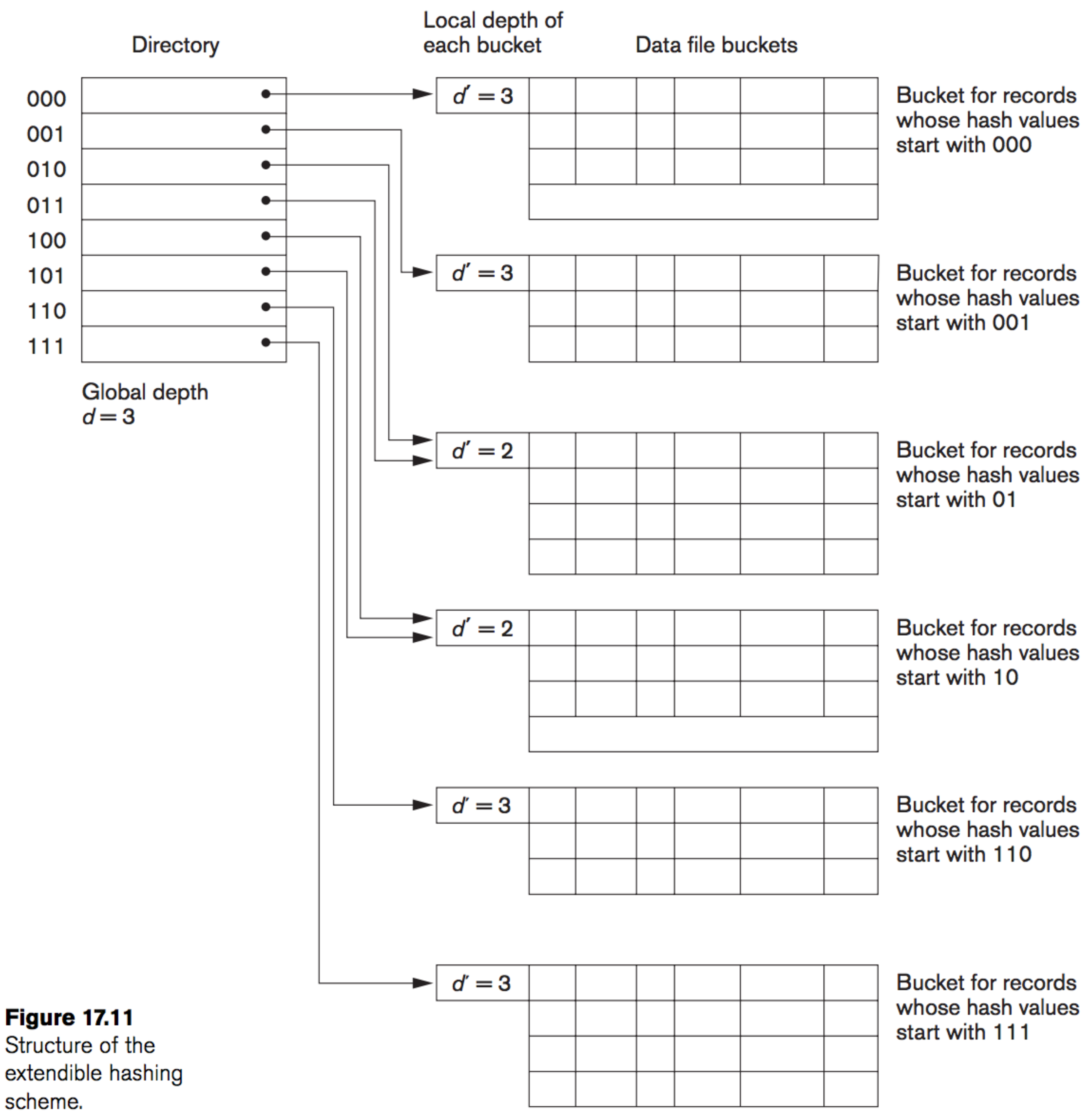

Directory
Local depth of each bucket
Data file buckets
000
001
010
011
100
101
110
111
Global depth $d = 3$
$d' = 3$
Bucket for records whose hash values start with 000
$d' = 3$
Bucket for records whose hash values start with 001
$d' = 2$
Bucket for records whose hash values start with 01
$d' = 2$
Bucket for records whose hash values start with 10
$d' = 3$
Bucket for records whose hash values start with 110
$d' = 3$
Bucket for records whose hash values start with 111

Figure 17.11
Structure of the extendible hashing scheme.

Example: Instructor Table

Instructor(<u>ID</u>, Lname, Department, Salary) and want to build a secondary index on the department attribute

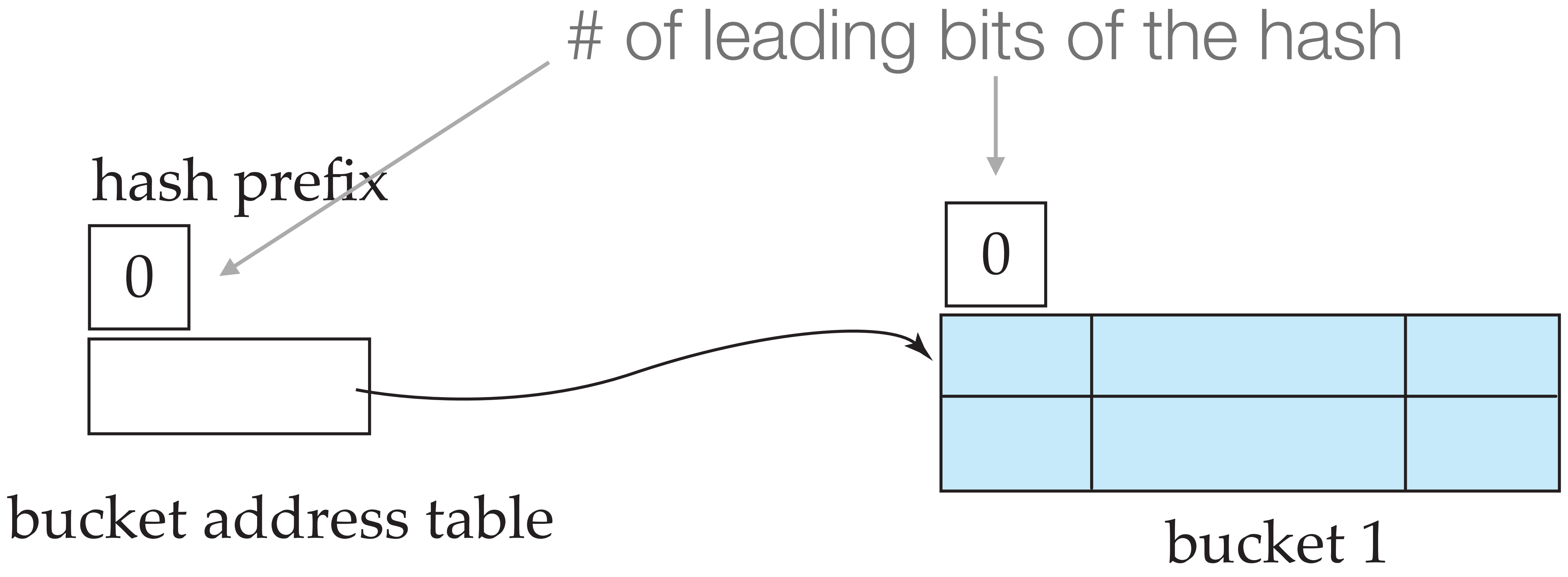
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Example: Hash Index For Dept

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

We will use high-order bits (left —> right)

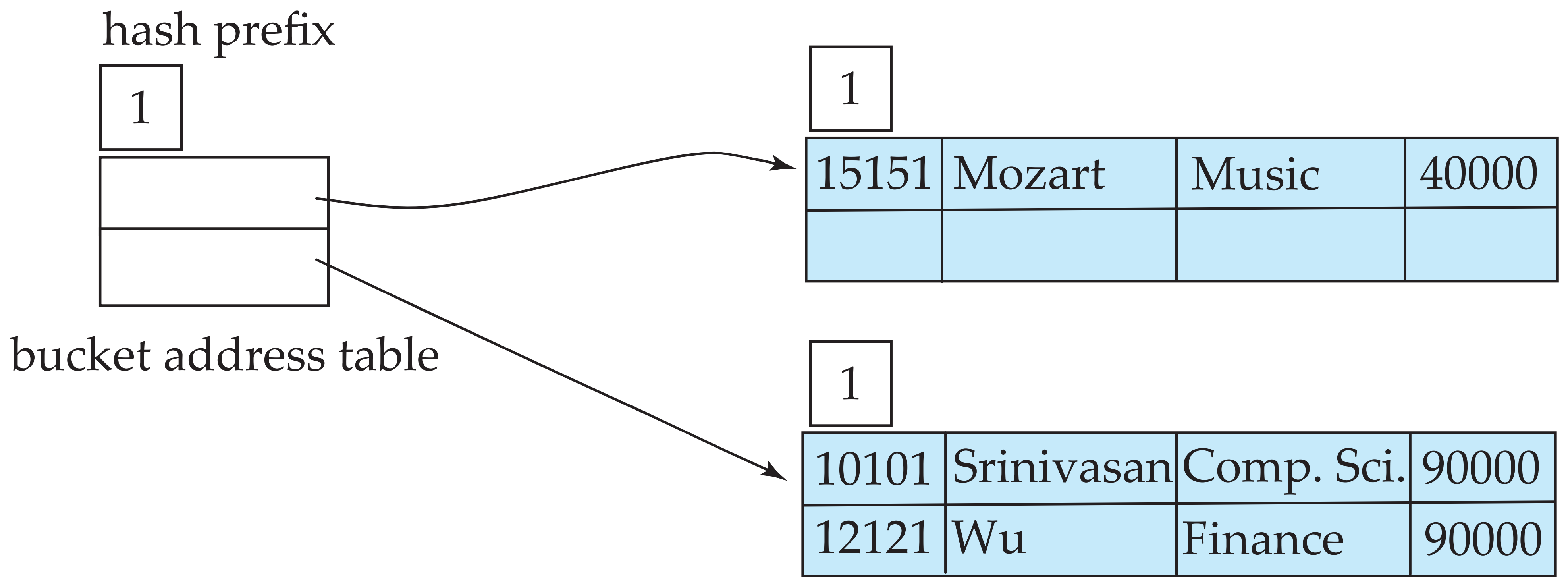
Example: Extendible Hashing Structure



http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/PPT-dir/ch11.ppt

Example: Extendible Hashing Structure (2)

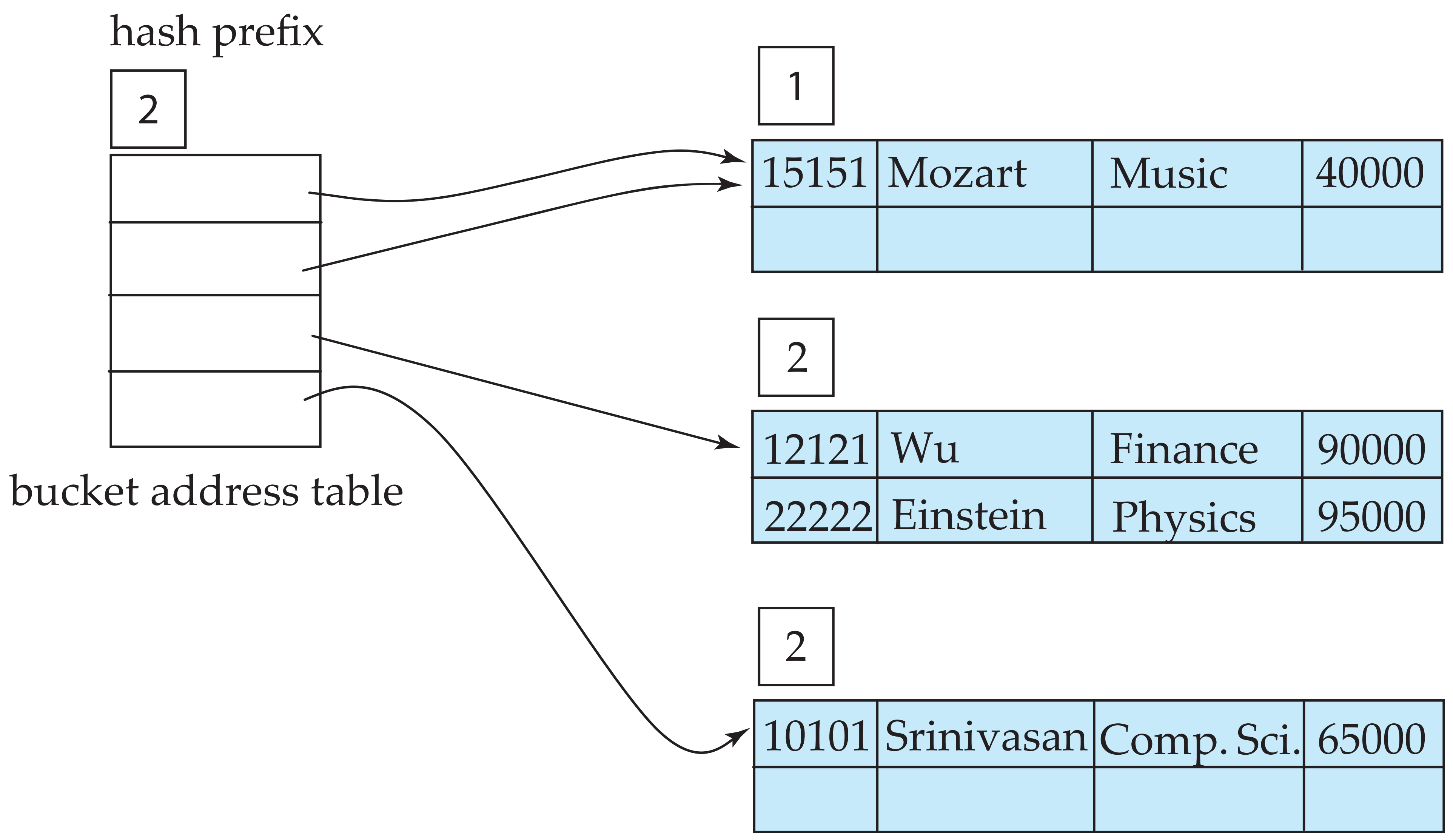
Insert 3 new tuples: (15151, Mozart, Music, 40000), (10101, Srinivasan, Comp. Sci, 90000), (12121, Wu, Finance, 90000)



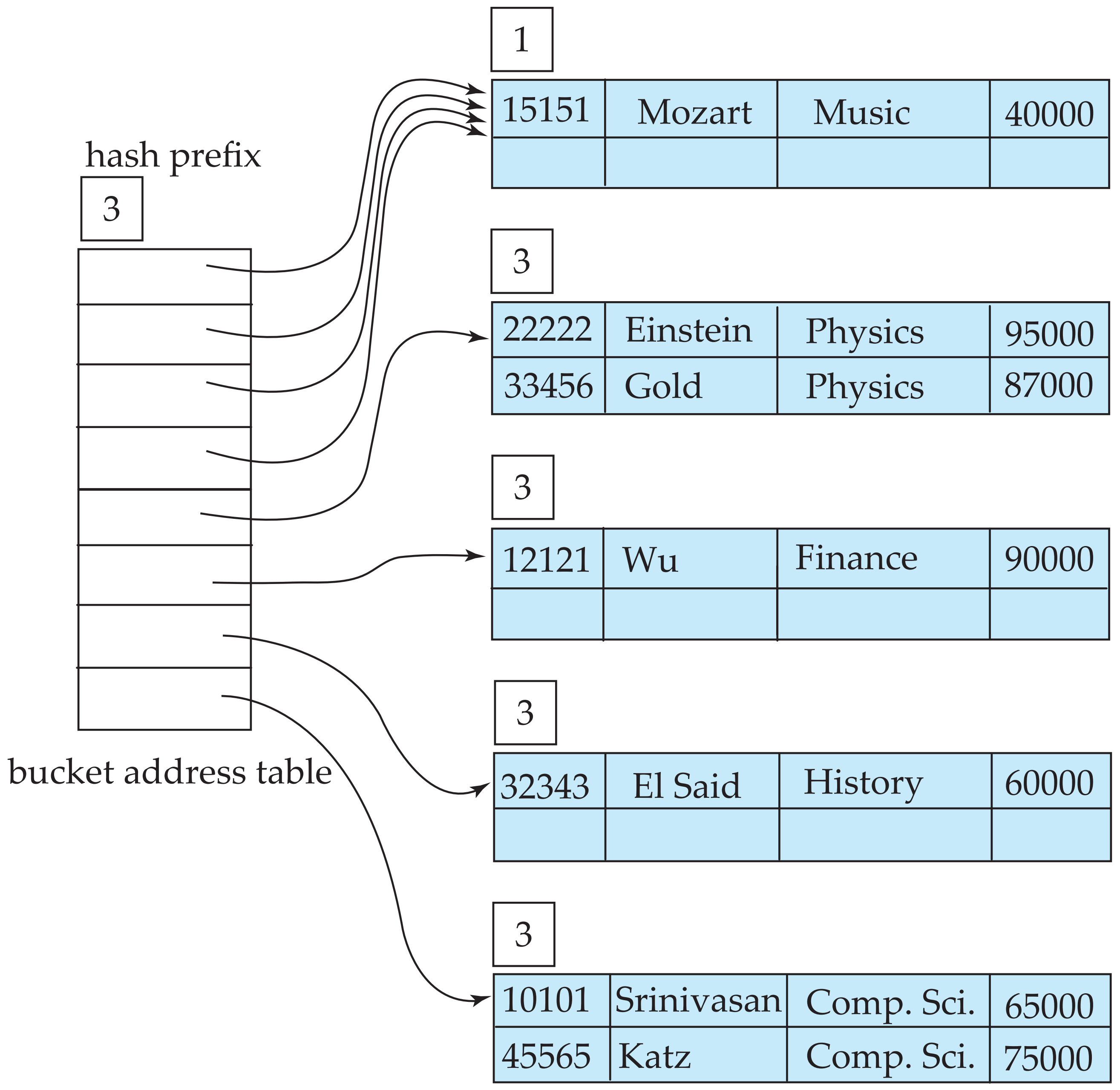
Insert next tuple (22222, Einstein, Physics, 95000) with leading hash bit = 1

Example: Extendible Hashing Structure

Matches the 2nd bucket and overflows => increase the hash prefix and have the new bucket



Example: Extendible Hashing Structure


hash prefix
3
bucket address table
1
15151 Mozart Music 40000
3
22222 Einstein Physics 95000
33456 Gold Physics 87000
3
12121 Wu Finance 90000
3
32343 El Said History 60000
3
10101 Srinivasan Comp. Sci. 65000
45565 Katz Comp. Sci. 75000

Other Dynamic Hashing Schemes

- Dynamic hashing (Larson, 1978): precursor to extendible hashing with the main difference in the organization of the directory with tree-structured directory with internal nodes and leaf nodes
- Linear hashing (Litwin, 1980): allows incremental growth without needing a directory at the cost of more bucket overflows

Dynamic Hashing Properties

- Benefits
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages
  - Additional level of indirection on lookup (2 block access instead of one)

Ordered Files vs Hashing

- Relative frequency of insertions and deletions
  - Ordered files are much more expensive to keep sorted
  - Cost of periodic re-organization in hashing
- Is average access time more important than worst-case access time?

Ordered Files vs Hashing

- What types of queries are expected?
  - Hashing is good for equality
  - Ordered files are preferred if range queries are common

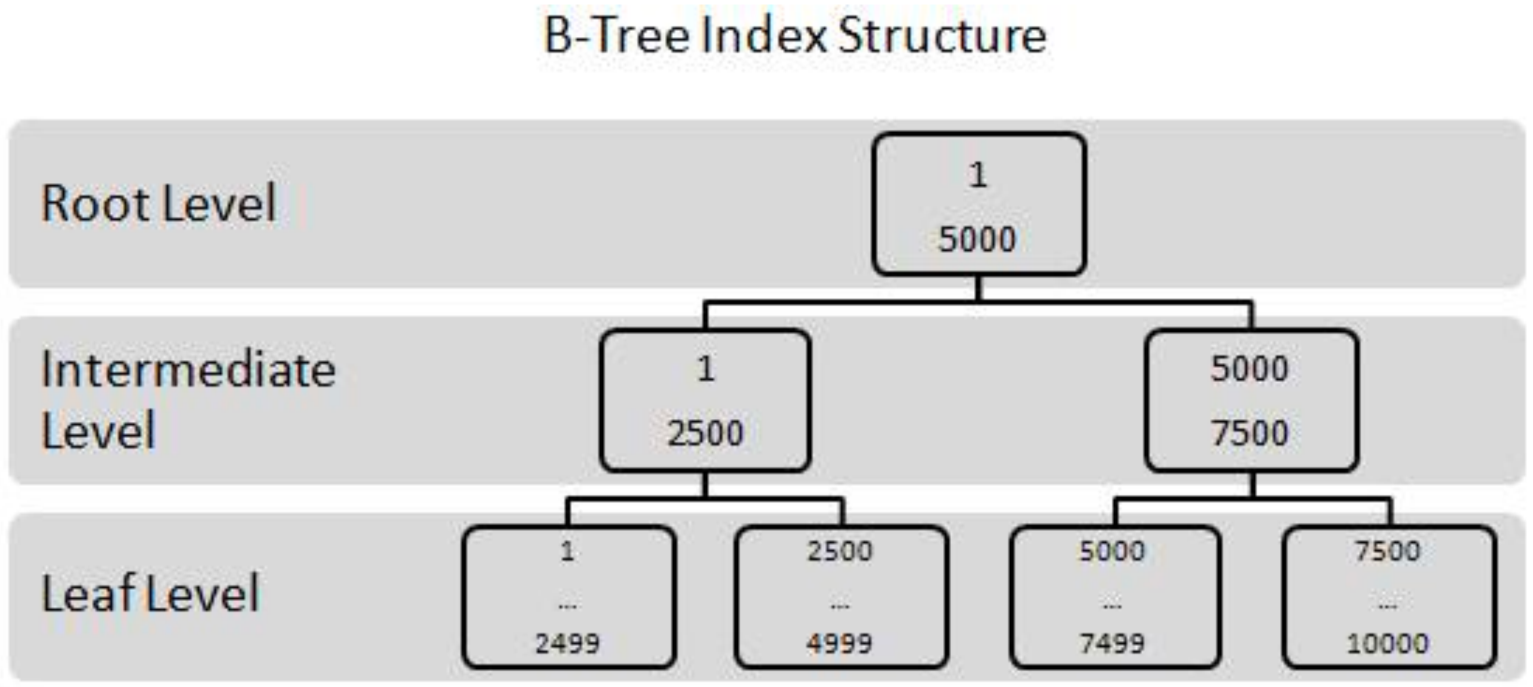
B^+-Tree

B^+-Tree

- Dynamic, multi-level tree data structure
    - Adjusted to be height-balanced (all leaf nodes are at same depth)
    - Good performance guarantee — supports efficient equality and range search
- Widely used in DBMS

B^+-Tree Basics

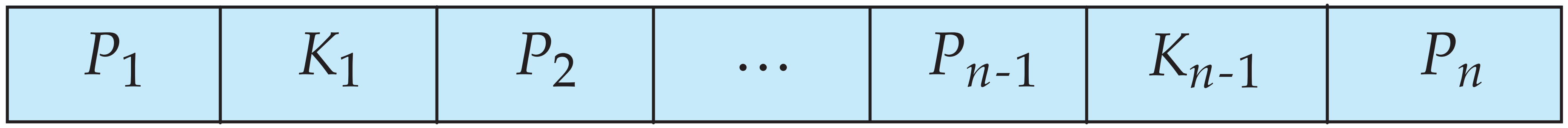
- Order p: maximum number of children at each node
- Every node contains m entries, with
    - Minimum 50% occupancy
    - Only exception is root node



https://www.simple-talk.com/iwritefor/articlefiles/735-image012.jpg

B$^+$-Tree Node Structure

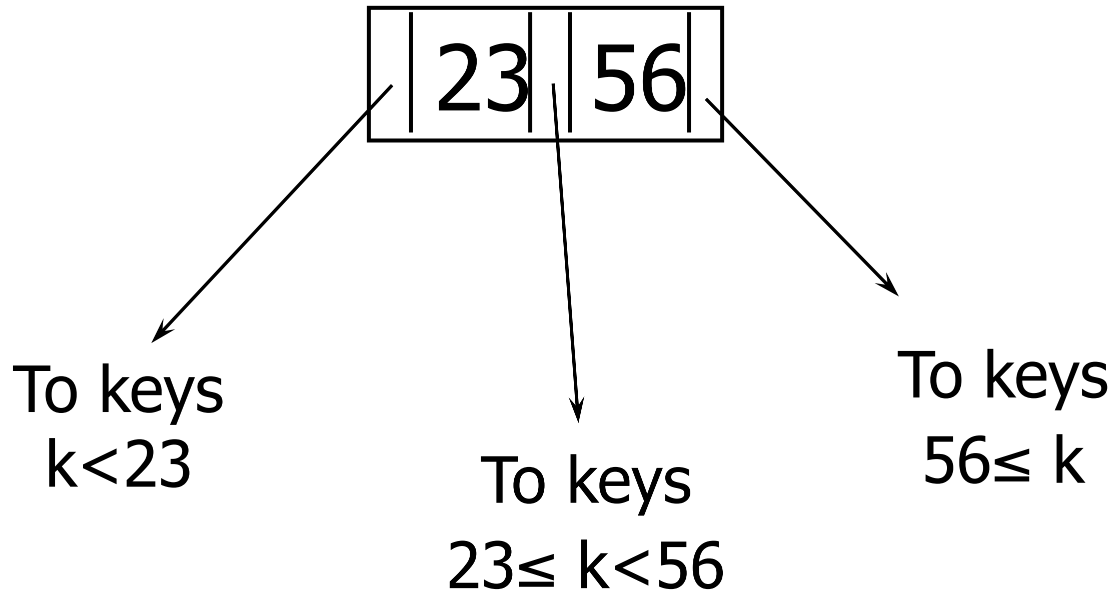
- Typical node



  - K_i are the search-key values
  - P_i are the points to the children (for non-leaf nodes) or pointers to records (for leaf nodes)
- Search keys in nodes are ordered
  - $K_1 < K_2 < \ldots < K_{n-1}$

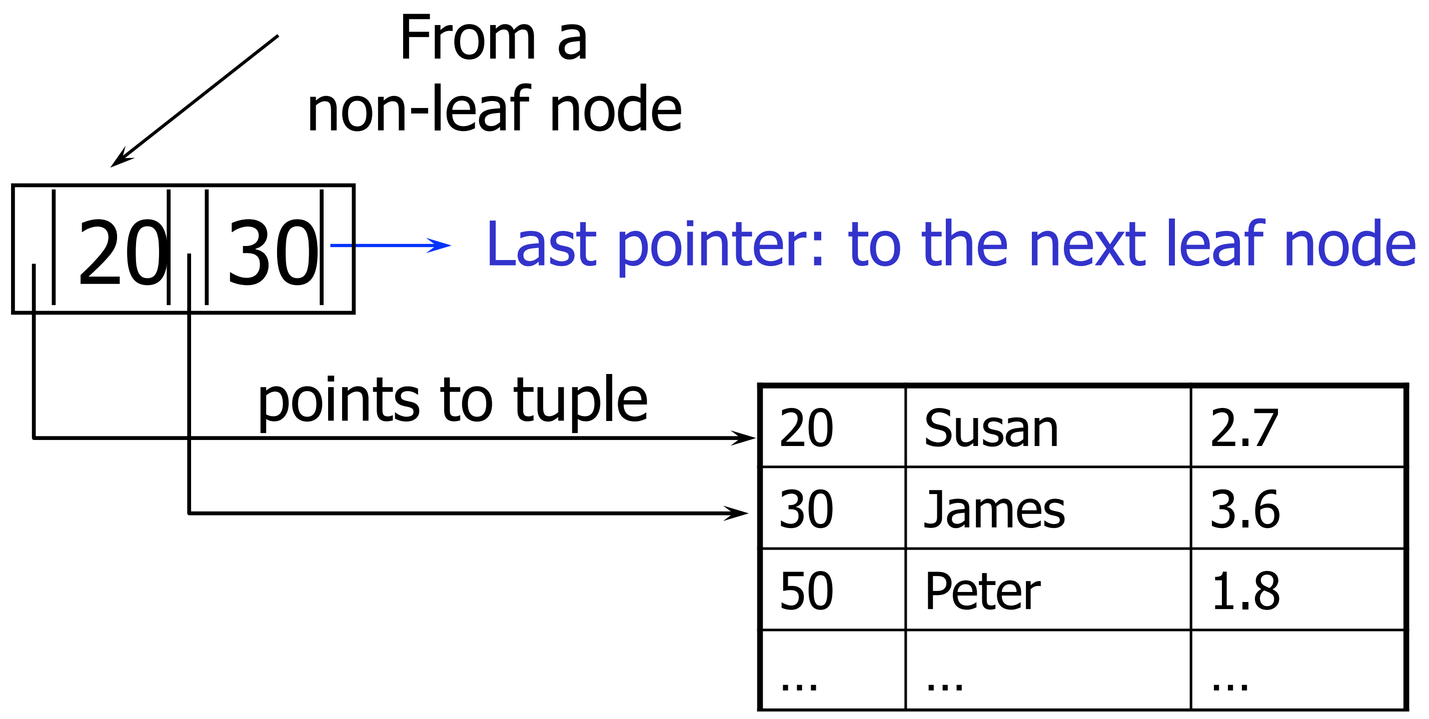
B$^+$-Tree: Non-Leaf Node

- Multi-level sparse index on the leaf nodes
- All search-keys in the subtree to which P_1 points to are less than K_1
- All search-keys in the subtree to which P_n points to have values greater than K_{n-1}



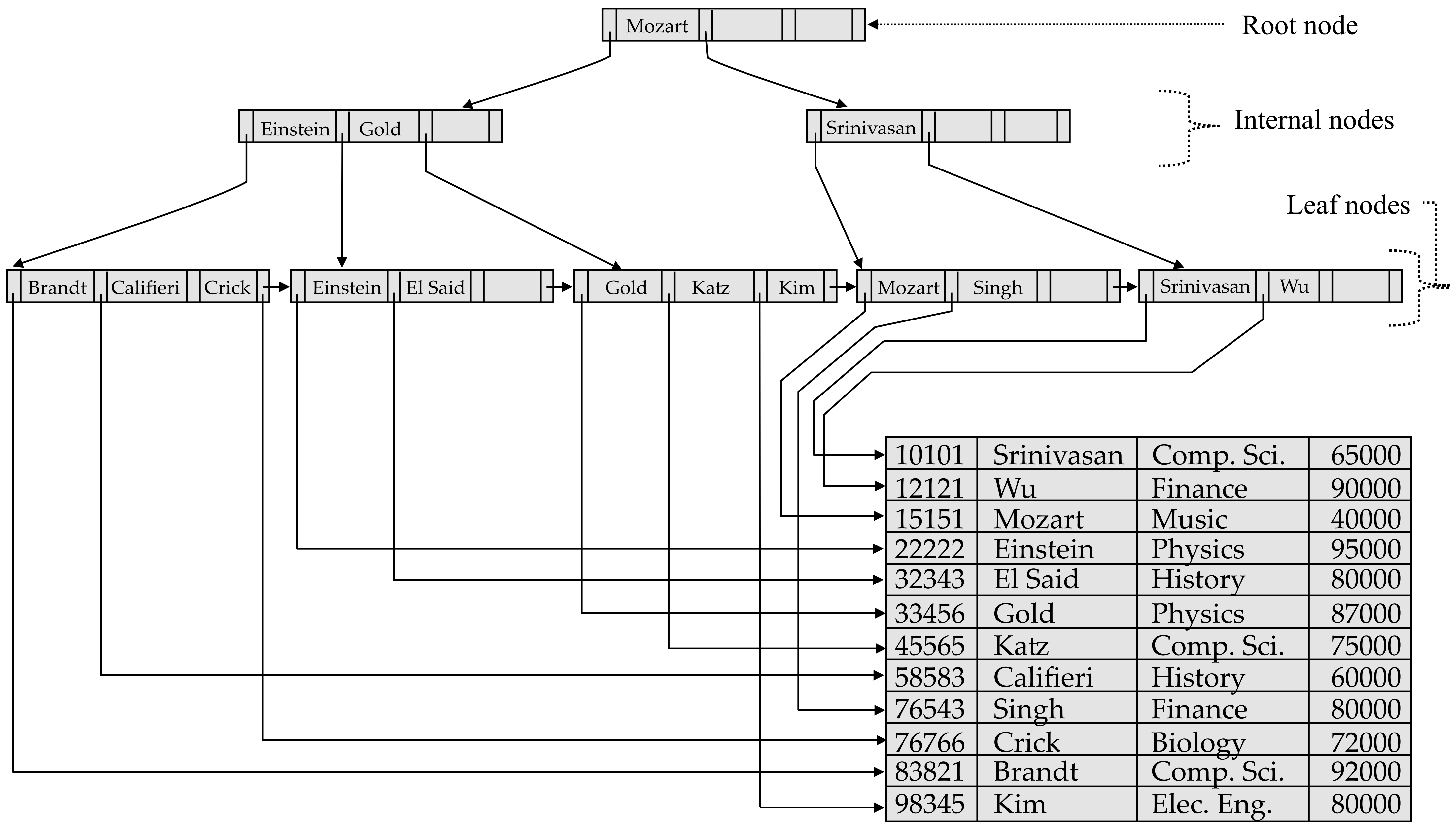
B$^+$-Tree: Leaf Node

- All pointers (except the last one) point to tuples or records
    - Search key values are sorted in order
- Last pointer (P_n) points to next leaf node in search-key order



20	Susan	2.7
30	James	3.6
50	Peter	1.8
…	…	…

Example: B$^+$-Tree



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

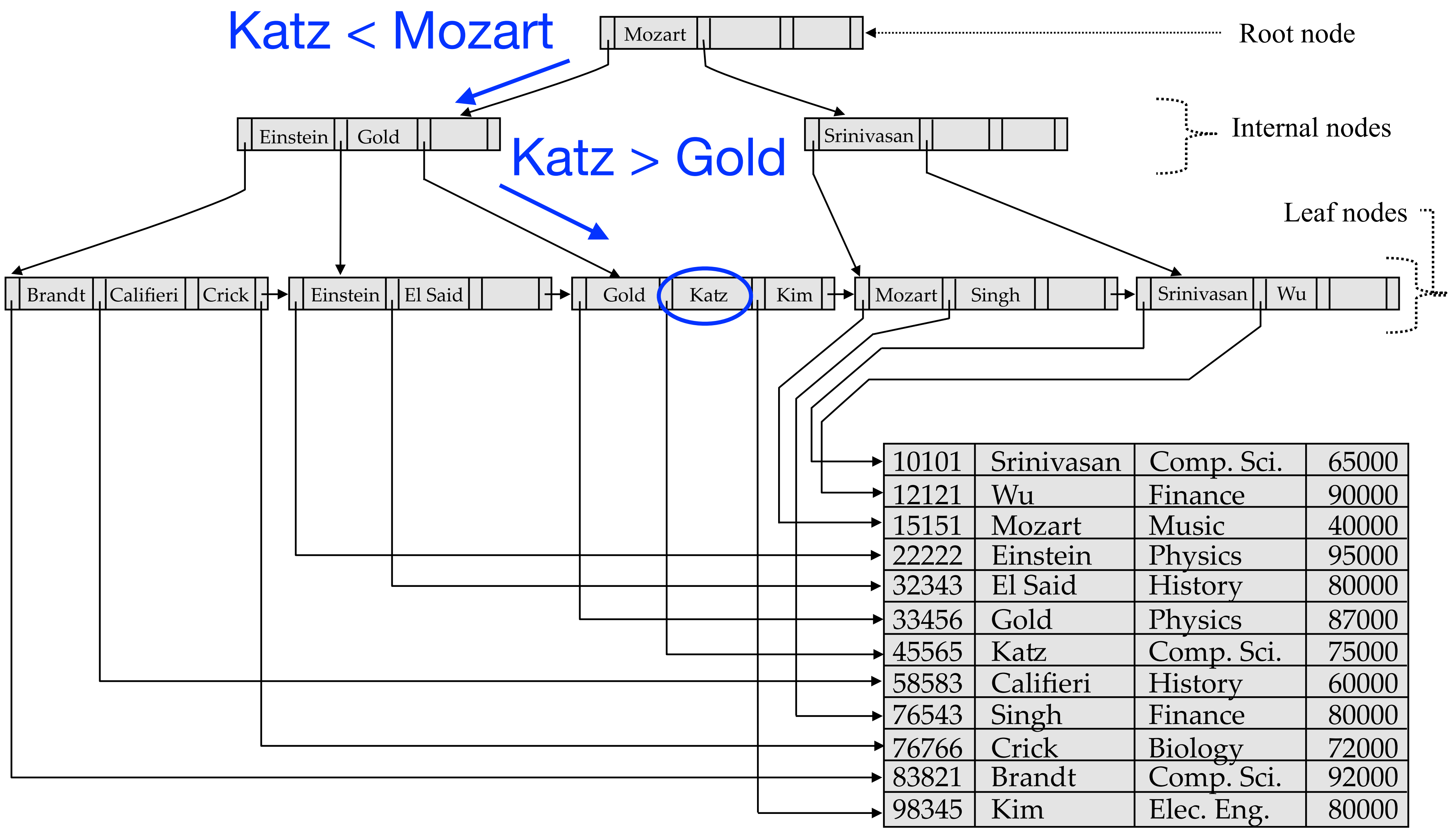
Figure from Database System Concepts book

B^+-Tree Search

- Start from root
- Examine index entries in non-leaf nodes to find the correct children
    - Searched using a binary or linear search
- Traverse down the tree until a leaf node is reached

Example: B$^+$-Tree Exact Query

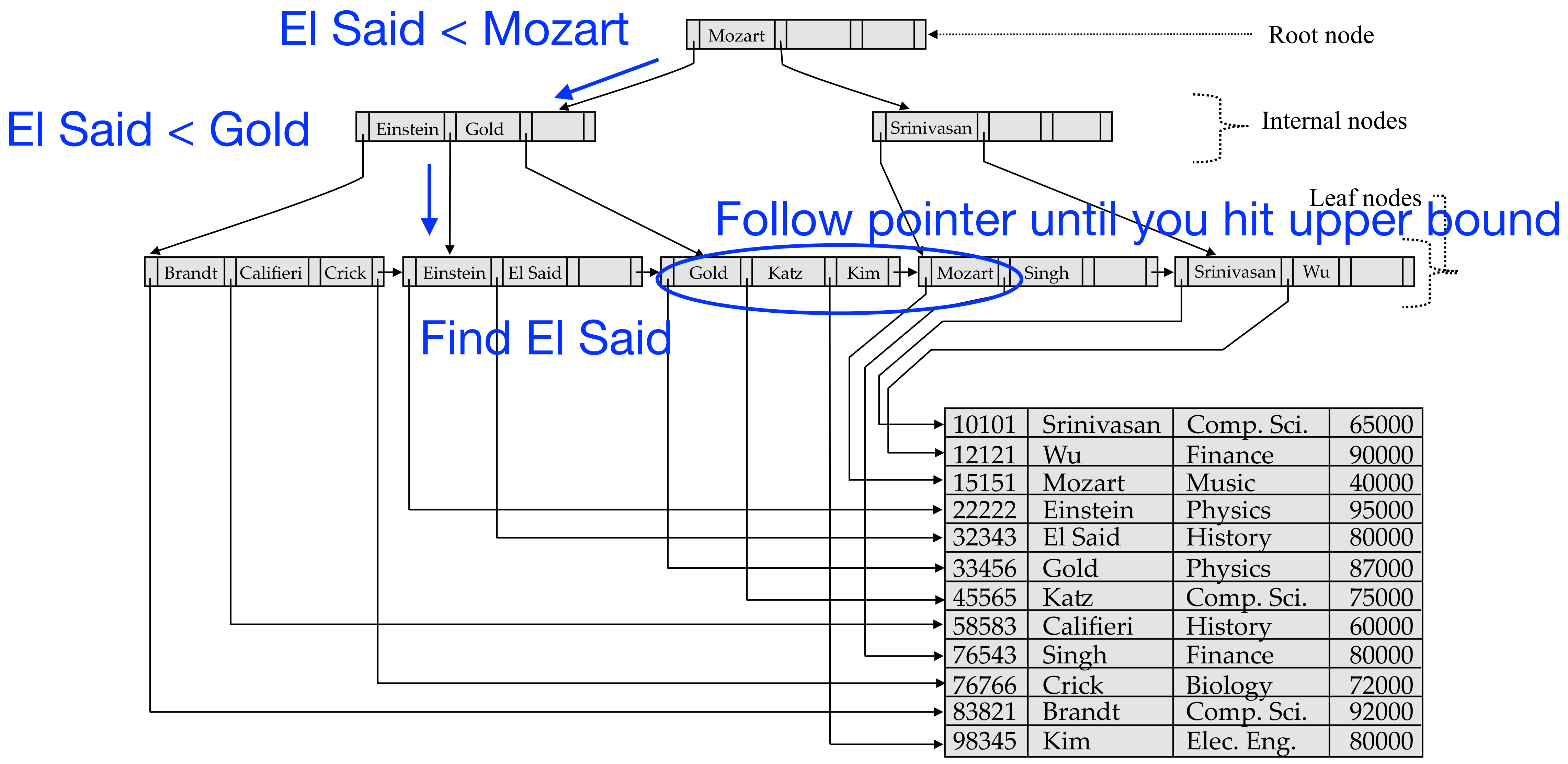
SELECT * FROM instructor WHERE name = ‘Katz’;



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

Example: B$^+$-Tree Range Query

SELECT * FROM instructor WHERE name > “El Said” AND name < “Singh”;



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

B$^+$-Tree Insert

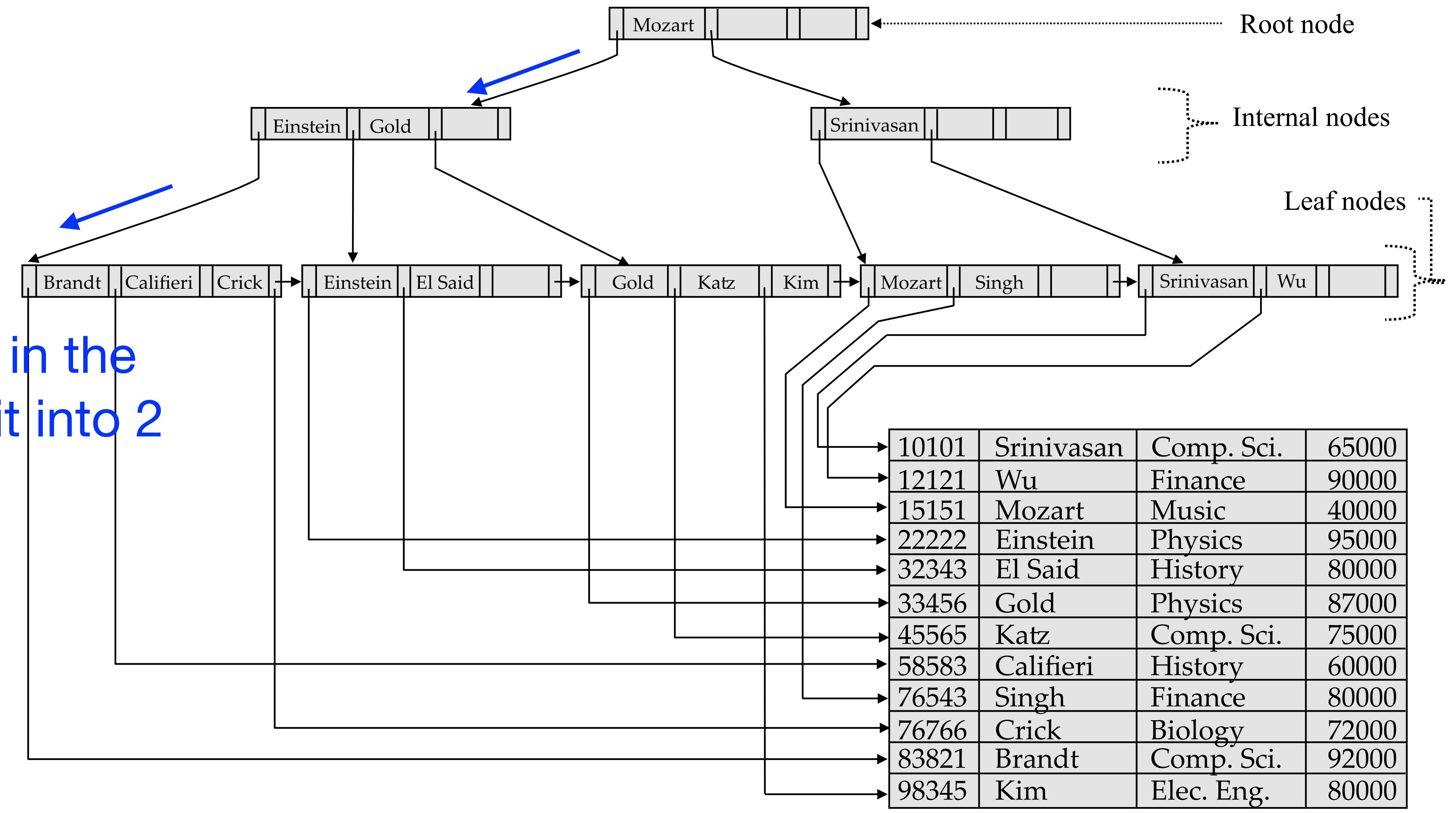
- Find the leaf node in which the search-key value would appear
- If the search-key value is already present in the leaf node
  - Add the record to the file
  - Add pointer to the bucket (if necessary)

B^+-Tree Insert

- If search-key value is not present
  - Add record to main file
  - If room in the leaf node, insert (key, pointer) pair in leaf node
  - Otherwise split the node along with the new (key, pointer) pair

Example: B$^+$-Tree Insert

INSERT INTO instructor(name) VALUES('Adams');

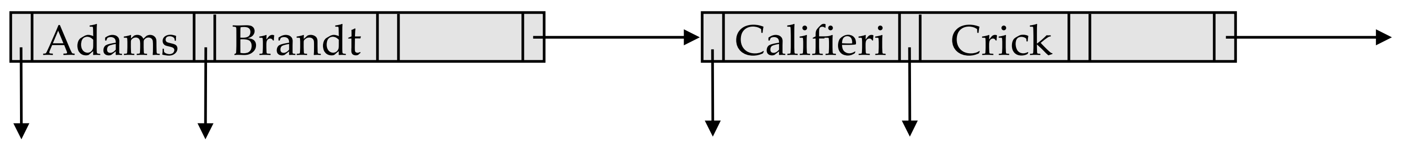


10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

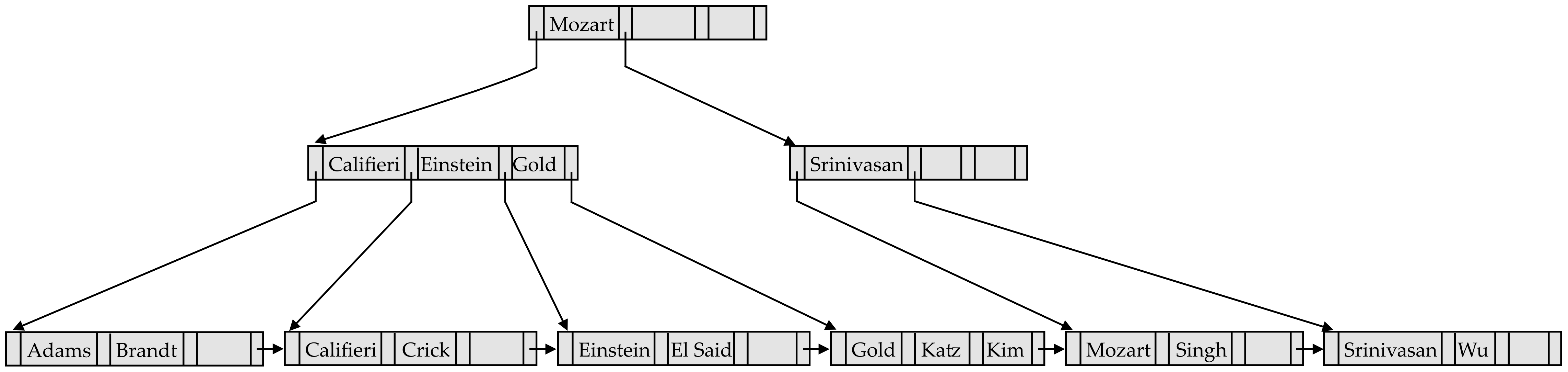
No room in the leaf - split into 2

Example: B$^+$-Tree Insert

Split so that Adams and Brandt on one side, Califieri and Crick on the other

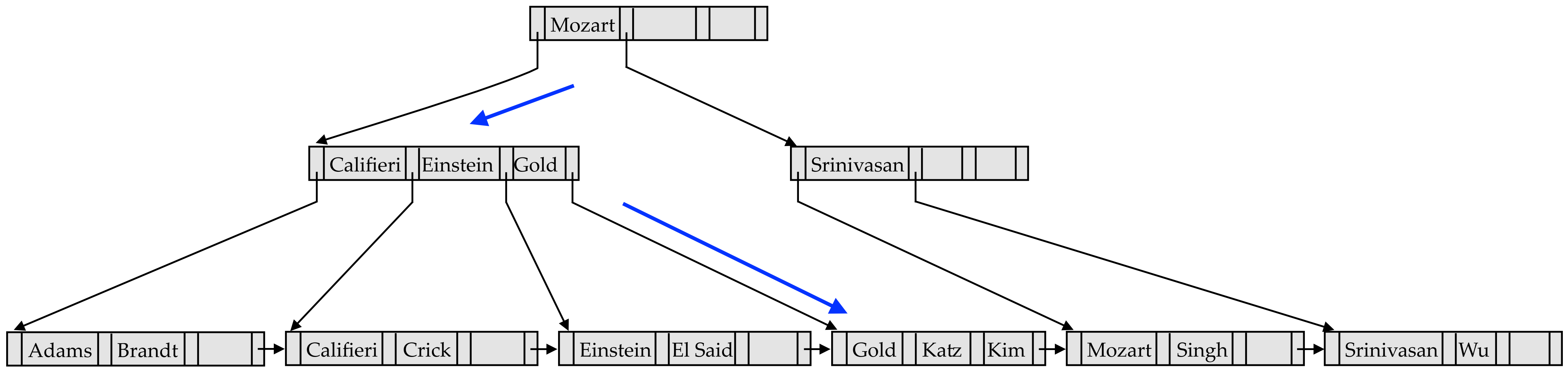


Since we are introducing new leaf node, we need to update the parent leaf…



Example: B$^+$-Tree Insert

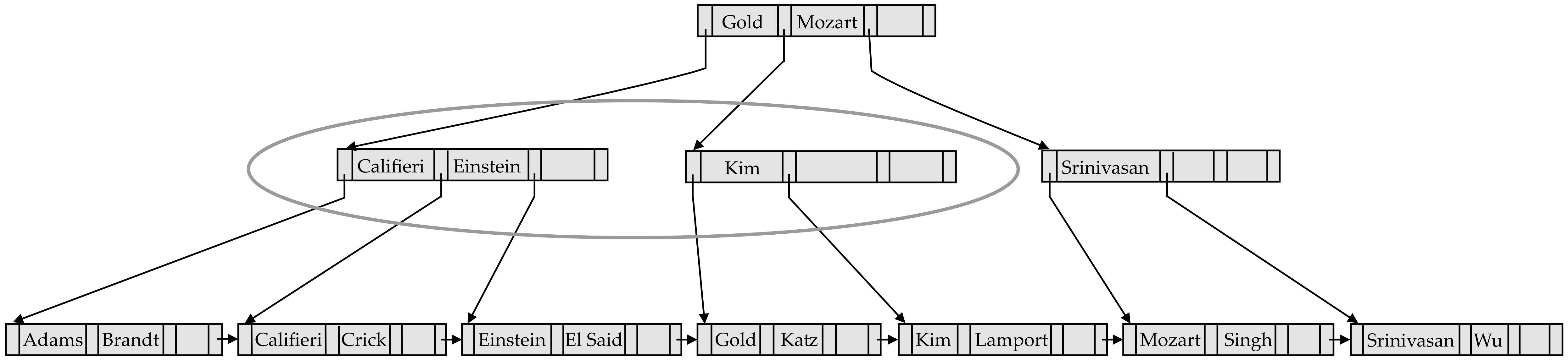
INSERT INTO instructor(name) VALUES('Lamport');



No room in the leaf - split into 2

Example: B$^+$-Tree Insert

But after split, there is no room in the parent node either, so parent needs to be split which affects the root

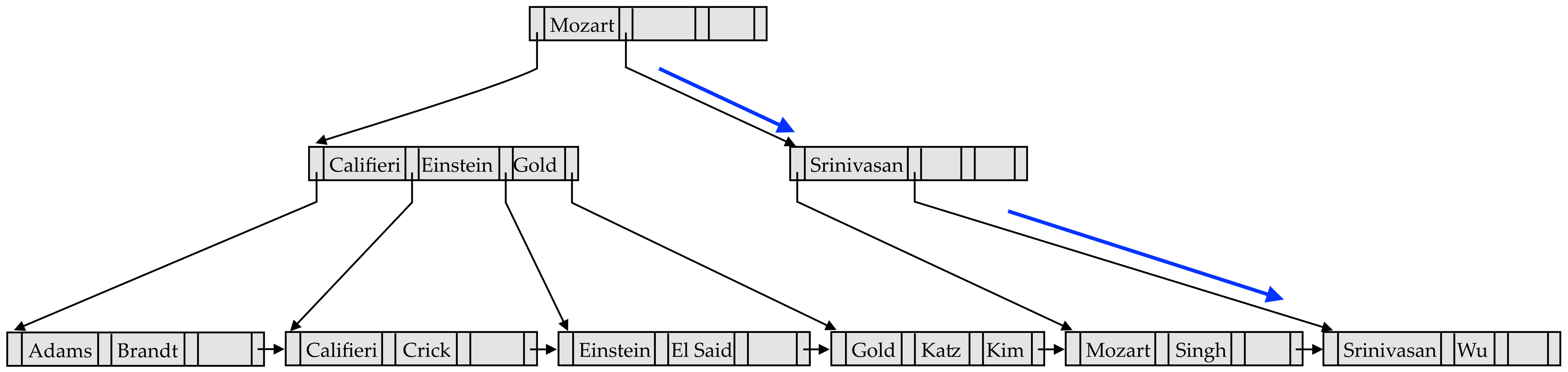


B^+-Tree Deletion

- Find the leaf node in which the search-key value appears and delete it from the main file and bucket
- Delete the (key, pointer) pair from the leaf node
    - If underflow occurs (leaf node is under minimum size)
        - Merge with sibling (reduce tree pointers from parent nodes)
        - Redistribute entries from left or right sibling if merge not possible

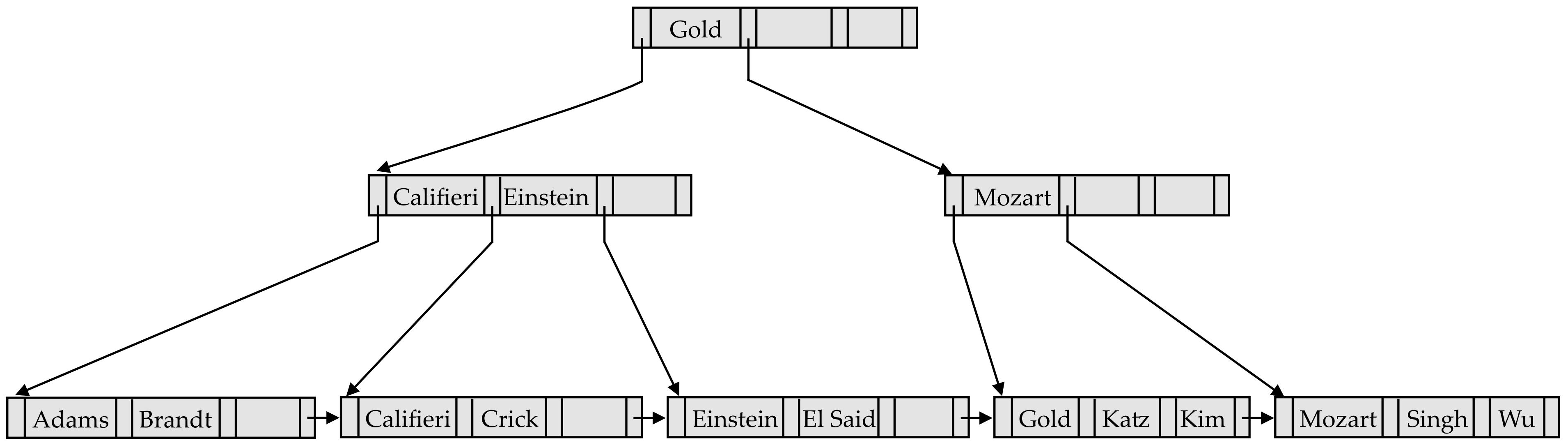
Example: B$^+$-Tree Delete

DELETE FROM instructor where name = ‘Srinivasan’;



After the deletion, only Wu is in the leaf node, and that is too empty since it needs to be at least 50% occupied => must merge with a sibling node or redistribute entries between the nodes

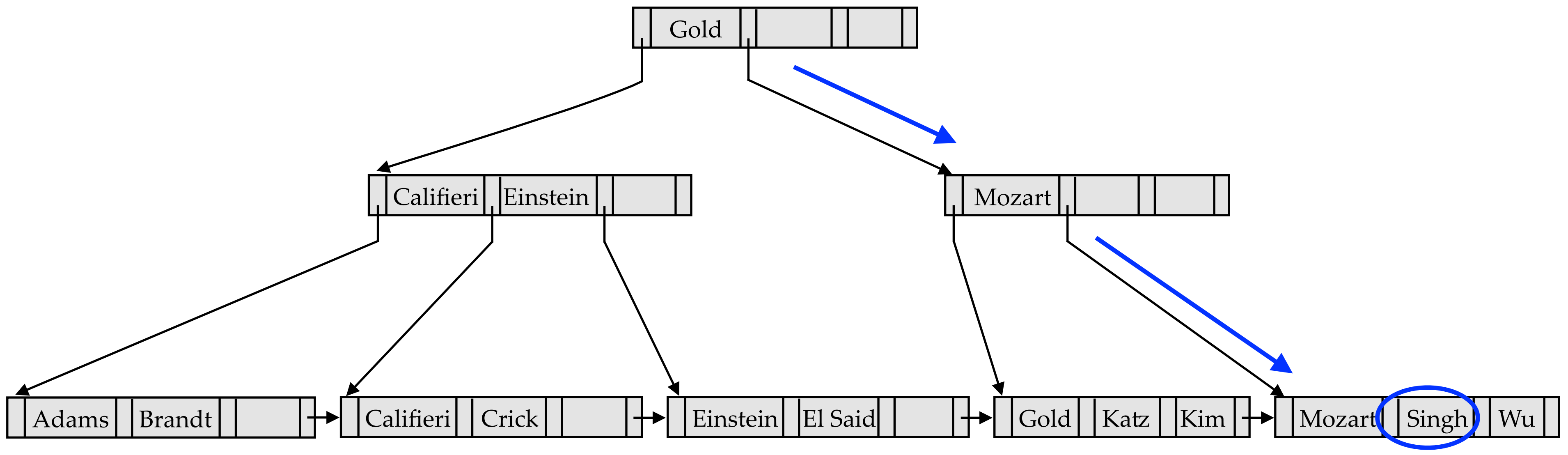
Example: B$^+$-Tree Delete



Merged with the previous sibling and delete from parent, but parent also only has one pointer so we must either merge or redistribute… since merging is not possible, must redistribute

Example: B$^+$-Tree Delete

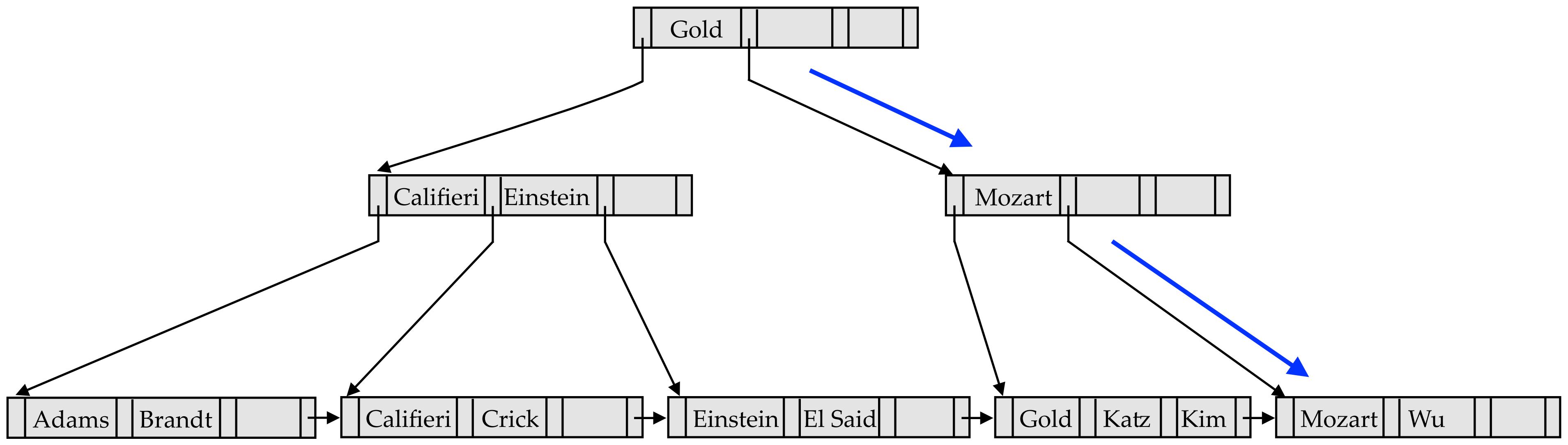
DELETE FROM instructor where name = 'Singh';



Easy case - just delete!

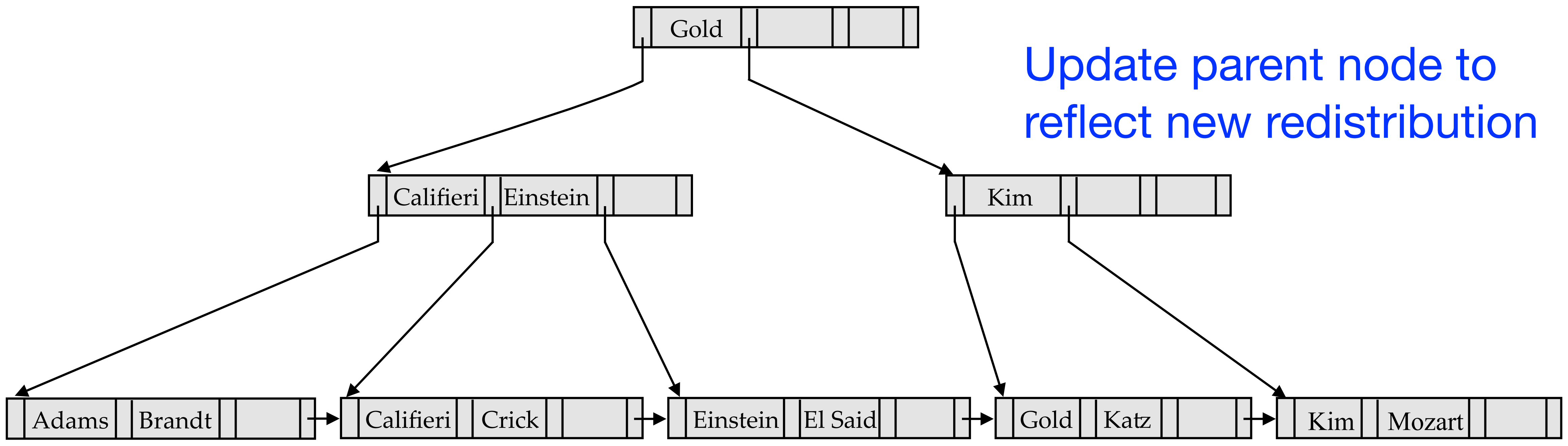
Example: B$^+$-Tree Delete

DELETE FROM instructor where name = ‘Wu’;



Deleting Wu makes the leaf undefiled and not possible to merge with sibling - so redistribute

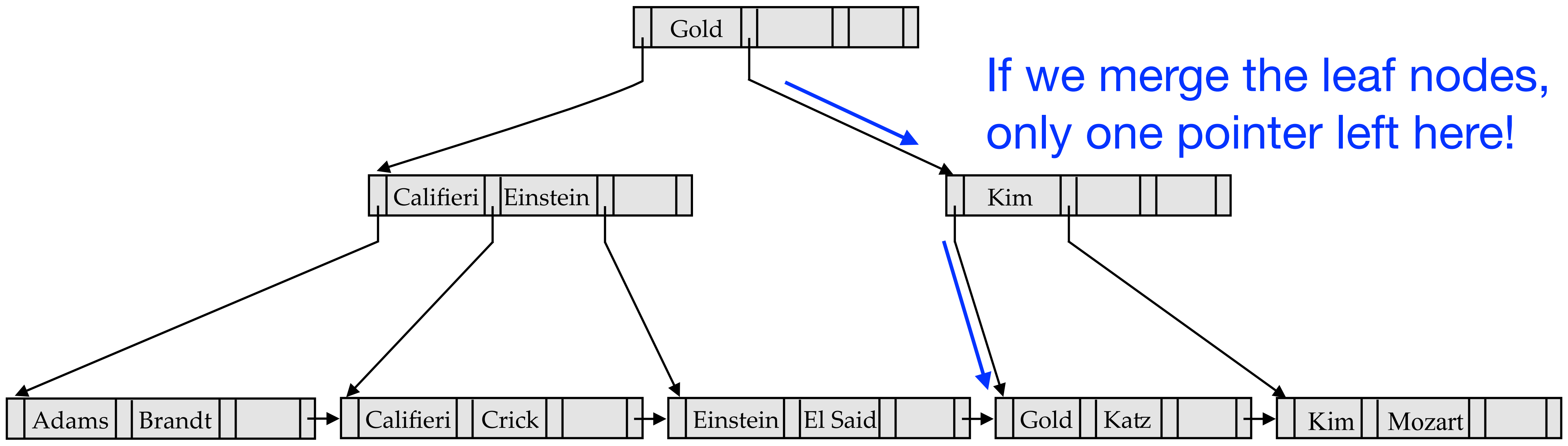
Example: B$^+$-Tree Delete



Update parent node to reflect new redistribution

Example: B$^+$-Tree Delete

DELETE FROM instructor where name = ‘Gold’;

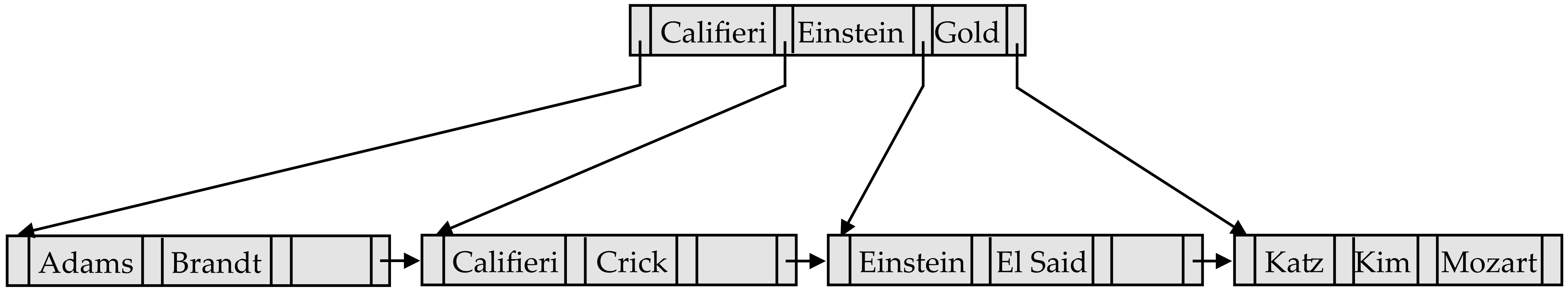


If we merge the leaf nodes, only one pointer left here!

Merge Katz with the sibling on the right

Example: B$^+$-Tree Delete

Delete original root node to avoid condition where root has only one child



Depth of tree has now decreased by 1!

B^+-Tree: Dealing with Duplicates

- Can have many data entries with the same key value (e.g., Year of a movie)
- Solution 1:
  - All entries with a given key value reside on a single page
  - Use overflow pages

B^+-Tree: Dealing with Duplicates

- Solution 2:
  - Allow duplicate key values in data entries
  - Modify search to deal with duplicates

B^+-Tree Performance

- How many I/O's are required for each operation?
  - Worst case cost of insertion / deletion are proportional to the height of the tree (more or less)
  - Height is roughly the $\log_{n/2}$ (number of records)
  - Fanout can be typically large (in the hundreds) - many keys and pointers can fit into one block
- A 4-level tree is enough for "typical" tables

B^+-Tree Index Files

- Alternative to indexed-sequential files
- Cool visualization - https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- (Pro) Automatically reorganizes itself with small, local changes when dealing with insertions/deletions
- (Pro) Reorganization of entire file is not required to maintain performance

B^+-Tree Index Files

- (Con) Extra insertion and deletion overhead
    - However, I/O operations is proportion to height of tree and therefore low
- (Con) Additional space required

Index Structures

- Hash index
  - Good for equality search
  - In expectation: O(1) I/Os and CPU performance for search and insert
- B+ tree index
  - Good for range and equality search
  - $O(\log_F(N))$ I/O cost for search, insert, and delete

SQL Index

- PRIMARY KEY declaration automatically creates a primary index
- UNIQUE key automatically creates a secondary index
- Additional secondary indexes can be created on non-key attributes
  CREATE INDEX <indexName> ON <Relation>(<attr>);
- Example: Company Database
  CREATE INDEX employeeAgeIdx ON Employee(Age);

Multiple-Key Access

- What if we want to access more than one attribute such as a combination of attributes?
- Example:
  SELECT * from EMPLOYEE WHERE dno = 4 AND age = 59;
- Assume that we may have created index on dno and/or age

Multiple-Key Access Strategies

- Dno has index: access tuples with dno=4 using index then do linear search to satisfy age = 59
- Age has index: access tuples with age = 59 using the index and then do linear search to satisfy dno = 4
- Both have indices: get pointer sets and find intersection

What if the number of records that meet each condition is individually large, but the intersection is small? Are the methods efficient?

Composite Search Keys

- Search on a combination of attributes (e.g., age and dno)
- Lexicographic ordering $(a_1, a_2) < (b_1, b_2)$ if either
  - $a_1 < b_1$ or
  - $a_1 = b_1$ and $a_2 < b_2$
- Suppose we have an index on (dno, age)
  - Can efficiently handle queries with dno = 4 and age = 59
  - Can also efficiently handle cases with dno = 4 and age < 59

Beyond B^+-Tree and Hashing

- Tree-based indexes: R-trees and variants, GIST, etc.
- Text indexes: inverted-list index, suffice arrays, etc.
- Other tricks: bitmap index, bit-sliced index, etc.

Choosing Indexes

- What indexes should we create?
  - Which relations should have indexes?
  - What field(s) should be in the search key?
  - Should we build several indexes?
- For each index, what kind should it be?
  - Clustered?
  - Hash/tree?

Choosing Indexes

- Consider best plan using current index and see if a better plan is possible with an additional index — if so, create
    - Must understand how DBMS evaluates queries and creates query evaluation plans
    - Consider tradeoffs: faster queries but slower updates and more storage

Choosing Indexes

- Attributes in WHERE clause are candidates for index keys
  - Exact match condition suggests hash index
  - Indexes also speed up joins
  - Range query suggests tree index
- Multi-attribute search keys should be considered when a WHERE clause contains several conditions
  - Order of attributes is important for range queries

Index Demo on IMDB

Tuning Indexes

- Initial choice of indexes may have to be revised
  - Certain queries may take too long to run without an index
  - Certain indexes may not get used at all
  - Certain indexes undergo too much updating because the attribute undergoes frequent changes

Indexing: Recap

- Index Overview
- Hash index
- B+-Tree
- Composite search keys
- Choosing indexes