BACS3183
Advanced Database Management

Chapter 7
Physical Database Design

Learning Outcomes

At the end of this class, you should be able to

- **Explain the different techniques for placing file records on disk.**
- **Give examples of the application of primary/clustering and secondary/clustering indexes.**
- **Distinguish between a non-dense index and a dense index.**
- **Use dynamic multilevel indexes using B^+-trees.**
- **Explain the theory of hashing techniques.**
- **Use hashing to facilitate dynamic file expansion.**

Introduction

- The database is stored as a collection of *files*.
- Main issues addressed generally in physical design
  - » Storage Media
  - » File organization - Many alternatives exist, each appropriate in some situation.
  - » Indexes
- Changing (deleting, inserting, updating) the database requires
  - » Maintaining the file structures
  - » Updating the indexes

Introduction

- **File organization**
  - Heap
  - Sequential
  - Hash
- **Indexing structures**
  - Hash Index
  - B^+-Tree Index
  - Bitmap index
  - Join index

1. Storage and file structures

- **The database is stored as a collection of *files*. Each file is a sequence of *records.***
- **Generally, a separate file is used to store records of a relation.**
- **Records are either fixed size or variable size**

Example of fixed size records

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000

Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - One or more fields have variable length
  - One or more fields are repeating
  - One or more fields are optional
  - File contains records of different types

File Organization

- Method of arranging a file of records on external storage.
  - Heap (unordered) files: Records are placed on disk in no particular order. New records are inserted at the end of the file.
  - Sequential (ordered) files: Records are ordered by the value of a specified field.
  - Hash files: Records are placed on disk according to a hash function.

A-217	Brighton	750
A-101	Downtown	500
A-110	Downtown	600
A-215	Mianus	700
A-102	Perryridge	400
A-201	Perryridge	900
A-218	Perryridge	700
A-222	Redwood	700
A-305	Round Hill	350

Block 1

Block 2

Block 3

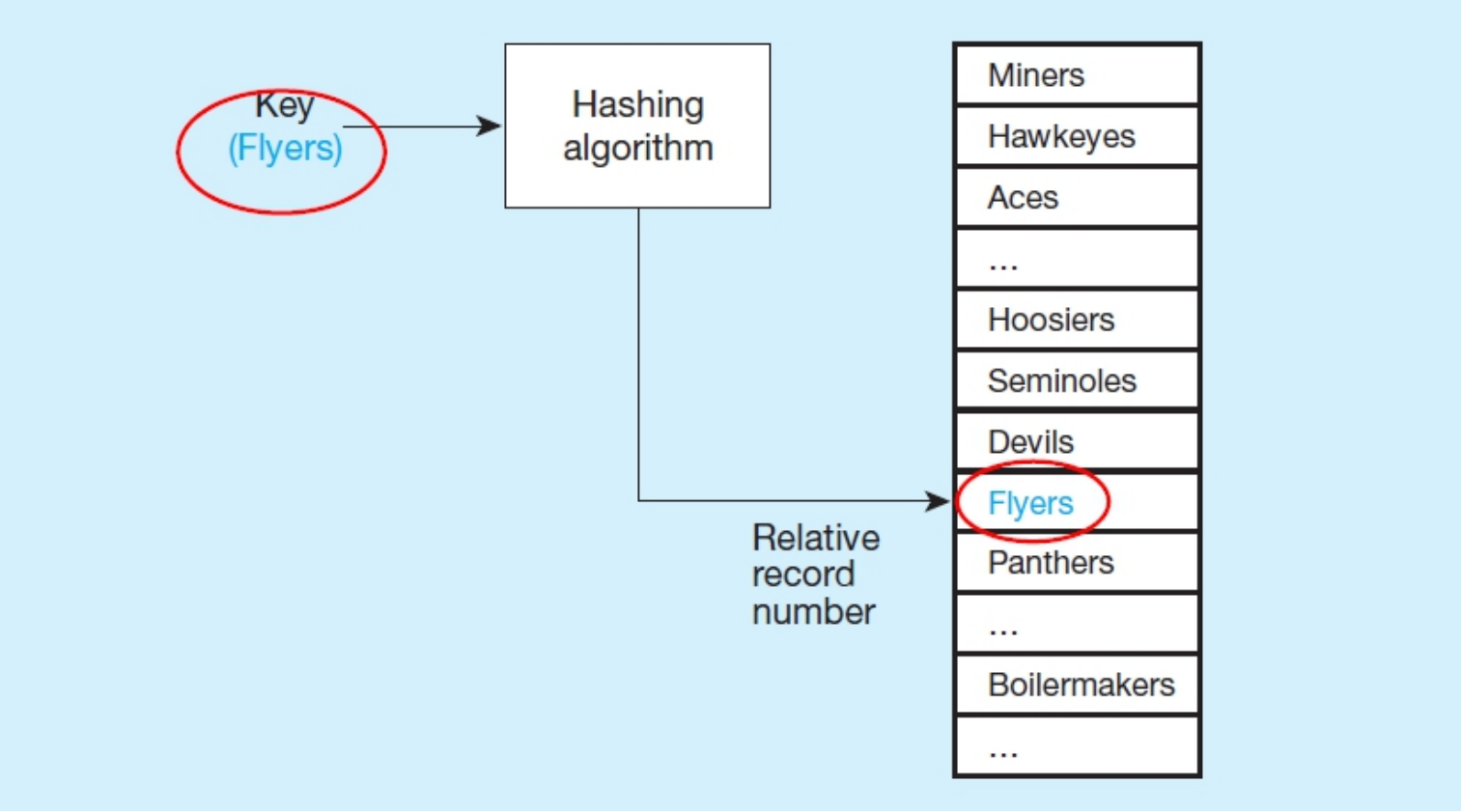
heap file

Sequential File

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

32222	Verdi	Music	48000

Hashed File


Key
(Flyers)
Hashing
algorithm
Relative
record
number
Miners
Hawkeyes
Aces
...
Hoosiers
Seminoles
Devils
Flyers
Panthers
...
Boilermakers
...

Example of Hashing

Assume a table with 8 slots:

Hash key = key % table size

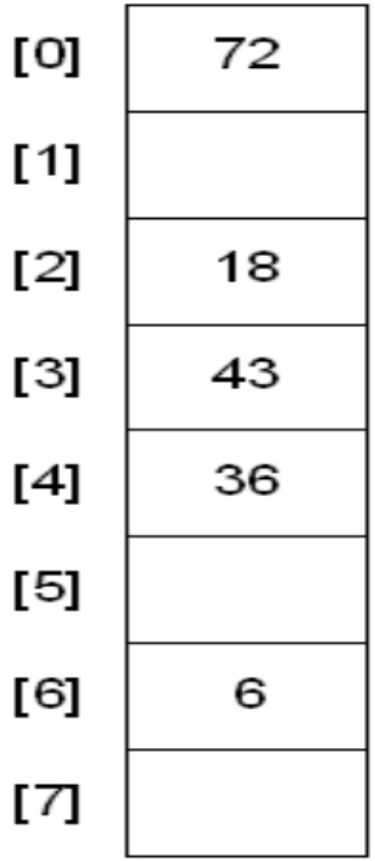
4 = 36 % 8

2 = 18 % 8

0 = 72 % 8

3 = 43 % 8

6 = 6 % 8



2. Indexes

- *B^+-tree indexes*
  - Most common type of index used in databases.
- *Hash indexes*
  - Good for simple and fast lookup operations.
- *Bitmap indexes*
  - Used in data warehouse applications
- *Join indexes*
  - Common for data warehouse applications

Index Evaluation Metrics

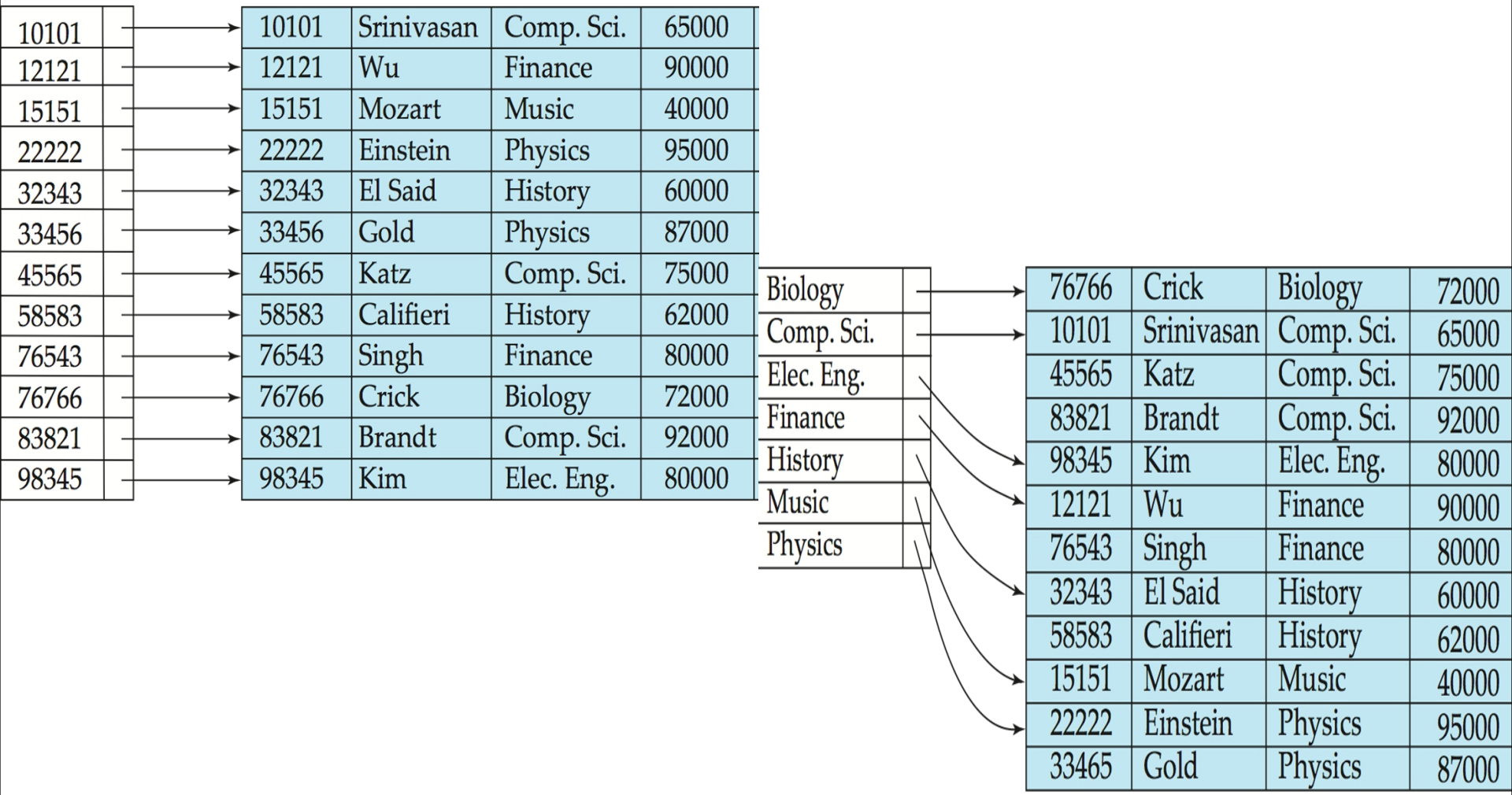
- **Access time** for:
  - **Equality searches** – records with a specified value in an attribute
  - **Range searches** – records with an attribute value falling within a specified range.
- **Insertion time –** Time to find correct place to insert the new data item + update the index structure
- **Deletion time -** Time to find the item to be deleted + update the index structure
- **Space overhead –** Space to store the index structure

2.1 Classification of Indexes

- **Clustering vs non-clustering**
  - Clustering index: ordering of index matches ordering of values of search key attribute in file
- **Dense vs sparse index**
  - Dense index: every value of the search key attribute found in the table also appears in the index

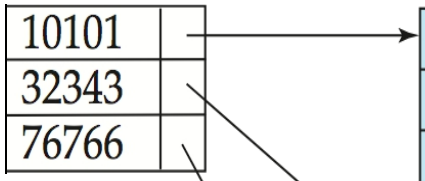
Dense Index

- Dense index: an index record appears for **every** search-key value in the file



Sparse Index

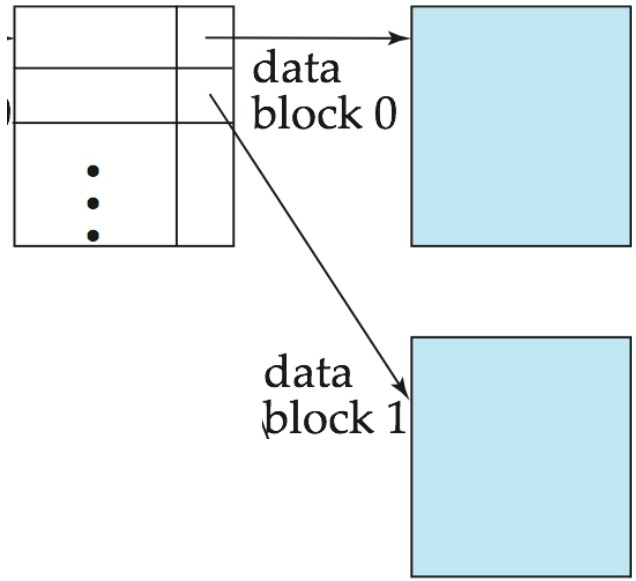
- **Sparse Index**: contains index records for only some search-key values.
- To locate a record with search-key value *K*, we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points



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

Sparse Index

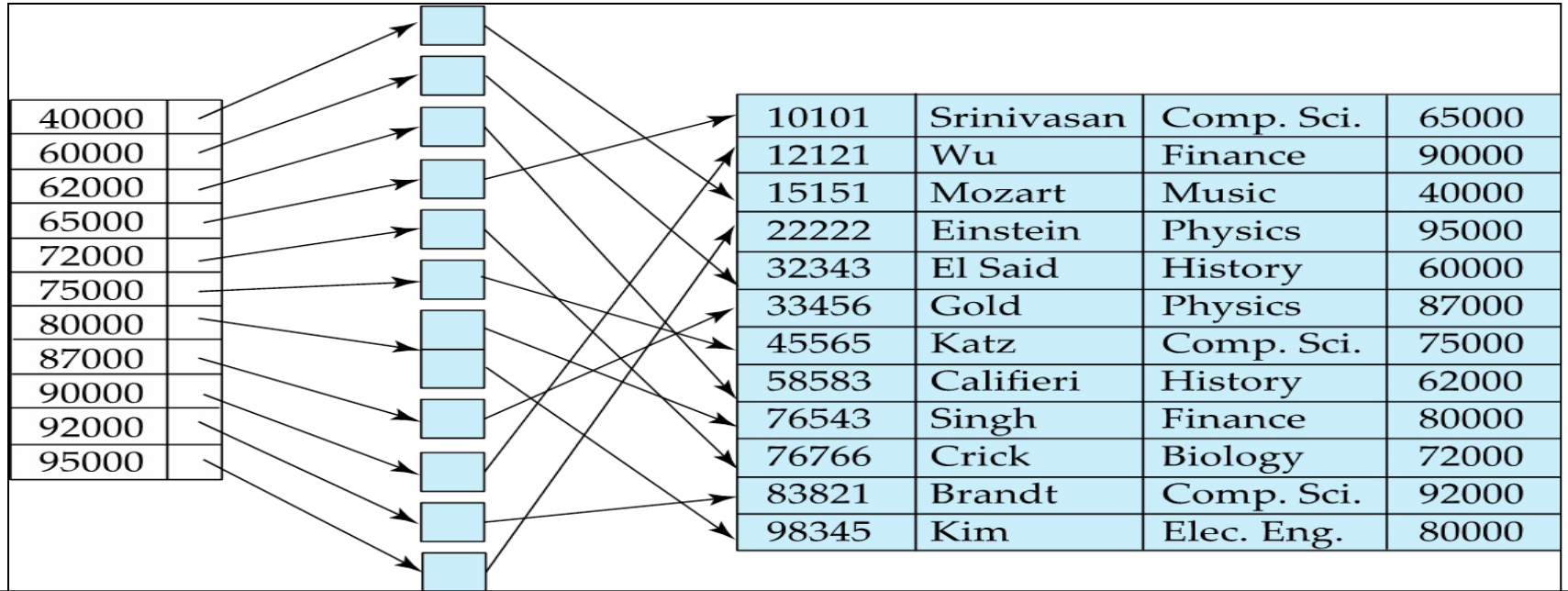
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions. With dense index, when a file is modified, every index on the file must be updated. Updating indices imposes overhead on database modification.
  - Generally slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Non-Clustering Index

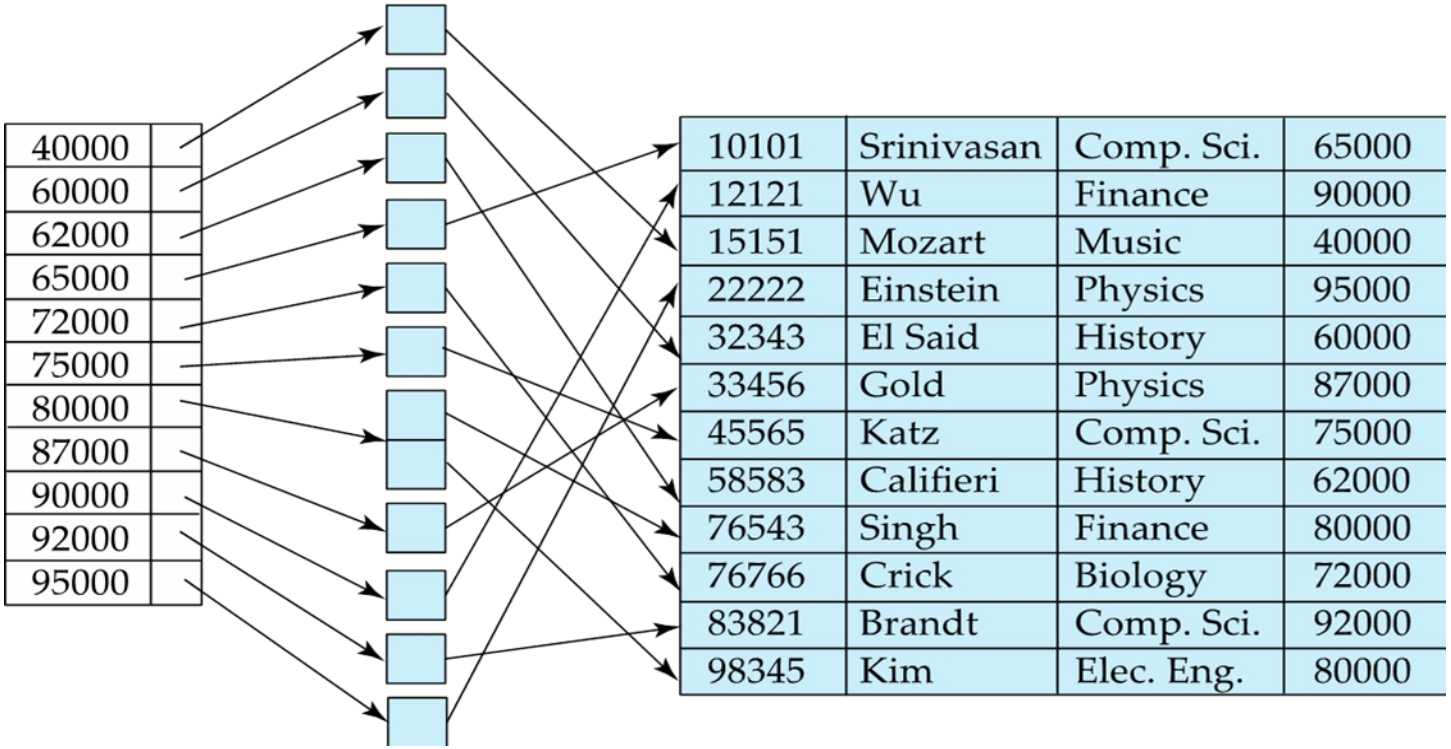
Clustering index: ordering of index matches ordering of values of search key attribute in file

- Frequently, one wants to find all the records whose values in a certain field and the file is not ordered on the field.
  - Eg 1: In the *instructor* relation stored sequentially by ID, find all instructors in a particular department
  - Eg 2: find all instructors with a specified salary or with salary in a specified range of values



Non-clustering Indices

- A non-clustering index must have an index record for every search-key value. The index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

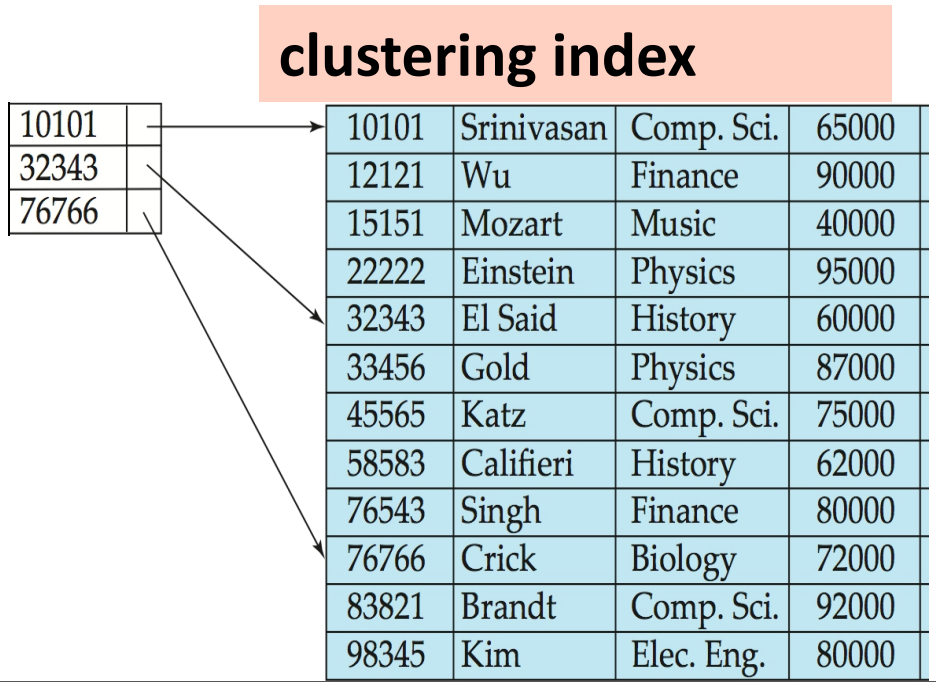


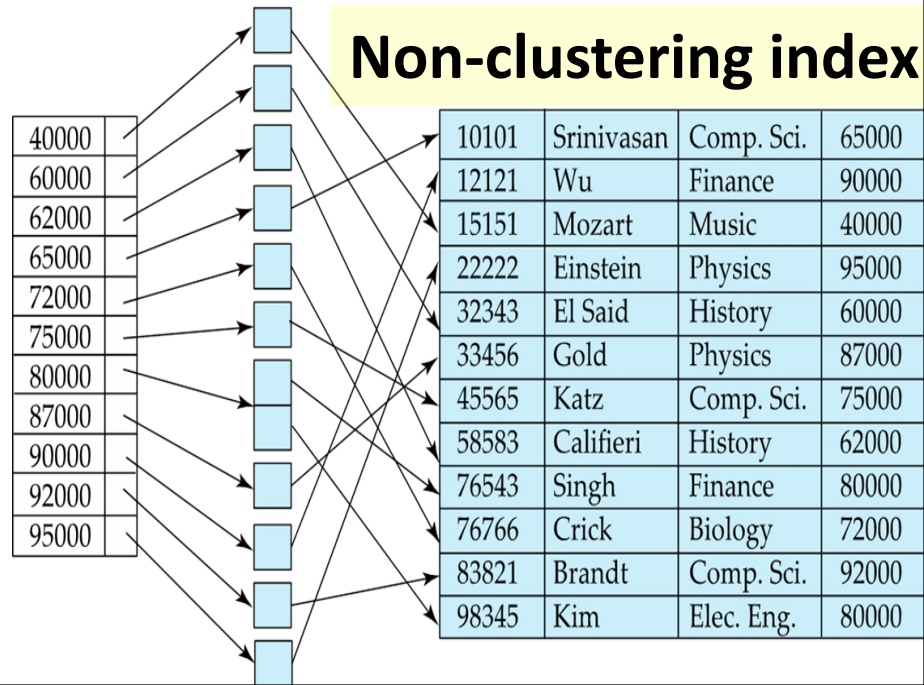
Bucket structure is used if search key is not a primary key, and file is not sorted in search key order.

Clustering and Non-clustering

Clustering index: ordering of index matches ordering of values of search key attribute in file

- **Clustering indices can be sparse - since it is always possible to find records with intermediate search-key values**
- **Non-clustering indices have to be dense.**

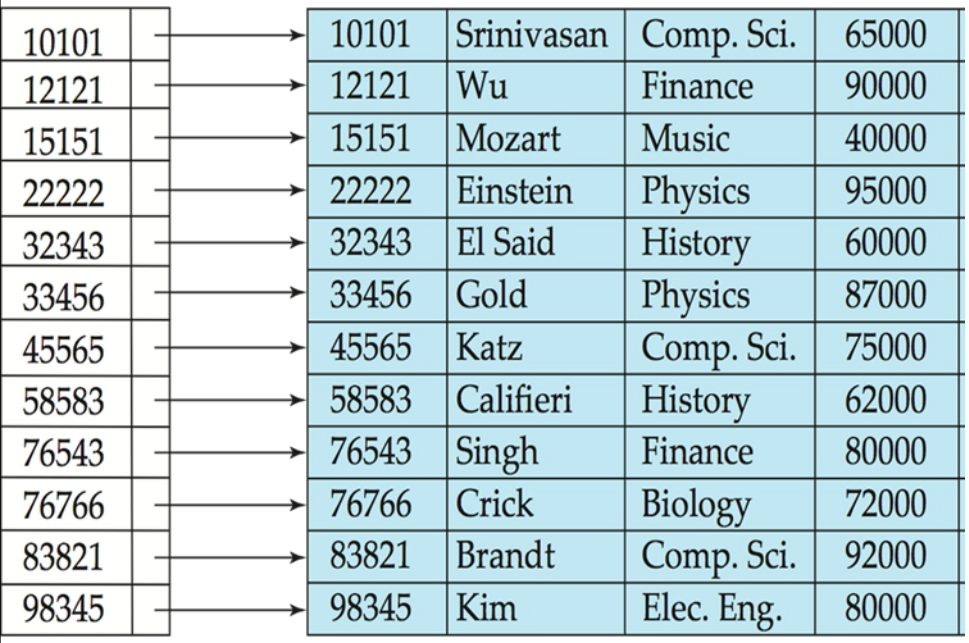




Clustering and Non-clustering

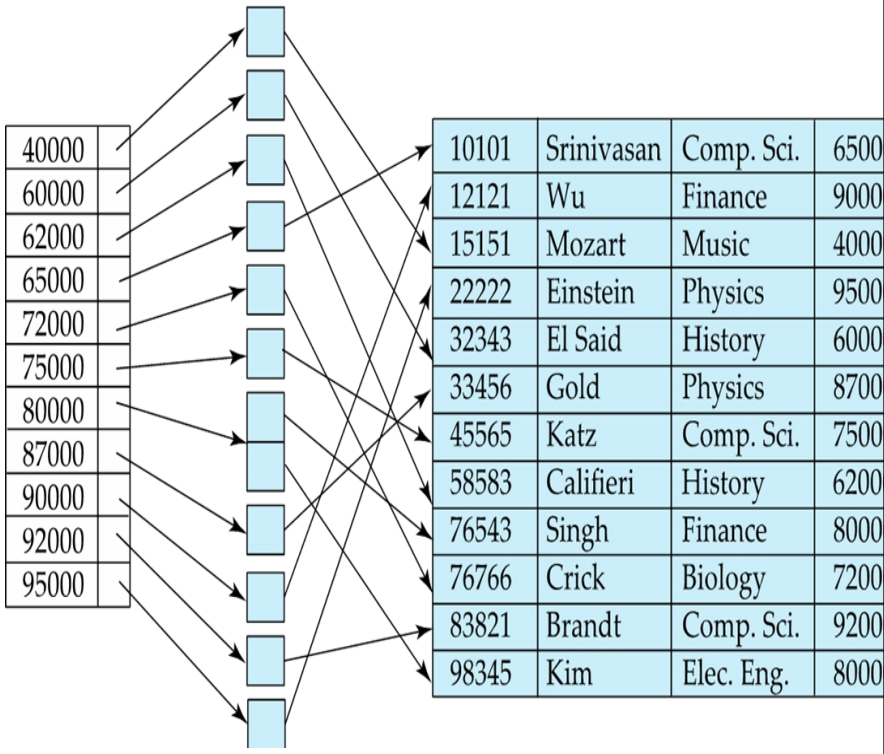
- **Sequential scan using clustering index is efficient, but a sequential scan using a non-clustering index is expensive – each record access may fetch a new block from disk.**

clustering index



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

Non-clustering index



40000	10101	Srinivasan	Comp. Sci.	6500
60000	12121	Wu	Finance	9000
62000	15151	Mozart	Music	4000
65000	22222	Einstein	Physics	9500
72000	32343	El Said	History	6000
75000	33456	Gold	Physics	8700
80000	45565	Katz	Comp. Sci.	7500
87000	58583	Califieri	History	6200
90000	76543	Singh	Finance	8000
92000	76766	Crick	Biology	7200
95000	83821	Brandt	Comp. Sci.	9200
	98345	Kim	Elec. Eng.	8000

Multilevel Index

Imagine this:
A file 100,000 records, 10 records in one block
Sparse index => 1 index record per block, how many records will the index have?

Index records are smaller than data records. If 100 index records fit into 1 block, how many blocks will be needed to store the index?

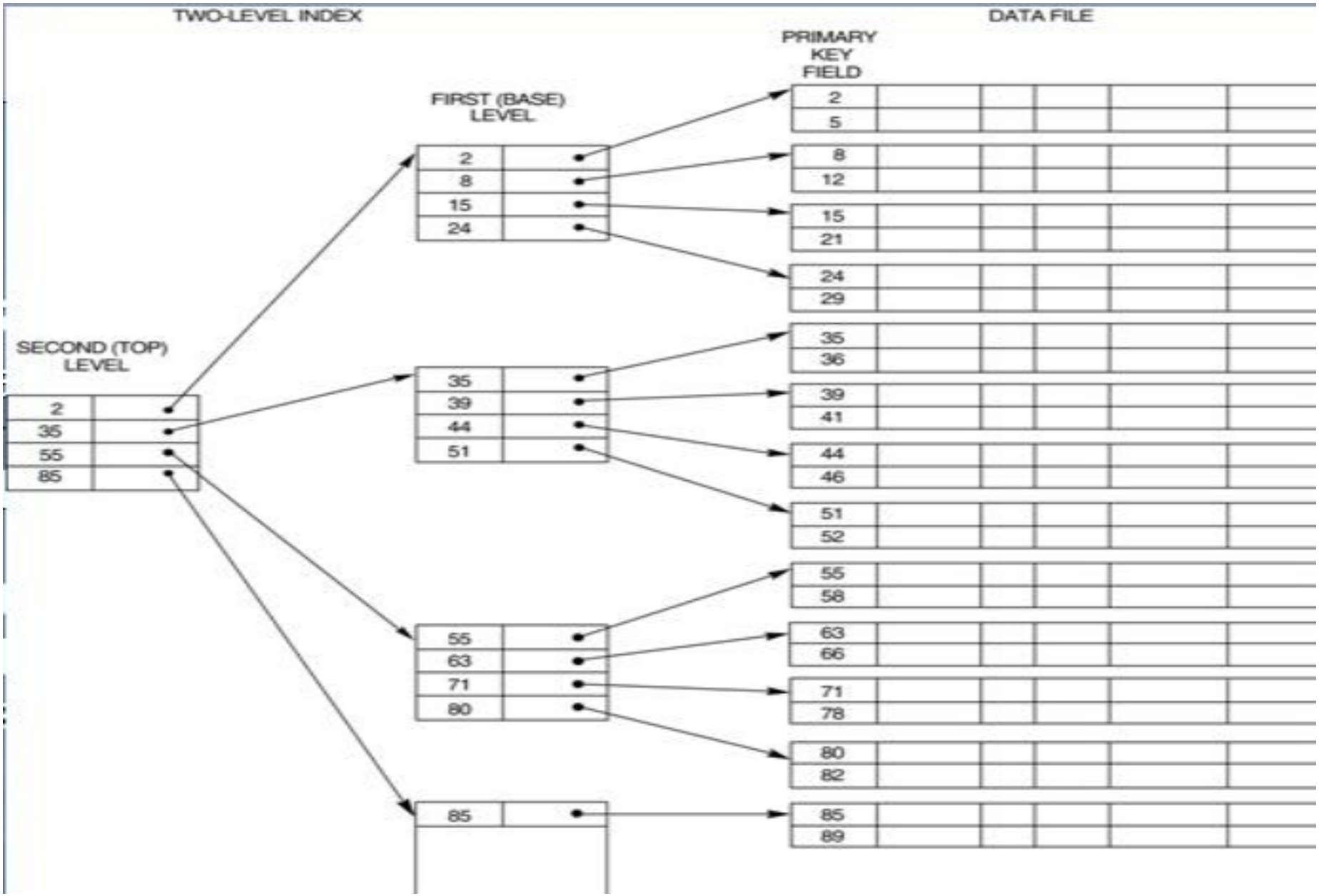
100000/10 blockks to store the data records.

There will be 100000/10 = 10000 index records.
Index will require 10000/100 = 100 blocks

Multilevel Index

- If an index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, construct
  - outer index – a sparse index on main index
  - inner index – the main 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.

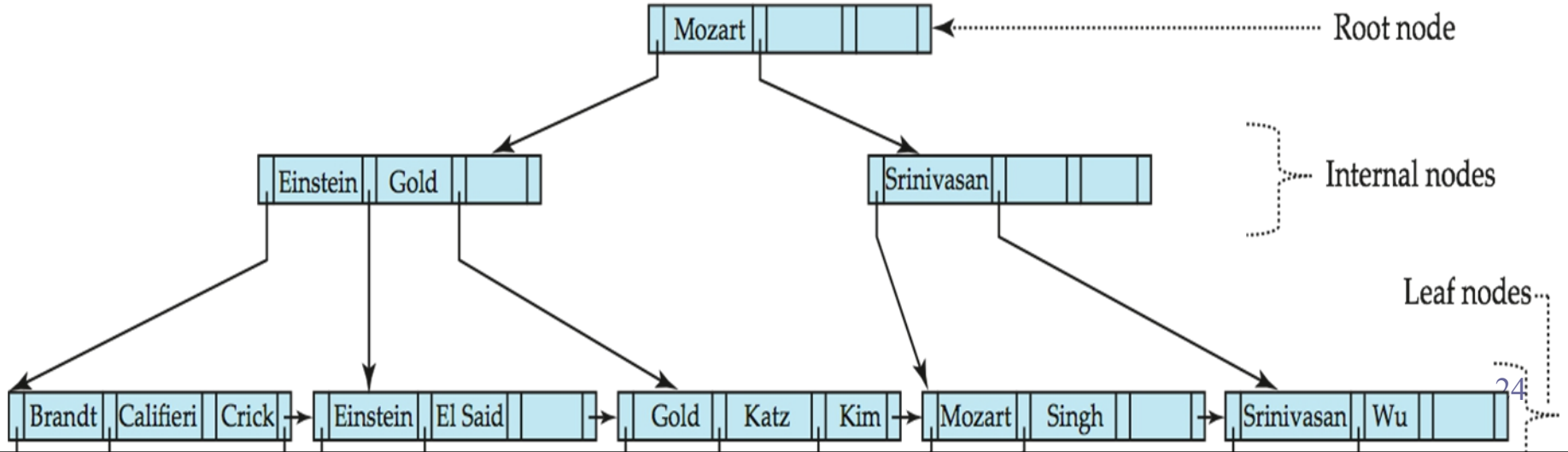
Example of Multilevel Index



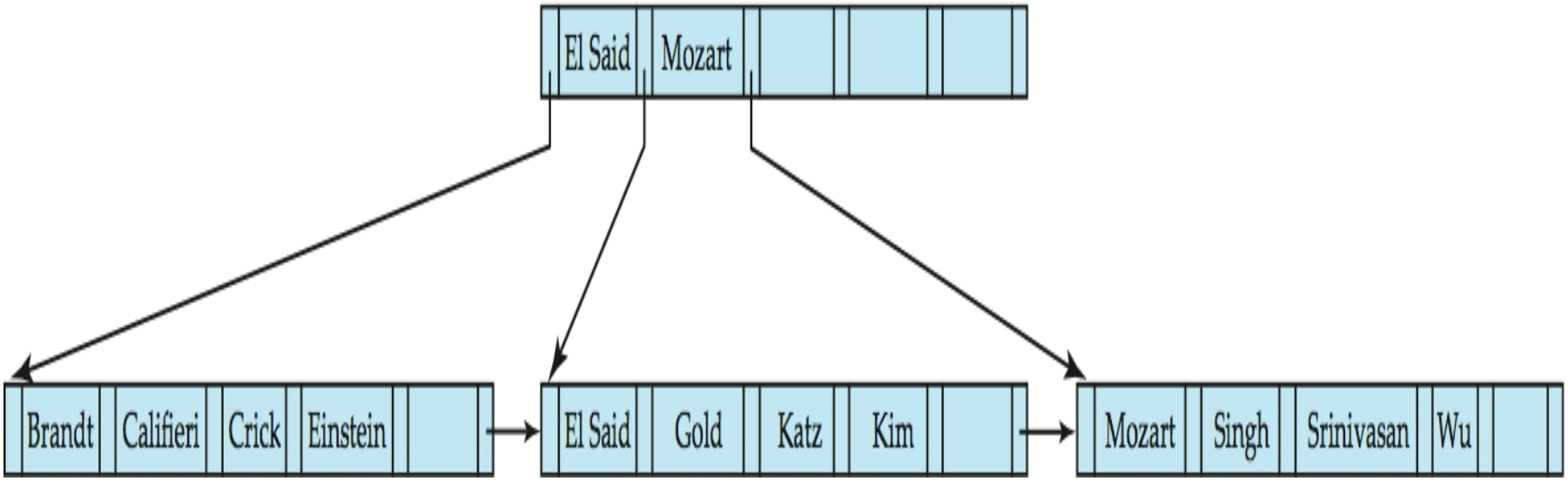
3. B+-Tree Index Files

A B+-tree must satisfy the following properties:

- **All paths from root to leaf are of the same length**
- **Each node that is not a root or a leaf (nonleaf node)has between $\lceil n/2 \rceil$ and n children where n is the maximum number of pointers per node. n is fixed for a particular tree**
- **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.**



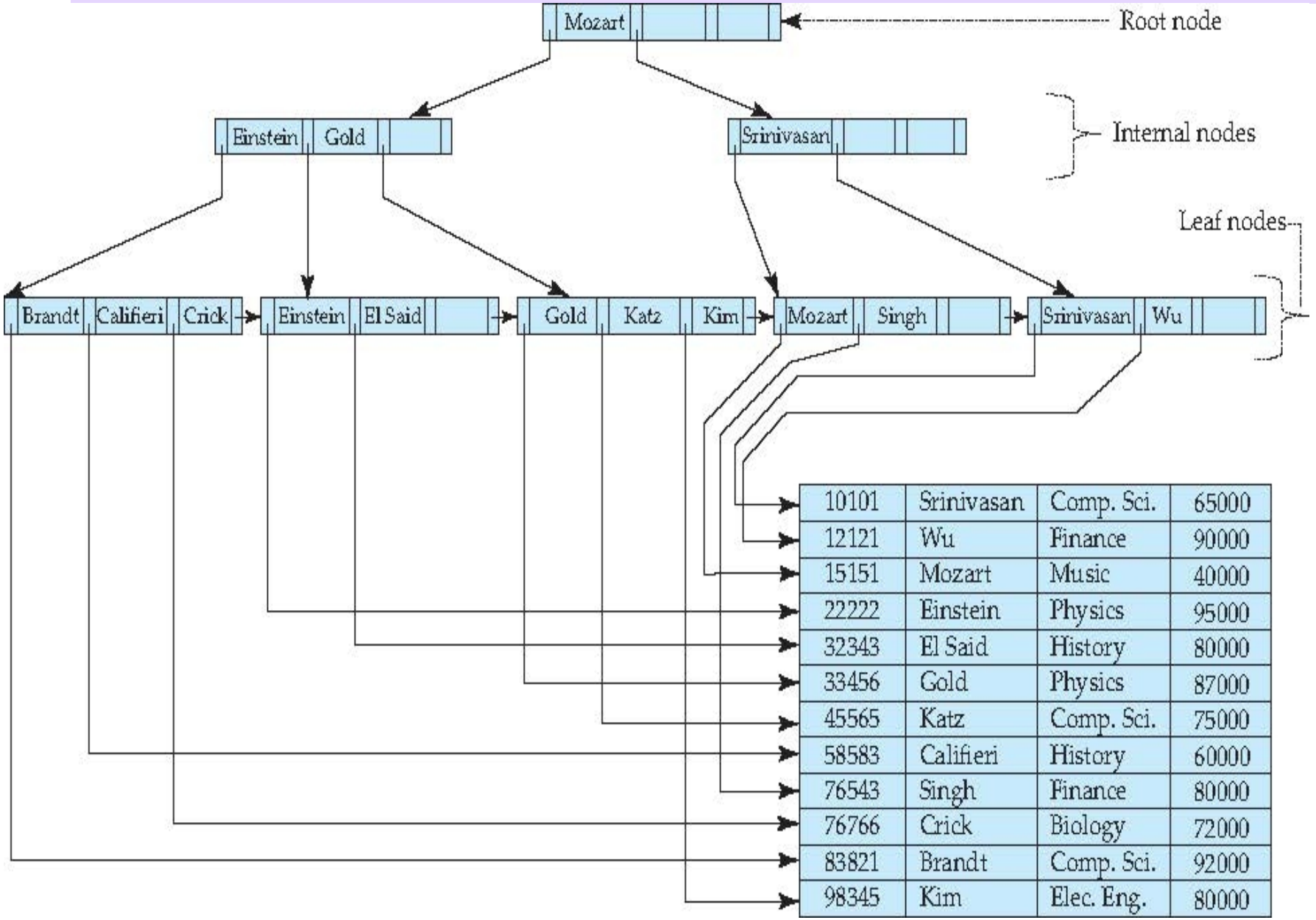
Example of B+-tree of order 6 (n=6)



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.**

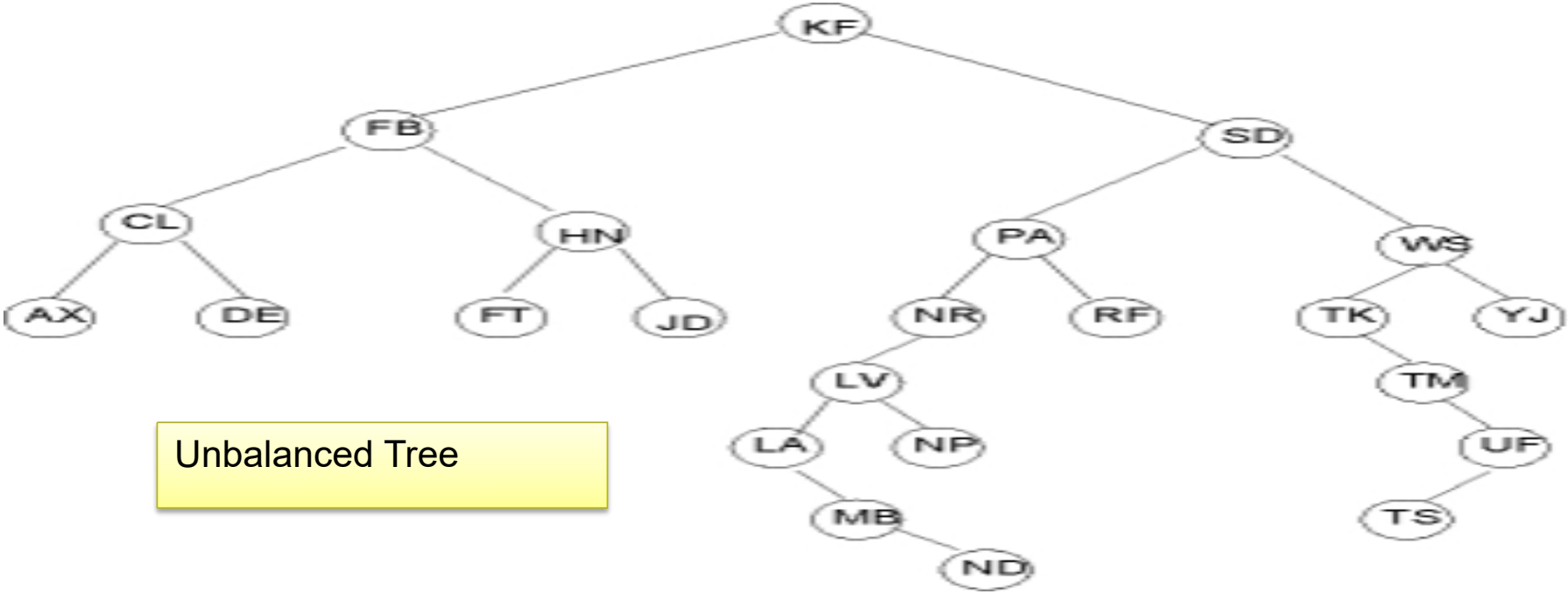
Example of B+-Tree



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

Balanced Tree

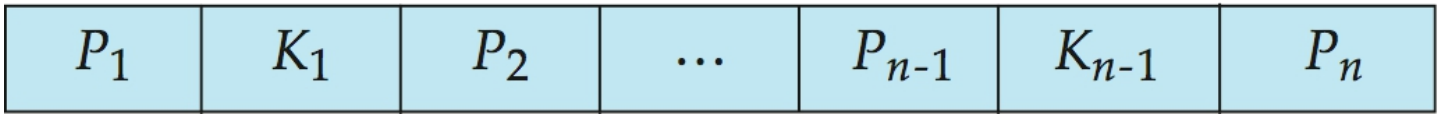
- A B^+-tree needs to be BALANCED, meaning that all of its leaf nodes are at the same level.



Unbalanced Tree

Keeping a tree balanced is important to guarantee that no node will be very high levels and hence require many block accesses during a tree search

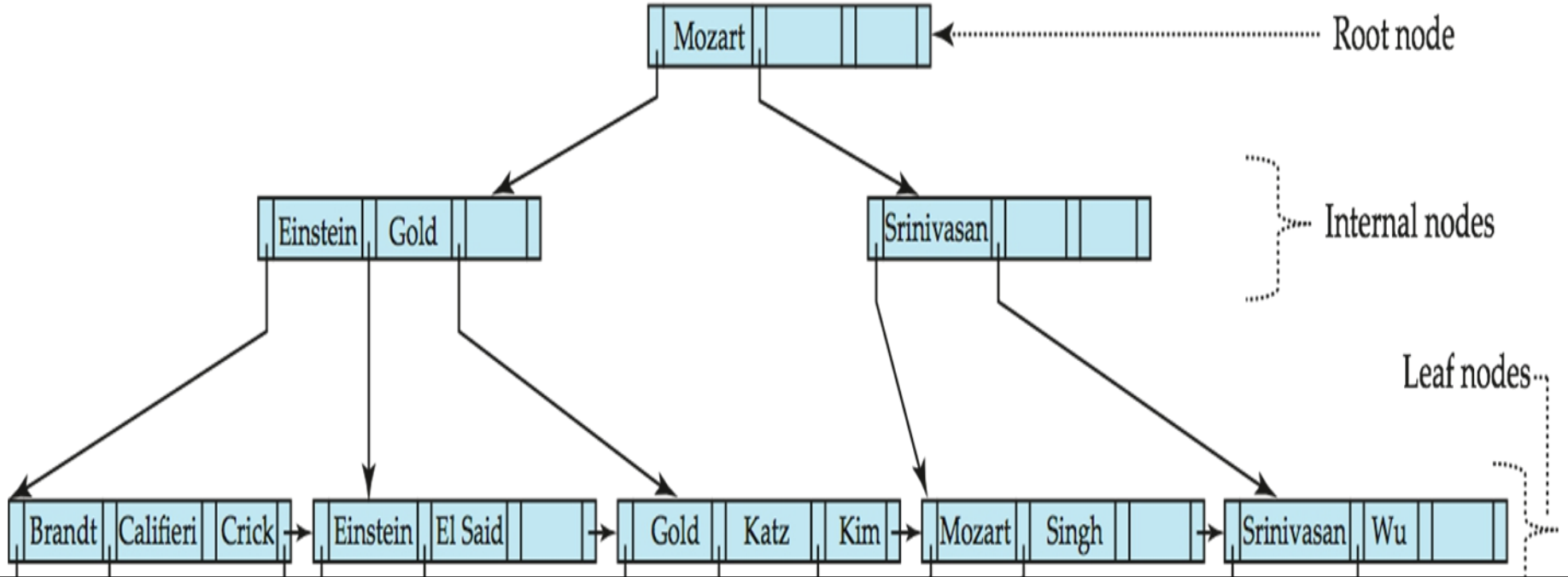
B+-Tree Node Structure



- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

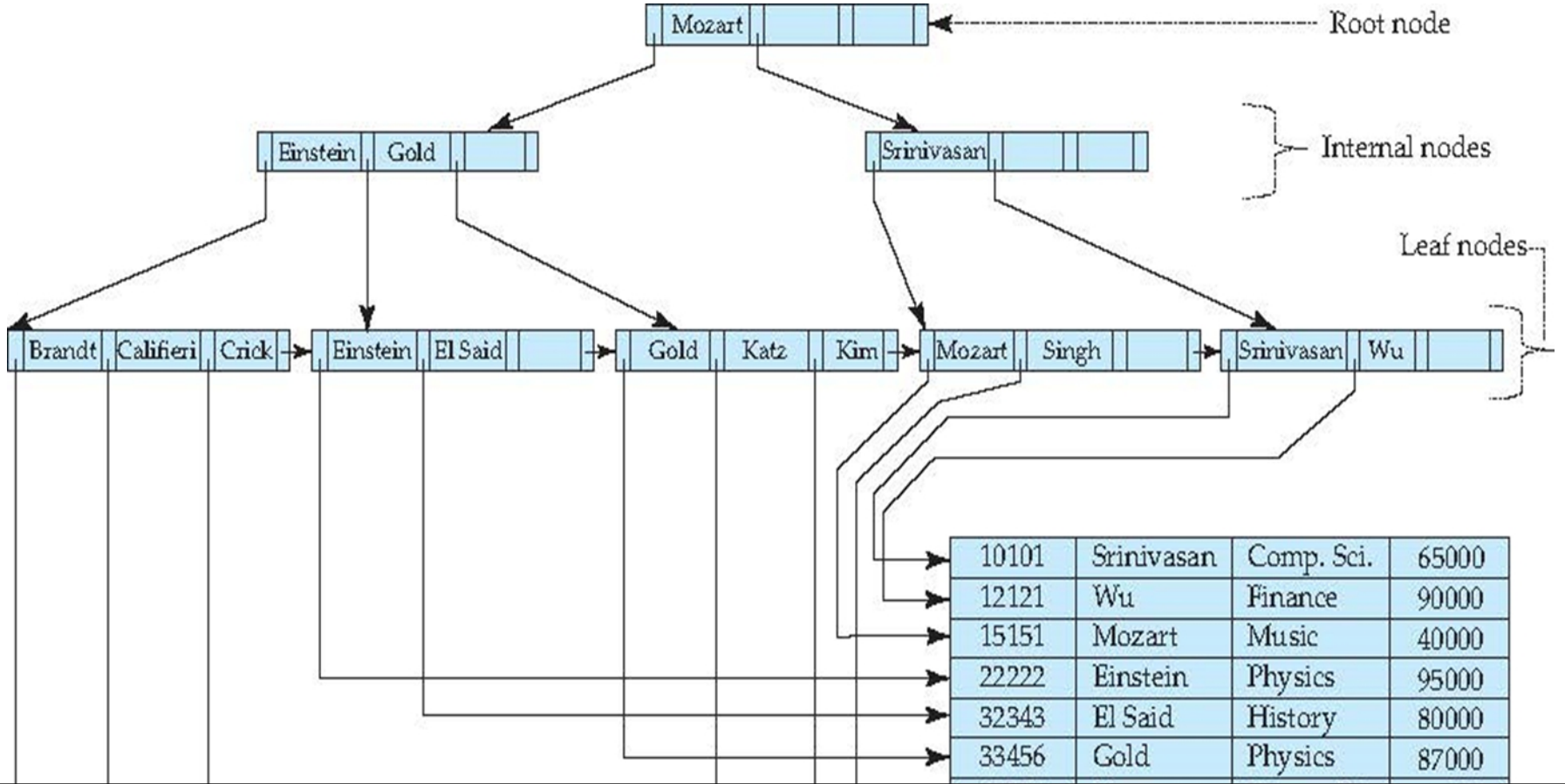
- The search-keys in a node are ordered

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



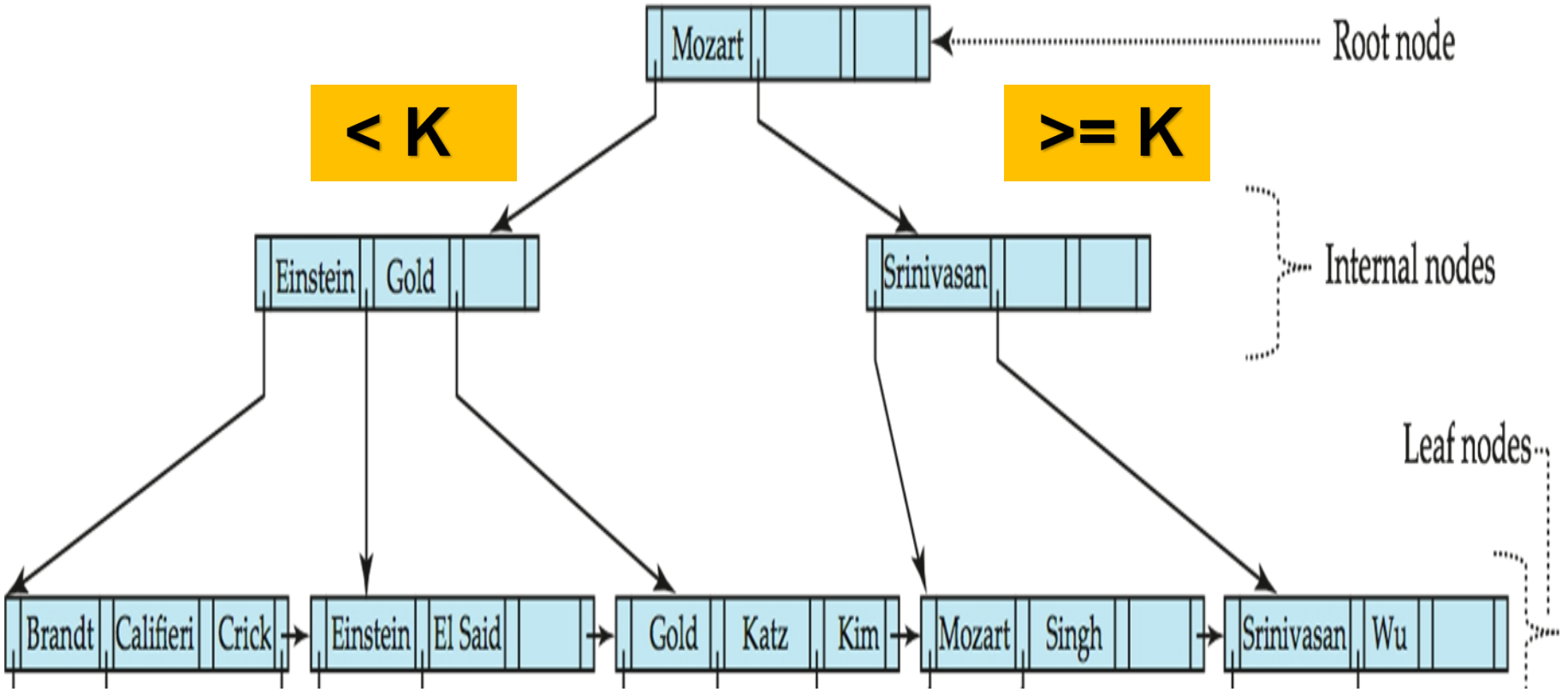
Leaf Nodes in B+-Trees

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



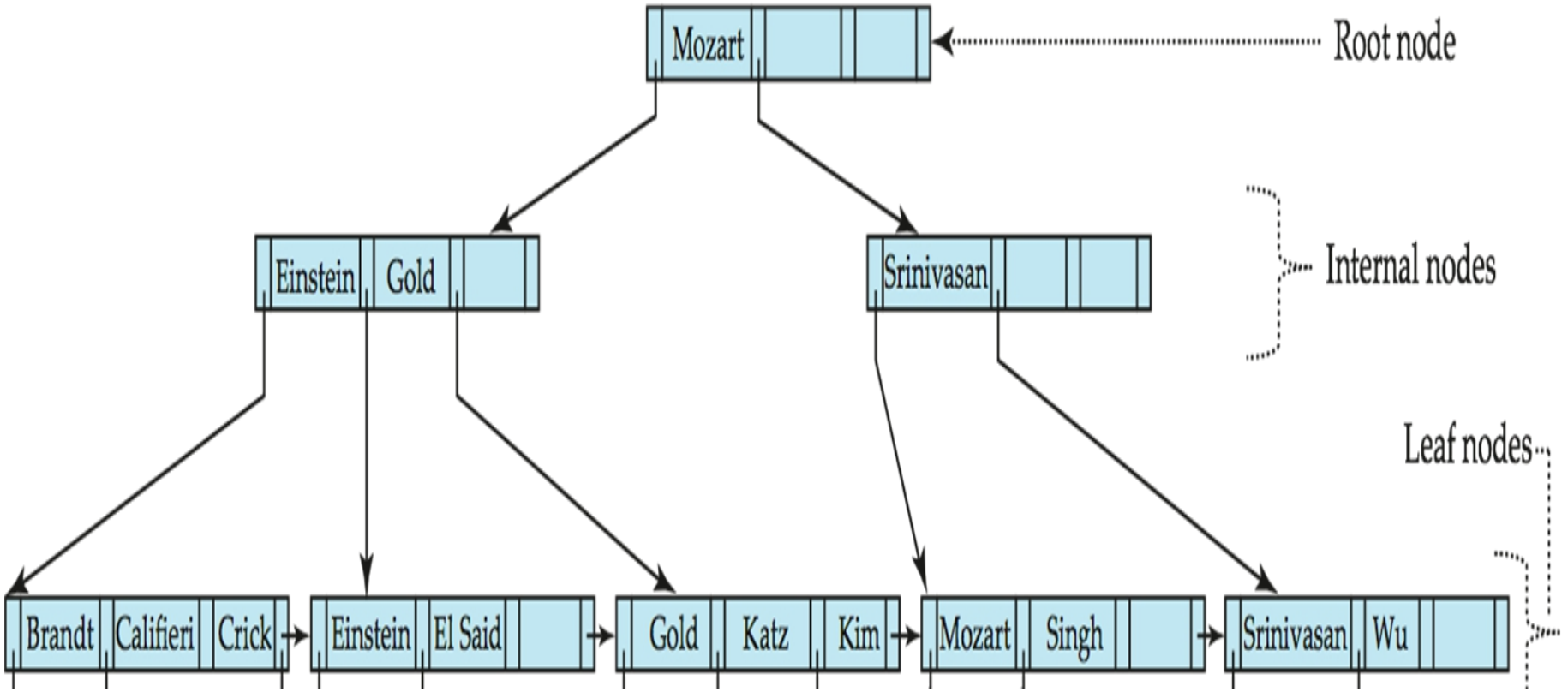
Non-Leaf Nodes in B+-Trees

- For a non-leaf node with n pointers:
  - All the search-keys in the subtree to which P_i points are less than K_i (ie on the left subtree)
  - All the search-keys in the subtree to which P_i points are greater than or equal to K_{i-1} (ie on the right subtree)



Queries on B^+-Trees

- Find all records with a search-key value of k.
  - Start with the root node
  - Examine the node for the smallest search-key value > k.
  - If such a value exists, assume it is K_i. Then follow P_i to the child node
  - Otherwise $k \geq K_{m-1}$, where there are m pointers in the node, Then follow P_m to the child node.
  - If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.
  - Eventually reach a leaf node. If key $K_i = k$, follow pointer P_i to the desired record or bucket. Else no record with search-key value k exists.



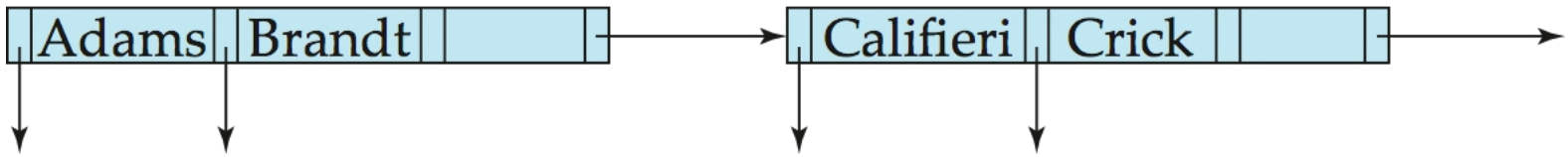
Select * from instructors where instructorLName = 'Kim';

Updates on B+-Trees: Insertion

- Find the leaf node in which the search-key value would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary pointer is inserted into bucket.
- If the search-key value is not there, then add the record to the main file and create bucket if necessary. Then:
  - if there is room in the leaf node, insert (search-key value, record/bucket pointer) pair into leaf node at appropriate position.
  - if there is no room in the leaf node, split it and insert (search-key value, record/bucket pointer) pair as discussed in the next slide.

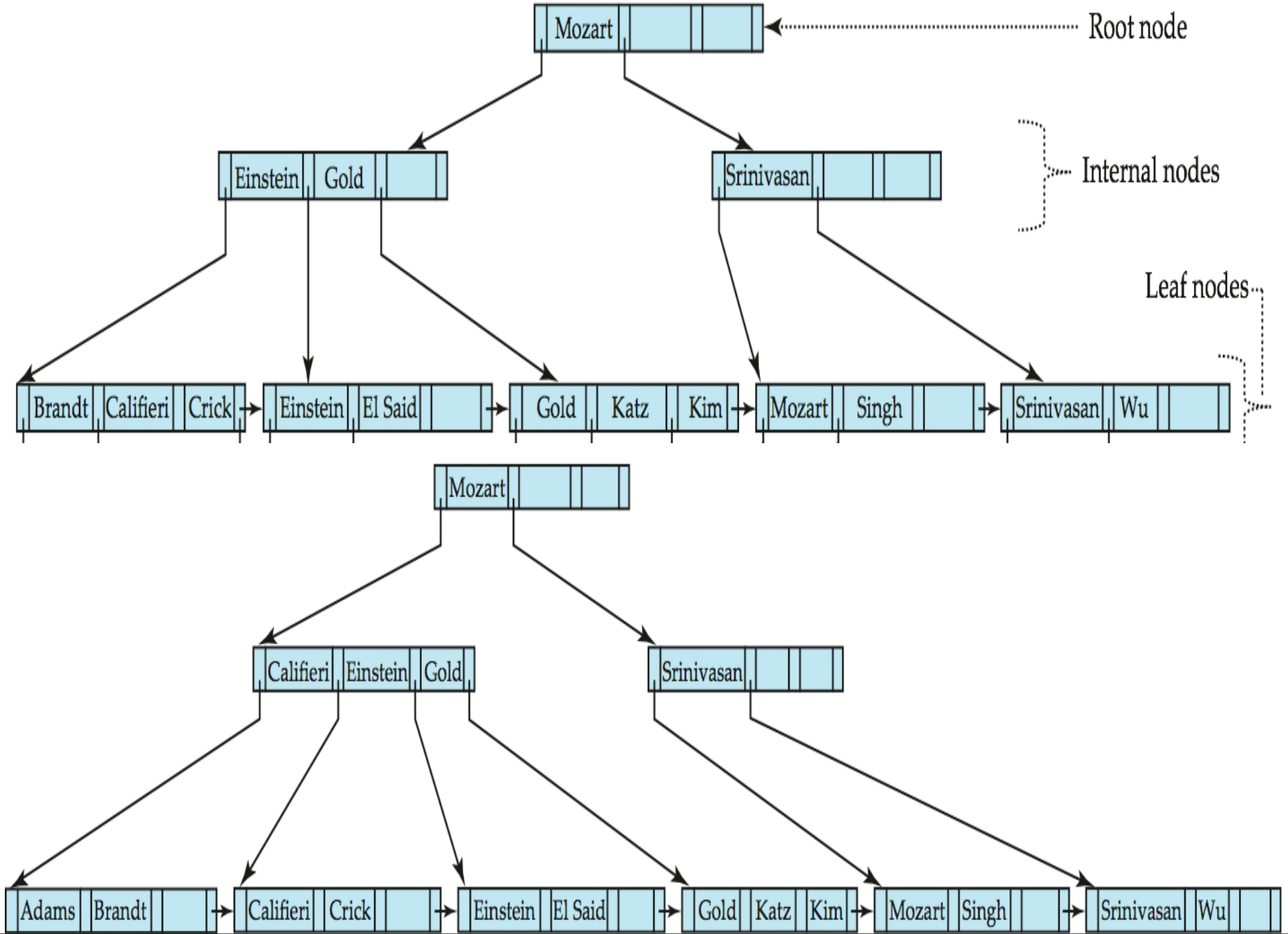
Updates on B+-Trees : Insertion

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

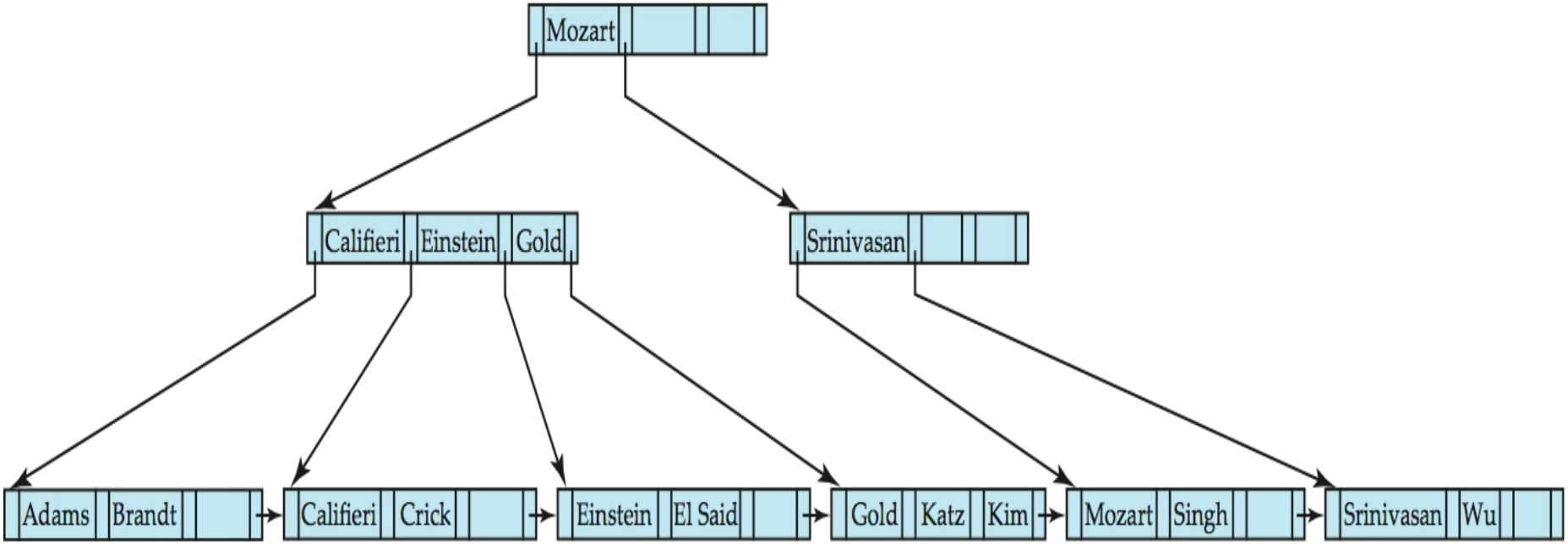


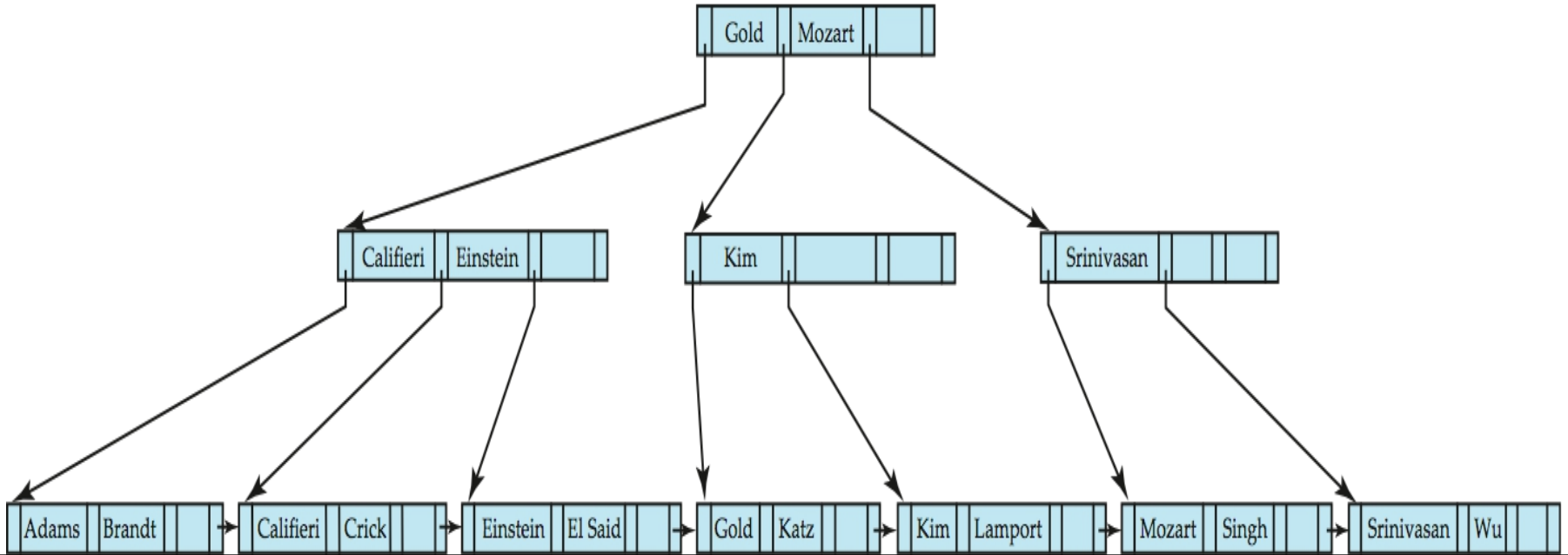
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri,pointer-to-new-node) into parent

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



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





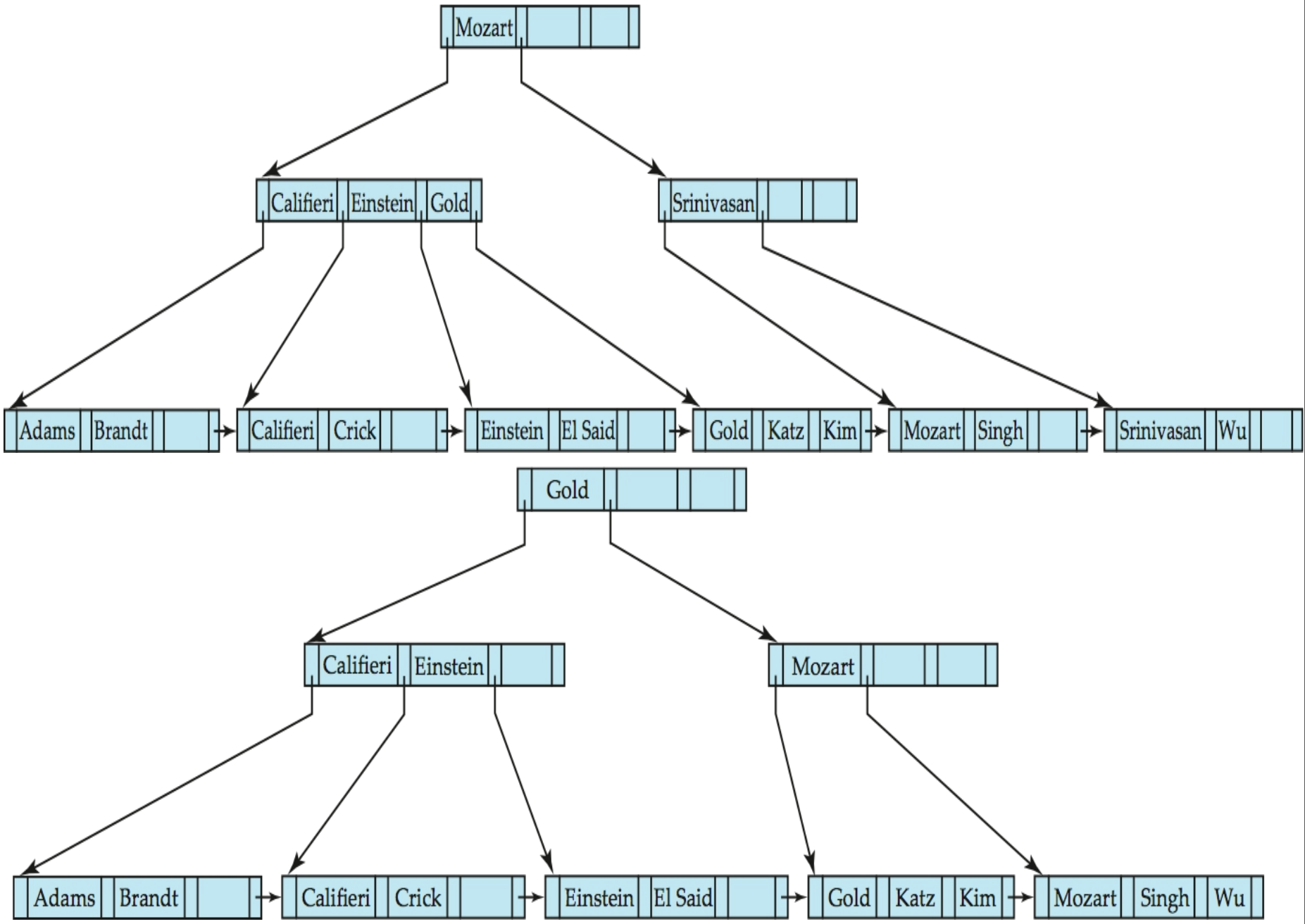
Updates on B+-Trees : Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

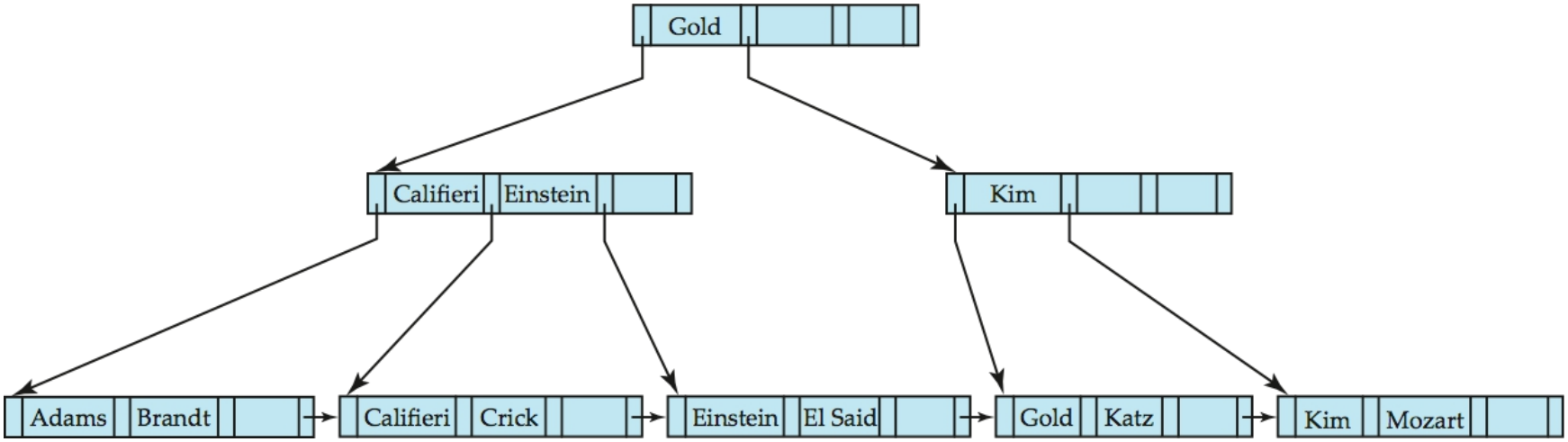
Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, and the entries in the node and a sibling don't fit into a single node, then
  - Redistribute the pointers between the node and a sibling such that both have at least 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.

B+-Tree before and after deletion of "Srinivasan"



Examples of B+-Tree Deletion

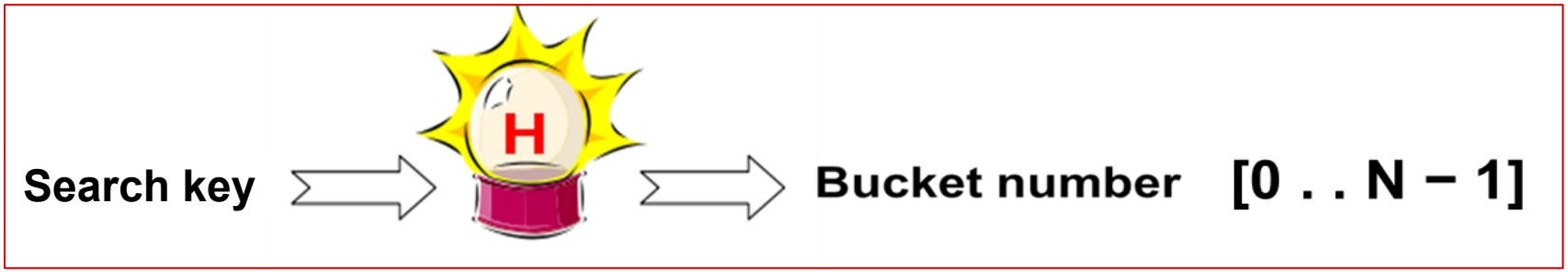


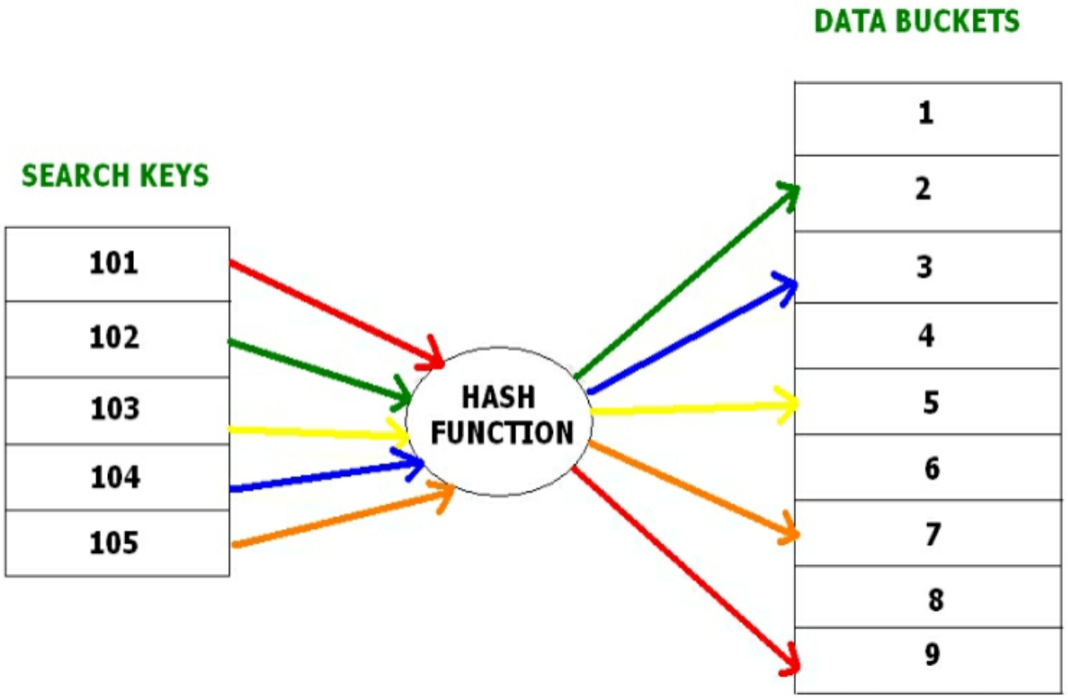
Deletion of “Singh” and “Wu” from result of previous example

- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result

4. Hash Index

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

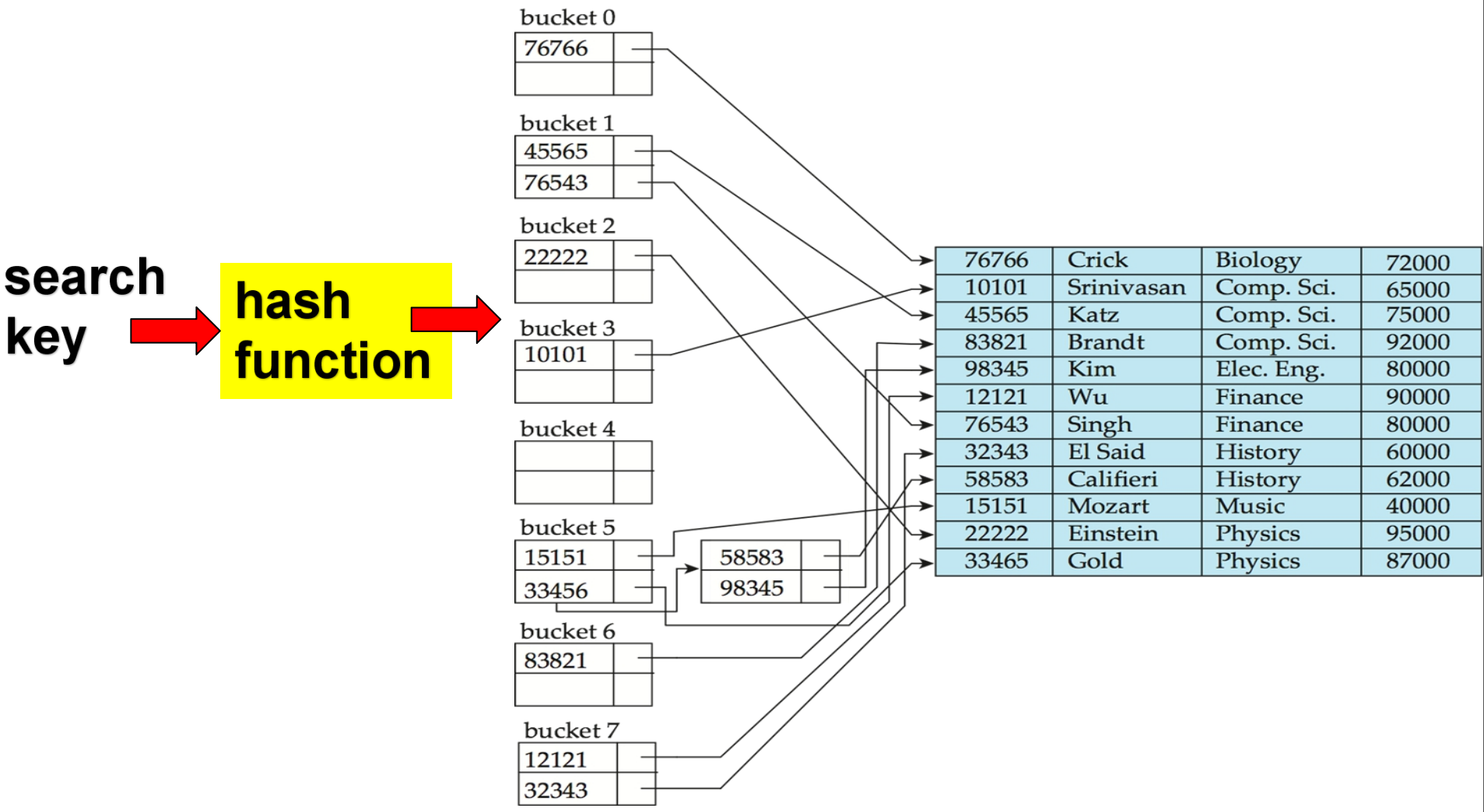




A bucket is typically a disk block.

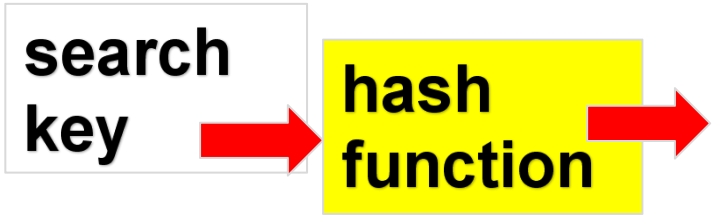
Hash Index

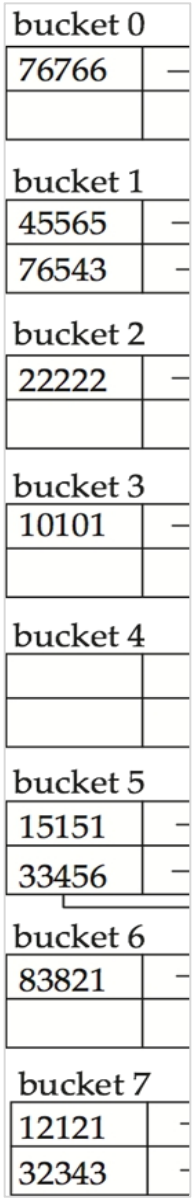
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.



Static Hashing

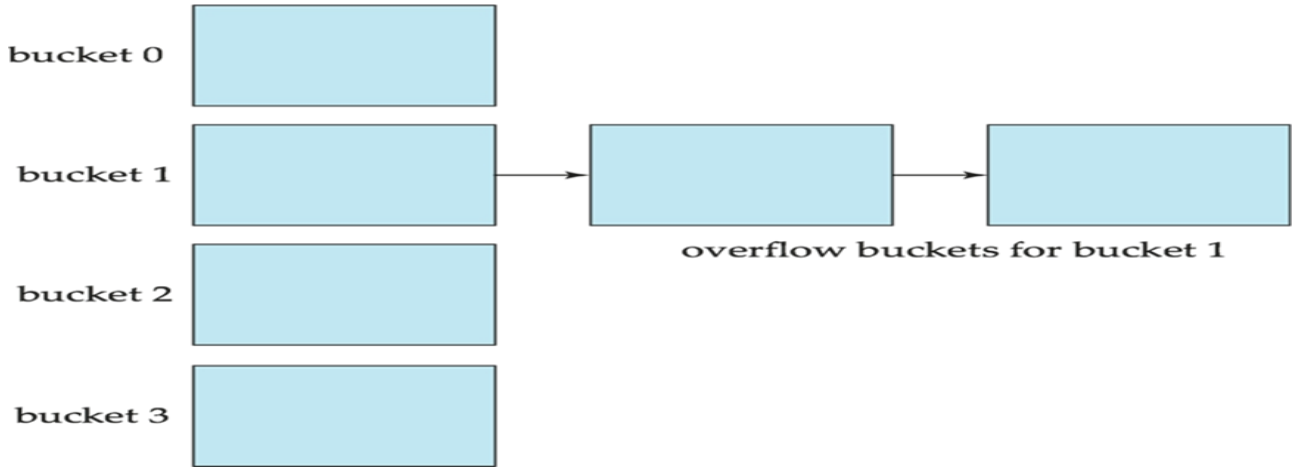
- In static hashing, function *h* maps search-key values to a fixed set of bucket addresses.
- Bucket overflow problem can happen due to
  - Insufficient buckets
  - Skew in distribution of records:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values





Bucket Overflow

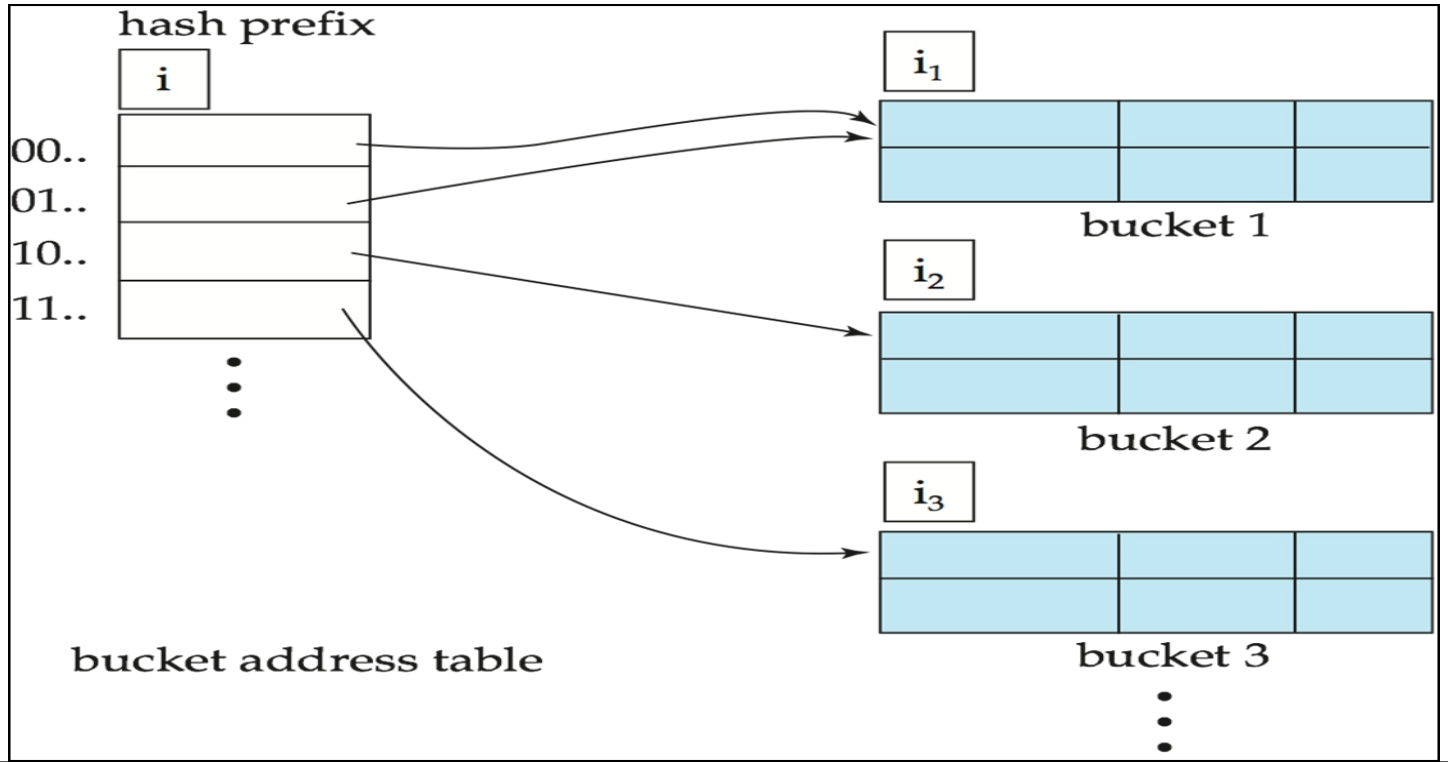
- Can be handled by using *overflow buckets* - the overflow buckets of a given bucket are chained together in a linked list.



- Deficiencies of static hashing
  - If database grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, significant amount of space will be wasted initially.
  - If database shrinks, space will be wasted.

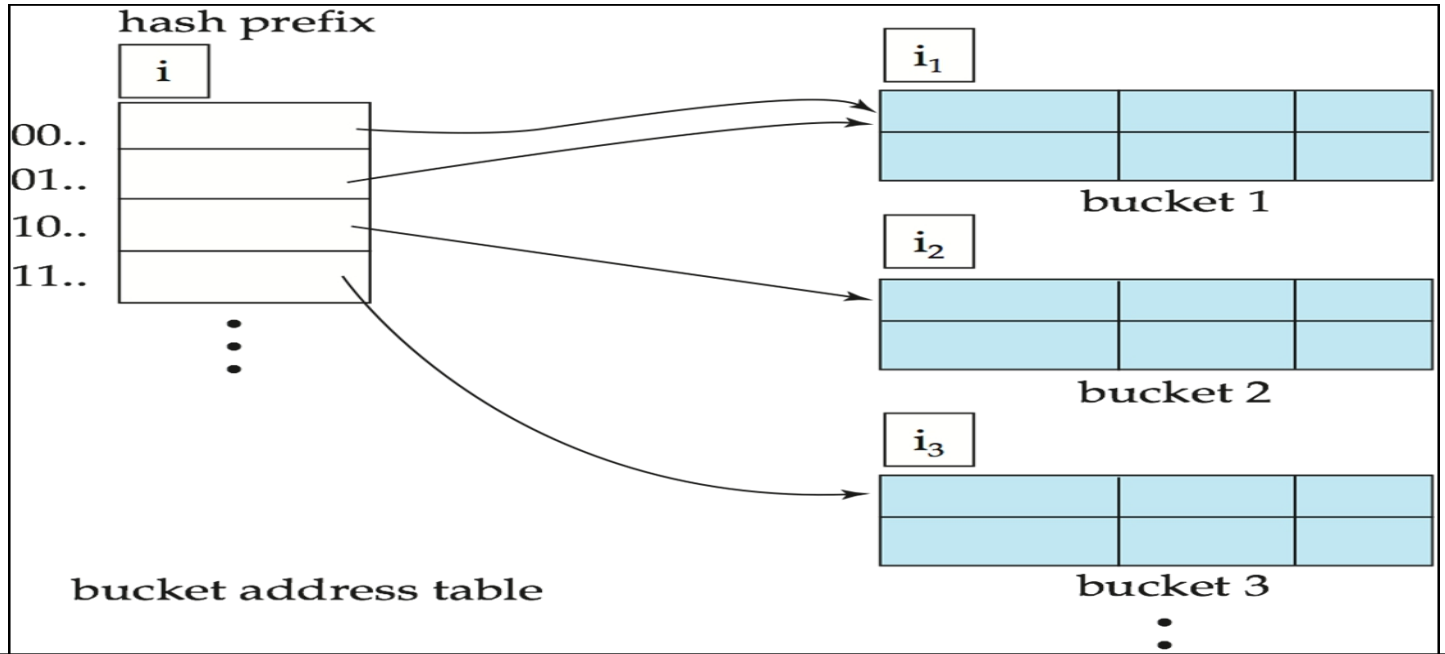
Dynamic Hashing

- Allows the hash function to be modified dynamically
- Extendable hashing – one form of dynamic hashing
  - Hash function generates values over a large range — typically 32-bit (Bucket address table size = 2^i, $0 \leq i \leq 32$)
  - The number of buckets also changes dynamically

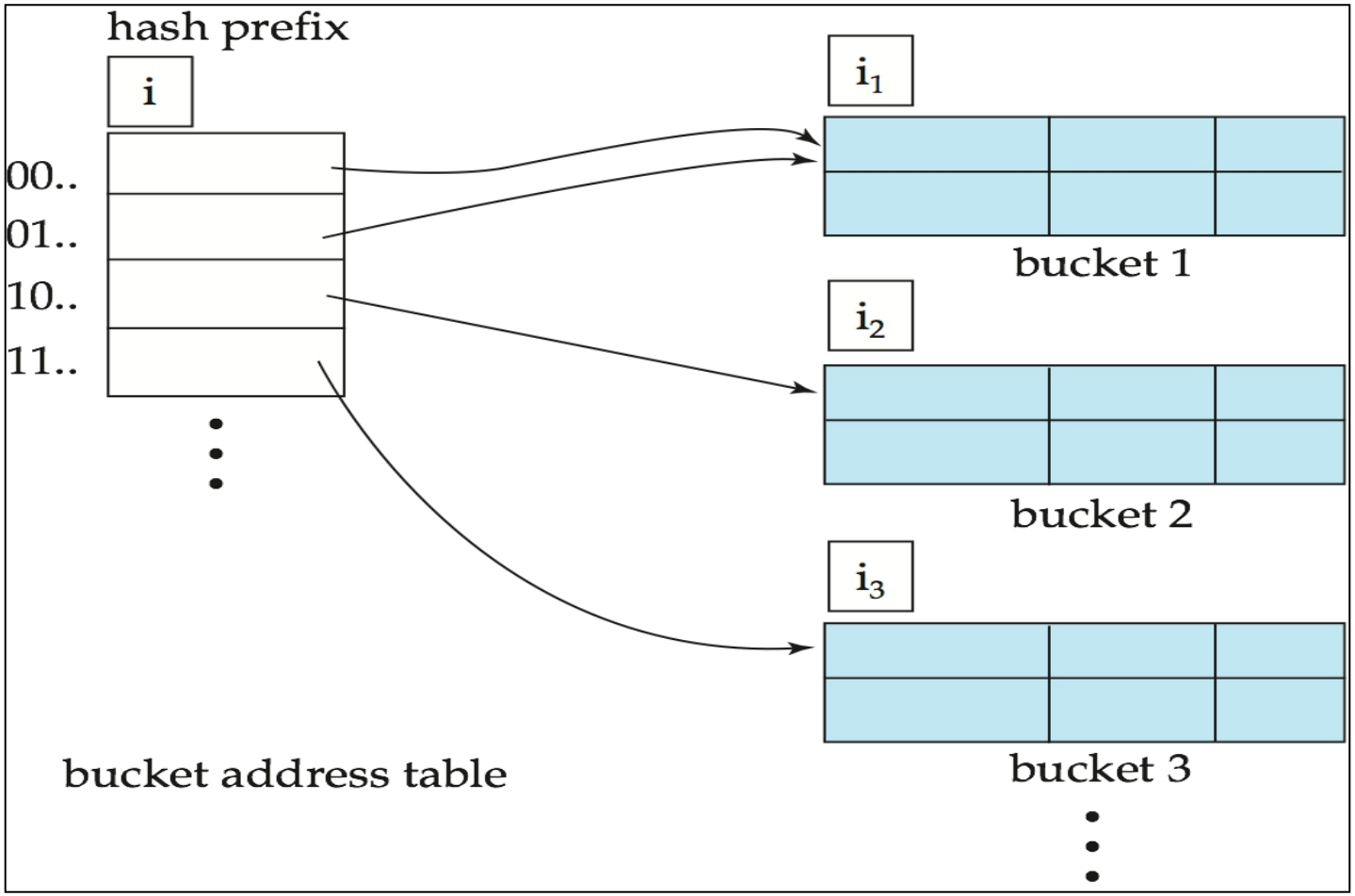


Dynamic Hashing

- 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.
- Actual number of buckets is $< 2^i$
  - The number of buckets also changes dynamically due to merging and splitting of buckets.



General Extendable Hash Structure



hash prefix – "global depth"= size of prefix used in BAT

i_j **- "local depth" = size of prefix used in jth bucket**

Extendable Hash Structure

- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first i high order bits of X as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say j.
  - If there is room in the bucket j insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)

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.

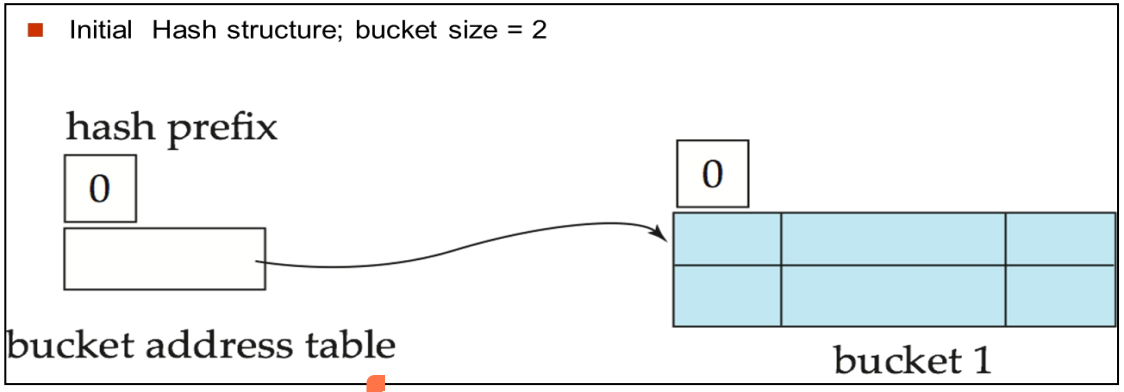
Deletion in Extendable Hash Structure

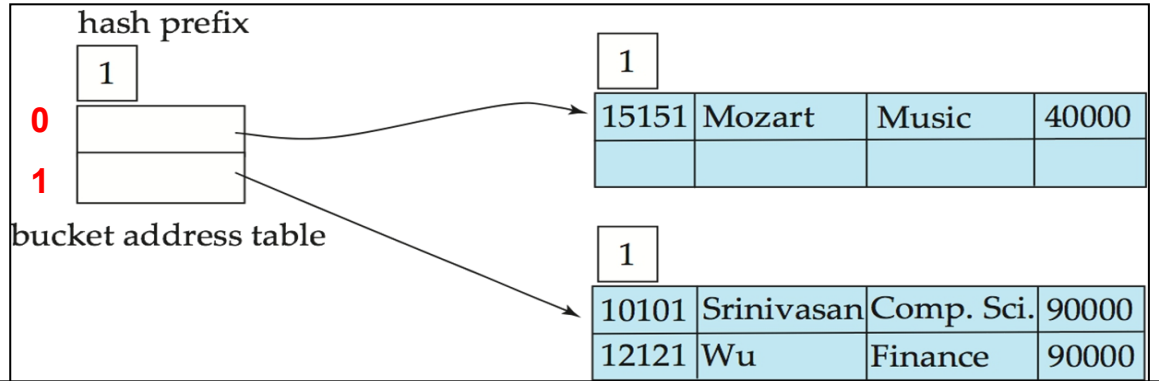
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can merge only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Use of Extendable Hash Structure: Example

dept_name	prefix	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

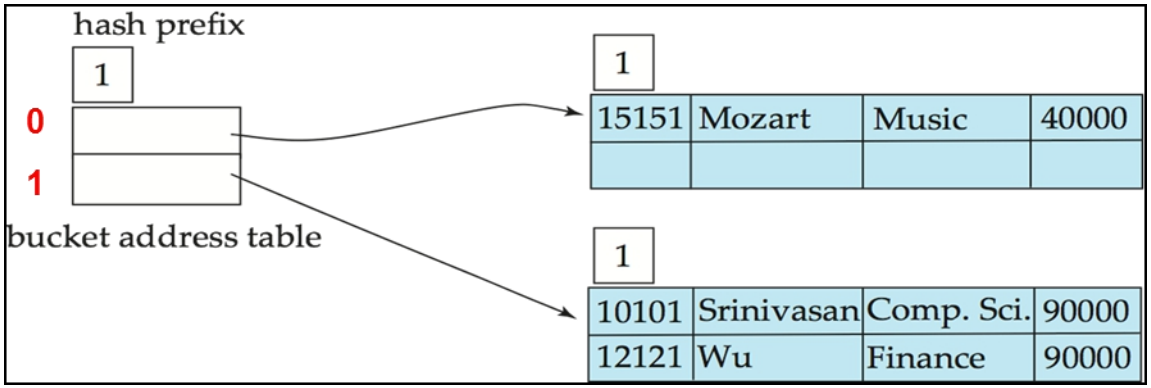
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000

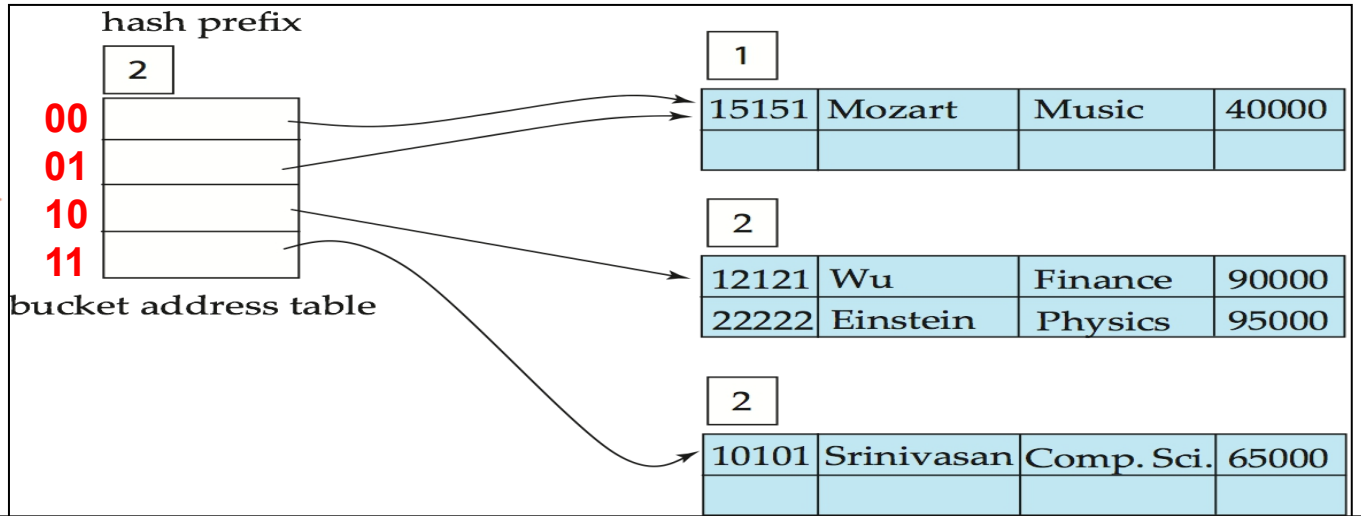




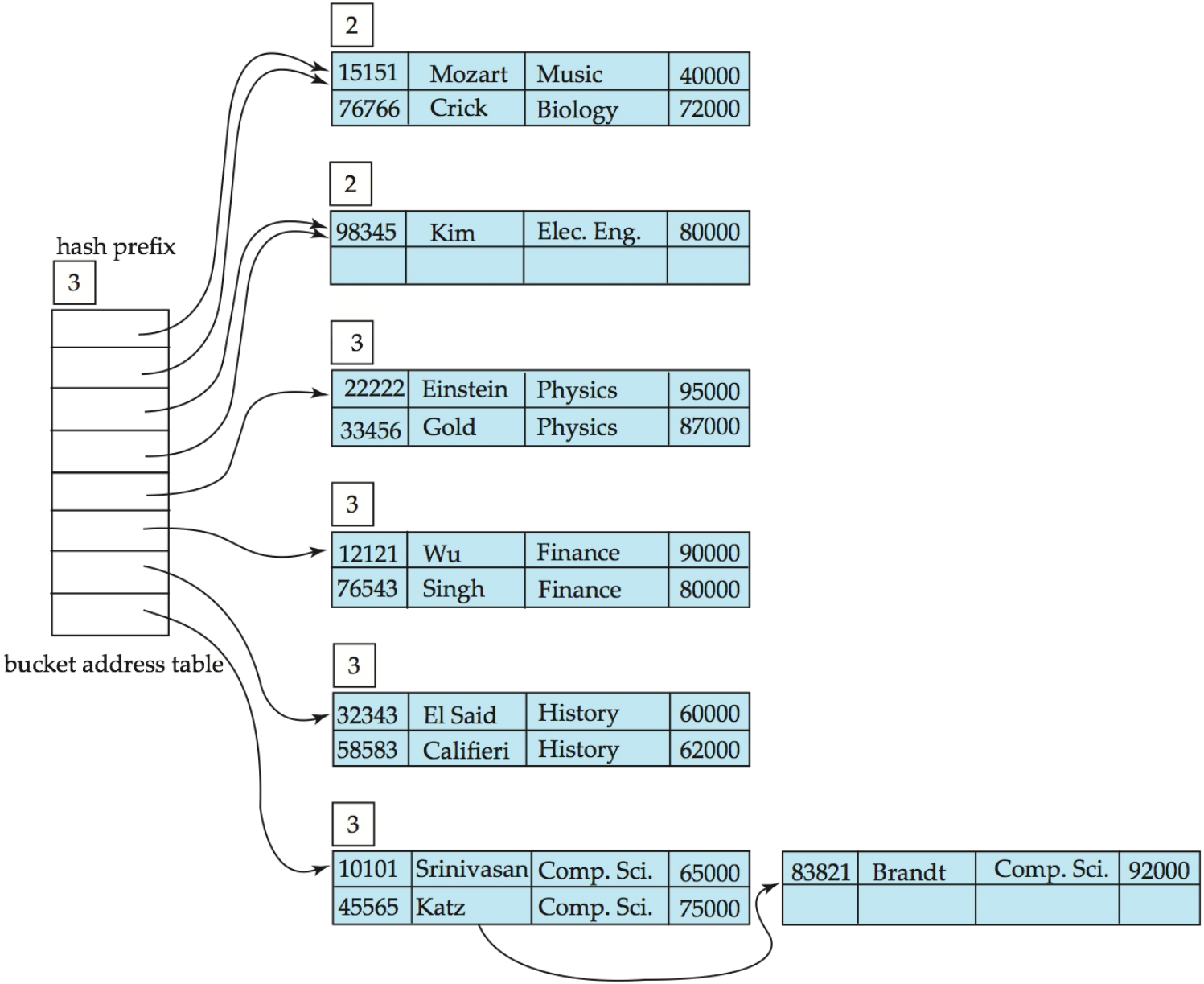
dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100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

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





Example


hash prefix
3
bucket address table
2
15151 Mozart Music 40000
76766 Crick Biology 72000
2
98345 Kim Elec. Eng. 80000
3
22222 Einstein Physics 95000
33456 Gold Physics 87000
3
12121 Wu Finance 90000
76543 Singh Finance 80000
3
32343 El Said History 60000
58583 Califieri History 62000
3
10101 Srinivasan Comp. Sci. 65000
45565 Katz Comp. Sci. 75000
83821 Brandt Comp. Sci. 92000

5. Rules for Using Indexes

1. Use on larger tables
2. Index the primary key of each table
3. Index search fields (fields frequently in WHERE clause)
4. Fields in SQL ORDER BY and GROUP BY commands
5. When there are >100 values but not when there are <30 values

Rules for Using Indexes

6. Avoid use of indexes for fields with long values; perhaps compress values first
7. DBMS may have limit on number of indexes per table and number of bytes per indexed field(s)
8. Be careful of indexing attributes with null values; many DBMSs will not recognize null values in an index search

Comparison of the Indexing Techniques

Issues to consider:

- **Is the cost of periodic re-organization of indexes acceptable?**
- **Relative frequency of insertions and deletions**
- **Is it desirable to optimize average access time at the expense of worst-case access time?**

Comparison of the Indexing Techniques

- **Expected type of queries:**
  - **Hashing is generally better at retrieving records having a specified value of the key. (Equality search)**
  - **If range searches are common, B^+-Tree indices are to be preferred**

```
select instructorName, dept, salary
from instructor
where instructorID = 24587;

select instructorName, dept, salary
from instructor
where instructorID between 24503 and 41899;
```

Database Tuning

- Index tuning goals
  - Dynamically evaluate requirements
  - Reorganize indexes to yield best performance
- Reasons for revising initial index choice
  - Certain queries may take too long to run due to lack of an index
  - Certain indexes may not get utilized
  - Certain indexes may undergo too much updating if based on an attribute that undergoes frequent changes

References

- ***Fundamentals of Database Systems.*** **Elmasri, R. and Navathe, S.B. *Pearson***

- ***Database Systems: A Practical Approach to Design, Implementation and Management.*** **Connolly, T. M. and Begg, C. E. Pearson.**

- ***Database System Concepts*** **Silberschatz, A., Korth, H,. and Sudarshan, S. McGraw_Hill**