



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
- Tree Based Indexing
- Hashing (*next week*)



Motivation

- Many queries reference only a small proportion of the records in a file.
 - “Find all instructors in the Physics department”
 - “Find the total number of credits earned by the student with *ID* 22201”
references only a fraction of the *instructor* or *student* records.
- It is inefficient for the system to read every tuple in the
 - *instructor* relation to check if the *dept name* value is “Physics”.
 - *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 that we associate with files, called indices.



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.
 - Does not have to be unique
- 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
- There are two basic kinds of indices:
 - **Ordered indices:** search keys are stored in a sorted order
 - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.



Example

What if we want to return all books published after 1867?

The table might be very expensive to search over row-by-row...

Russian_Novels

BID	Title	Author	Published	Full_text
001	<i>War and Peace</i>	Tolstoy	1869	...
002	<i>Crime and Punishment</i>	Dostoyevsky	1866	...
003	<i>Anna Karenina</i>	Tolstoy	1877	...

```
SELECT *
FROM Russian_Novels
WHERE Published > 1867
```



Example (cont'd)

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

Russian_Novels

BID	Title	Author	Published	Full_text
001	<i>War and Peace</i>	Tolstoy	1869	...
002	<i>Crime and Punishment</i>	Dostoyevsky	1866	...
003	<i>Anna Karenina</i>	Tolstoy	1877	...

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Create Index By_Yr_Index
On Russian_Novels (Published)



Example (cont'd)

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

Russian_Novels

BID	Title	Author	Published	Full_text
001	<i>War and Peace</i>	Tolstoy	1869	...
002	<i>Crime and Punishment</i>	Dostoyevsky	1866	...
003	<i>Anna Karenina</i>	Tolstoy	1877	...

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	002
Tolstoy	Anna Karenina	003
Tolstoy	War and Peace	001

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures...

Create Index By_Author_Title_Index
On Russian_Novels (Author, Title)



Index Evaluation Metrics

- **Access types:** The types of access that are supported efficiently.
 - Records with a specified value in the attribute
 - Records with an attribute value falling in a specified range of values.
- **Access time:** The time to find a particular data item, or set of items, using the technique in question.
- **Insertion time:** The time to insert a new data item.
- **Deletion time:** The time to delete a data item.
- **Space overhead:** The additional space occupied by an index structure.



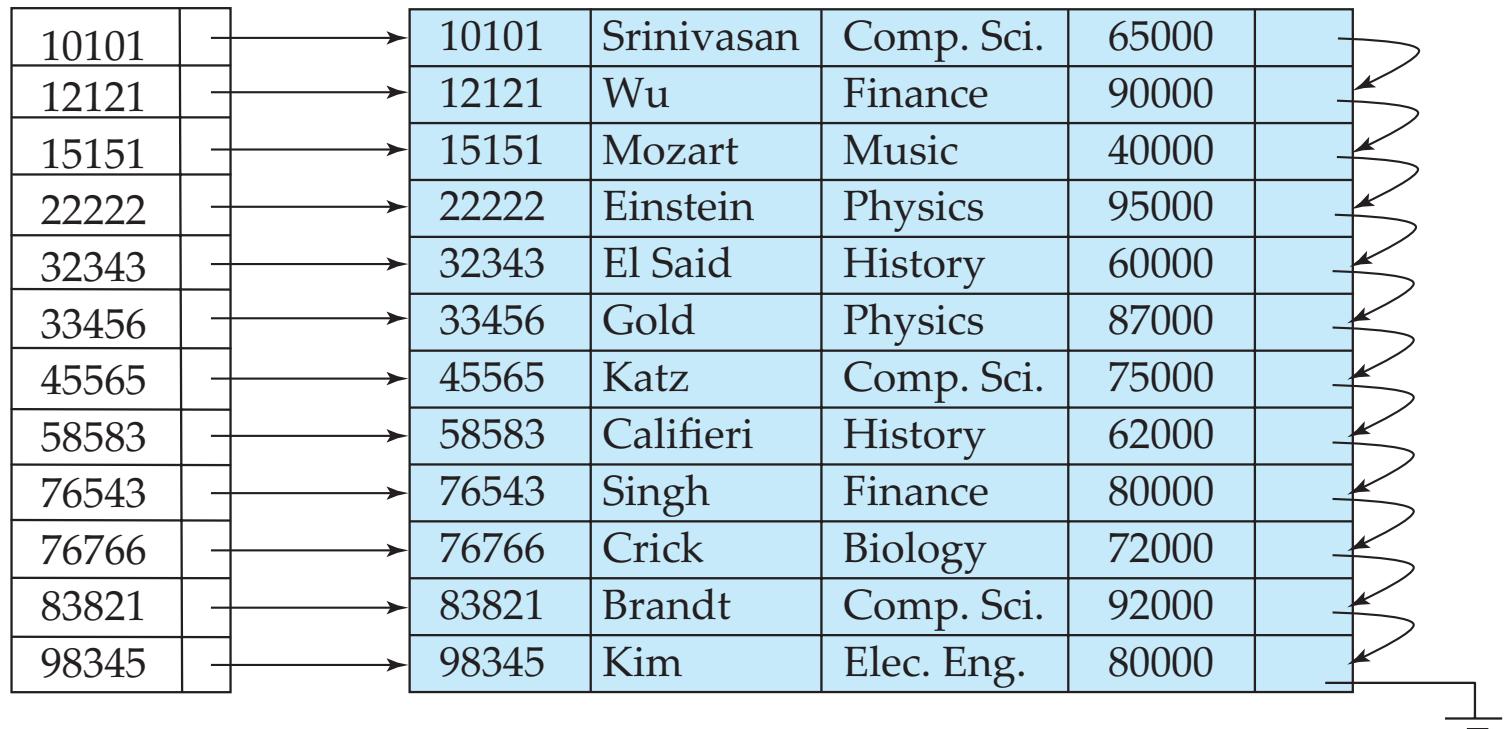
Ordered Indices

- 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

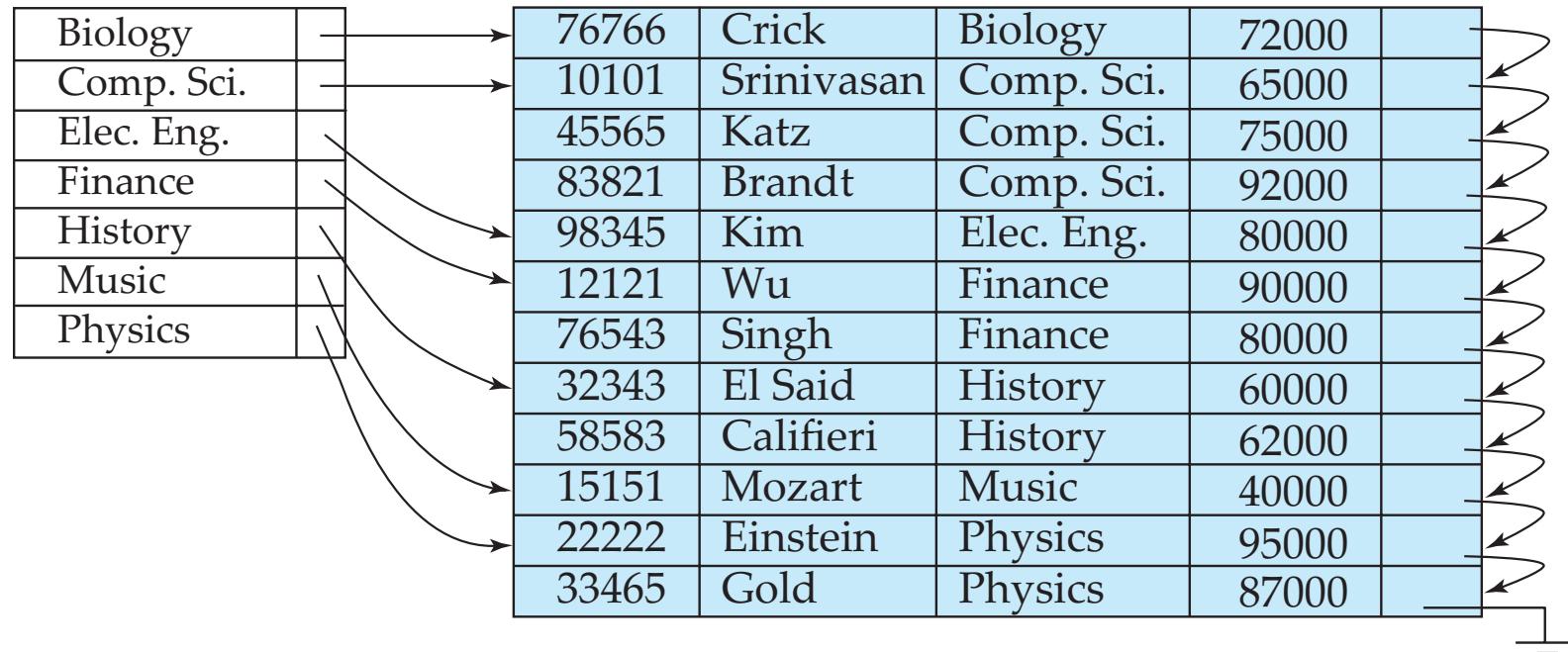
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation





Dense Index Files (Cont.)

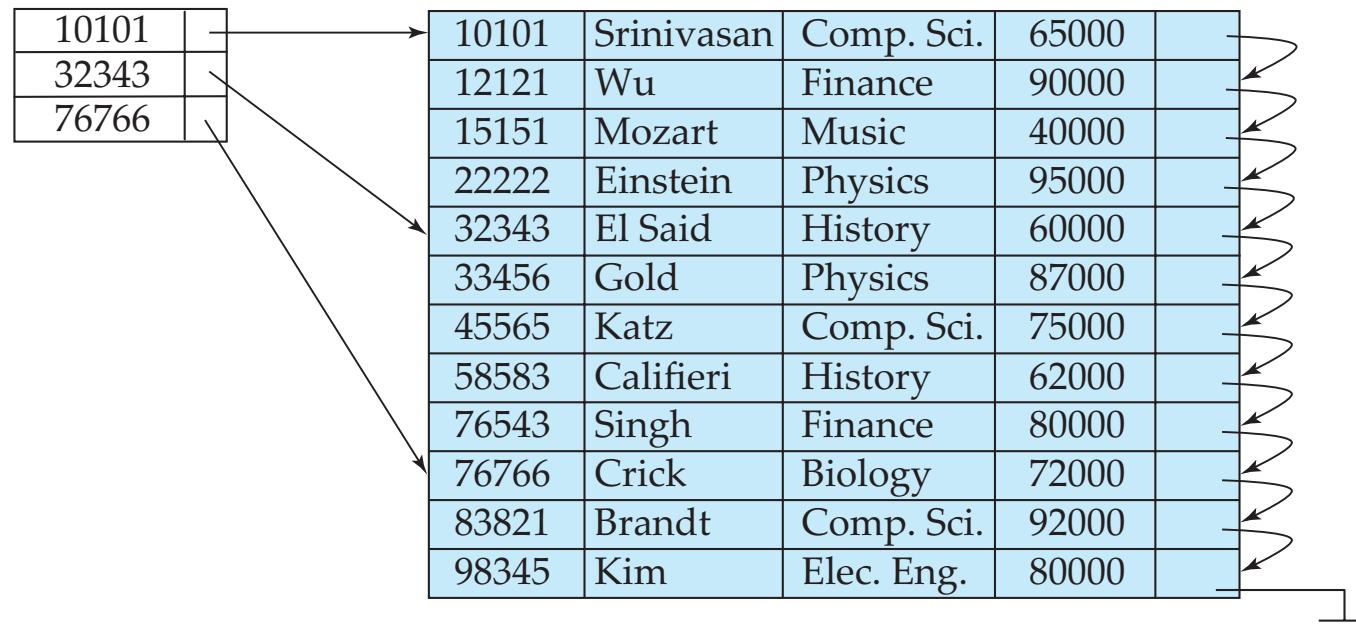
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*





Sparse Index Files

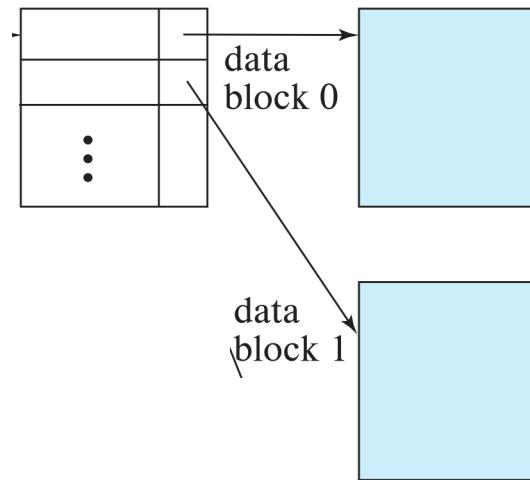
- **Sparse Index:** an index entry appears for only some of the search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K , we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to which the index record points





Sparse Index Files (Cont.)

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



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

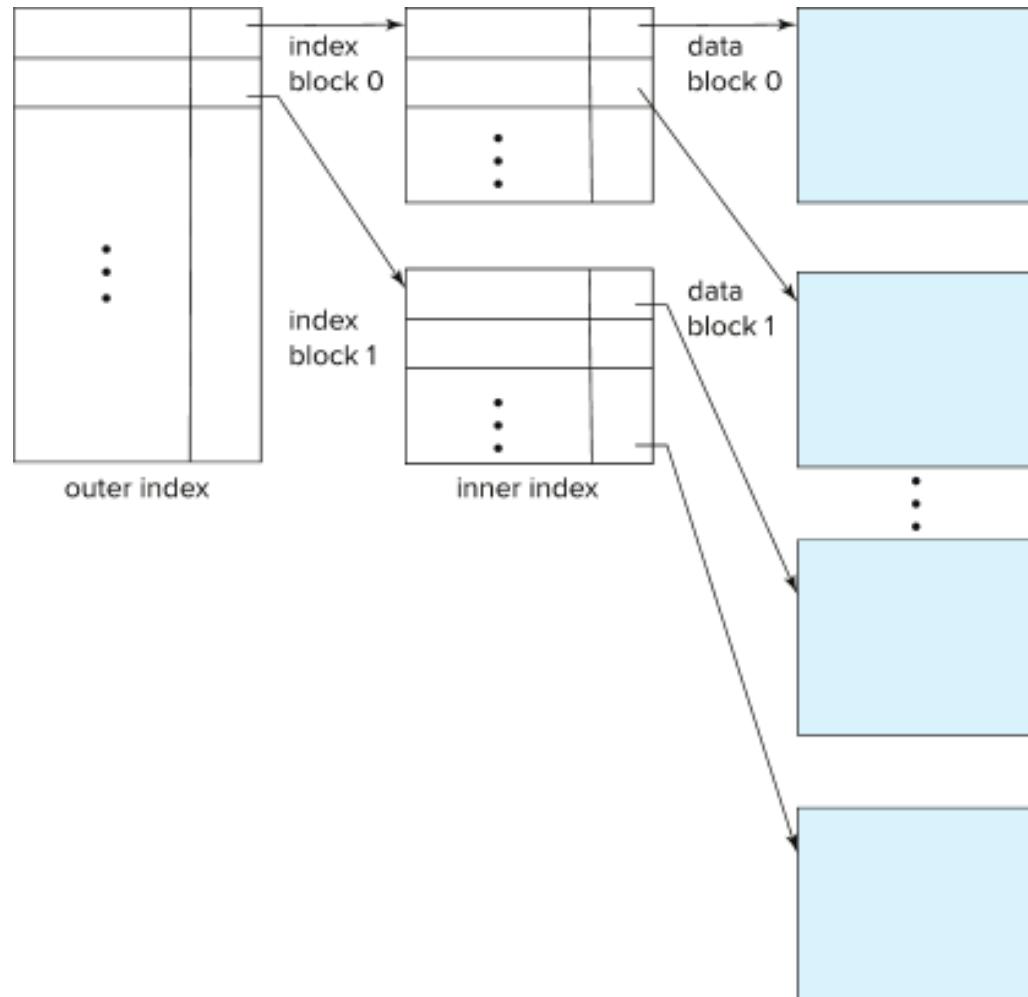


Multilevel Index

- 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.



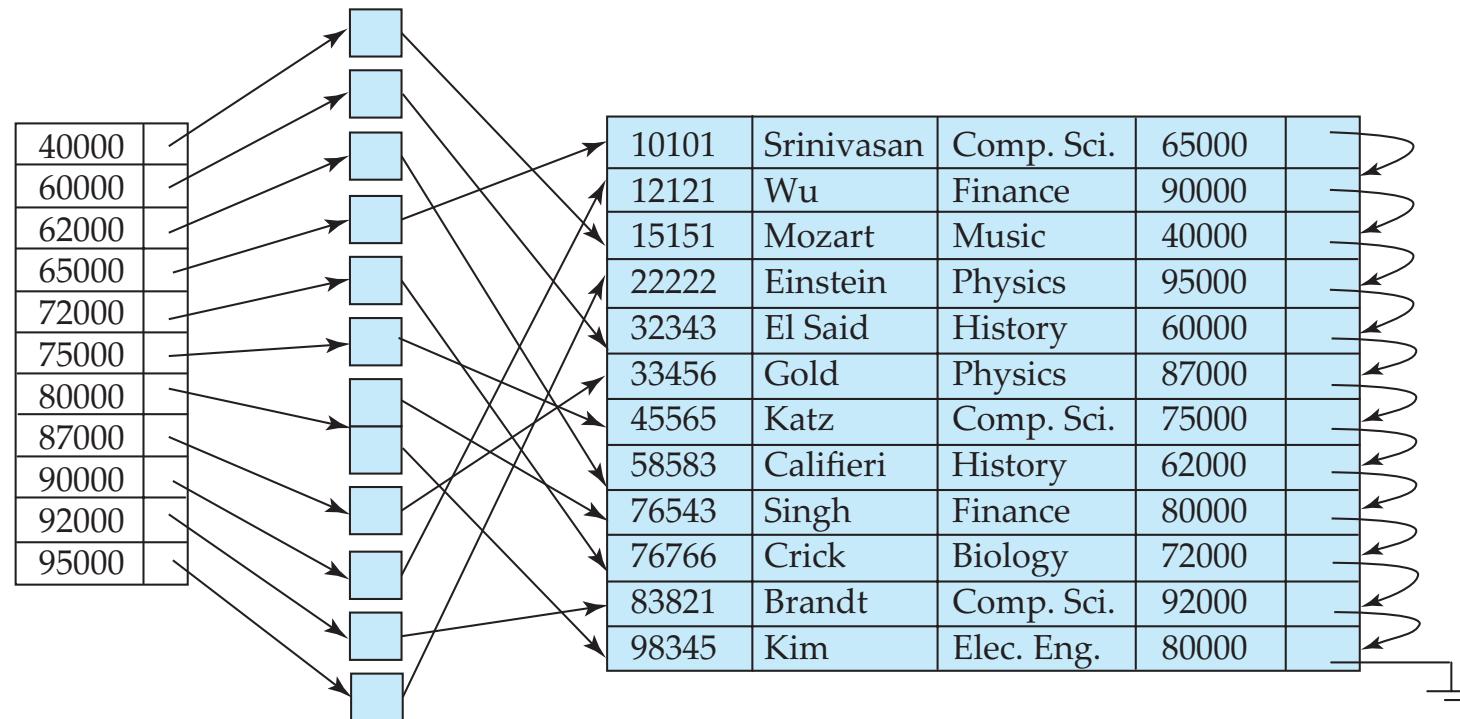
Multilevel Index (Cont.)





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



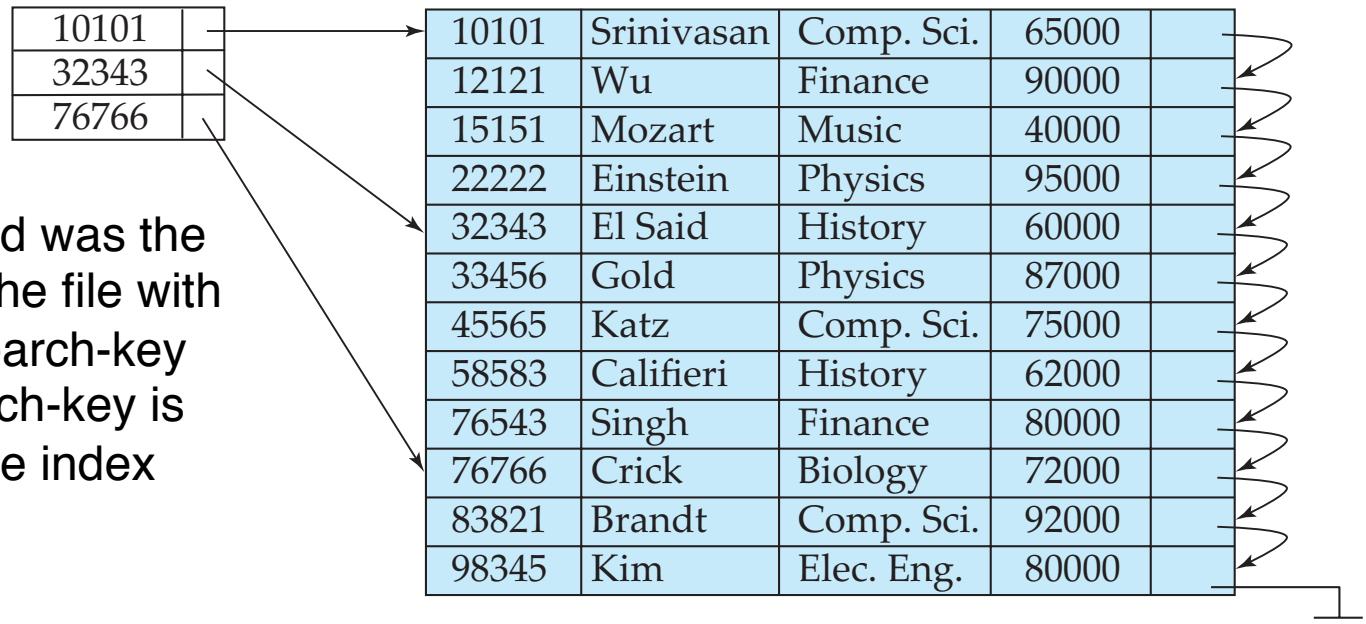
Clustering vs Nonclustering Indices

- Indices offer substantial benefits when searching for records.
- BUT: indices imposes overhead on database modification
 - When a record is inserted or deleted, every index on the relation must be updated
 - When a record is updated, any index on an updated attribute must be updated
- Sequential scan using clustering index is efficient, but a sequential scan using a secondary (nonclustering) index is expensive on magnetic disk
 - Each record access may fetch a new block from disk
 - Each block fetch on magnetic disk requires about 5 to 10 milliseconds



Index Update: Deletion

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



- Single-level index entry deletion:**
 - Dense indices** – deletion of search-key is similar to file record deletion.
 - Sparse indices** –
 - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
 - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.



Index Update: Insertion

- **Single-level index insertion:**
 - Perform a lookup using the search-key value of the record to be inserted.
 - **Dense indices** – if the search-key value does not appear in the index, insert it
 - Indices are maintained as sequential files
 - Need to create space for new entry, overflow blocks may be required
 - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
 - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- **Multilevel insertion and deletion:** algorithms are simple extensions of the single-level algorithms



Indices on Multiple Keys

- **Composite search key** contains more than one attribute
 - E.g., index on *instructor* relation on attributes (*name*, *ID*)
 - Values are sorted lexicographically
 - E.g. (John, 12121) < (John, 13514) and (John, 13514) < (Peter, 11223)
 - Can query on just *name*, or on (*name*, *ID*)

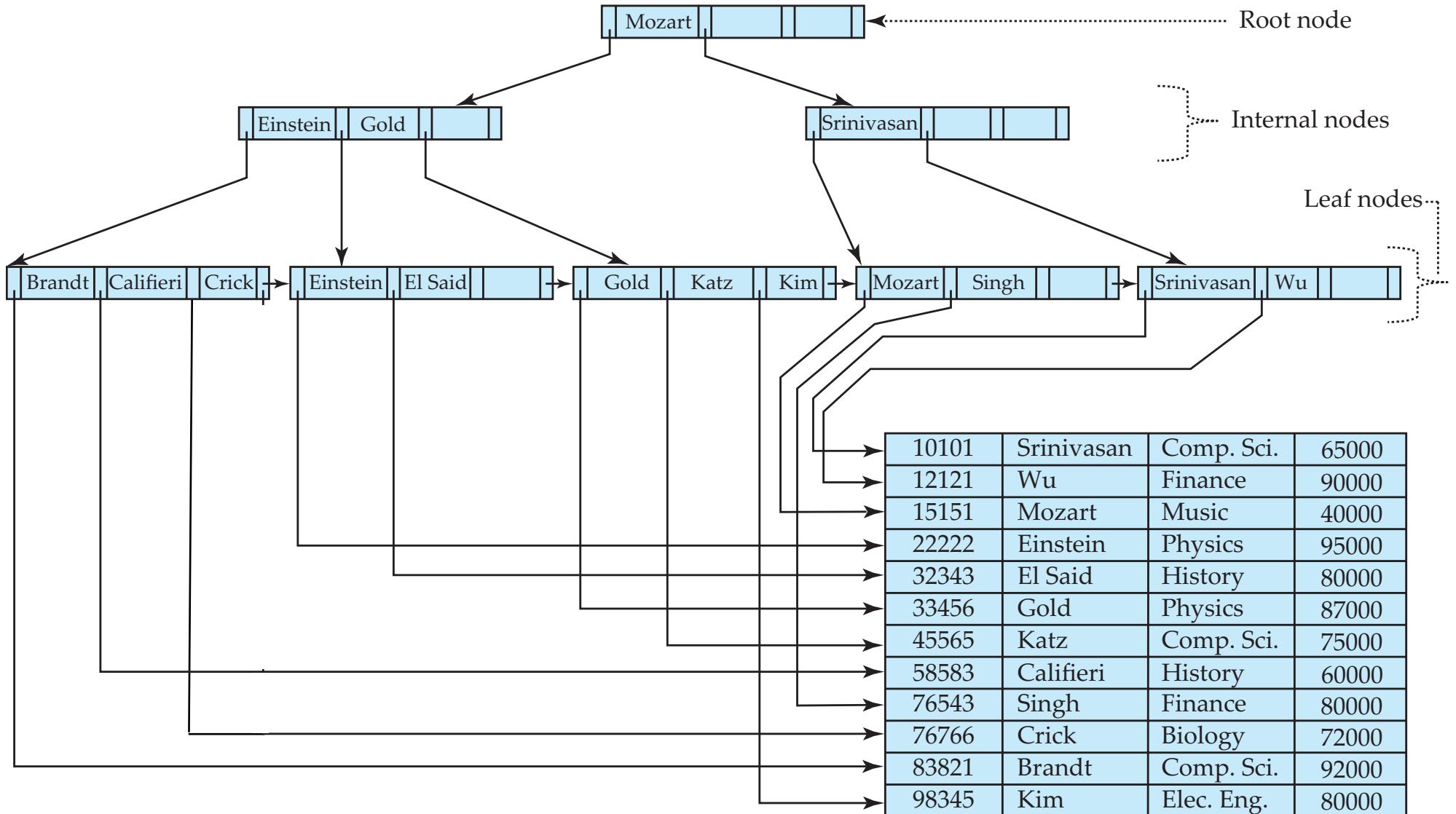


B+-Tree Index Files

- 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
 - takes the form of a **balanced tree**
- 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	\dots	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).
- There are up to $n-1$ search-key values and n pointers.
- The search-keys in a node are ordered

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

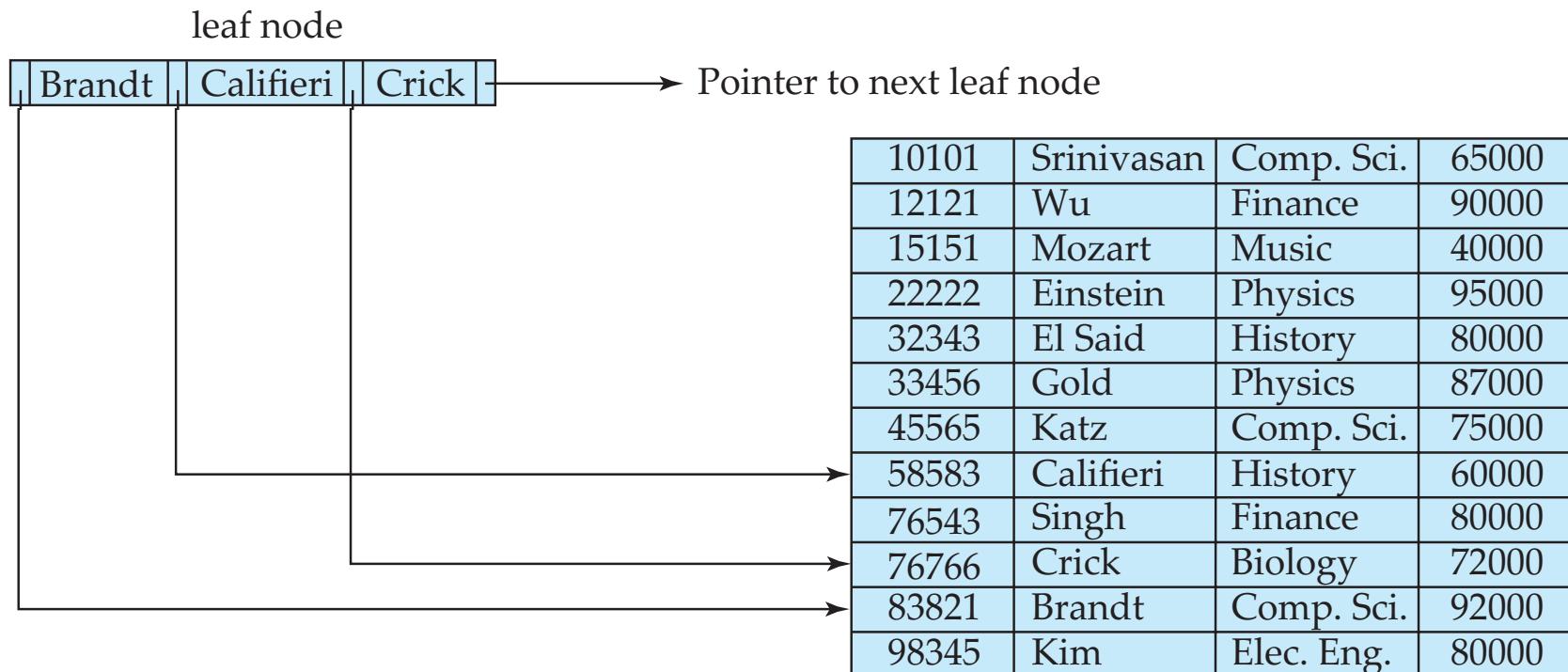
(Initially assume no duplicate keys, address duplicates later)



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





Non-Leaf Nodes in B+-Trees

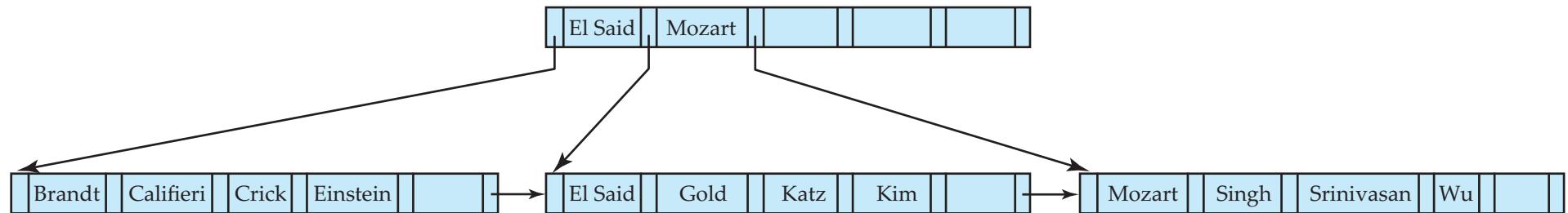
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
 - All the search-keys in the subtree to which P_1 points are less than K_1
 - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to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

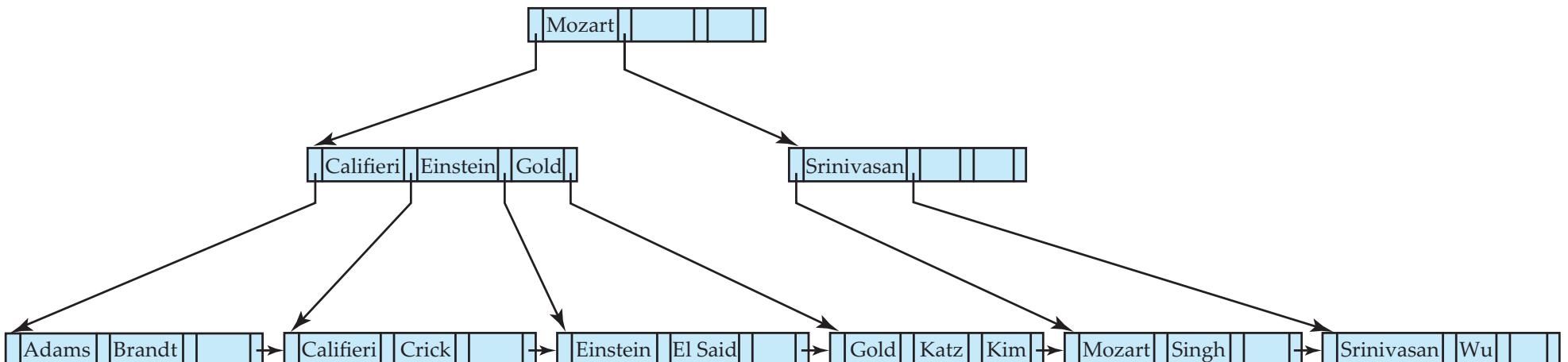
- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B⁺-tree form a hierarchy of sparse indices.
- The B⁺-tree contains a relatively small number of levels
 - Level below root has at least $2 * \lceil n/2 \rceil$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



Queries on B⁺-Trees

function *find(v)*

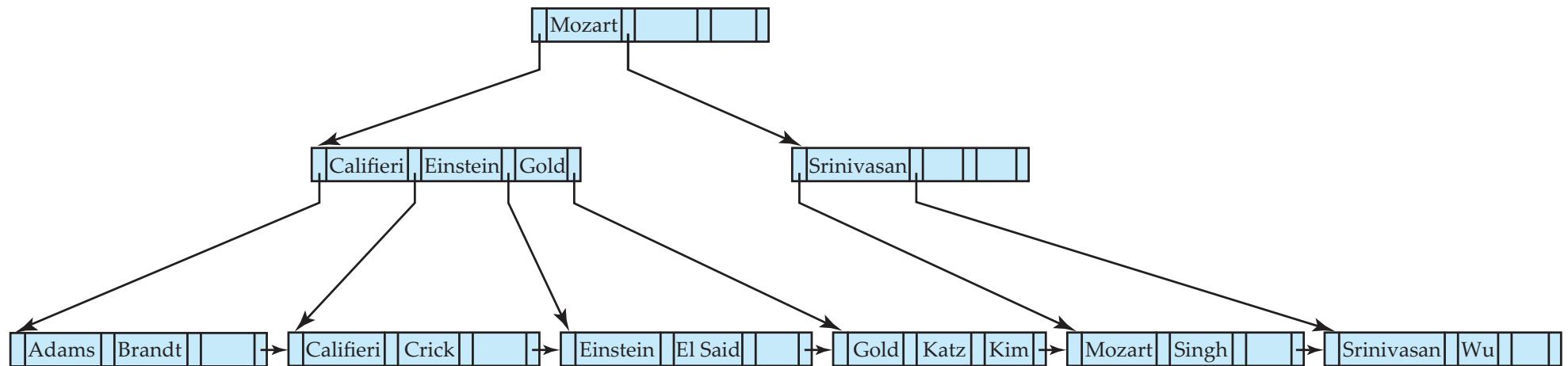
1. *C=root*
2. **while** (*C* is not a leaf node)
 1. Let *i* be least number s.t. $V \leq K_i$.
 2. **if** there is no such number *i* **then**
 3. Set *C = last non-null pointer in C*
 4. **else if** (*v = C.K_i*) Set *C = P_{i+1}*
 5. **else set** *C = C.P_i*
3. **if** for some *i*, $K_i = V$ **then 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
- E.g., search for key values between Einstein and Singh.
 - Locate Einstein by following the pointers
 - Then, sequentially read data from Einstein up to Singh by using the pointers P_n , i.e., P_4 in this case.





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.
- High fan-out = Low IO cost
 - The fan-out is defined as the number of pointers to child nodes coming out of a node
 - This means that the depth of the tree is small, i.e., getting to any element requires very few IO operations!



Updates on B+-Trees: Insertion

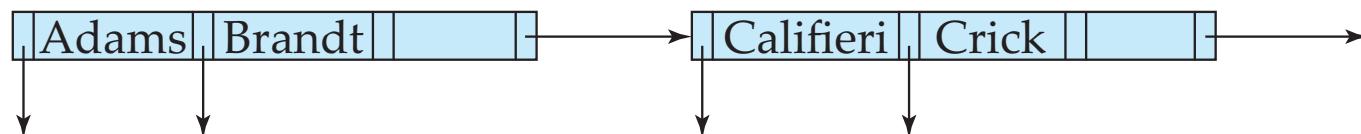
Assume record already added to the file. Let

- pr be pointer to the record, and let
 - v be the search key value of the record
-
- Find the leaf node in which the search-key value would appear
 - If there is room in the leaf node, insert (v, pr) pair in the leaf node
 - 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.)

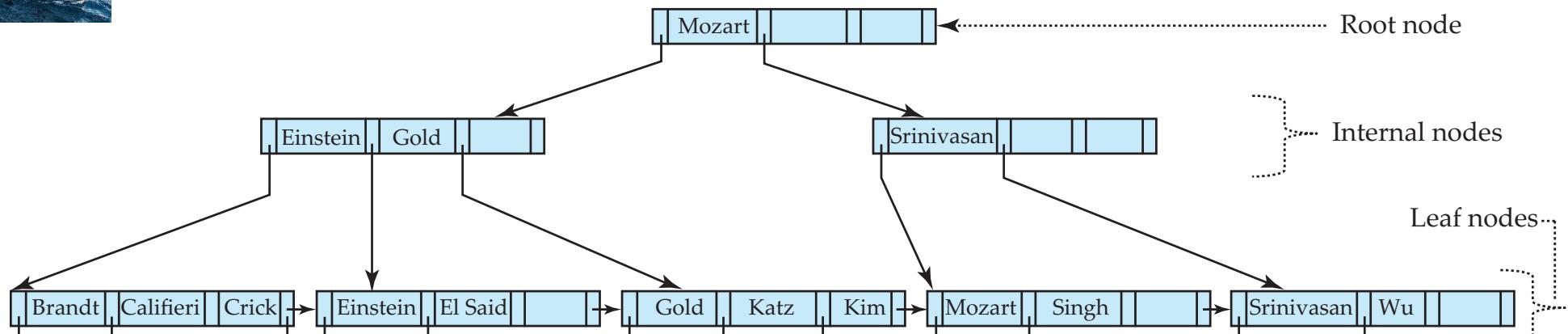
- Splitting a leaf node:
 - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
 - let the new node be p , and let k be the least key value in p . Insert (k,p) in the parent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



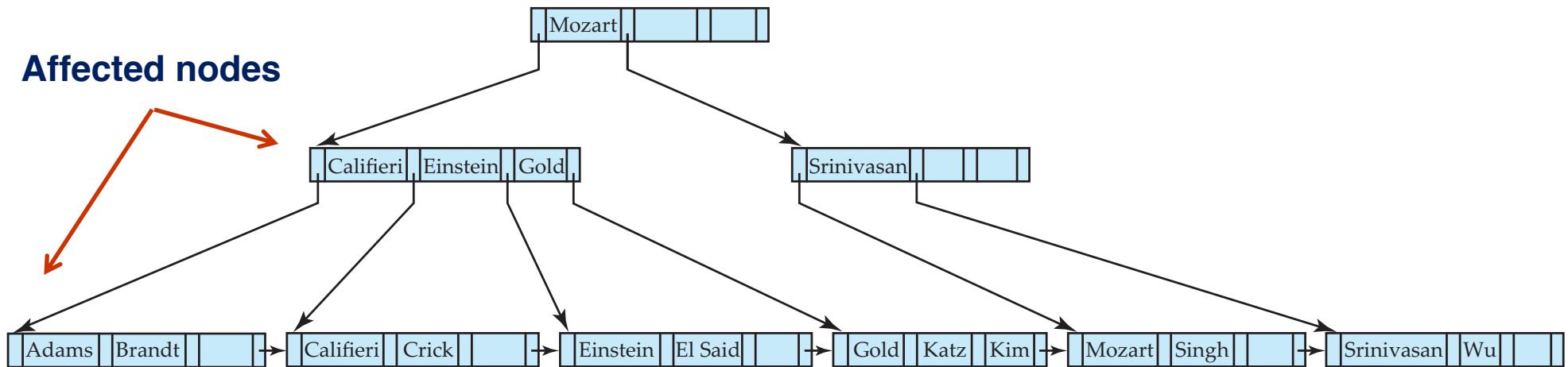
Result of splitting node containing Brandt, Califieri and Crick on inserting Adams
Next step: insert entry with (Califieri, pointer-to-new-node) into parent



B⁺-Tree Insertion



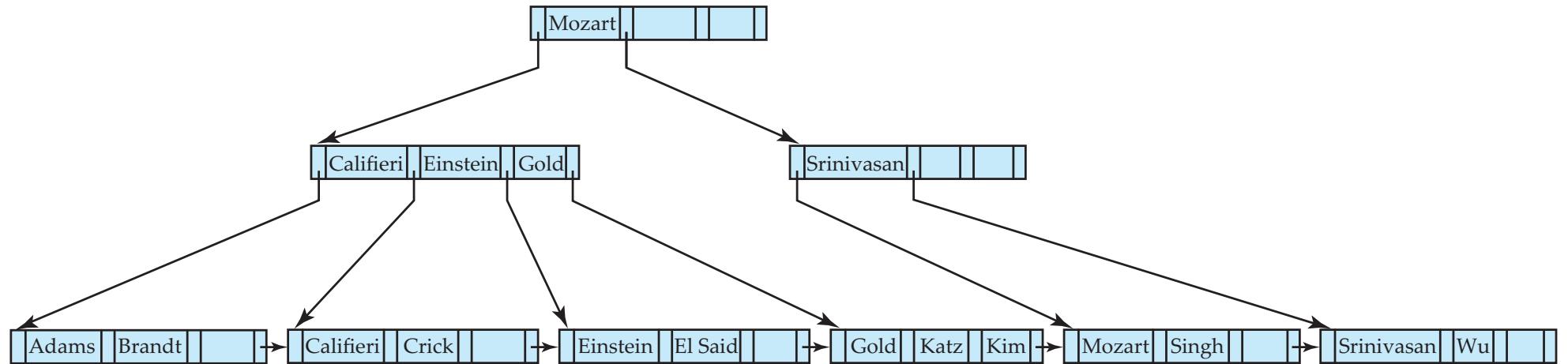
Affected nodes



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

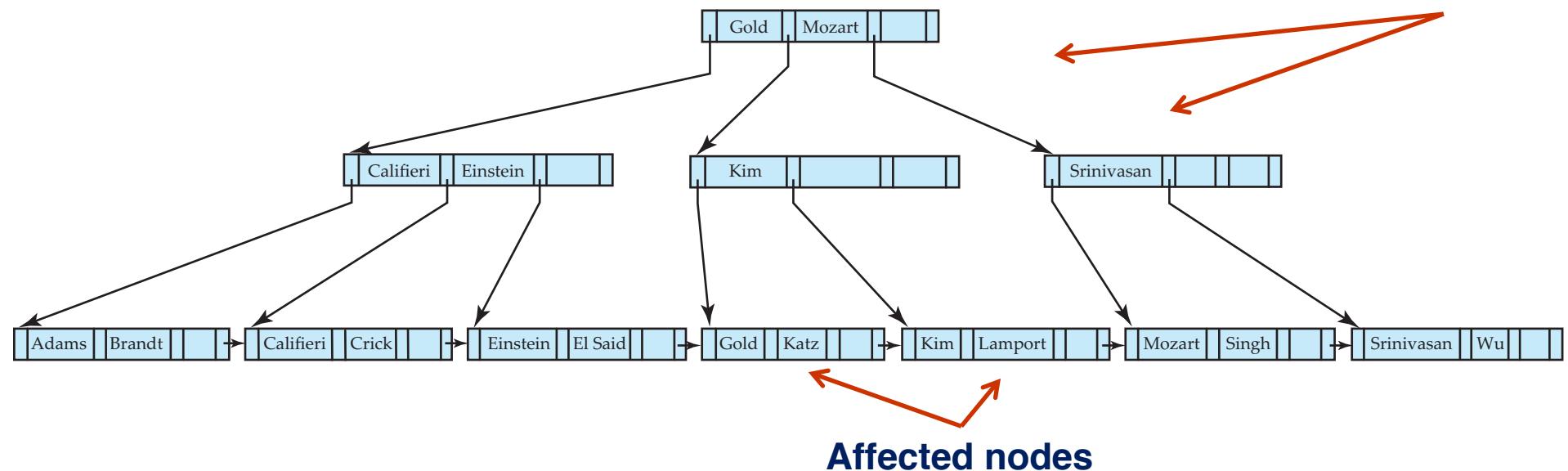


B+-Tree Insertion



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

Affected nodes

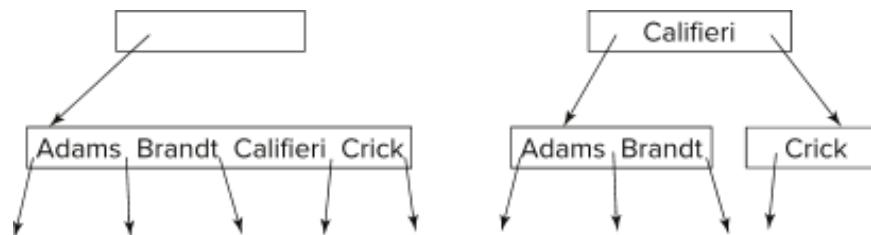


Affected nodes



Insertion in B+-Trees (Cont.)

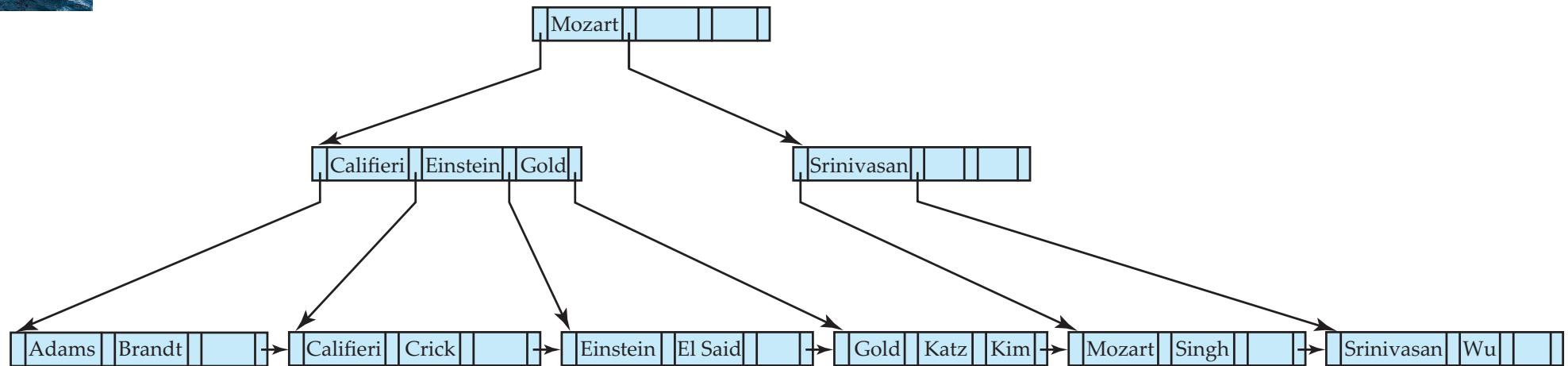
- Splitting a non-leaf node: when inserting (k, p) into an already full internal node N
 - Copy N to an in-memory area M with space for $n+1$ pointers and n keys
 - Insert (k, p) into M
 - Copy $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
 - Copy $P_{\lceil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



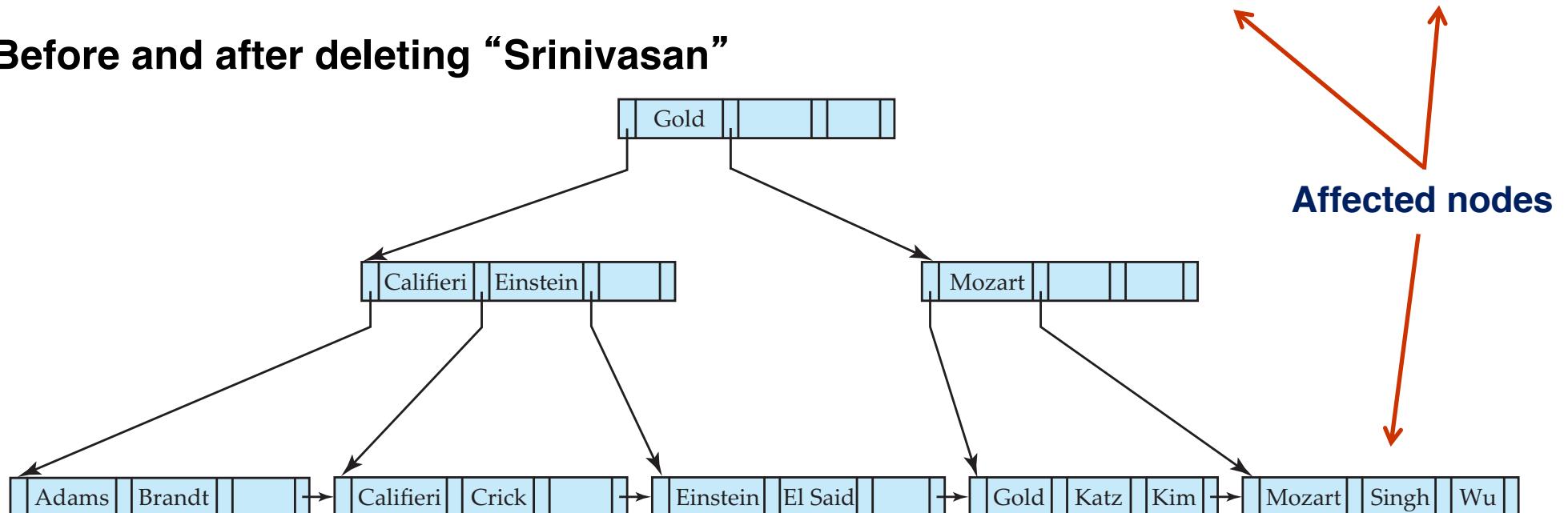
- Read pseudocode in book!



Examples of B+-Tree Deletion



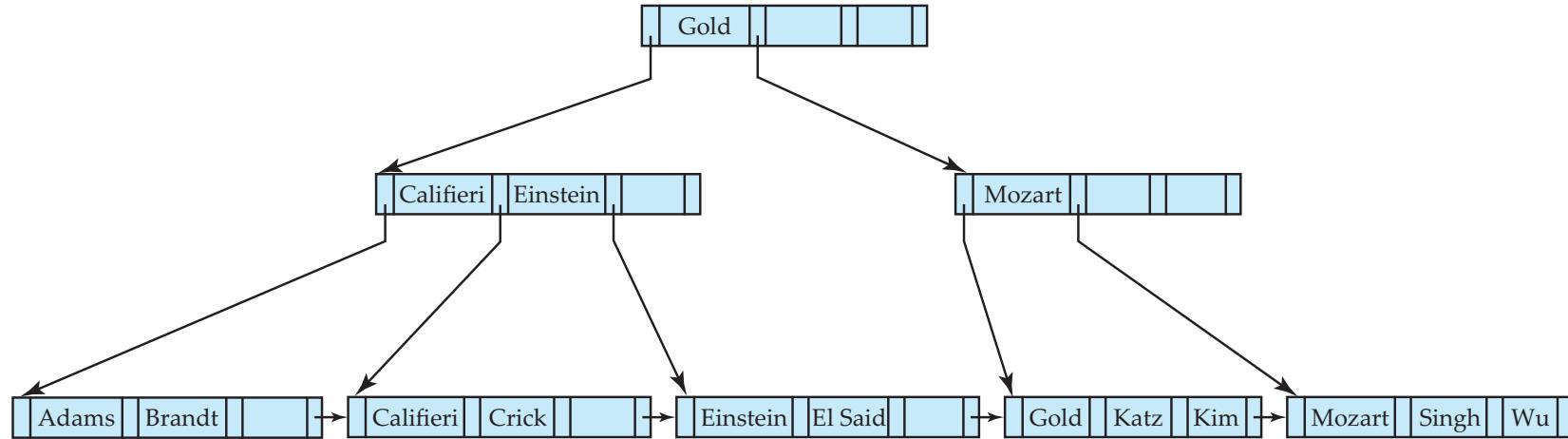
Before and after deleting “Srinivasan”



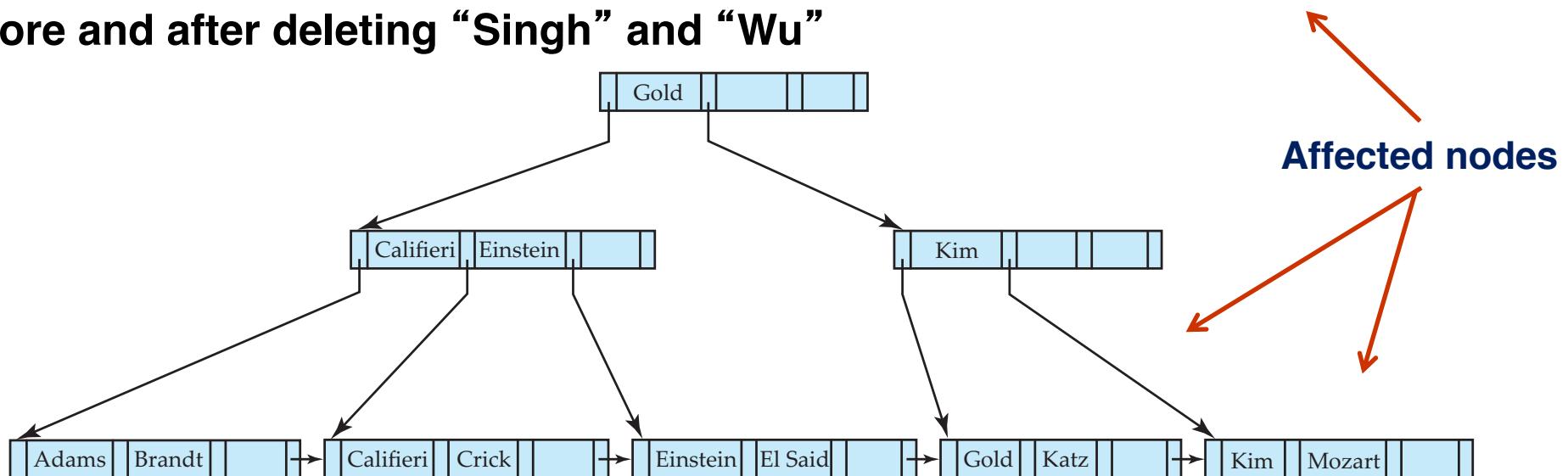
- Deleting “Srinivasan” causes **merging** of under-full leaves



Examples of B+-Tree Deletion (Cont.)



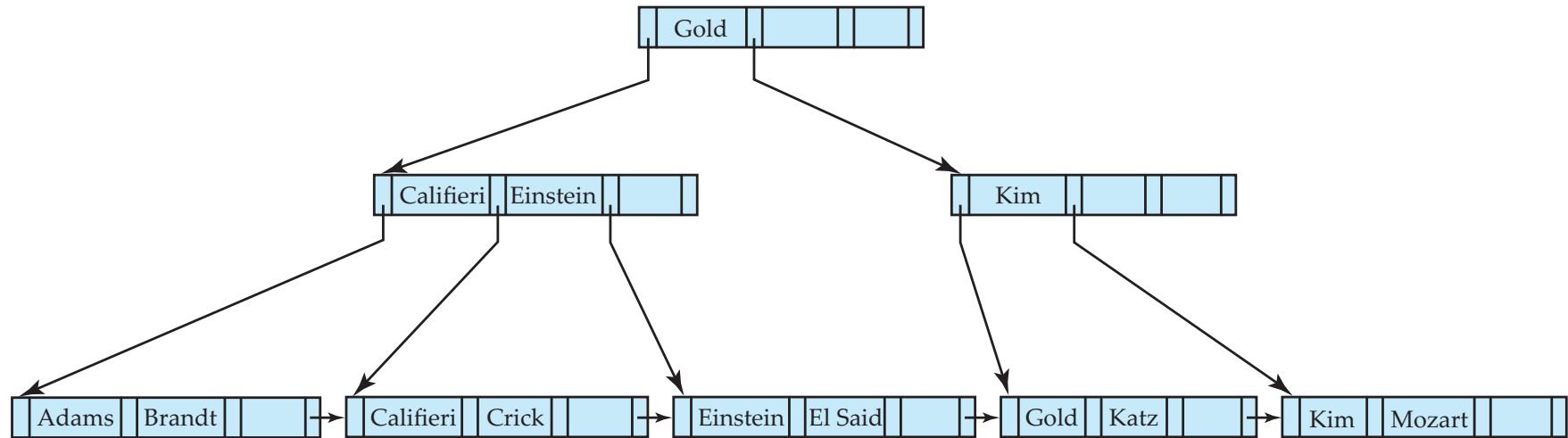
Before and after deleting “Singh” and “Wu”



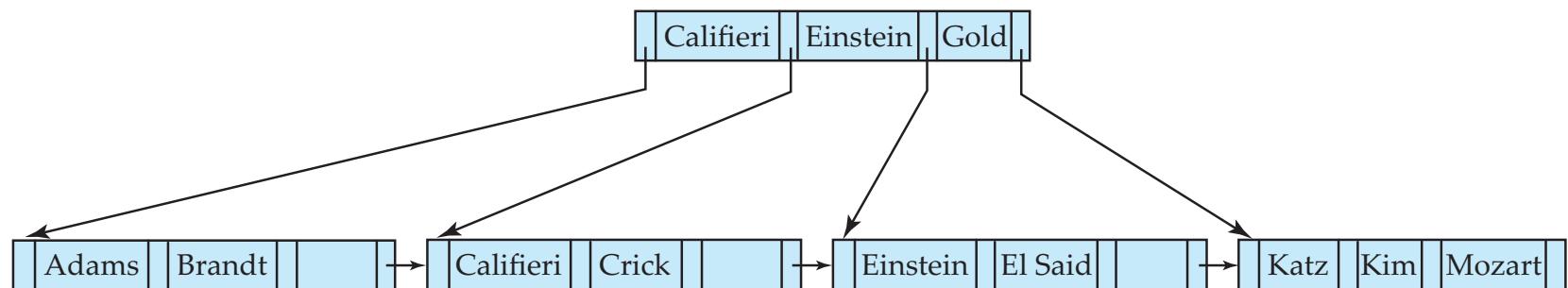
- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result



Example of B⁺-tree Deletion (Cont.)



Before and after deletion of “Gold”



- 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



Updates on B+-Trees: Deletion

Assume record already deleted from file. Let V be the search key value of the record, and Pr be the pointer to the record.

- Remove (Pr, V) from the leaf node
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
 - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
 - Delete the pair (K_{i-1}, P_i) , where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.



Updates on B+-Trees: Deletion

- 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.



Complexity of Updates

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



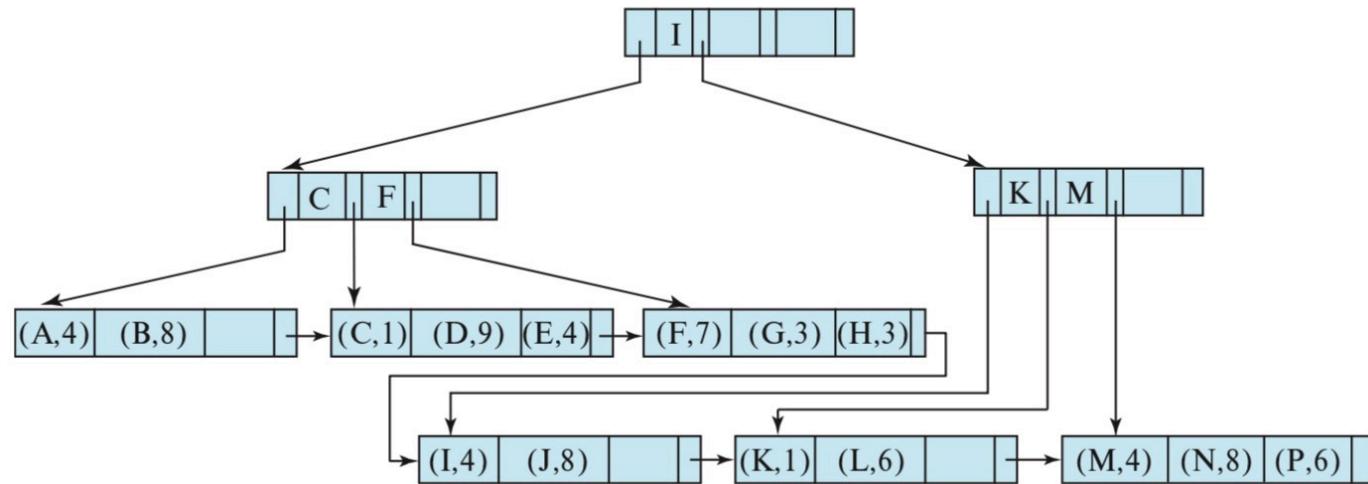
B+-Tree File Organization

- B+-Tree File Organization:
 - Leaf nodes in a B+-tree file organization store records, instead of pointers
 - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
 - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a 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 search key of B+-tree file organization instead of record pointer in secondary index
 - Add record-id if B+-tree file organization search key is non-unique
 - Extra traversal of file organization to locate record
 - Higher cost for queries, but node splits are cheap



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 that will be discussed later)
 - 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
 - Implemented as part of bulk-load utility by most database systems



B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node (a)



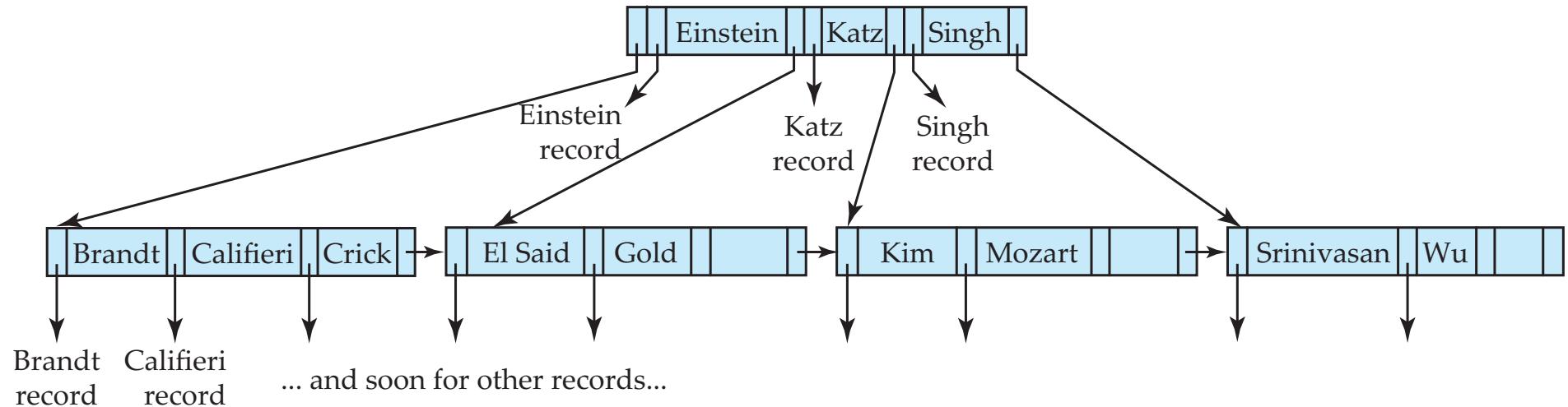
(a)



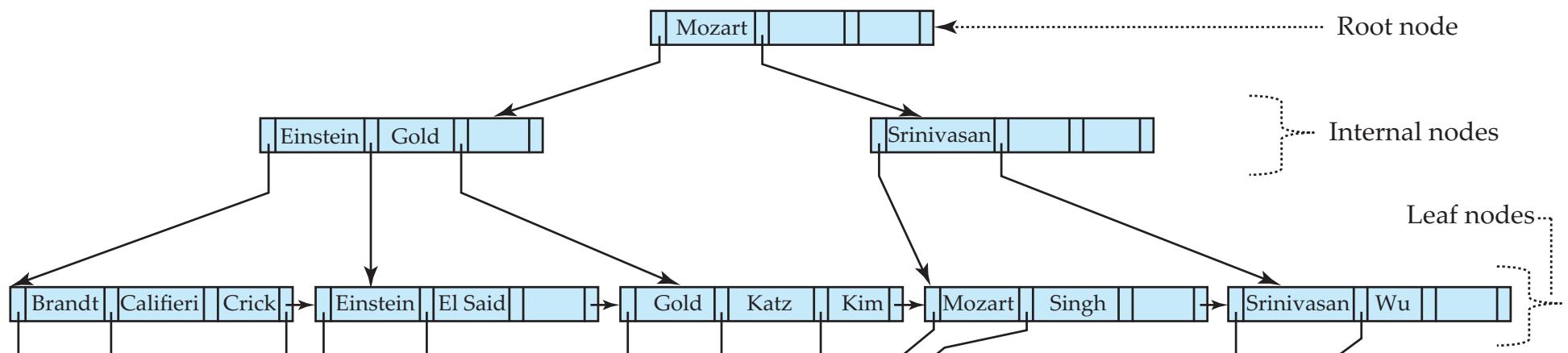
- Nonleaf node (b) – pointers B_i are the ^(b) bucket or file record pointers.
- There are $n - 1$ keys in the leaf node, but there are $m - 1$ keys in the nonleaf node.



B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data





B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
 - May use less tree nodes than a corresponding B⁺-Tree.
 - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
 - Only small fraction of all search-key values are found early
 - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B⁺-Tree
 - Insertion and deletion more complicated than in B⁺-Trees
 - Implementation is harder than B⁺-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.



Indexing in Main Memory

- Random access in memory
 - Much cheaper than on disk/flash
 - But still expensive compared to cache read
 - Data structures that make best use of cache preferable
 - Binary search for a key value within a large B⁺-tree node results in many cache misses
- B⁺- trees with small nodes that fit in cache line are preferable to reduce cache misses
- Key idea: use large node size to optimize disk access, but structure data within a node using a tree with small node size, instead of using an array.



Next week, we will continue with

Hashing