ALL

Chapter 14: Indexing

Database System Concepts, 7th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use

Outline

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- Hashing

Basic Concepts

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

Basic Concepts

- An index for a file in a database system works in much the same way as the index in the textbook.
- Database-system indices play the same role as book indices in libraries.
- For example, to retrieve a *student* record given an ID, the database system would look up an index to find on which disk block the corresponding record resides, and then fetch the disk block, to get the appropriate *student* record..

Basic Concepts

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

search-key	pointer

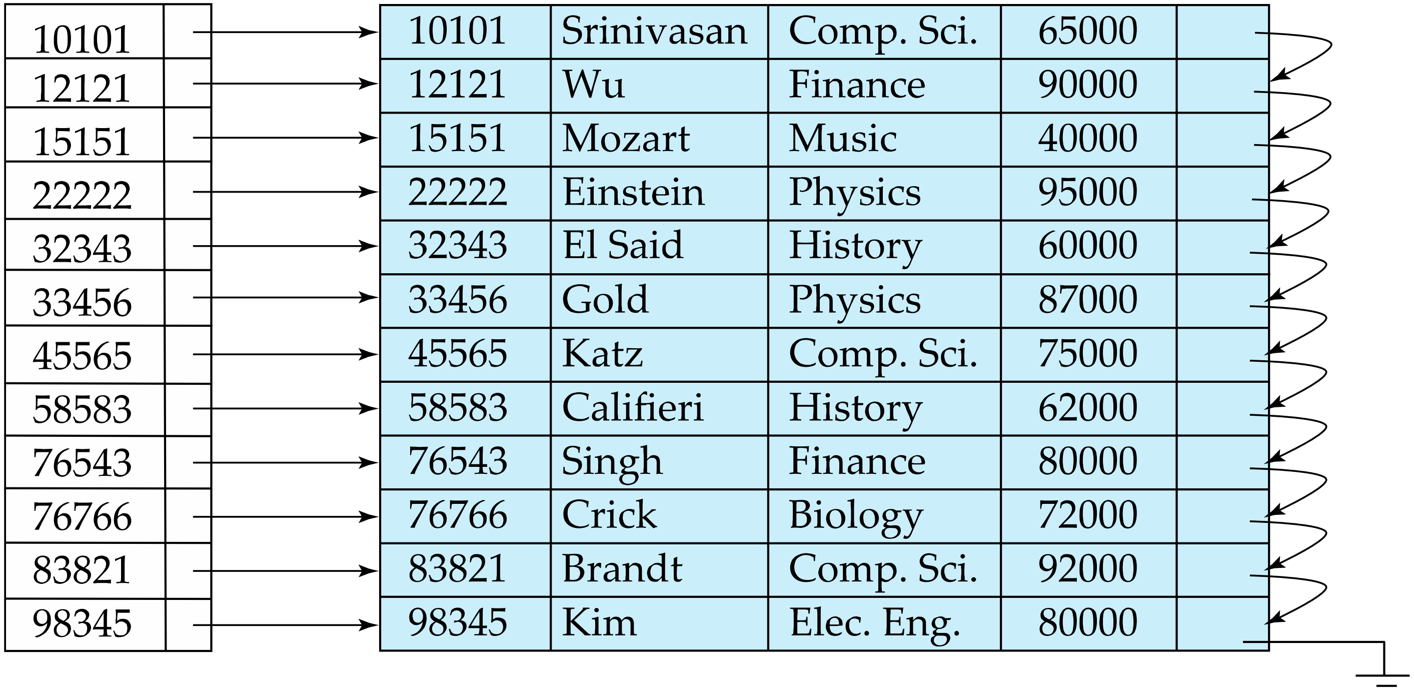
- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".

Ordered Indices

- In an **ordered index,** index entries are stored sorted on the search key value.
- **Clustering 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 primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **nonclustering 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

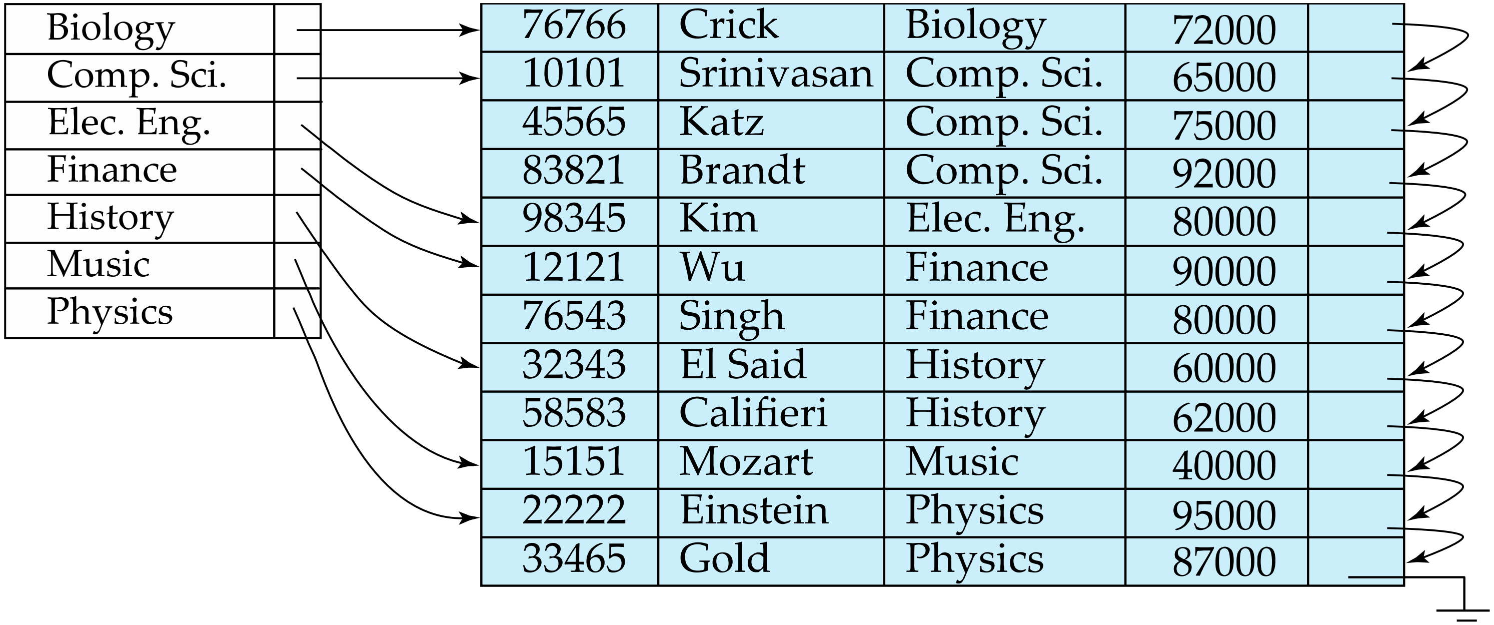


10101	
12121	
15151	
22222	
32343	
33456	
45565	
58583	
76543	
76766	
83821	
98345	

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	

Dense Index Files (Cont.)

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

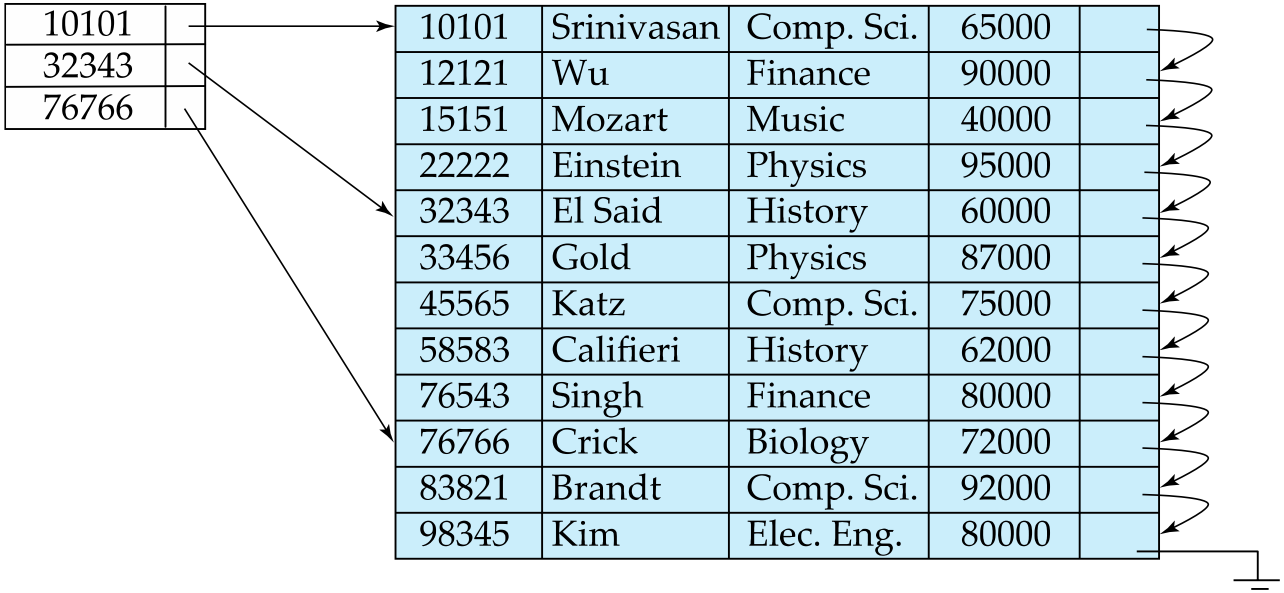


Biology	
Comp. Sci.	
Elec. Eng.	
Finance	
History	
Music	
Physics	

76766	Crick	Biology	72000	
10101	Srinivasan	Comp. Sci.	65000	
45565	Katz	Comp. Sci.	75000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	
12121	Wu	Finance	90000	
76543	Singh	Finance	80000	
32343	El Said	History	60000	
58583	Califieri	History	62000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
33465	Gold	Physics	87000	

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

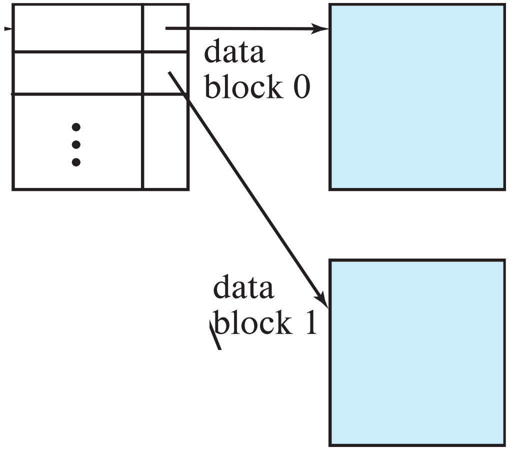


10101	
32343	
76766	

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

Sparse Index Files (Cont.)

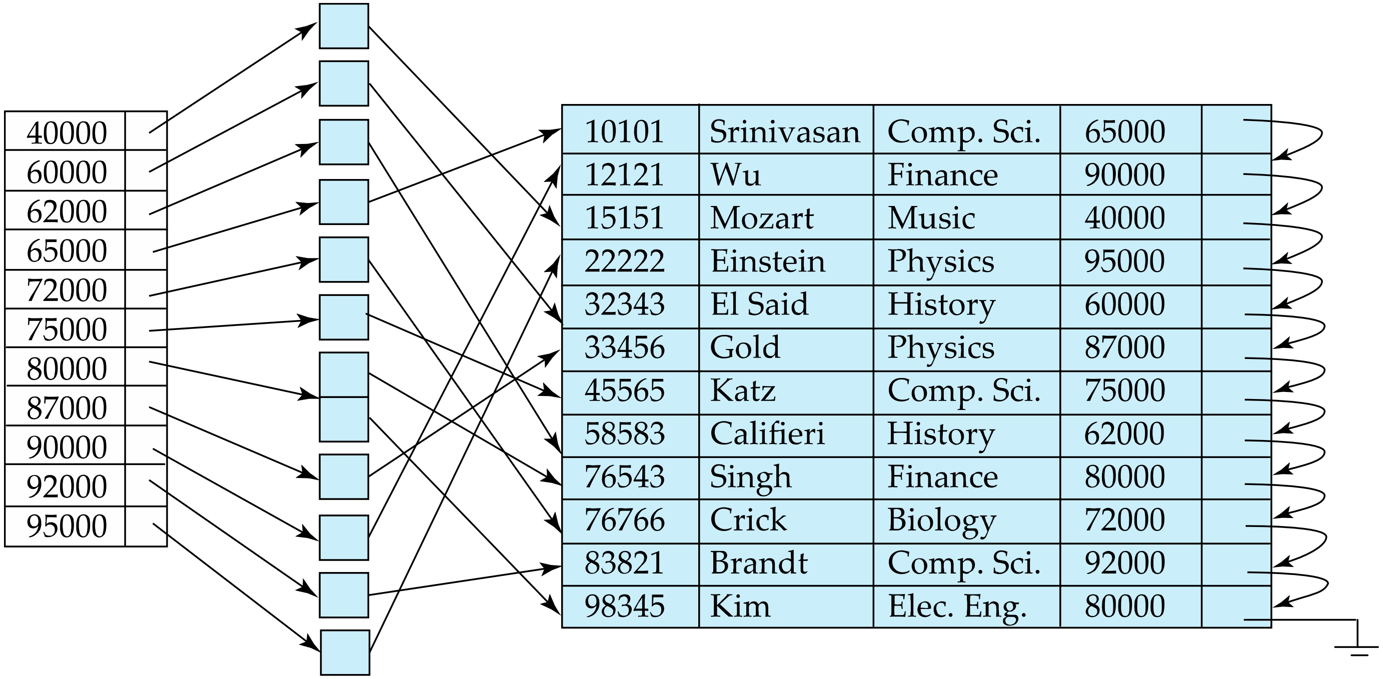
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**:
  - for clustered index: sparse index with an index entry for every block in file, corresponding to least search-key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



  - For unclustered index: sparse index on top of dense index (multilevel index)

Secondary Indices Example

- Secondary index on salary field of instructor



40000	
60000	
62000	
65000	
72000	
75000	
80000	
87000	
90000	
92000	
95000	

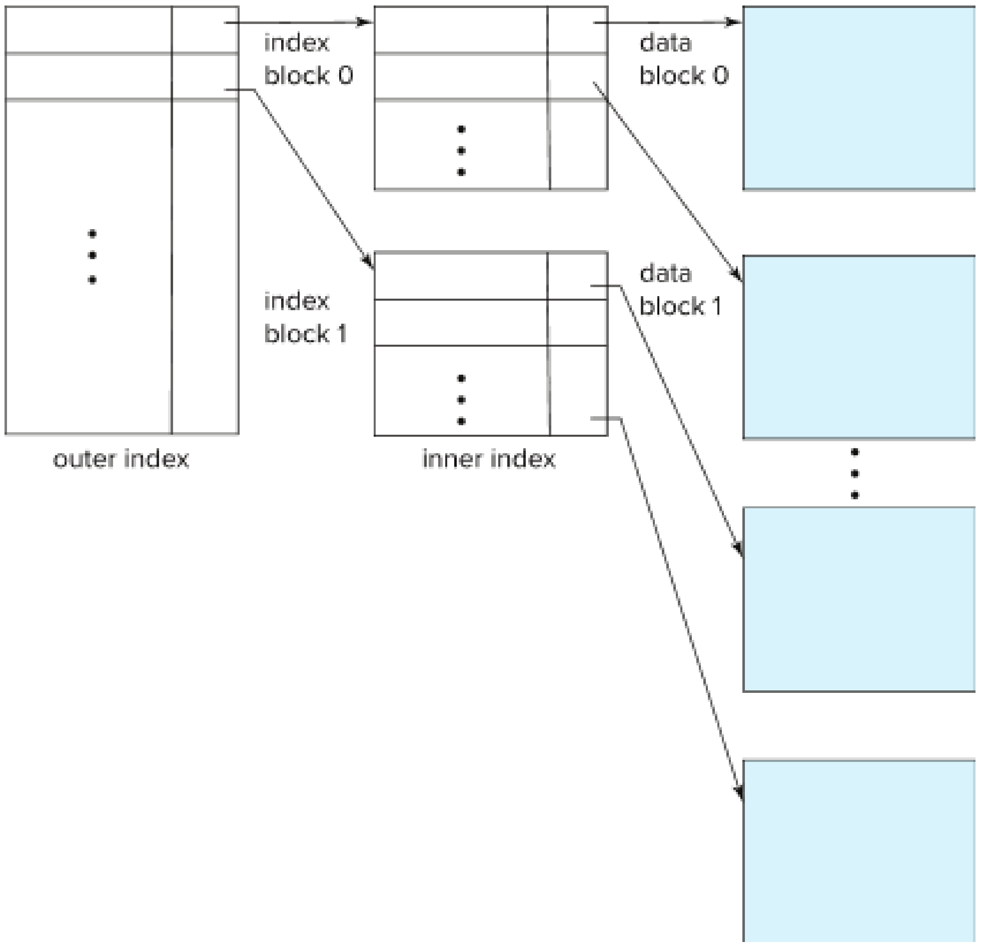
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

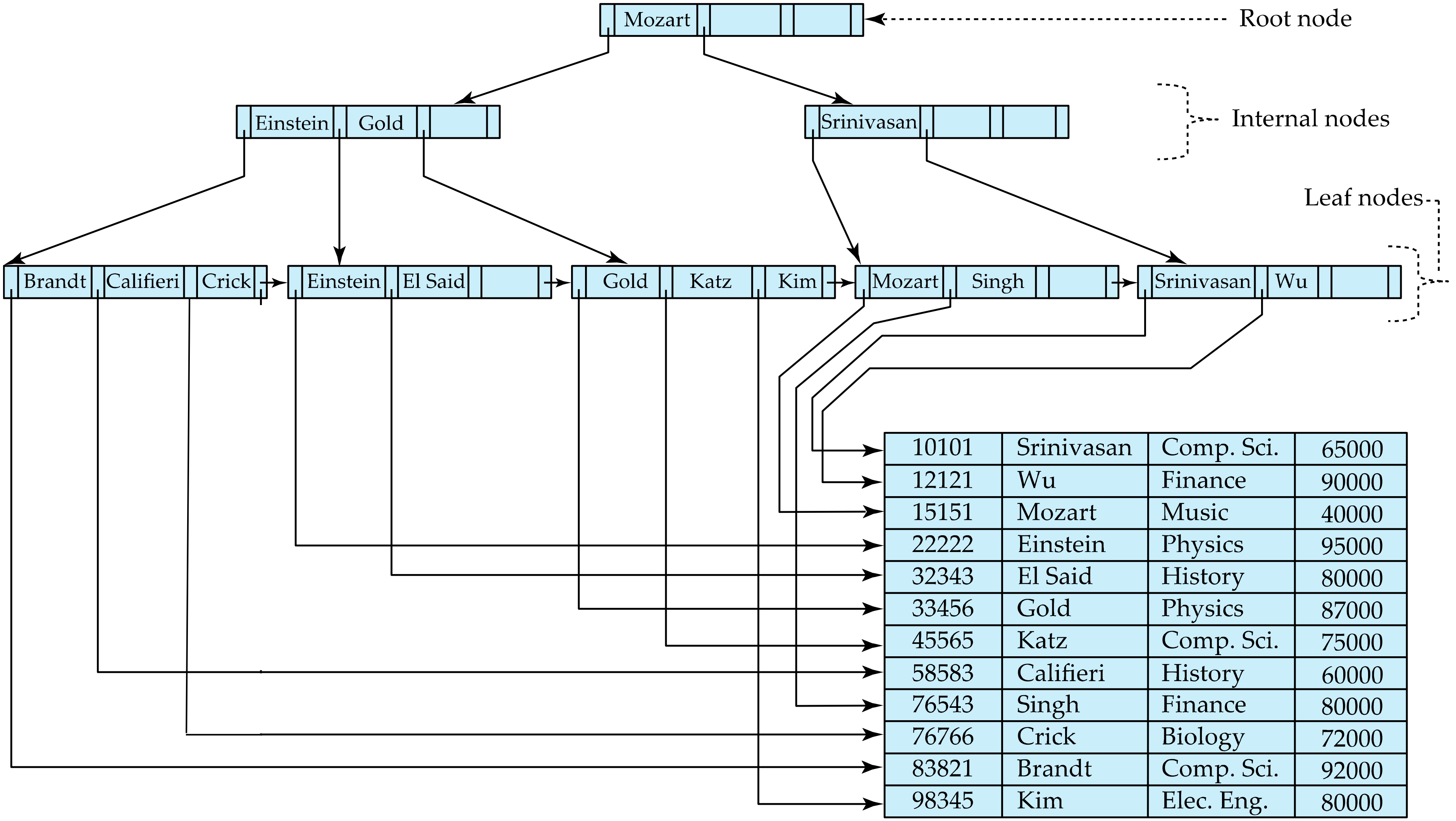
Multilevel Index

- 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



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

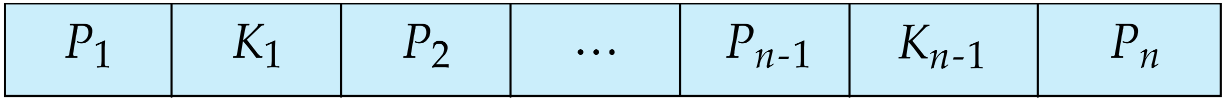
B^+-Tree Index Files (Cont.)

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



P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

  - 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

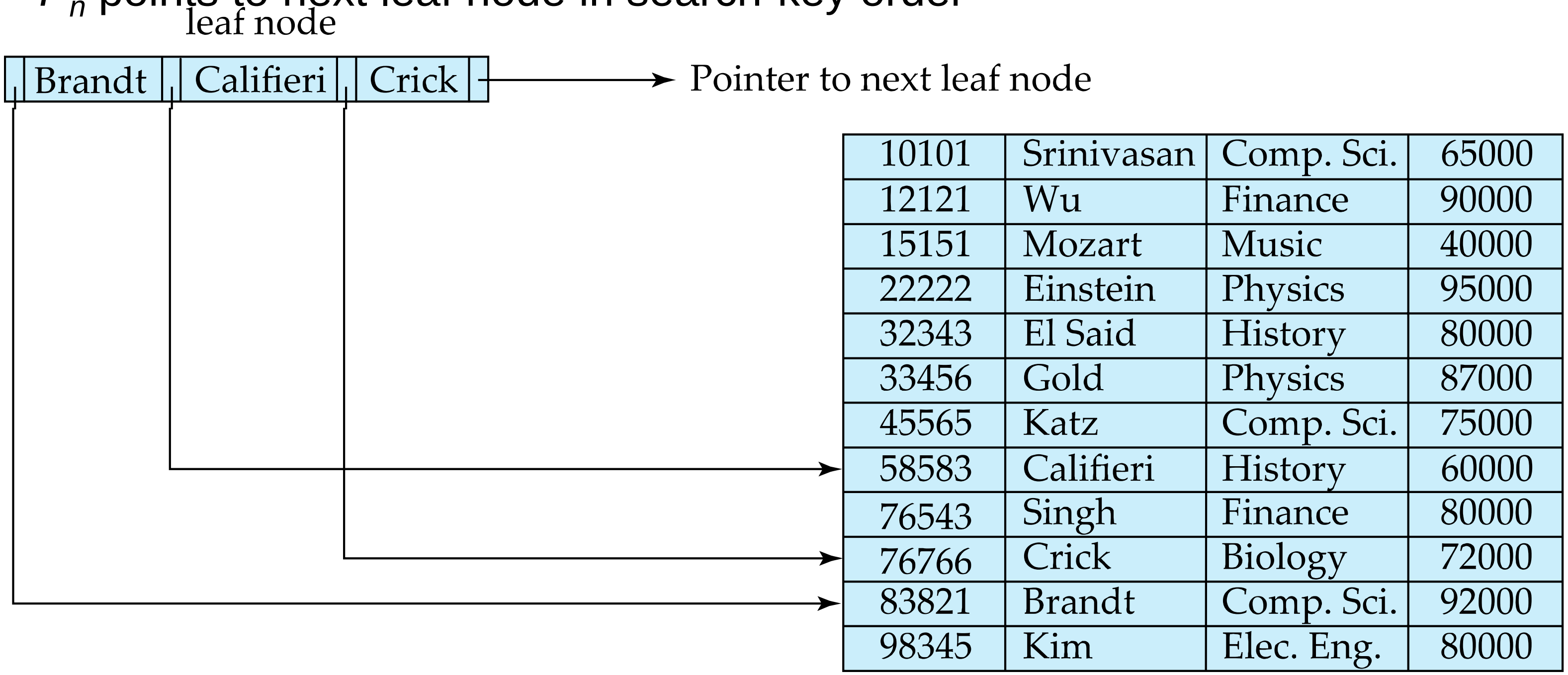
$$K_1 < K_2 < K_3 < . . . < K_{n–1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B^+-Trees

Properties of a leaf node:

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



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

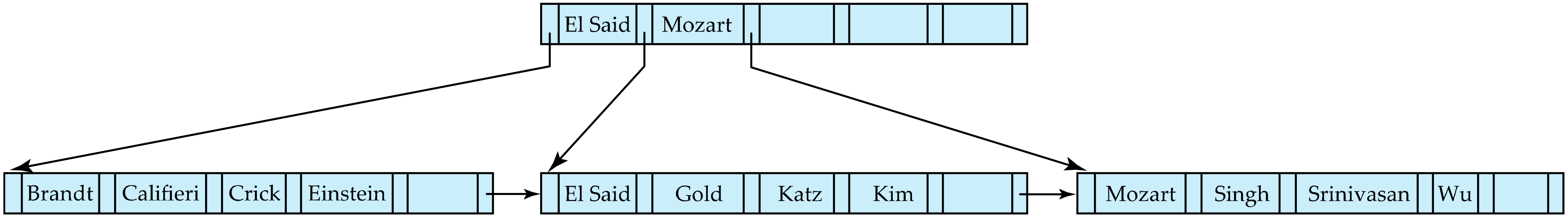
Non-Leaf Nodes in B$^+$-Trees

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

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

Example of B$^+$-tree

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



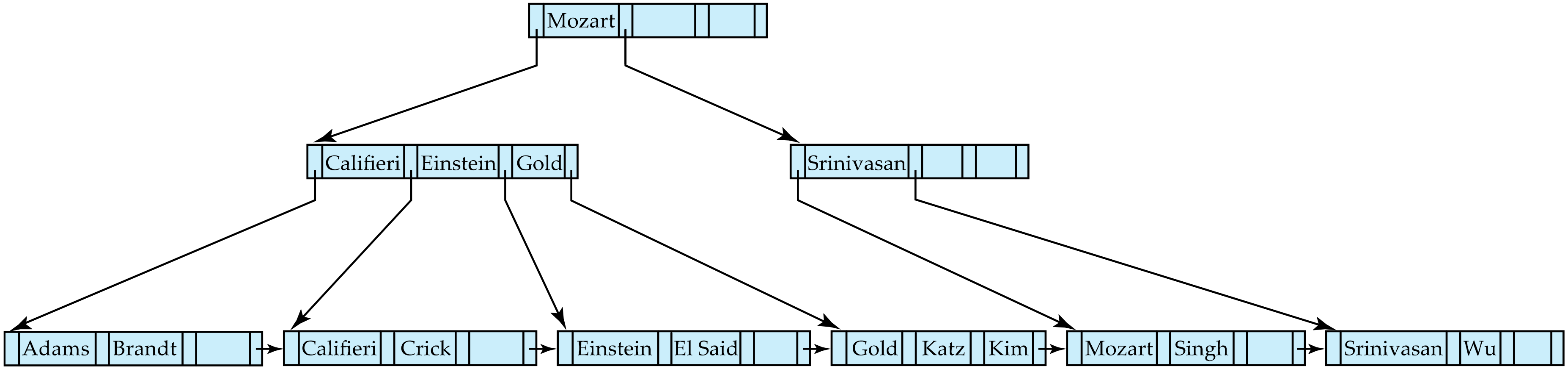
- Leaf nodes must have between 3 and 5 values ($\lceil(n–1)/2\rceil$ and n –1, with n = 6).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil(n/2\rceil$ and n with n =6).
- Root must have at least 2 children.

Observations about B^+-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
    - Level below root has at least $2*\lceil n/2 \rceil$ values
    - Next level has at least $2*\lceil n/2 \rceil * \lceil n/2 \rceil$ 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).

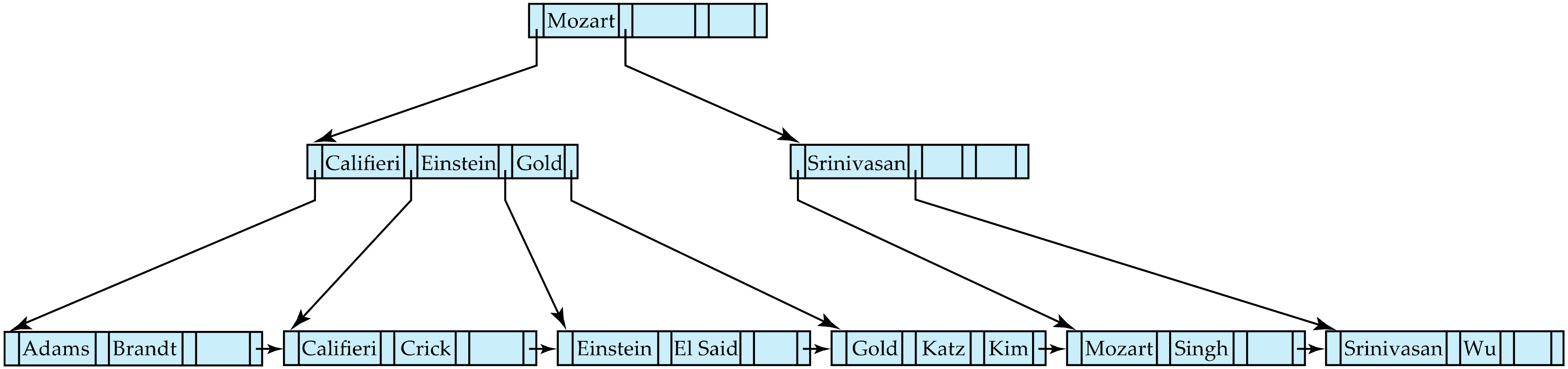
Queries on B$^+$-Trees

```
function find(v)
1.  C=root
2.  while (C is not a leaf node)
    1.  Let i be least number s.t. $V \leq K_i$.
    2.  if there is no such number i then
    3.      Set C = last non-null pointer in C
    4.  else if ($v = C.K_i$ ) Set C = $P_{i+1}$
    5.  else set $C = C.P_i$
3.  if for some i, $K_i = V$  then return $C.P_i$
4.  else return null /* no record with search-key value v exists. */
```



Queries on B$^+$-Trees (Cont.)

- **Range queries** find all records with search key values in a given range
  - See book for details of **function** *findRange*(*lb, ub*) which returns set of all such records
  - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next*() function



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

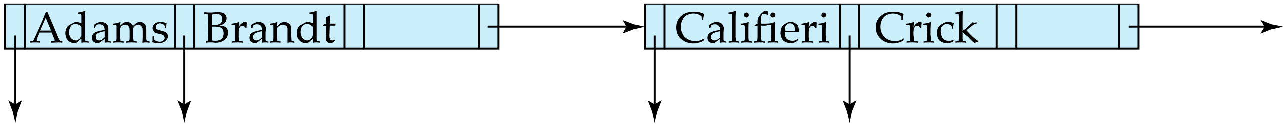
Assume record already added to the file. Let

- *pr* be pointer to the record, and let
- v be the search key value of the record

1. Find the leaf node in which the search-key value would appear
    1. If there is room in the leaf node, insert (v, *pr*) pair in the leaf node
    2. Otherwise, split the node (along with the new (*v, pr*) entry) as discussed in the next slide, and propagate updates to parent nodes.

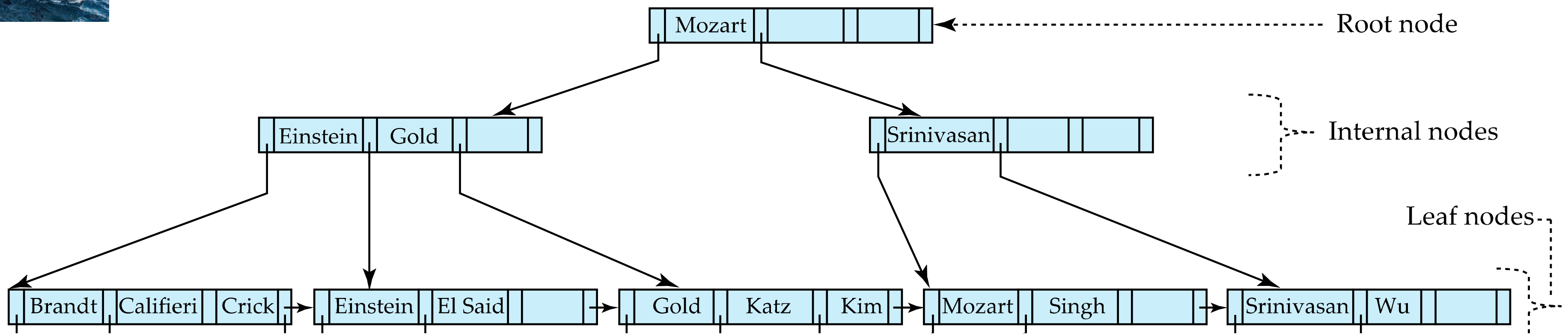
Updates on B$^+$-Trees: Insertion (Cont.)

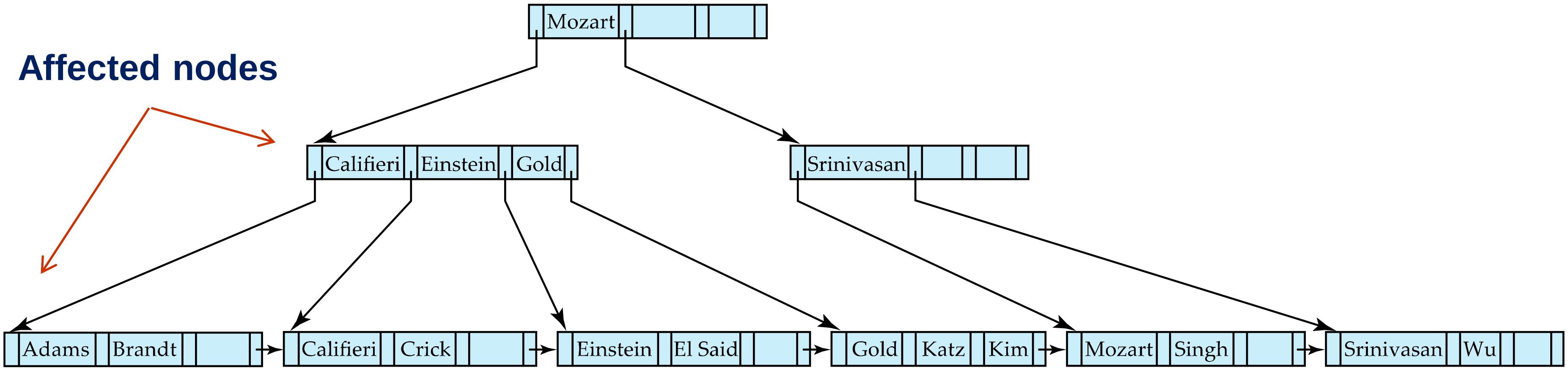
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k*,*p*) 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.



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

B+-Tree Insertion

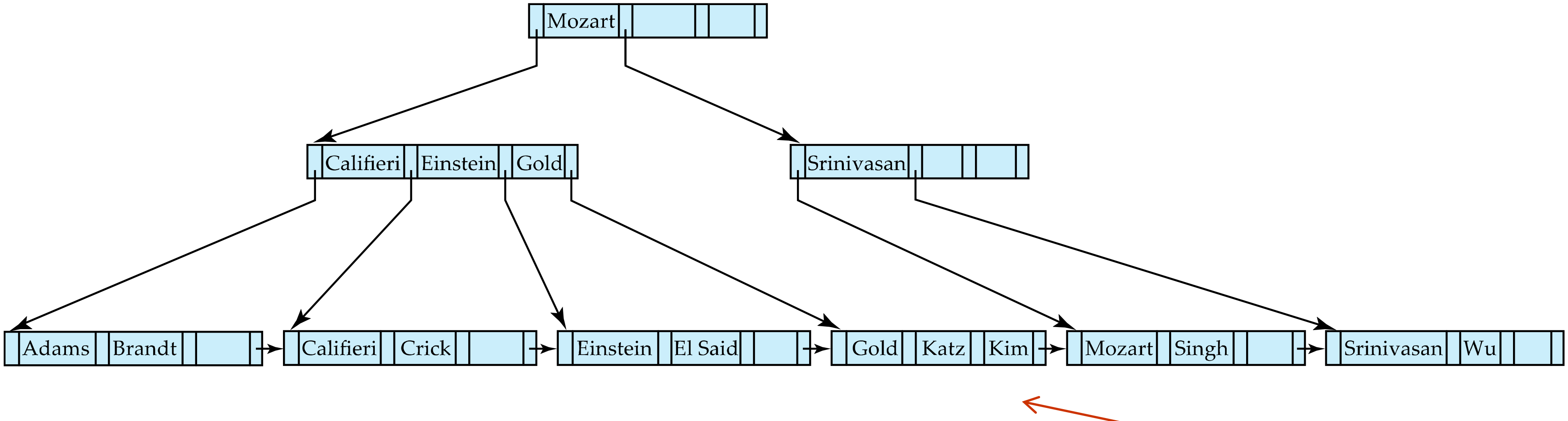




Affected nodes

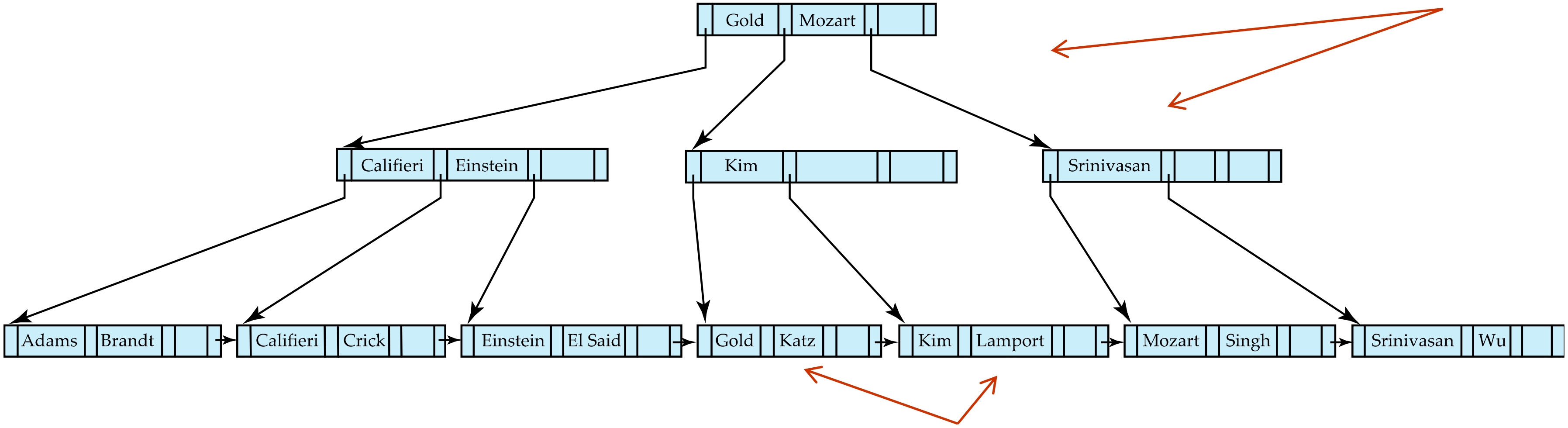
B+-Tree before and after insertion of “Adams”

B$^+$-Tree Insertion



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

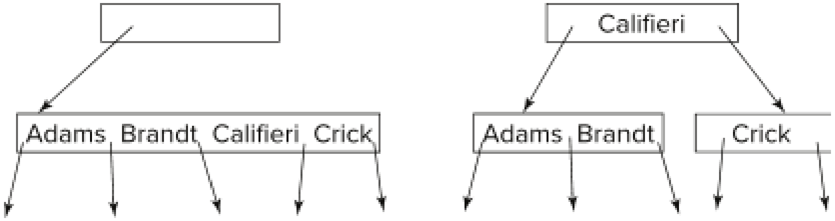
Affected nodes



Affected nodes

Insertion in B^+-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1,K_1, \ldots, K_{\lceil n/2\rceil-1},P_{\lceil n/2\rceil}$ from M back into node N
  - Copy $P_{\lceil n/2\rceil+1},K_{\lceil n/2\rceil+1},\ldots,K_n,P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2\rceil},N')$ into parent N
- Example

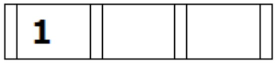


- **Read pseudocode in book!**

Insertion in B$^+$-Trees (Cont.)

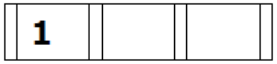
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

- Insert 1



Insertion in B^+-Trees (Cont.)

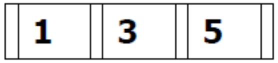
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



- Insert 3, 5

Insertion in B⁺-Trees (Cont.)

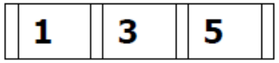
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



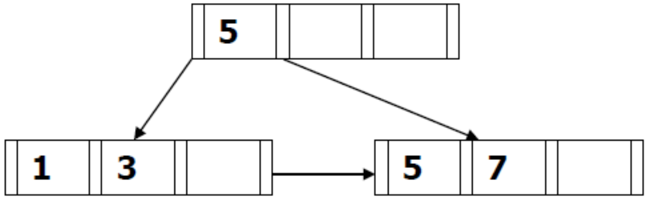
- Insert 7

Insertion in B+-Trees (Cont.)

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

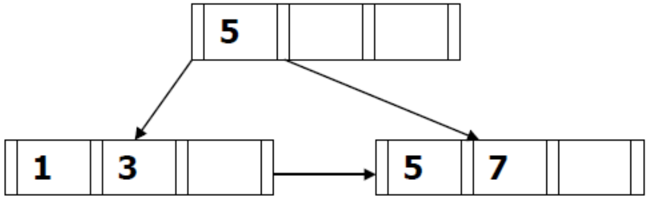


- Insert 7



Insertion in B+-Trees (Cont.)

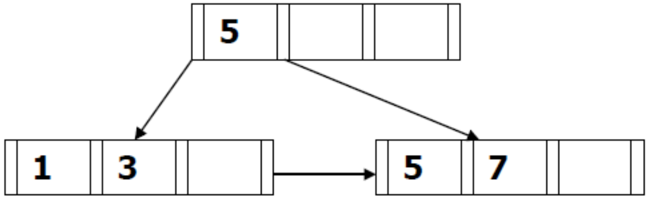
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



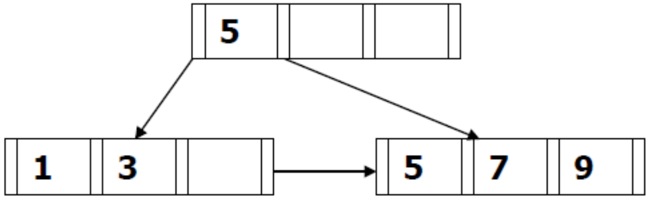
- Insert 9

Insertion in B+-Trees (Cont.)

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

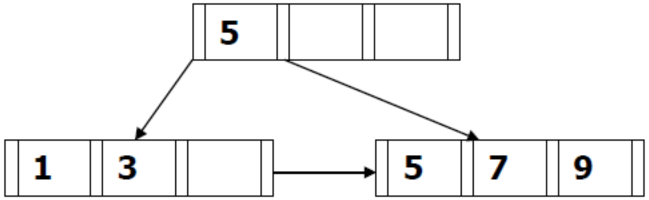


- Insert 9

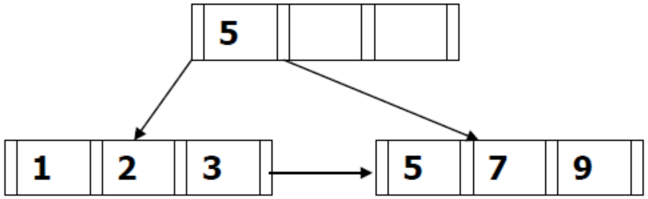


Insertion in B$^+$-Trees (Cont.)

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

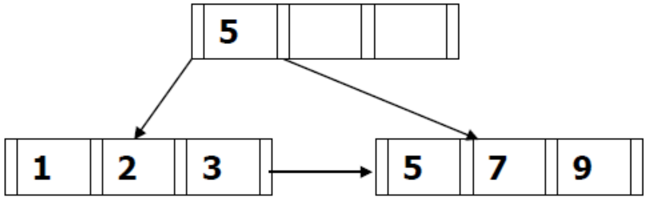


- Insert 2



Insertion in B^+-Trees (Cont.)

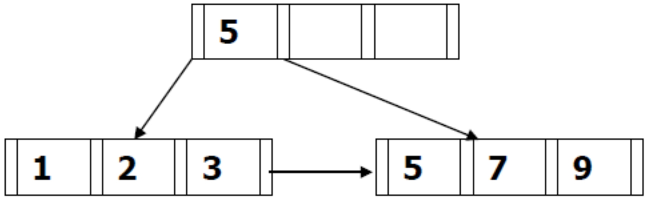
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



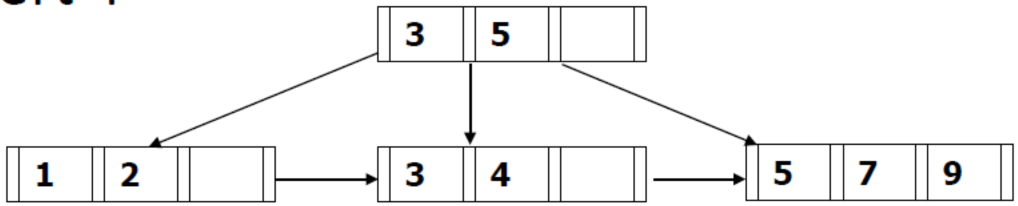
- Insert 4

Insertion in B$^+$-Trees (Cont.)

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

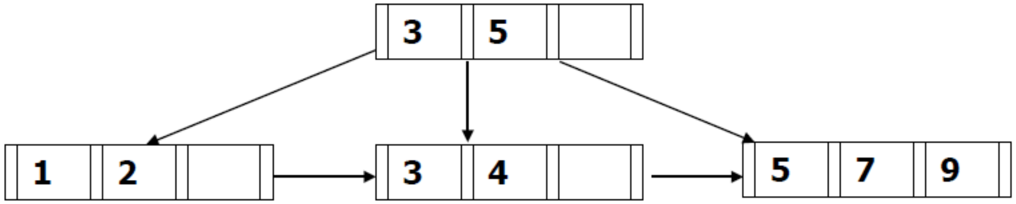


- Insert 4



Insertion in B+-Trees (Cont.)

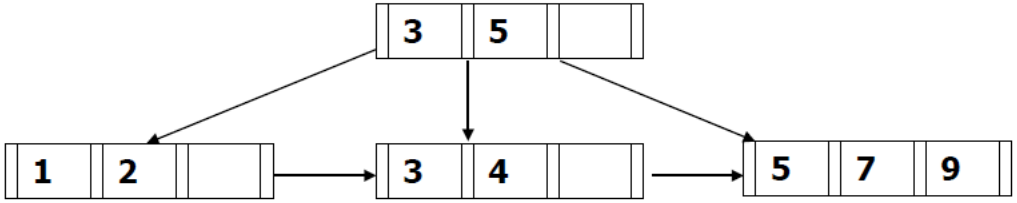
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



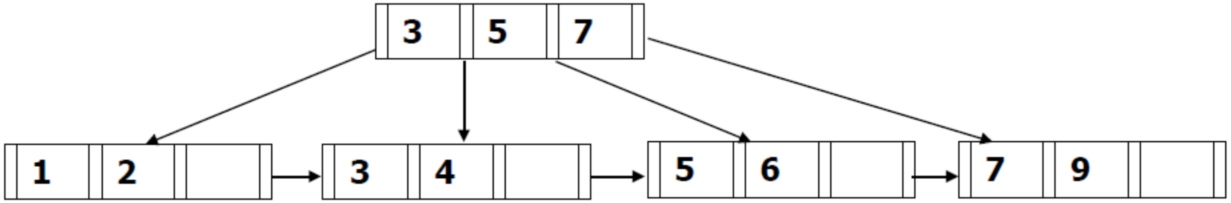
- Insert 6

Insertion in B+-Trees (Cont.)

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

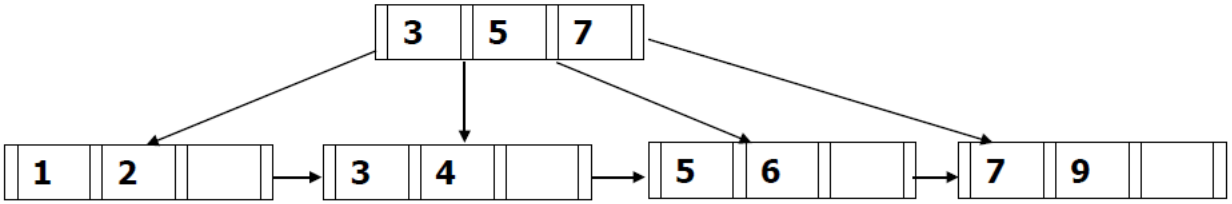


- Insert 6



Insertion in B+-Trees (Cont.)

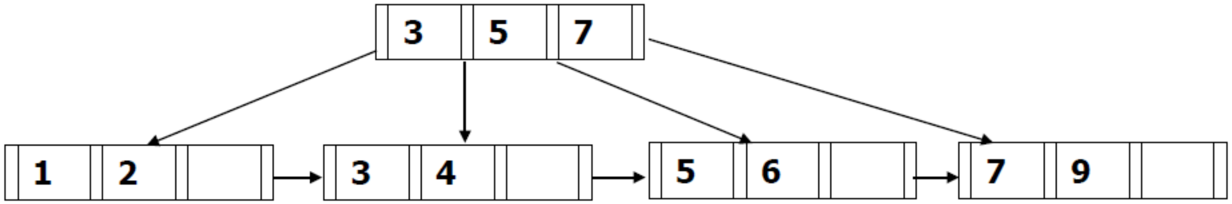
Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10



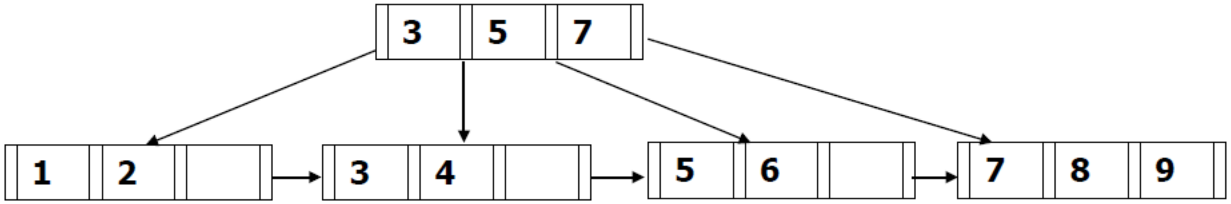
- Insert 8

Insertion in B+-Trees (Cont.)

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

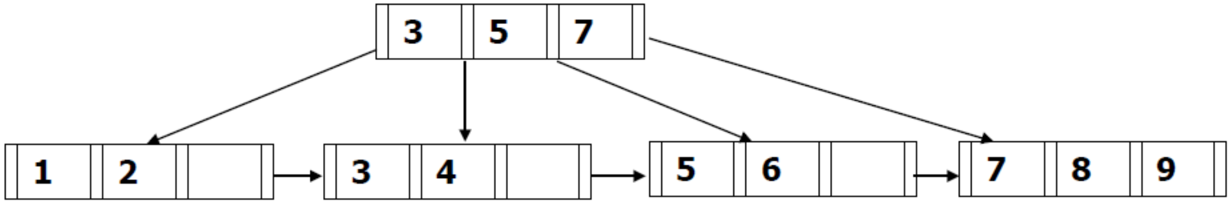


- Insert 8

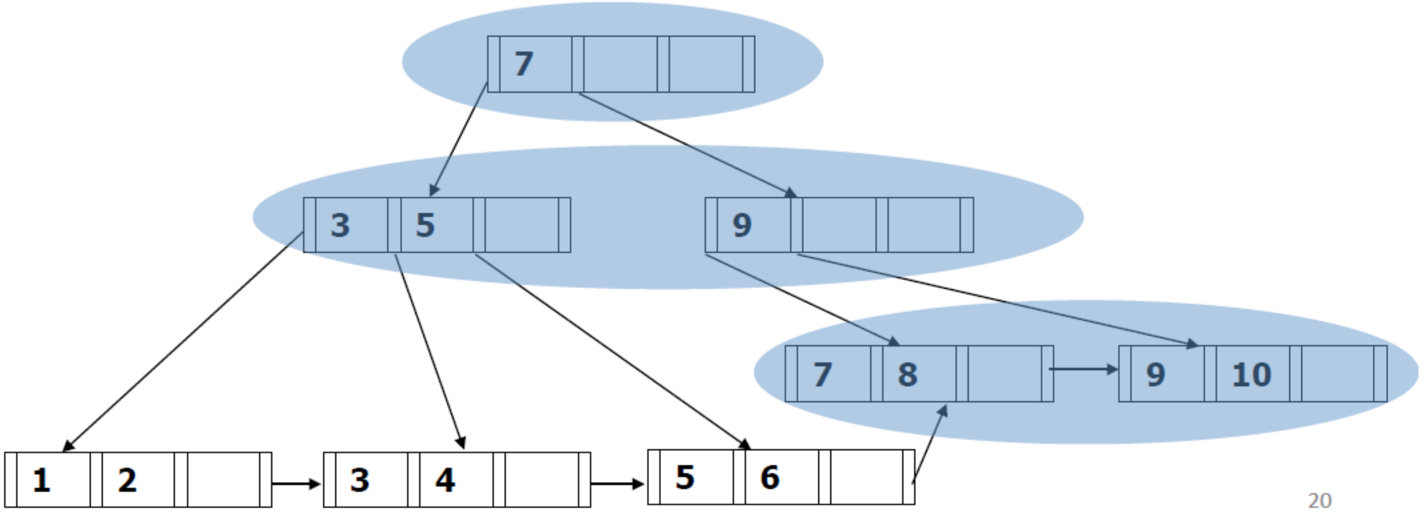


Insertion in B+-Trees (Cont.)

Insert 1, 3, 5, 7, 9, 2, 4, 6, 8, 10

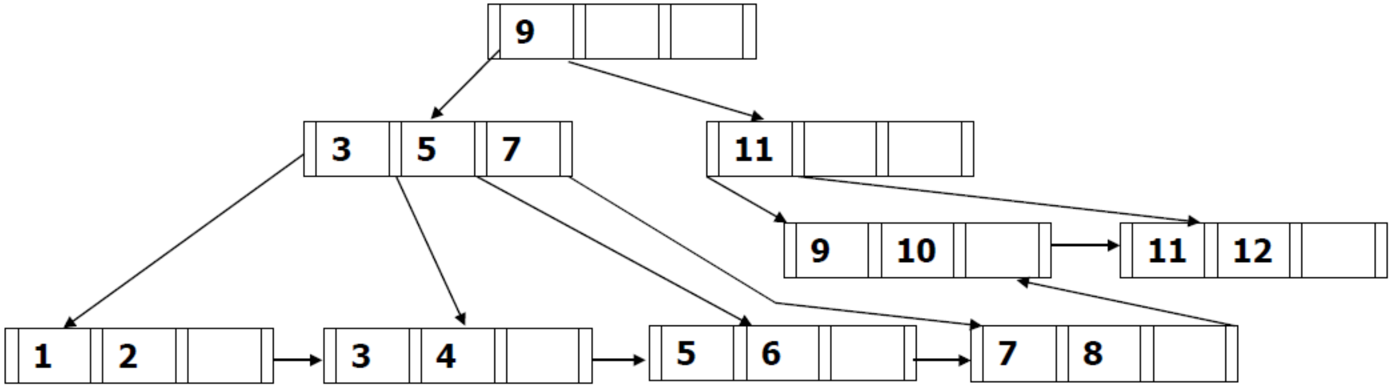


- Insert 10



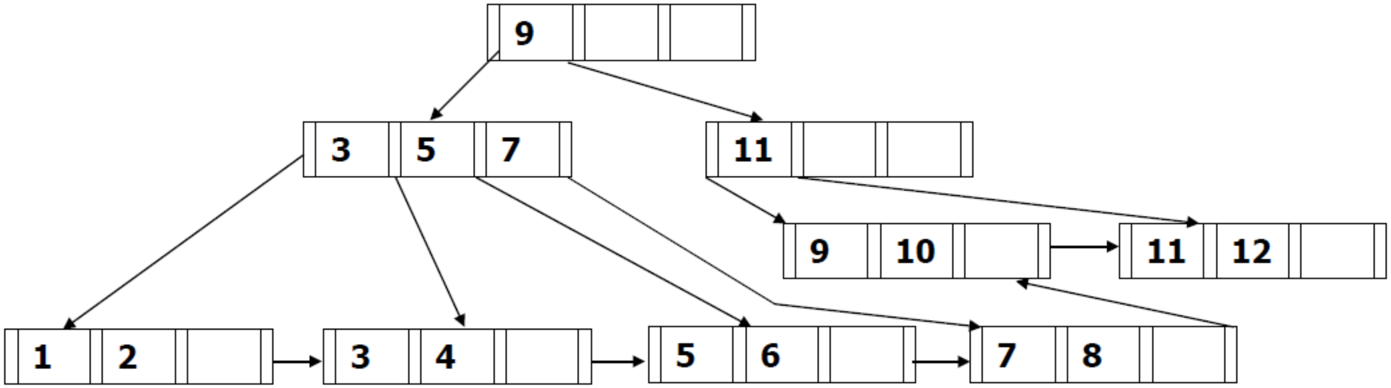
Deletion in B$^+$-Trees (Cont.)

Show the tree after deletions

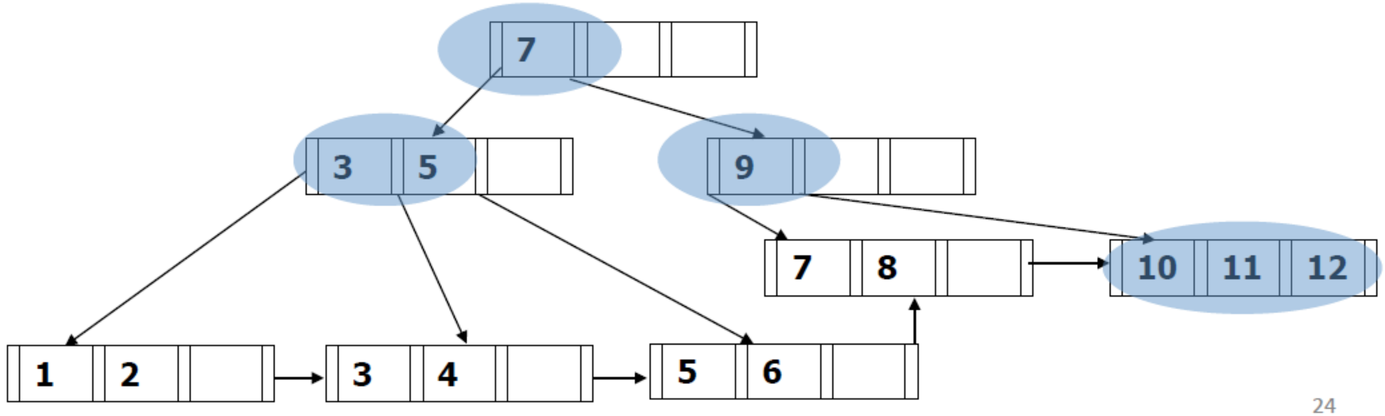


- Remove 9, 7, 8

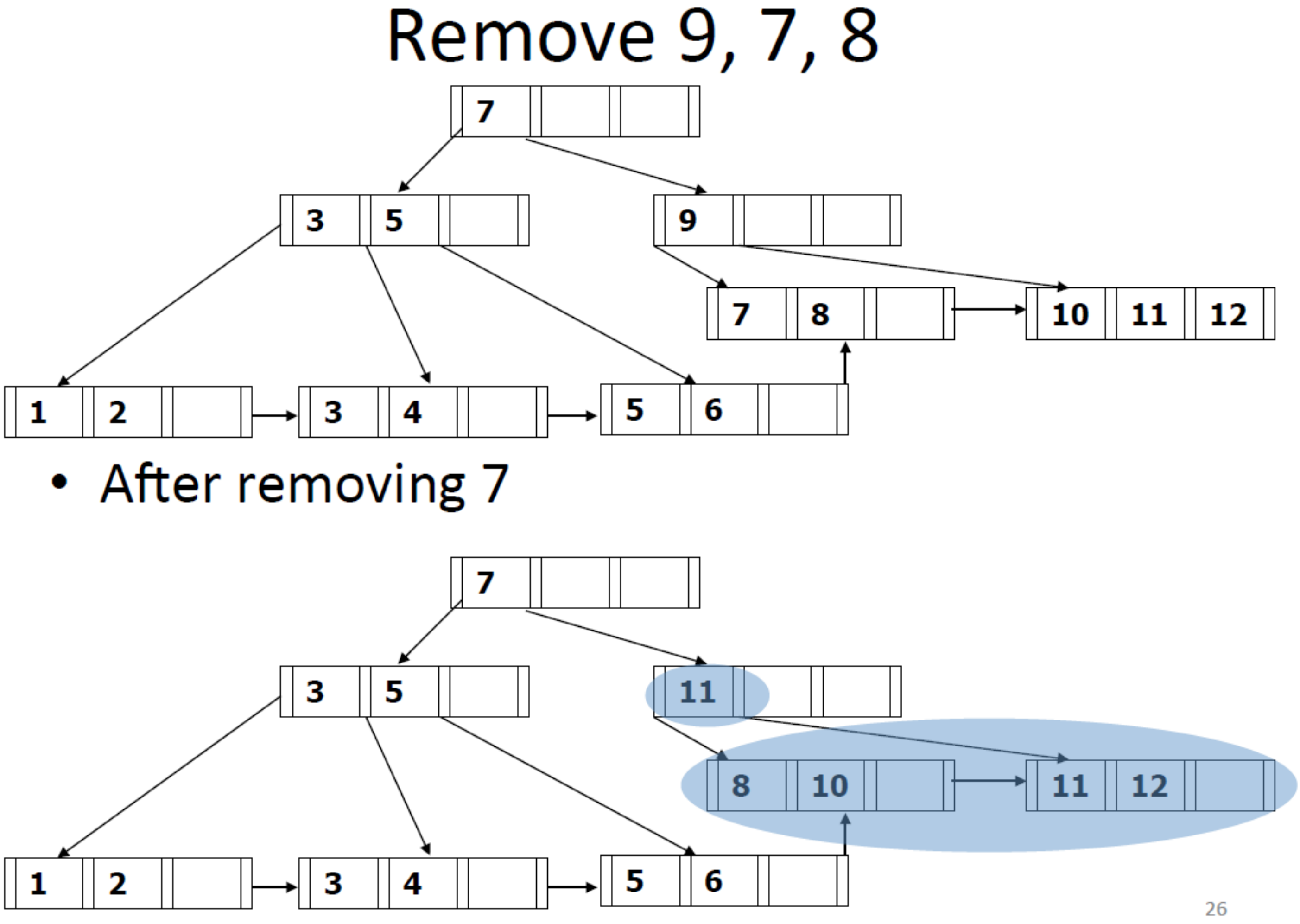
Deletion in B$^+$-Trees (Cont.)



- After removing 9



Deletion in B$^+$-Trees (Cont.)

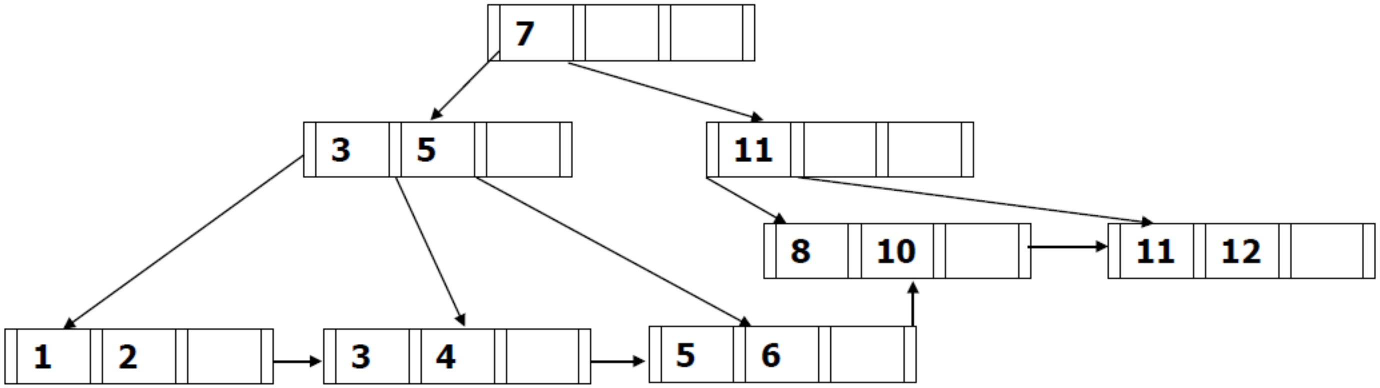


Remove 9, 7, 8

- After removing 7

Deletion in B⁺-Trees (Cont.)

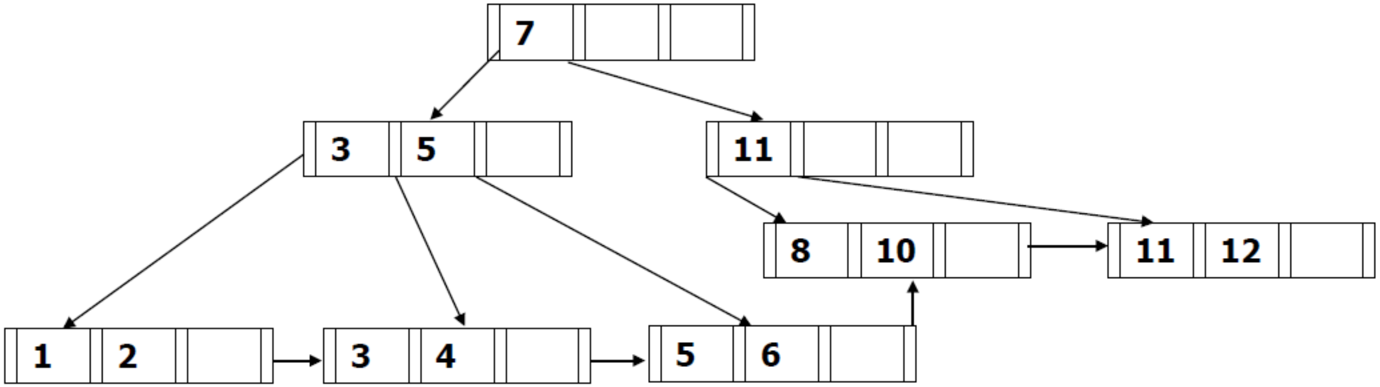
Remove 9, 7, 8



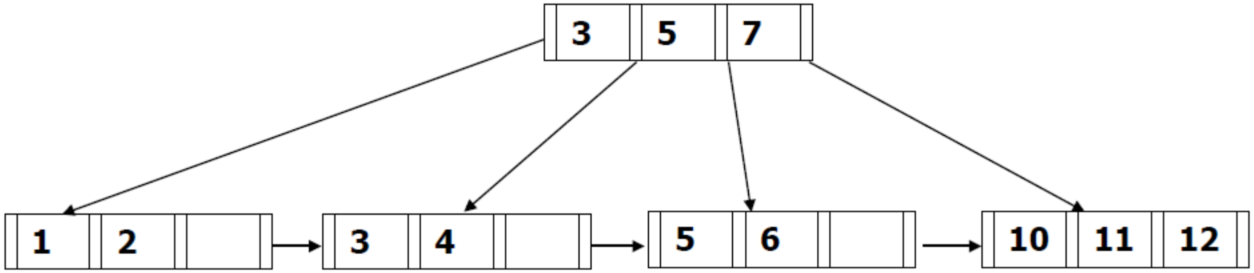
- After removing 8

Deletion in B+-Trees (Cont.)

Remove 9, 7, 8



- After removing 8



28

Static Hashing

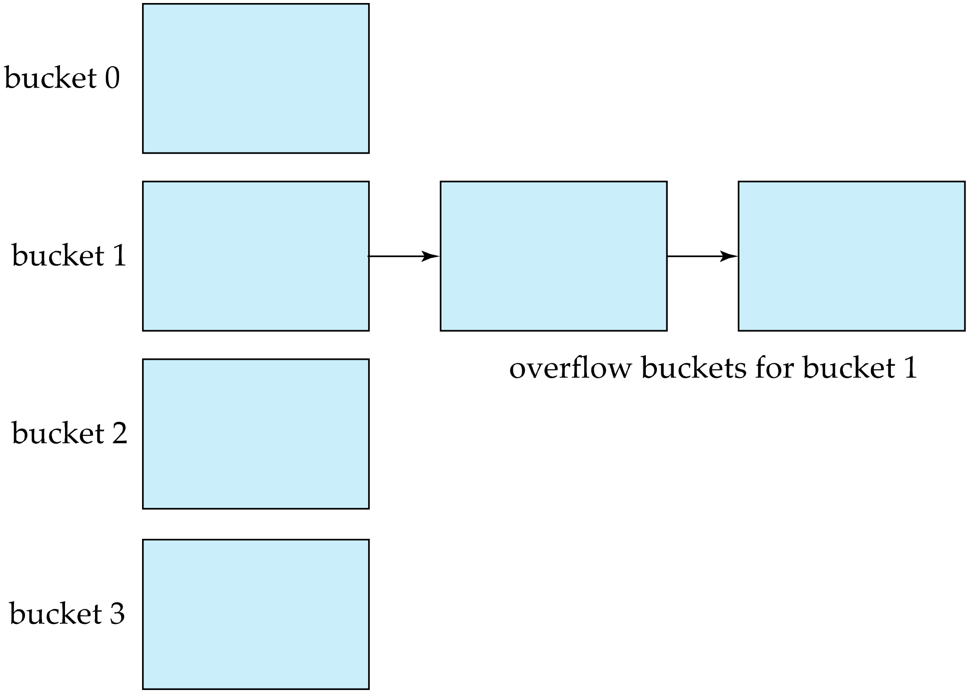
- A **bucket** is a unit of storage containing one or more entries (a bucket is typically a disk block).
  - we obtain the bucket of an entry from its search-key value using a **hash function**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *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

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.)

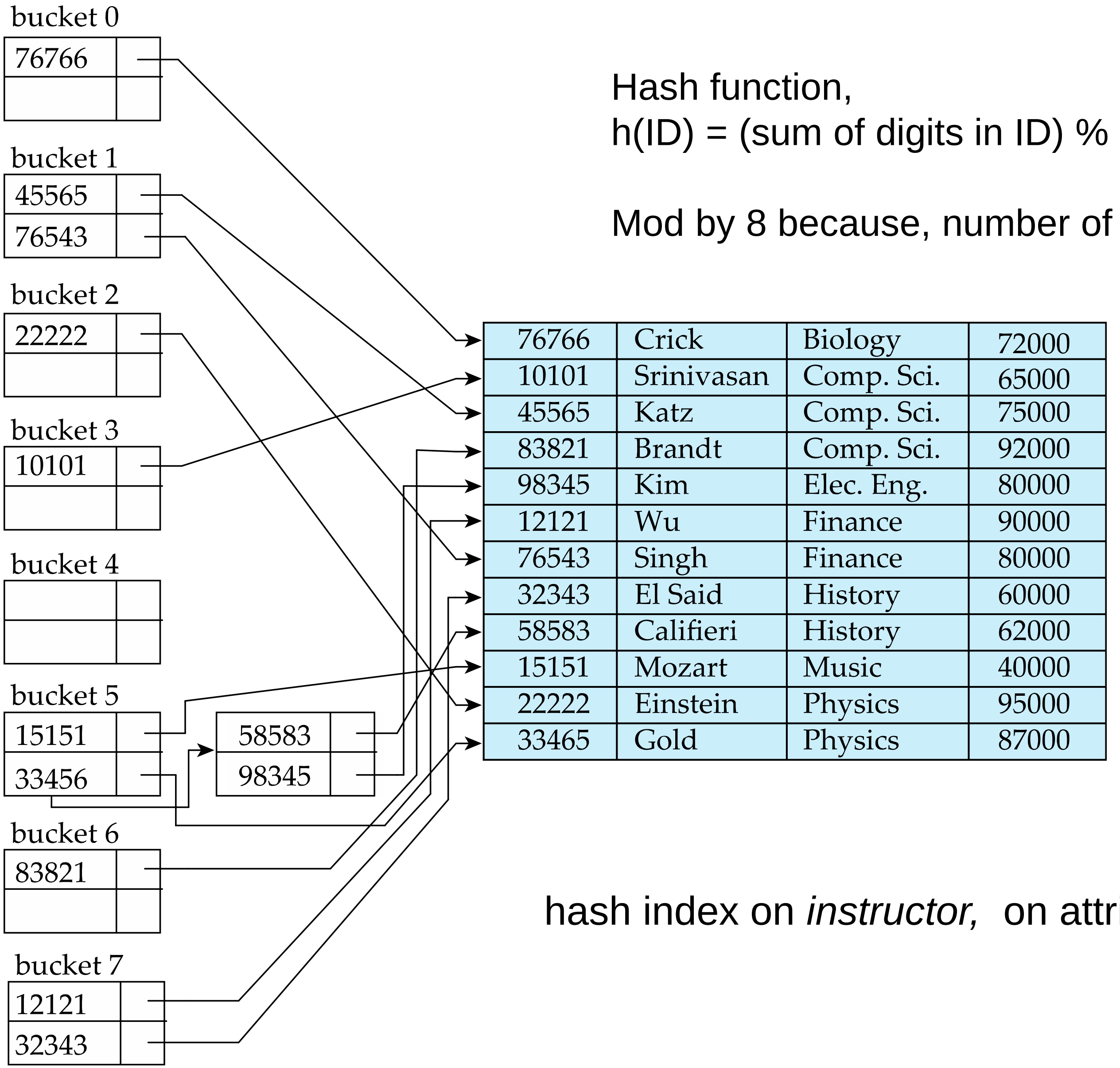
- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed addressing (**also called **closed hashing or open hashing** depending on the book you use**)**
  - An alternative, called **open addressing (**also called **open hashing** or **closed hashing** depending on the book you use) which does not use overflow buckets, is not suitable for database applications.



Example of Hash Index

Hash function,
h(ID) = (sum of digits in ID) % 8

Mod by 8 because, number of hash buckets is 8.



76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

hash index on *instructor,* on attribute *ID*