



Chapter 11: Indexing and Hashing

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Chapter 12: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B⁺-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access



Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author 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



- 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. E.g.,
 - records with a specified value in the attribute
 - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead



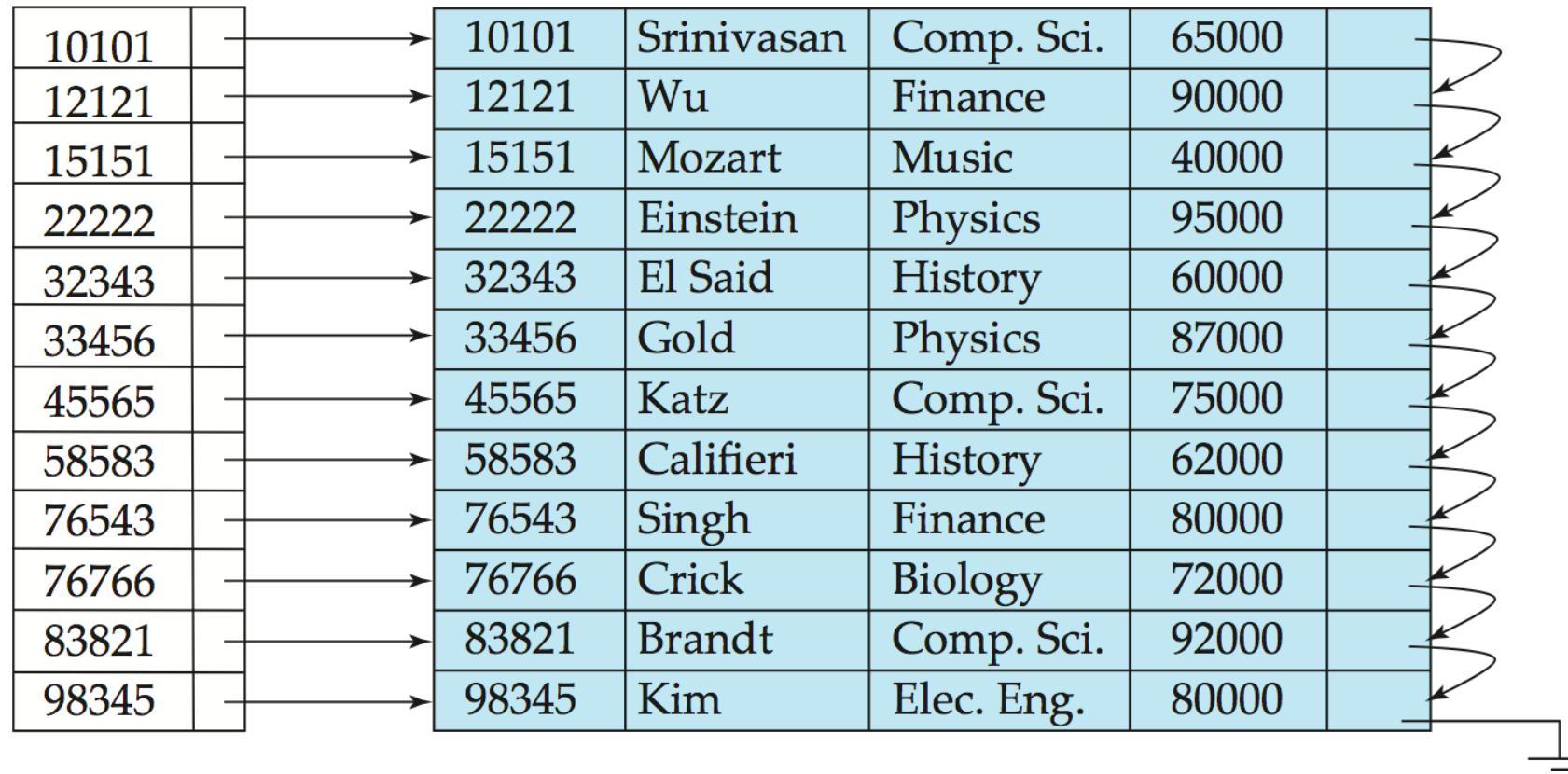
Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
 - Also called **clustering index**
 - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called **non-clustering index**.
- **Index-sequential file**: ordered sequential file with a primary index.



Dense Index Files

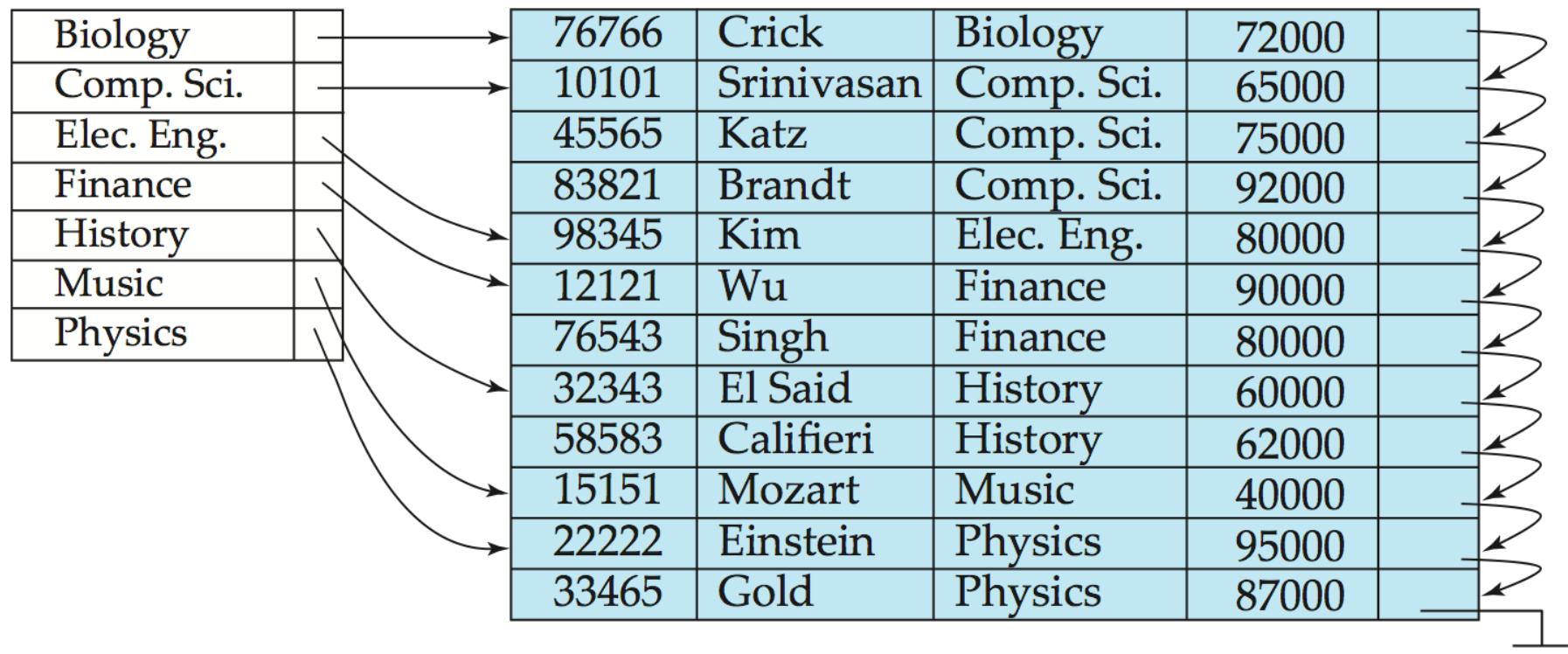
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation





Dense Index Files (Cont.)

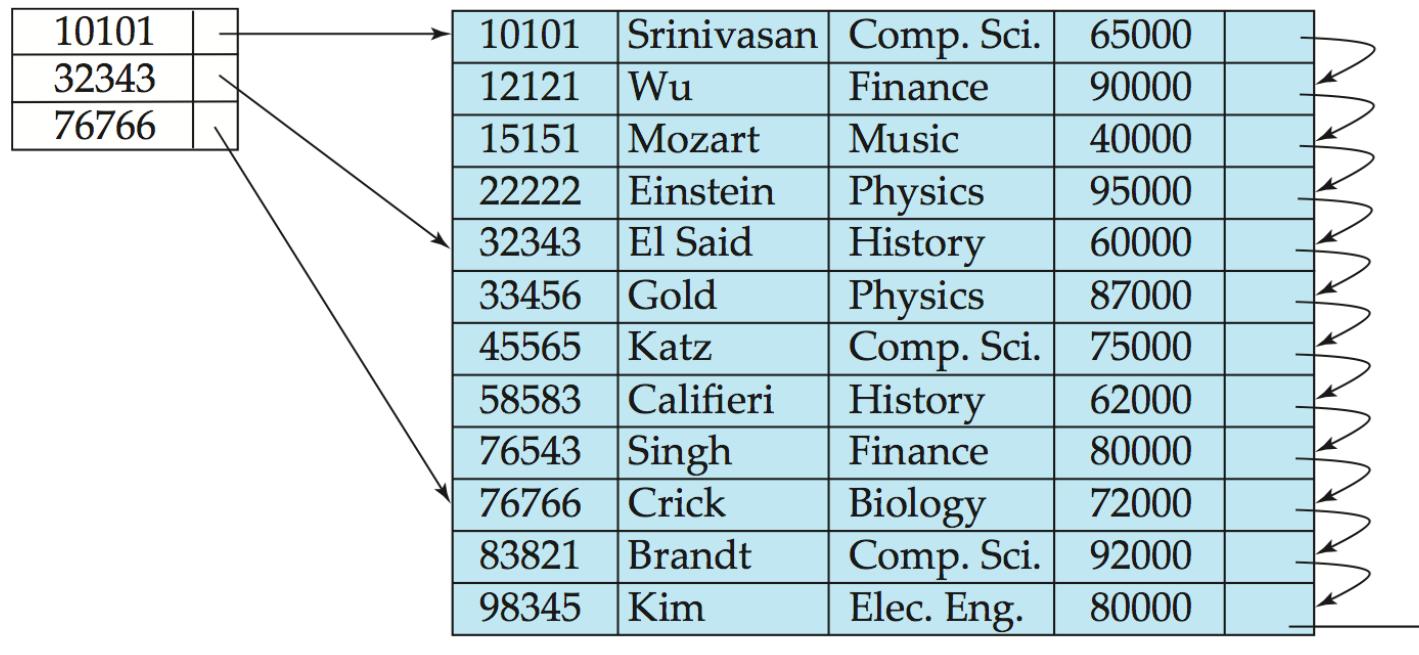
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*





Sparse Index Files

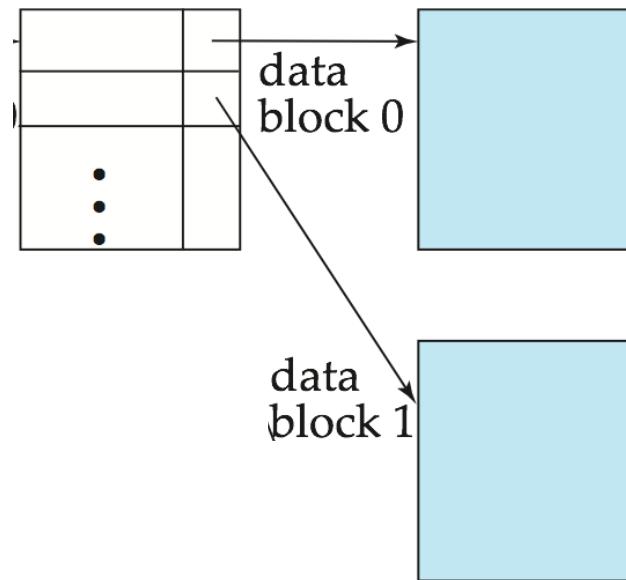
- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to 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:** sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



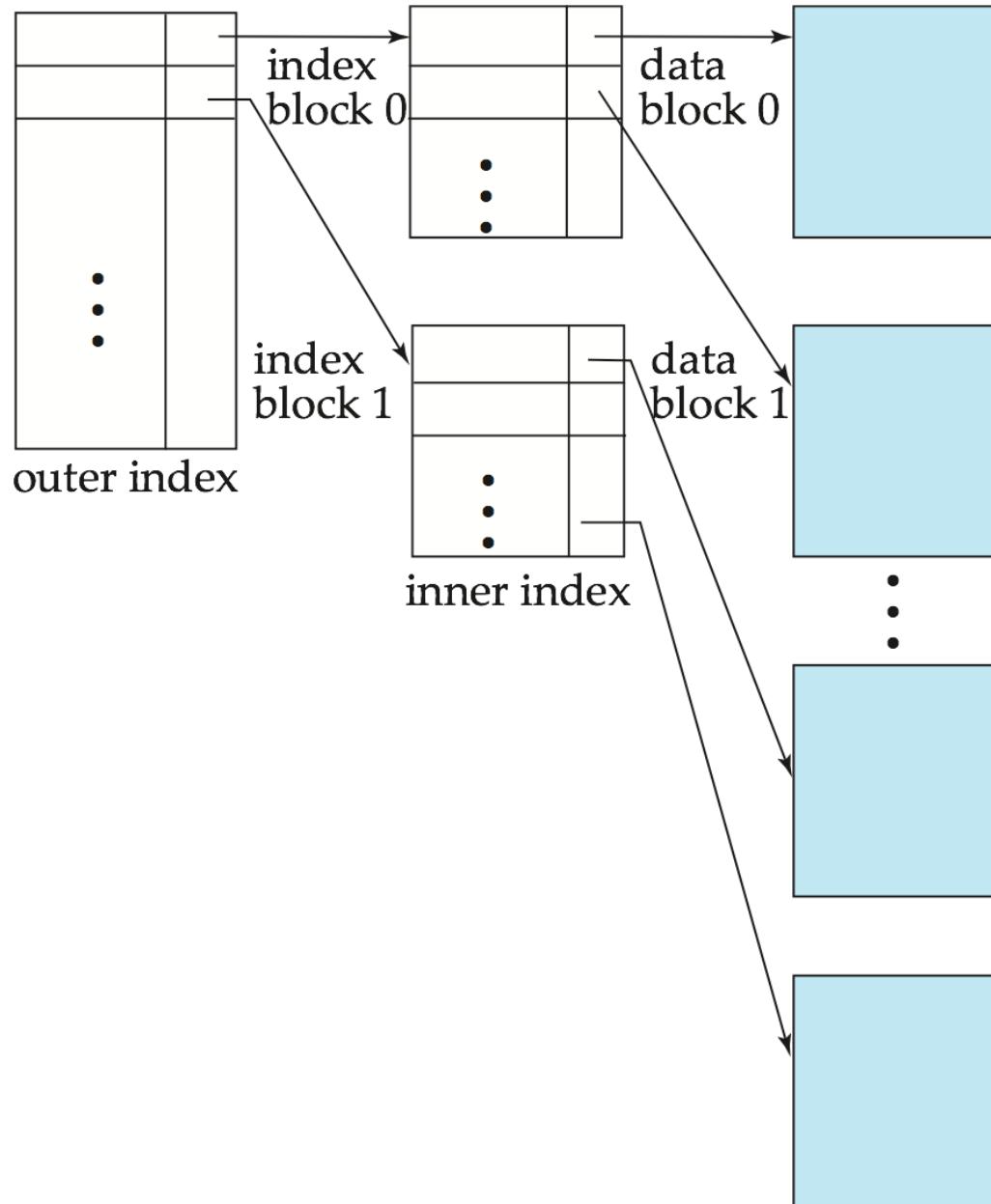


Multilevel Index

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

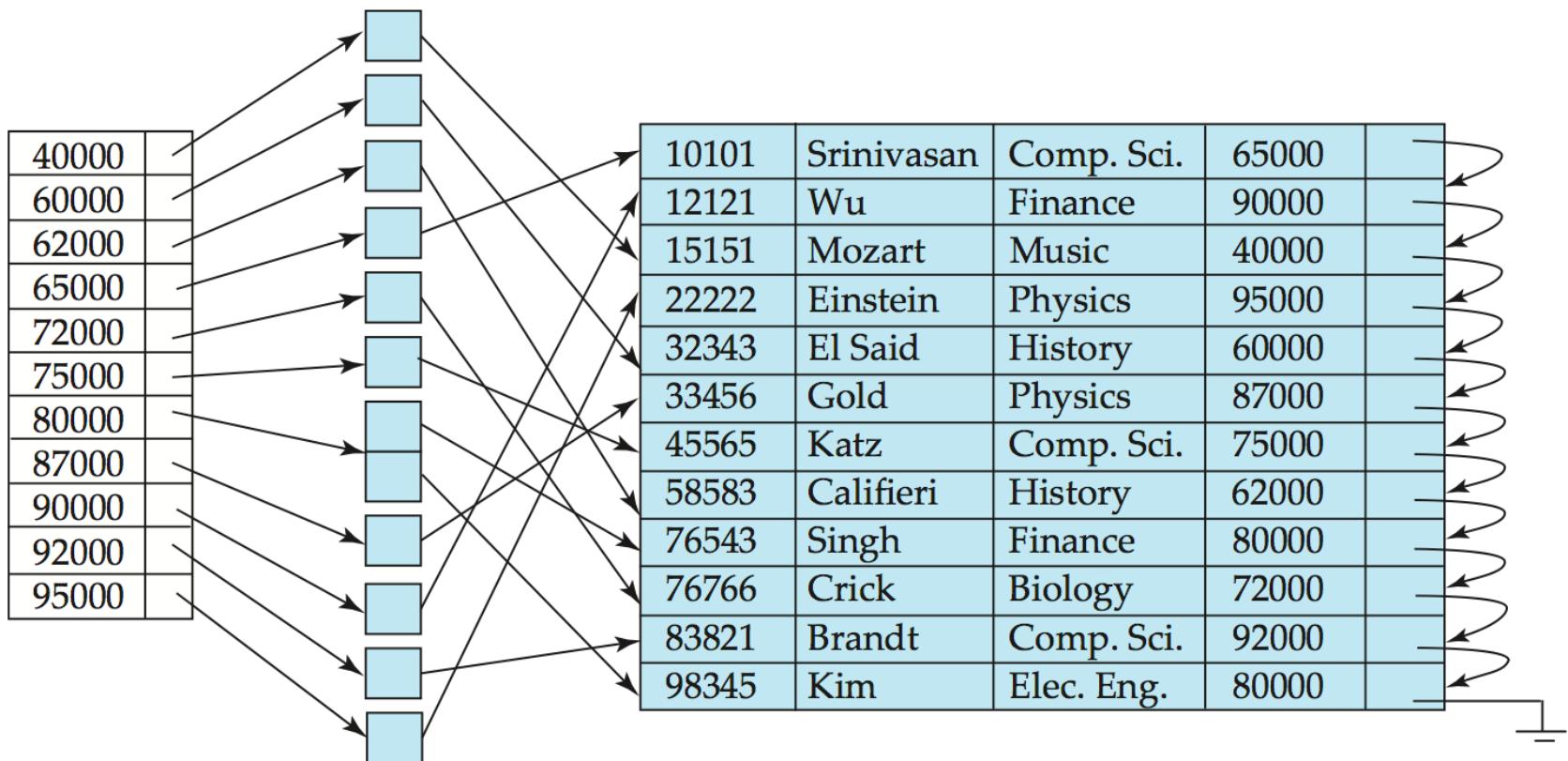


Multilevel Index (Cont.)





Secondary Indices Example



Secondary 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.
- Secondary indices have to be dense



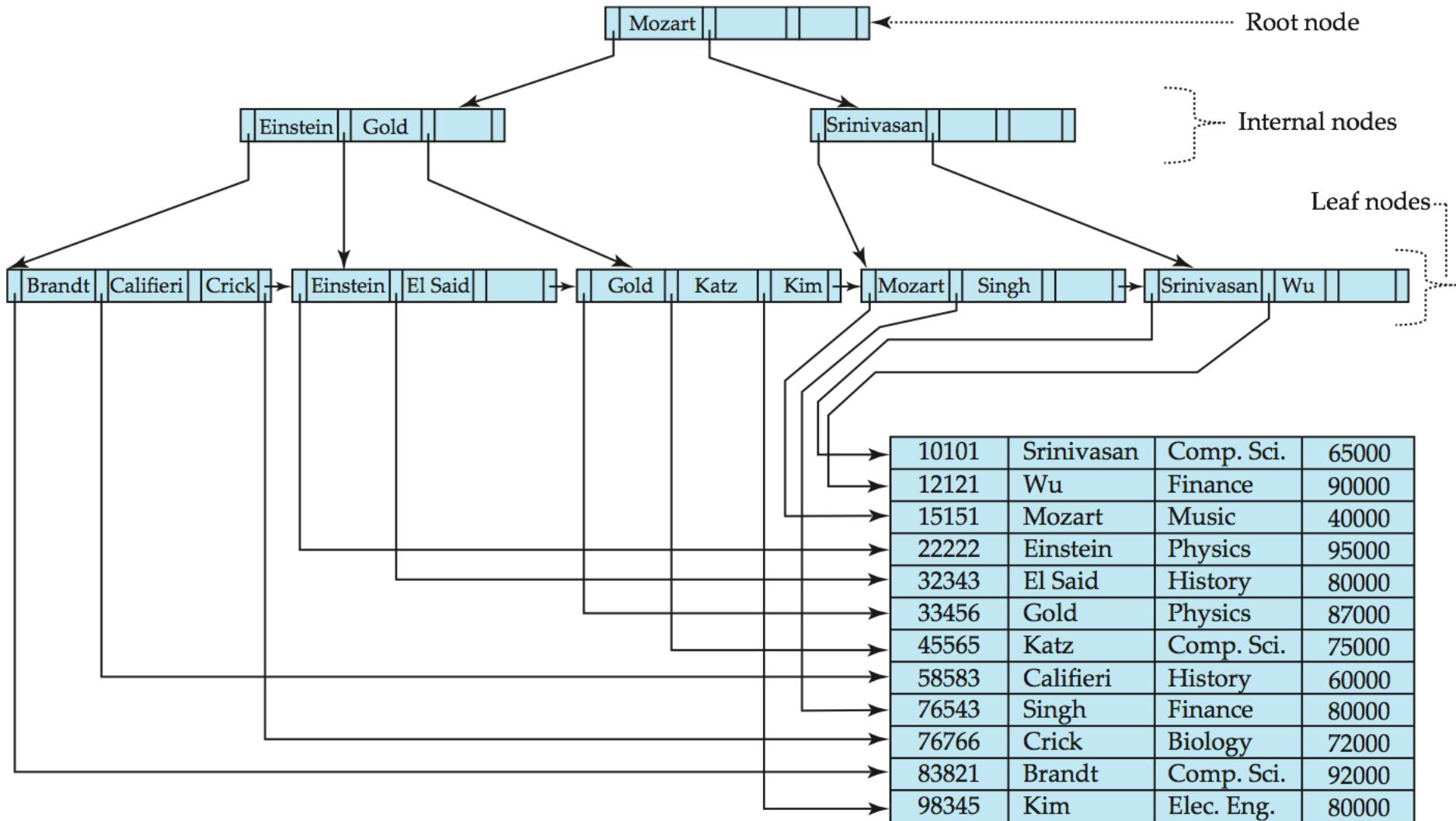
B⁺-Tree Index Files

B⁺-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
 - performance degrades as file grows, since many overflow blocks get created.
 - Periodic reorganization of entire file is required.
- Advantage of B⁺-tree index files:
 - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
 - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B⁺-trees:
 - extra insertion and deletion overhead, space overhead.
- Advantages of B⁺-trees outweigh disadvantages
 - B⁺-trees are used extensively



Example of B+-Tree





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

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

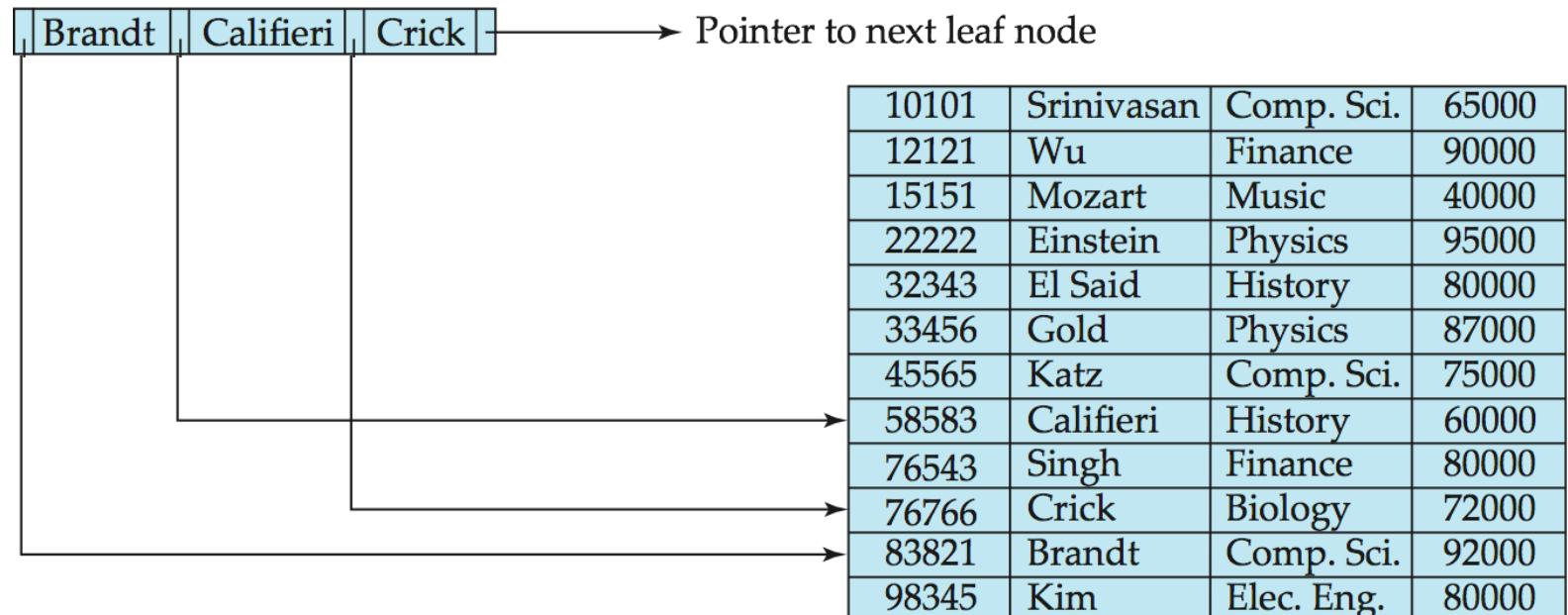
(Initially assume no duplicate keys, address duplicates later)



Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \dots, n-1$, 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
leaf node





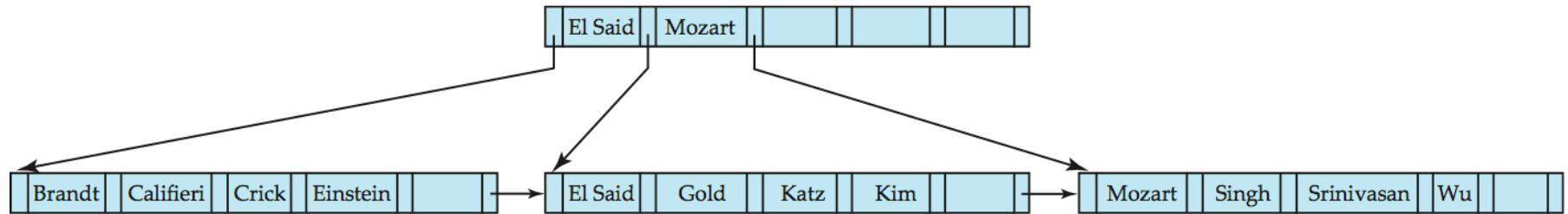
Non-Leaf Nodes in B⁺-Trees

- 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}

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

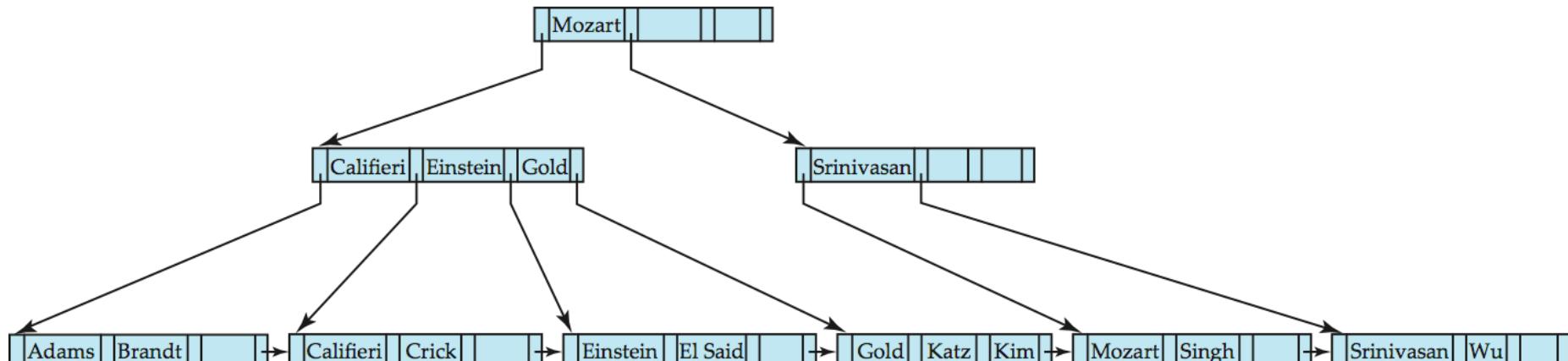
- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B+-tree form a hierarchy of sparse indices.
- The B+-tree contains a relatively small number of levels
 - ▶ Level below root has at least $2 * \lceil n/2 \rceil$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

- Find record with search-key value V .

1. $C = \text{root}$
2. While C is not a leaf node {
 1. Let i be least value s.t. $V \leq K_i$.
 2. If no such exists, set $C = \text{last non-null pointer in } C$
 3. Else { if ($V = K_i$) Set $C = P_{i+1}$ else set $C = P_i$ }
4. Let i be least value s.t. $K_i = V$
5. If there is such a value i , follow pointer P_i to the desired record.
6. Else no record with search-key value k exists.





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.
- 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



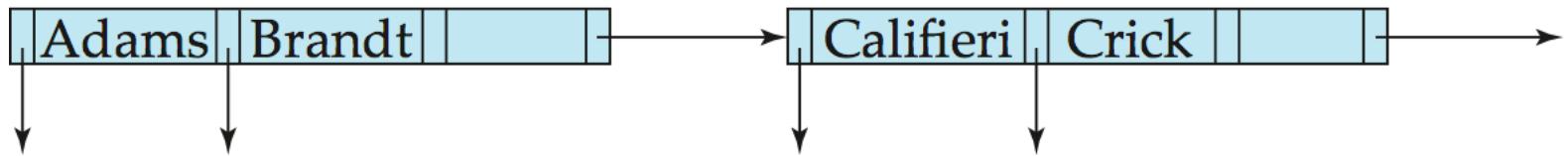
Updates on B+-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
 1. Add record to the file
 2. If necessary add a pointer to the bucket.
3. If the search-key value is not present, then
 1. add the record to the main file (and create a bucket if necessary)
 2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
 3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.



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

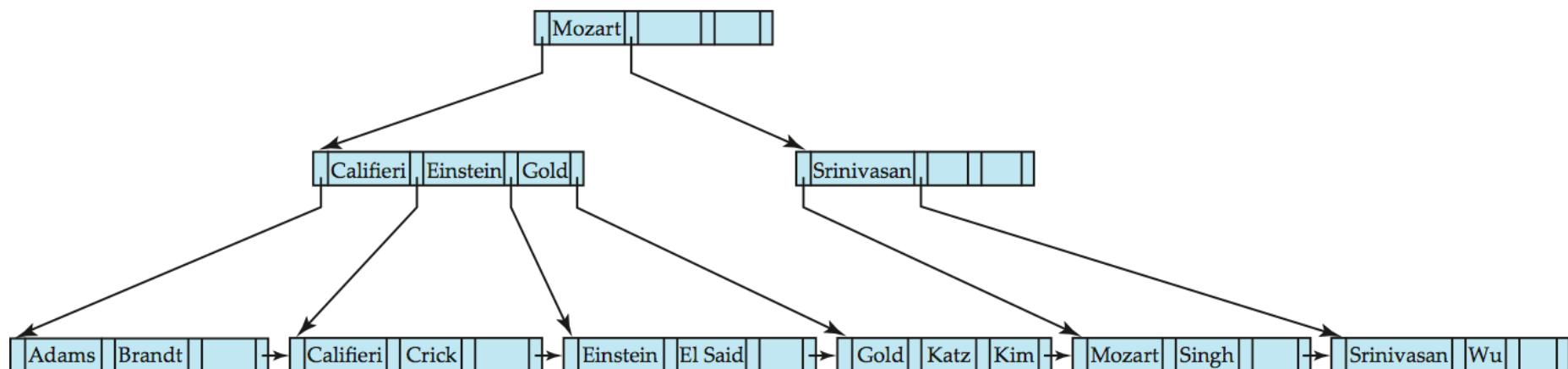
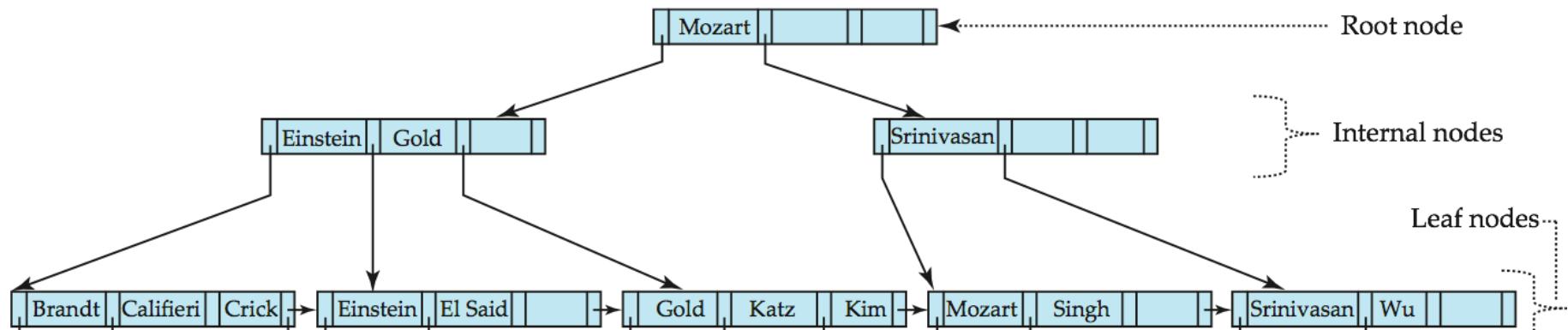
- Splitting a leaf node:
 - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
 - let the new node be p , and let k be the least key value in p . Insert (k,p) in the parent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



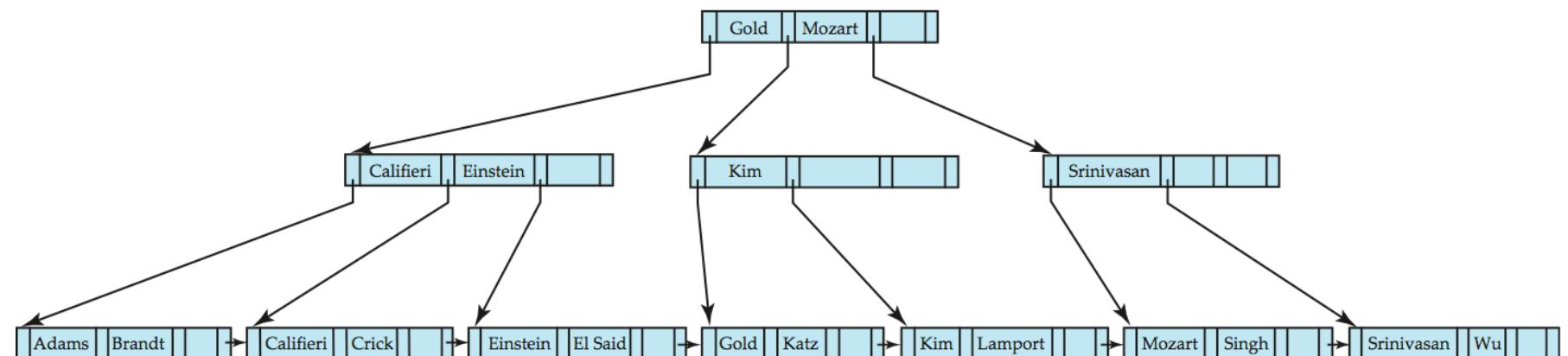
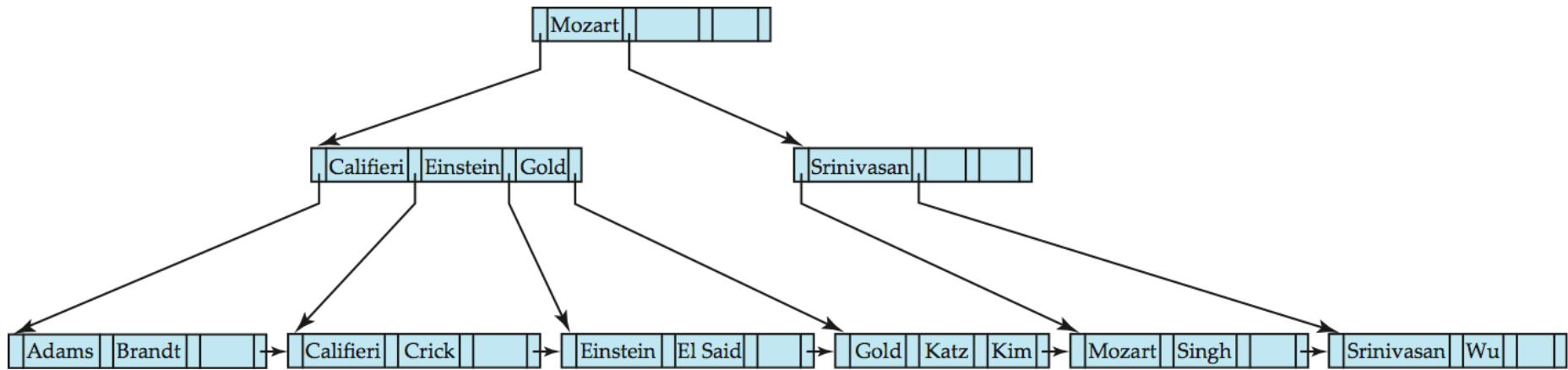
B⁺-Tree Insertion



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



B⁺-Tree Insertion

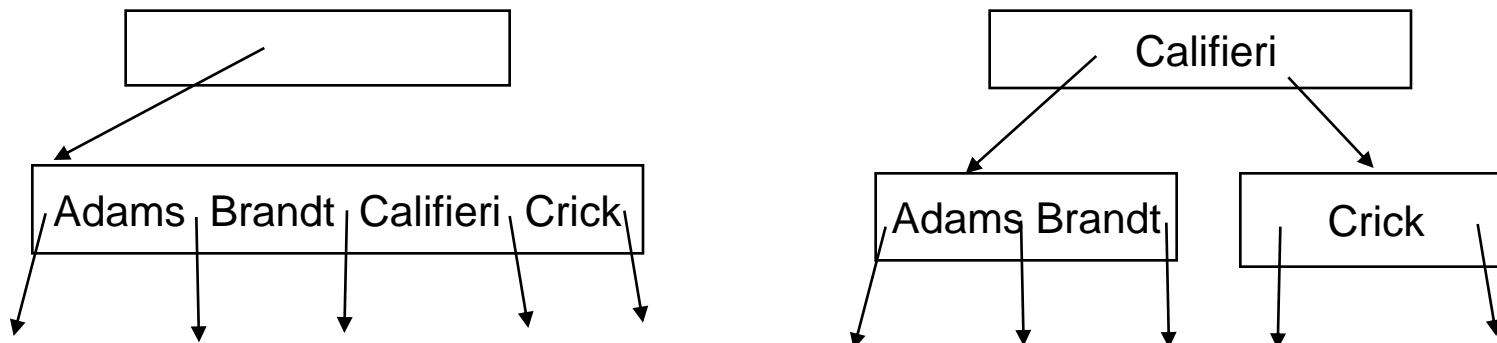


B⁺-Tree before and after insertion of "Lamport"



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 $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
 - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \dots, K_n, P_{n+1}$ from M into newly allocated node N'
 - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N
- **Read pseudocode in book!**





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 ***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.

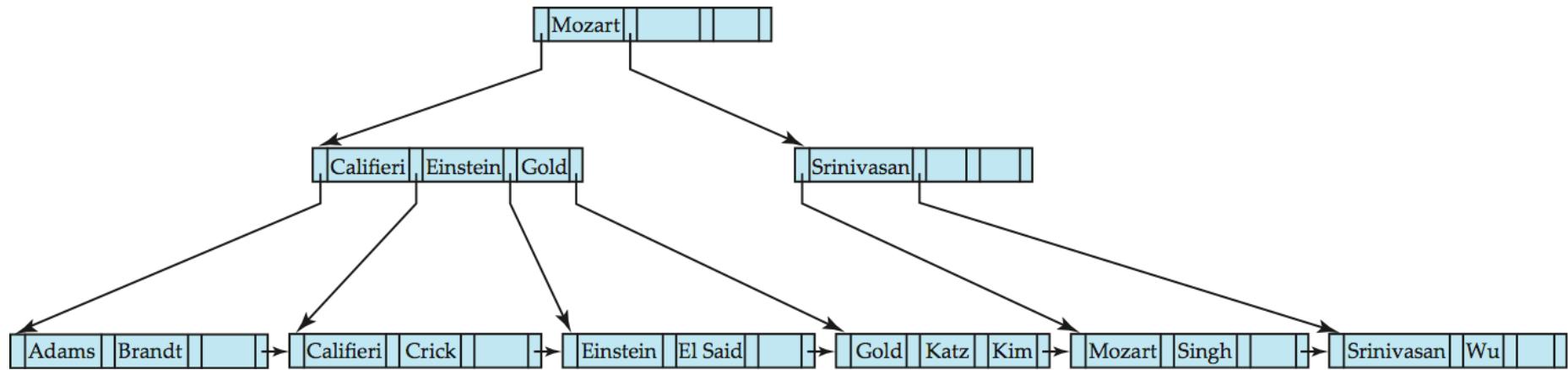


Updates on B+-Trees: Deletion

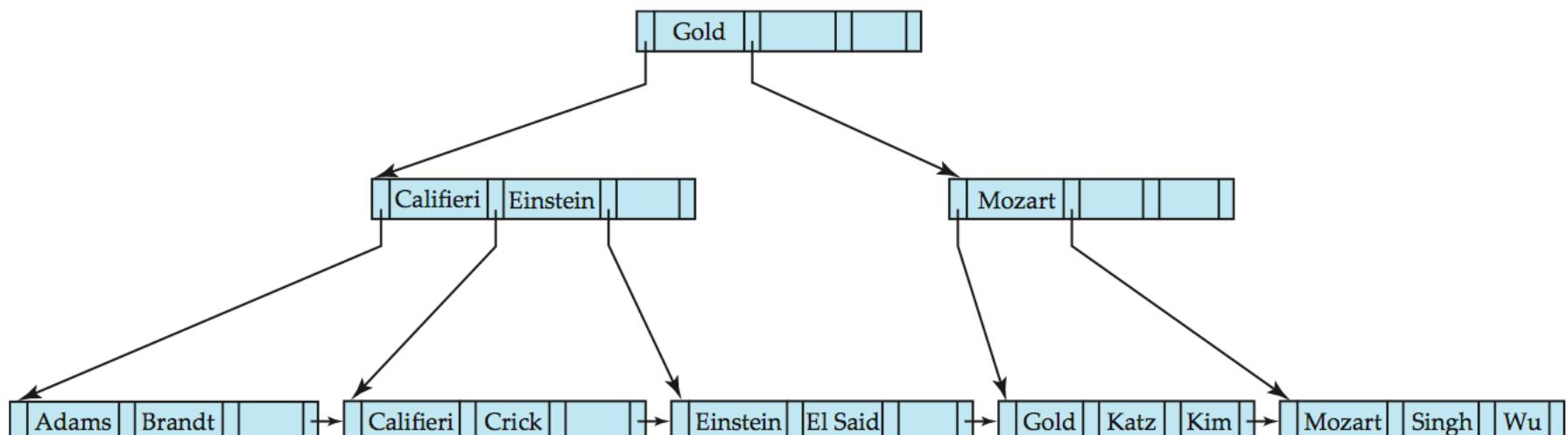
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
 - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
 - Update 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



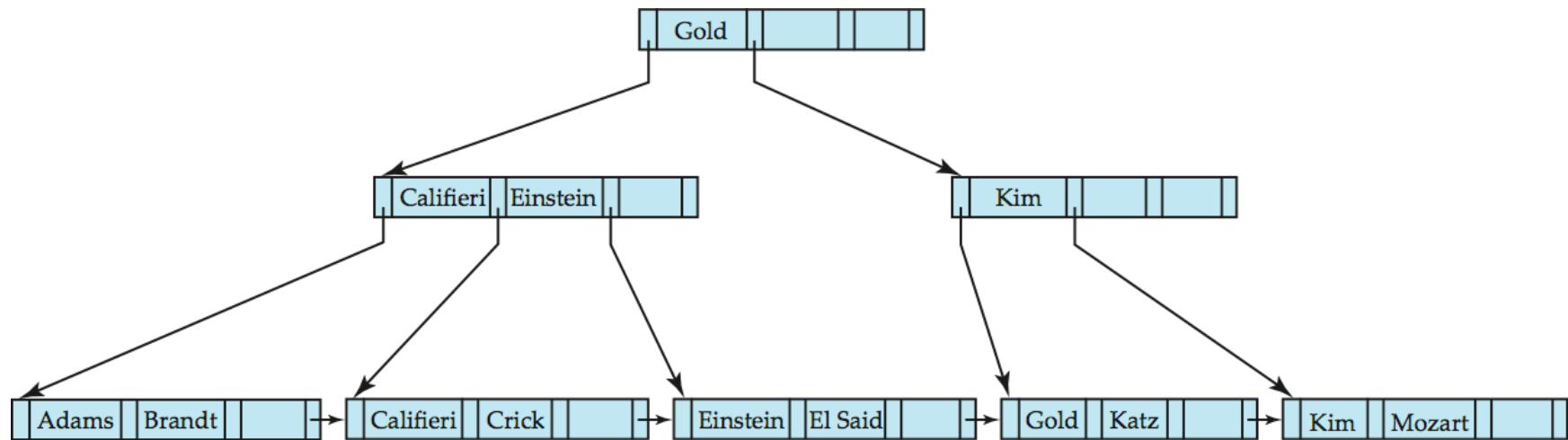
Before and after deleting “Srinivasan”



- Deleting “Srinivasan” causes merging of under-full leaves



Examples of B+-Tree Deletion (Cont.)

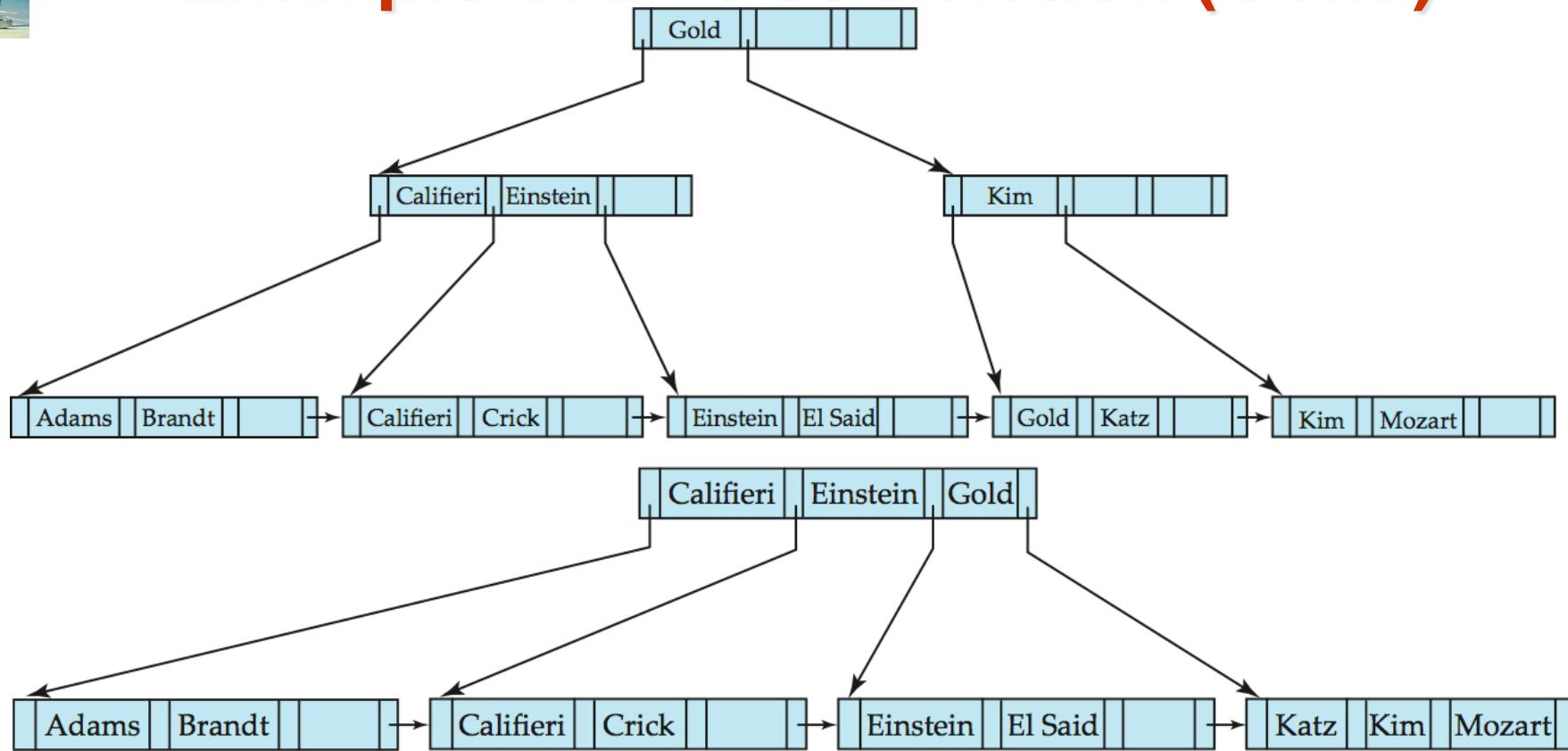


Deletion of “Singh” and “Wu” from result of previous example

- 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



Example of B+-tree Deletion (Cont.)



Before and after deletion of “Gold” from earlier example

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

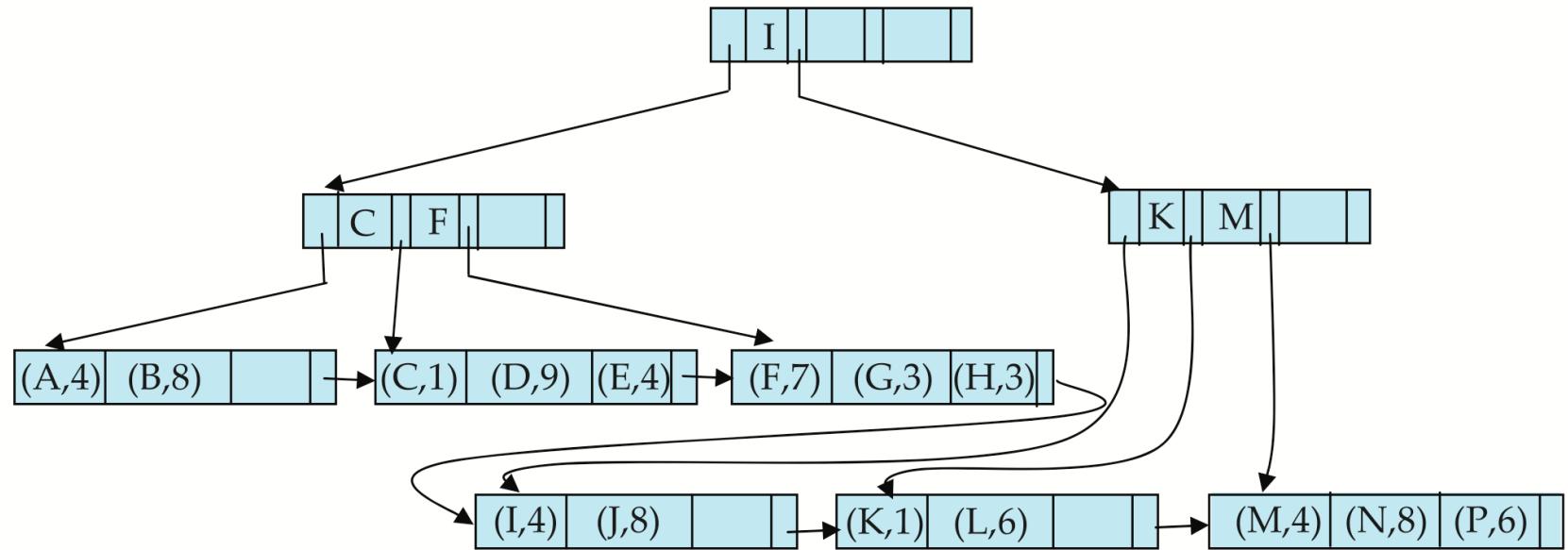


B⁺-Tree File Organization

- Index file degradation problem is solved by using B⁺-Tree indices.
- Data file degradation problem is solved by using B⁺-Tree File Organization.
- The leaf nodes in a B⁺-tree file organization store records, instead of pointers.
- 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 nonleaf 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.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
 - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries



Other Issues in Indexing

■ Record relocation and secondary indices

- If a record moves, all secondary indices that store record pointers have to be updated
- Node splits in B⁺-tree file organizations become very expensive
- *Solution:* use primary-index search key instead of record pointer in secondary index
 - ▶ Extra traversal of primary index to locate record
 - Higher cost for queries, but node splits are cheap
 - ▶ Add record-id if primary-index search key is non-unique



Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B⁺-tree requires ≥ 1 IO per entry
 - assuming leaf level does not fit in memory
 - can be very inefficient for loading a large number of entries at a time
(bulk loading)
- Efficient alternative 1:
 - sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4)
 - insert in sorted order
 - ▶ insertion will go to existing page (or cause a split)
 - ▶ much improved IO performance, but most leaf nodes half full
- Efficient alternative 2: **Bottom-up B⁺-tree construction**
 - As before sort entries
 - And then create tree layer-by-layer, starting with leaf level
 - ▶ details as an exercise
 - Implemented as part of bulk-load utility by most database systems



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** $balance = 80000$

- May fetch many records that satisfy the first but not the second condition



Index Definition in SQL

- Create an index

```
create index <index-name> on <relation-name>  
      (<attribute-list>)
```

E.g.: **create index b-index on branch(branch_name)**

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.

- Not really required if SQL **unique** integrity constraint is supported

- To drop an index

```
drop index <index-name>
```

- Most database systems allow specification of type of index, and clustering.



Other Kinds of Indices

- Bitmap indices
 - Used for attributes with very few distinct values
 - Primary use: answering some aggregate queries very efficiently
- Hash indices
 - Used widely for in-memory indices, but not much for on-disk indices
- R-trees
 - Used to index geographical and other multi-dimensional data
 - E.g. find all shops within specified region (rectangle or circle)
 - ▶ Cannot use B+-tree to efficiently answer such queries



End of Chapter

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Handling Duplicate Search Keys

- With duplicate search keys
 - In both leaf and internal nodes,
 - ▶ we cannot guarantee that $K_1 < K_2 < K_3 < \dots < K_{n-1}$
 - ▶ but can guarantee $K_1 \leq K_2 \leq K_3 \leq \dots \leq K_{n-1}$
 - Search-keys in the subtree to which P_i points
 - ▶ are $\leq K_i$, but not necessarily $< K_i$
 - ▶ To see why, suppose same search key value V is present in two leaf node L_i and L_{i+1} . Then in parent node K_i must be equal to V



Handling Duplicate Search Keys (Cont.)

- We modify find procedure as follows
 - traverse P_i even if $V = K_i$
 - As soon as we reach a leaf node C check if C has only search key values less than V
 - ▶ if so set C = right sibling of C before checking whether C contains V
- Procedure printAll
 - uses modified find procedure to find first occurrence of V
 - Traverse through consecutive leaves to find all occurrences of V

**** Errata note: modified find procedure missing in first printing of 6th edition**



Handling Duplicate Search Keys (Cont.)

- Alternatives to scheme described earlier
 - Buckets on separate block (bad idea)
 - List of tuple pointers with each key
 - ▶ Extra code to handle long lists
 - ▶ Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
 - ▶ Low space overhead, no extra cost for queries
 - Make search key unique by adding a record-identifier
 - ▶ Extra storage overhead for keys
 - ▶ Simpler code for insertion/deletion
 - ▶ Widely used