KOREA UNIVERSITY
DATAX LAB

Chapter 14: Indexing (1/2)

Outline

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

- Hashing
- Write-optimized indices
- Spatio-Temporal Indexing

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - an attribute or a 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".

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - Records with a specified value in the attribute
  - 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.

- **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.
- **Non-clustering index**: an index whose search key specifies an order different from the sequential order of the file.
  - Also called **secondary 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

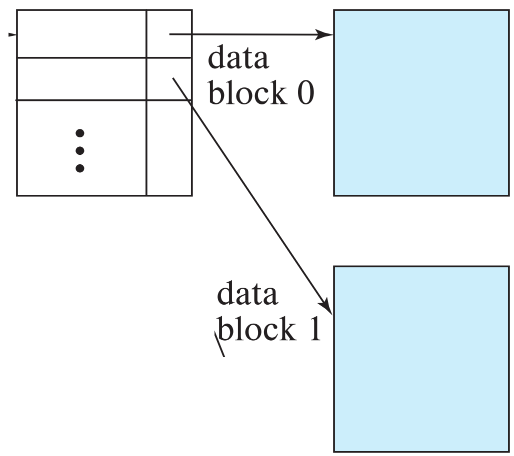
- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record 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.)

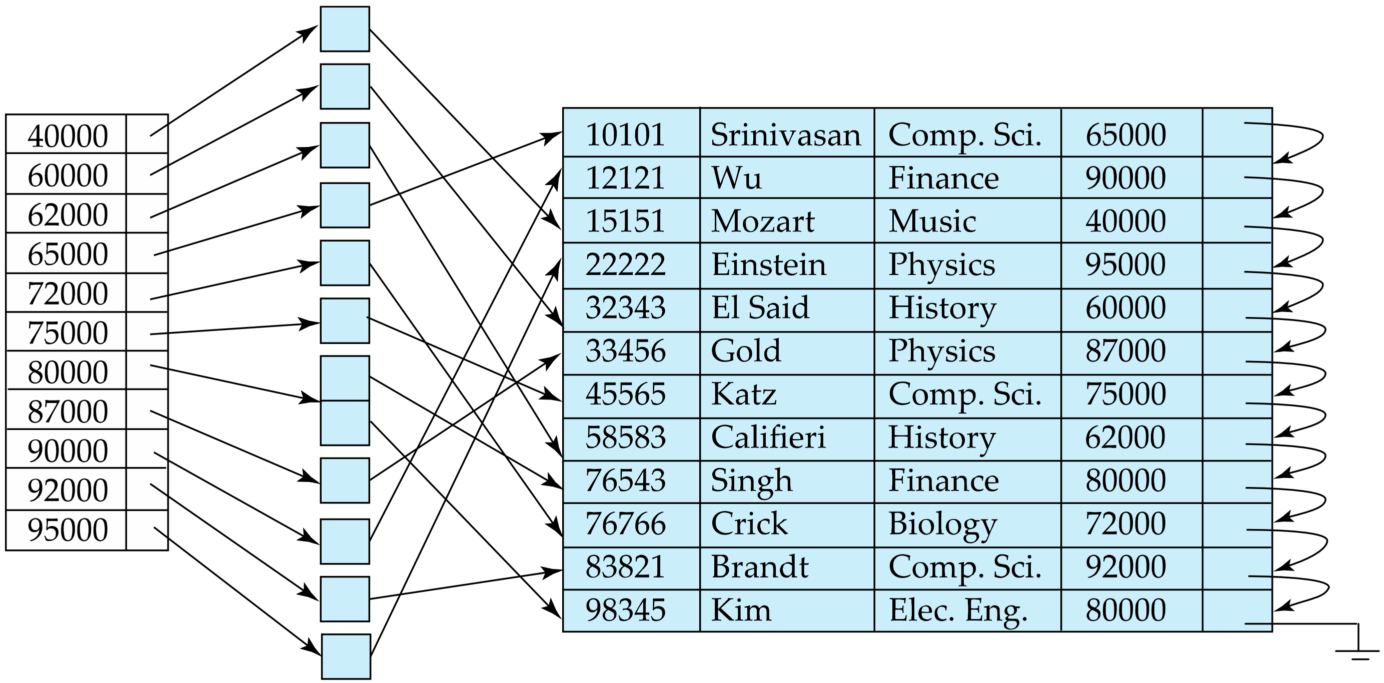
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**:
  - For clustering index: sparse index with an index entry for every block in file, corresponding to least search-key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



  - For non-clustering index: sparse index on top of dense index (multilevel index). (See Slide #12, #13)

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

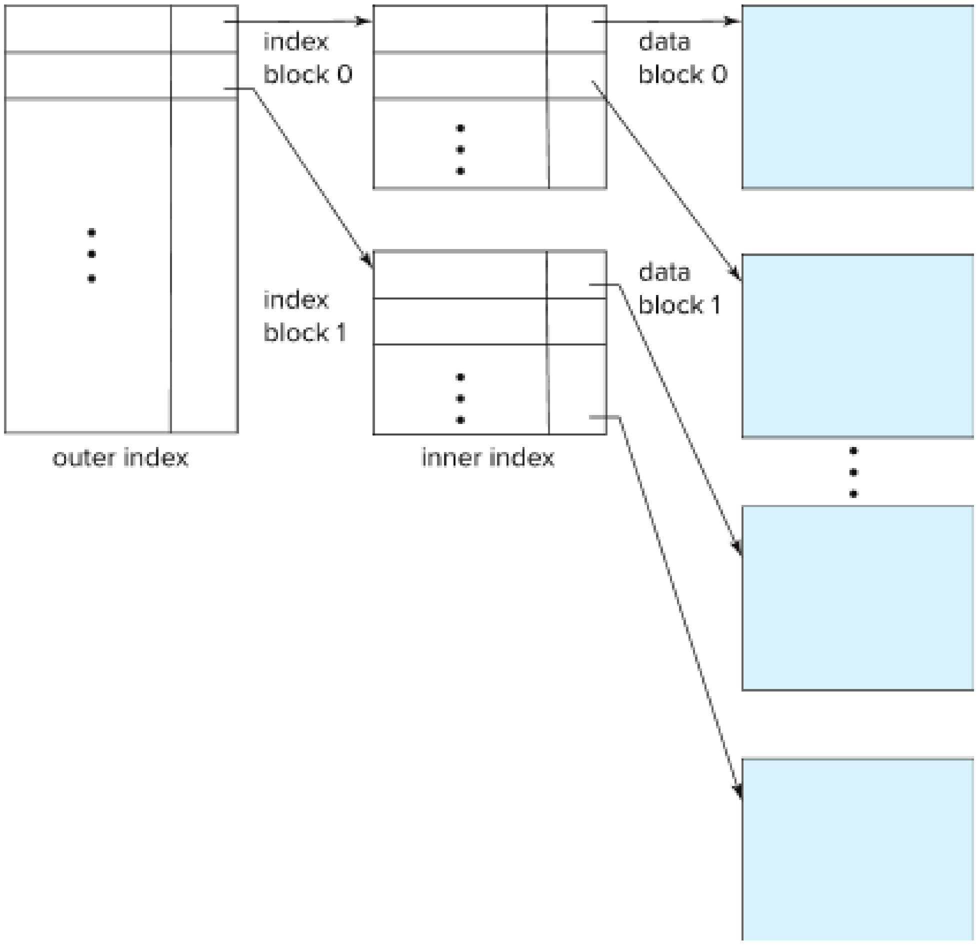
Enforcement of clustering indices is implemented in some DBMS's; PSQL does not support it except the 'cluster' command.

'cluster *table_name* using *idx_name*'
'cluster *table_name*' (afterwards)

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.)



Index Update: Deletion

- **Single-level index entry deletion:**
  - **Dense indices** – deletion of search-key is similar to file record deletion.
  - **Sparse indices** –
    - 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.
      - 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.

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	

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

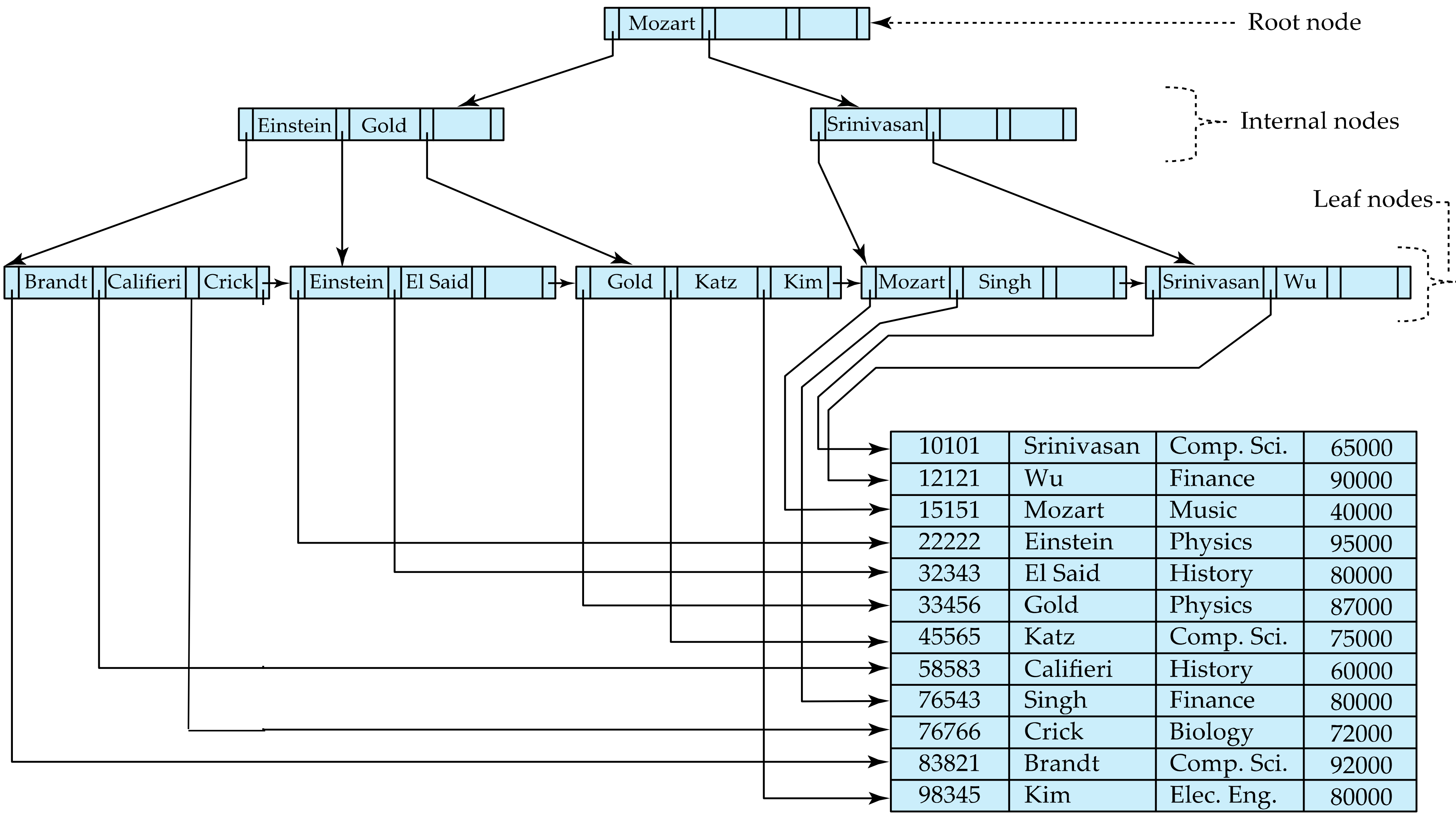
- **Composite search key**
  - E.g., index on *instructor* relation on attributes (*name, ID*)
  - Values are sorted lexicographically
    - E.g. (John, 12121) < (John, 13514) and
      (John, 13514) < (Peter, 11223)
  - Can query on just *name*, or on (*name, ID*)

B^+-Tree Index

- Disadvantage of index-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:
  - 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 *(n=4)*



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 (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

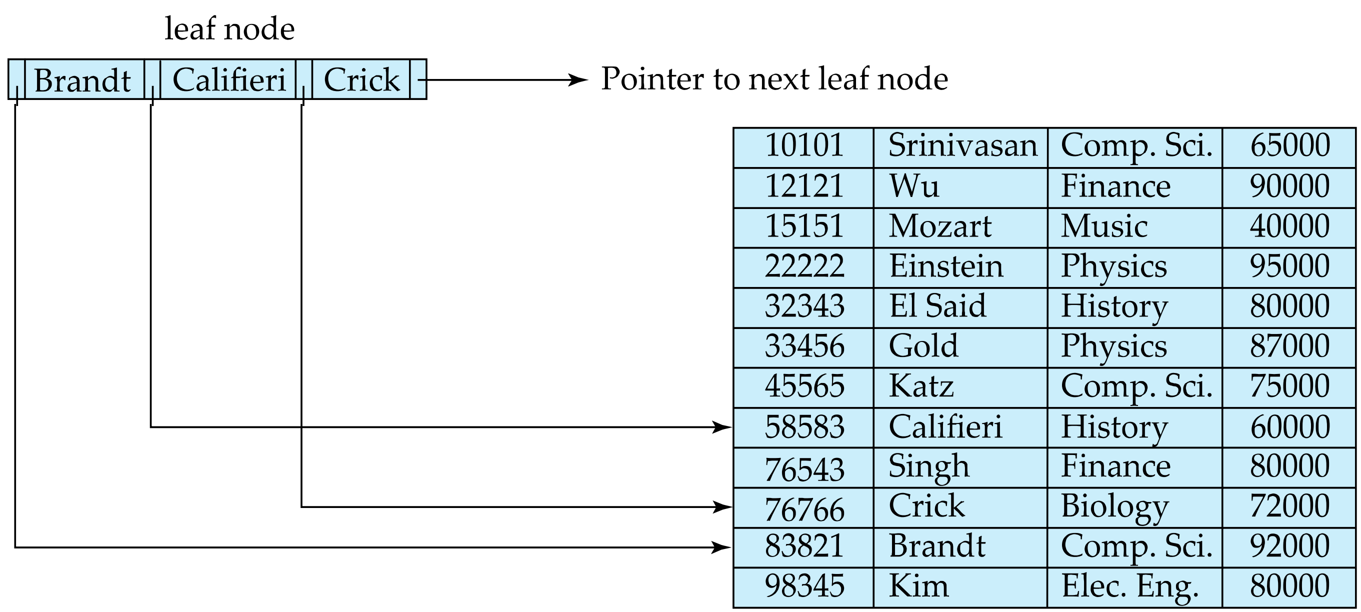
$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$

(Initially assume no duplicate keys; duplicates will be addressed 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

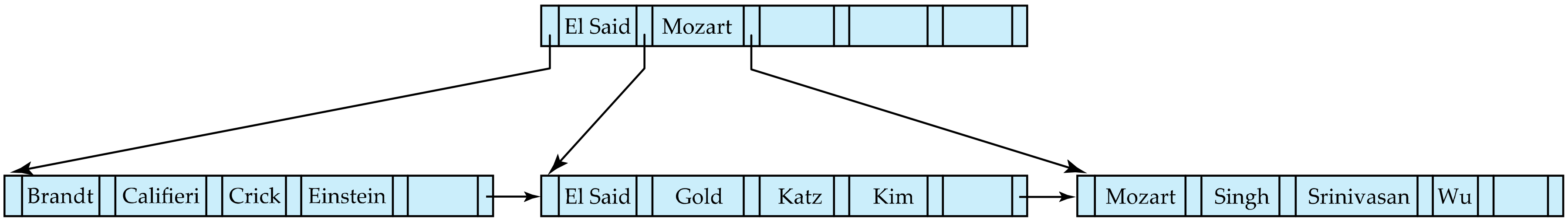
- Non leaf nodes form a multi-level, sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal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

KOREA UNIVERSITY
DATAX LAB

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.

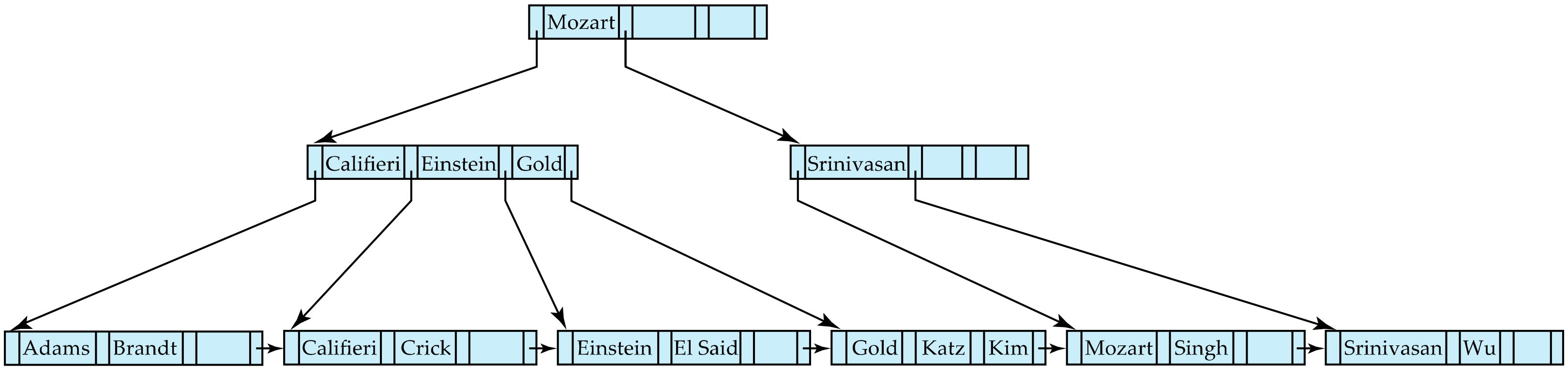
Observations about B^+-trees

- 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 $2 * \lceil n/2 \rceil$ children
    - Next level has at least $2 * \lceil n/2 \rceil * \lceil n/2 \rceil$ children
    - .. 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

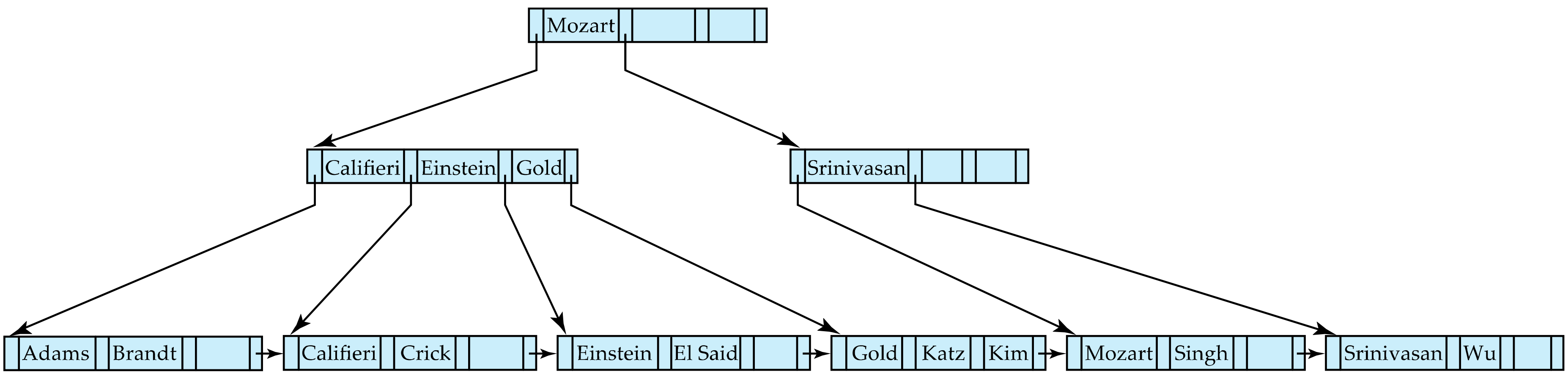
function *find*(*v*)

1. *C=root*
2. **while** (C is not a leaf node)
   1. Let *i* be least number s.t. $V \le K_i$.
   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
    - The above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Updates on B^+-Trees: Insertion

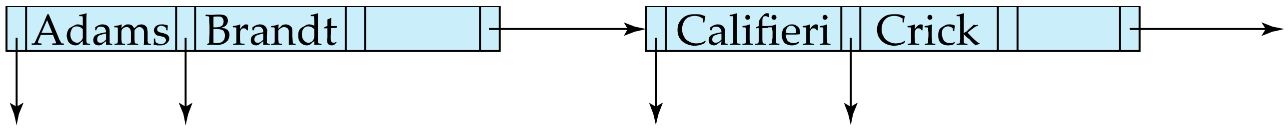
Assume record already added to the file.

- v be the search key value of the record
- *pr* be pointer to 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.

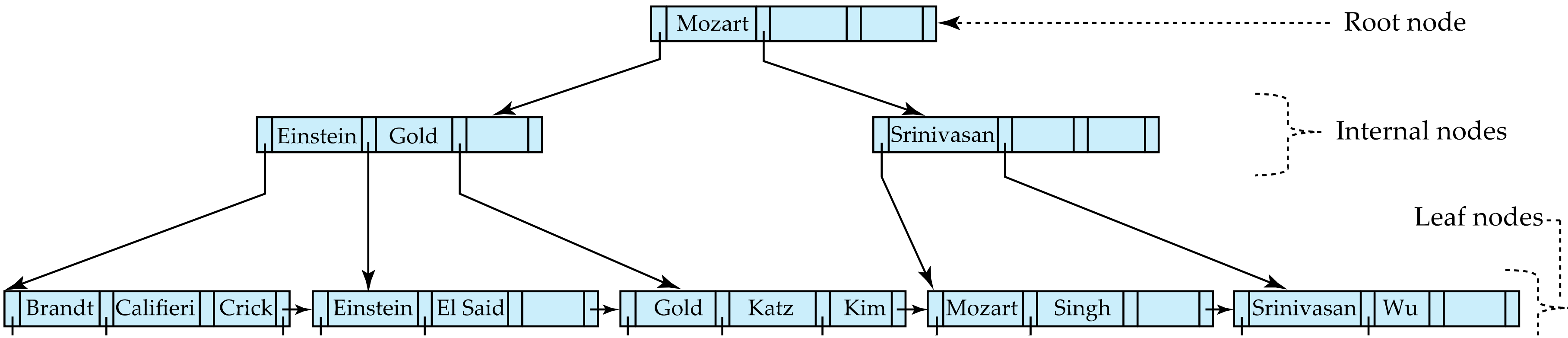
Updates on B$^+$-Trees: Insertion (Cont.)

- 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.



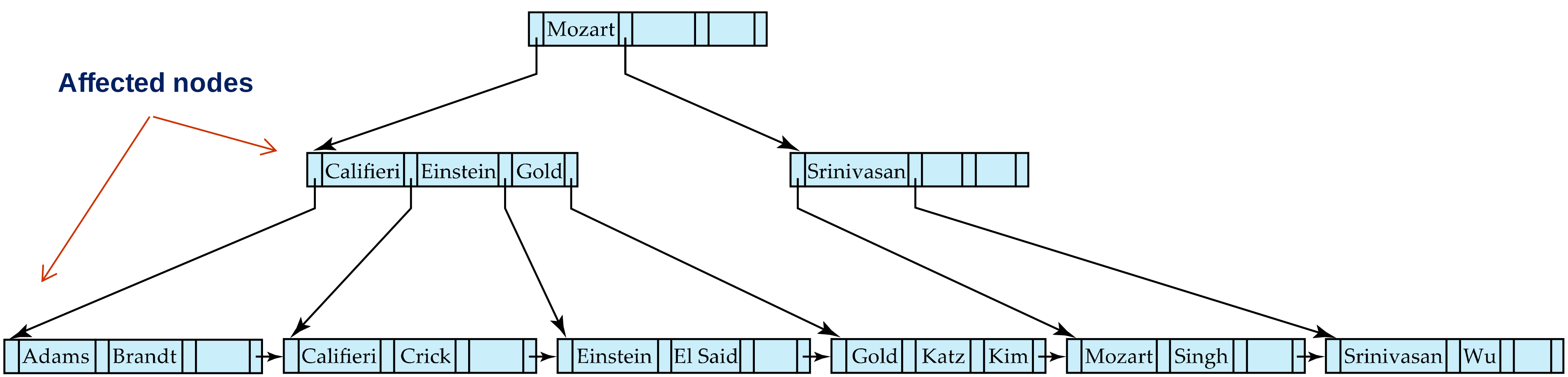
Result of splitting node containing (Brandt, Califieri, Crick) on inserting Adams
Next step: insert entry with (Califieri, pointer-to-new-node) into parent

B+-Tree Insertion

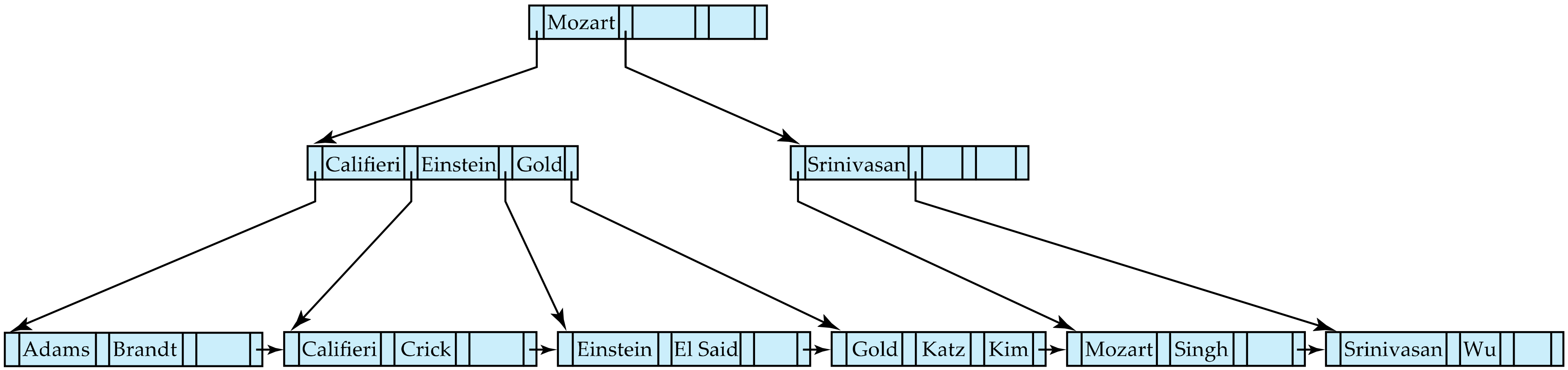


B+-Tree before and after insertion of "Adams"

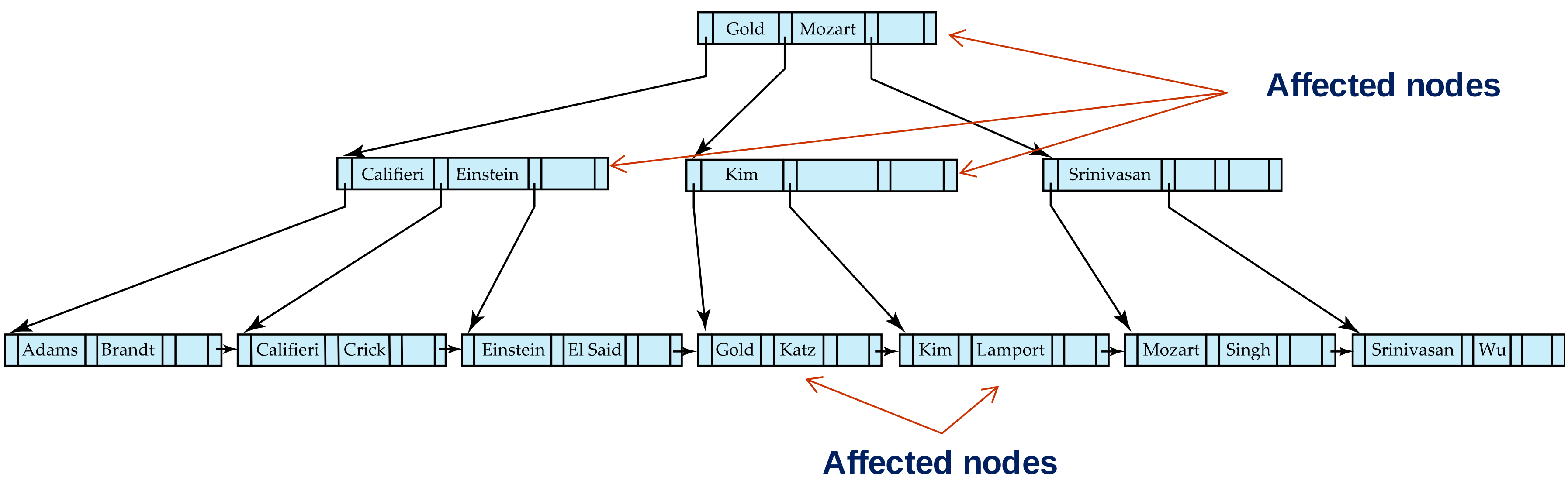
Affected nodes



B+-Tree Insertion



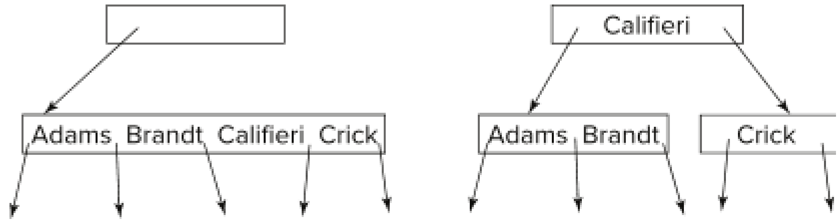
B+-Tree before and after insertion of "Lamport"



KOREA UNIVERSITY DATAX LAB

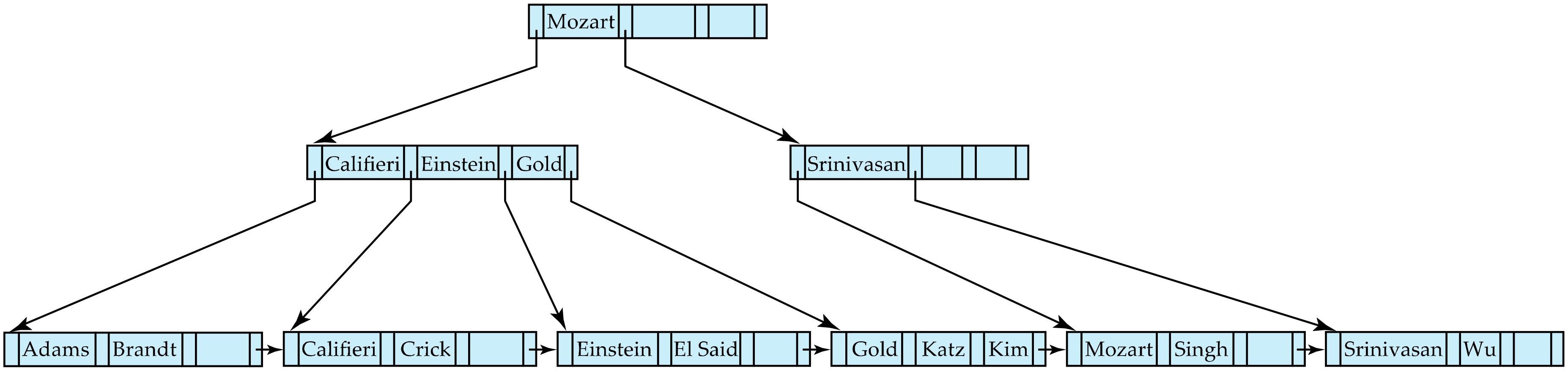
Insertion in B$^+$-Trees (Cont.)

- 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

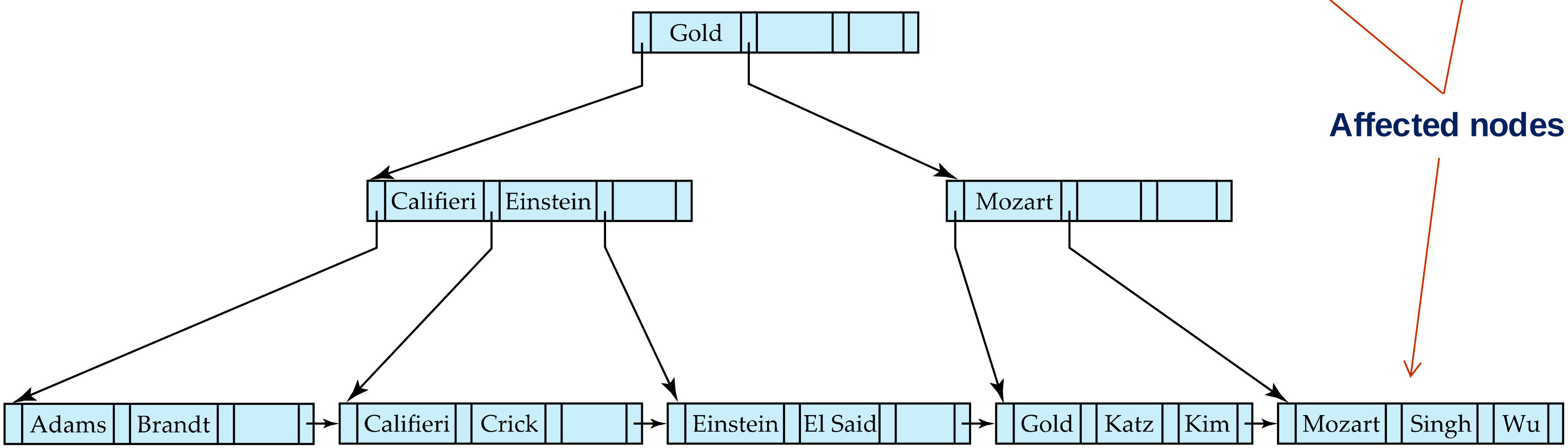


- **Refer to pseudocode in book!**

Examples of B+-Tree Deletion



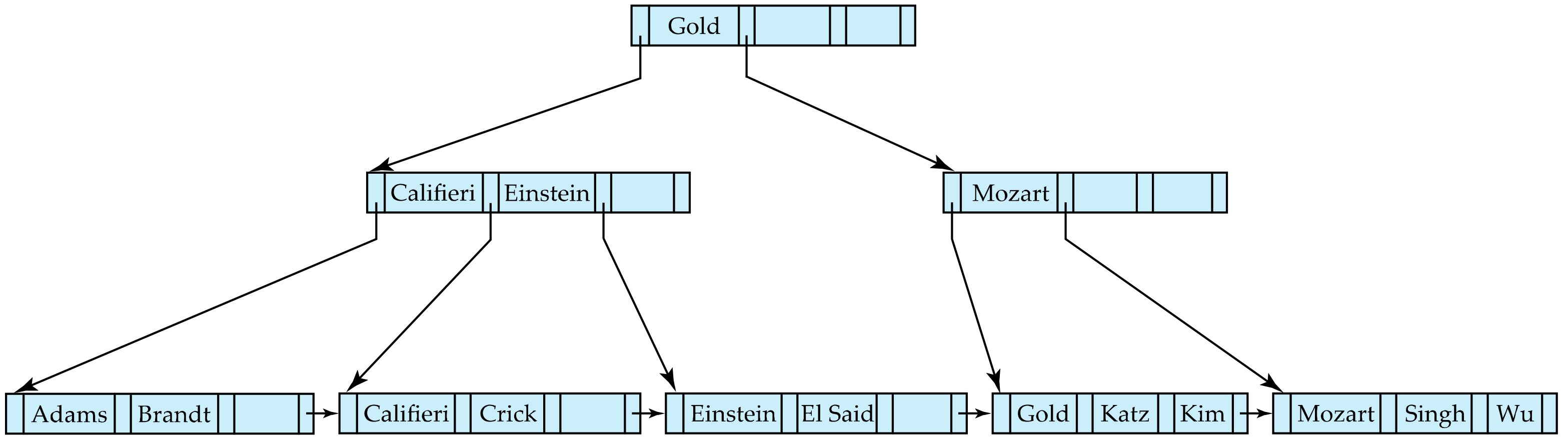
Before and after deleting "Srinivasan"



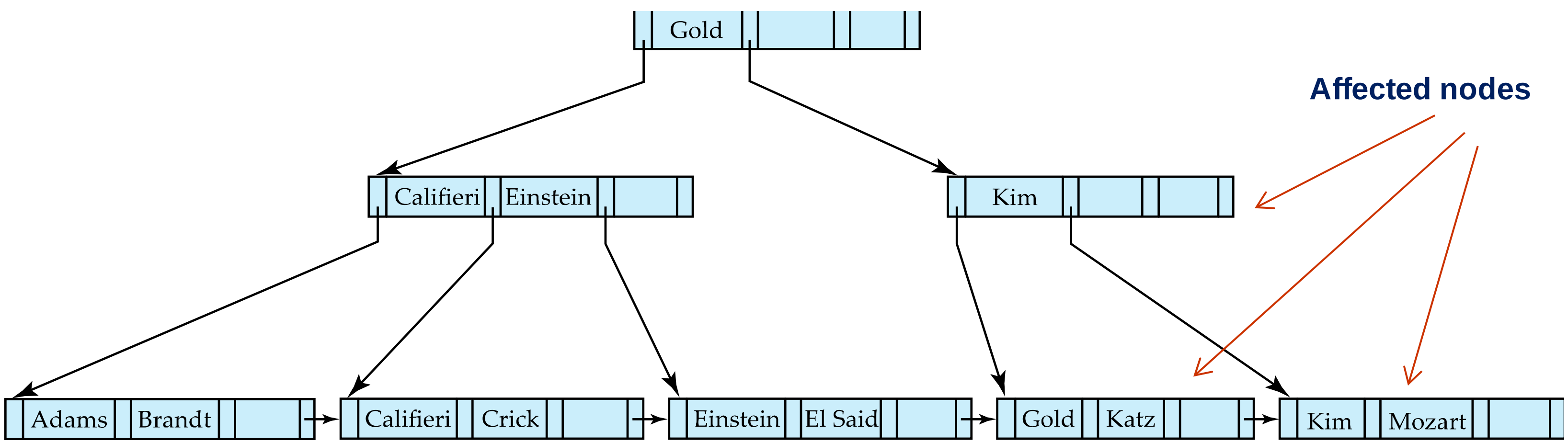
Affected nodes

- Deleting "Srinivasan" causes **merging** of underfull leaves

Examples of B$^+$-Tree Deletion (Cont.)

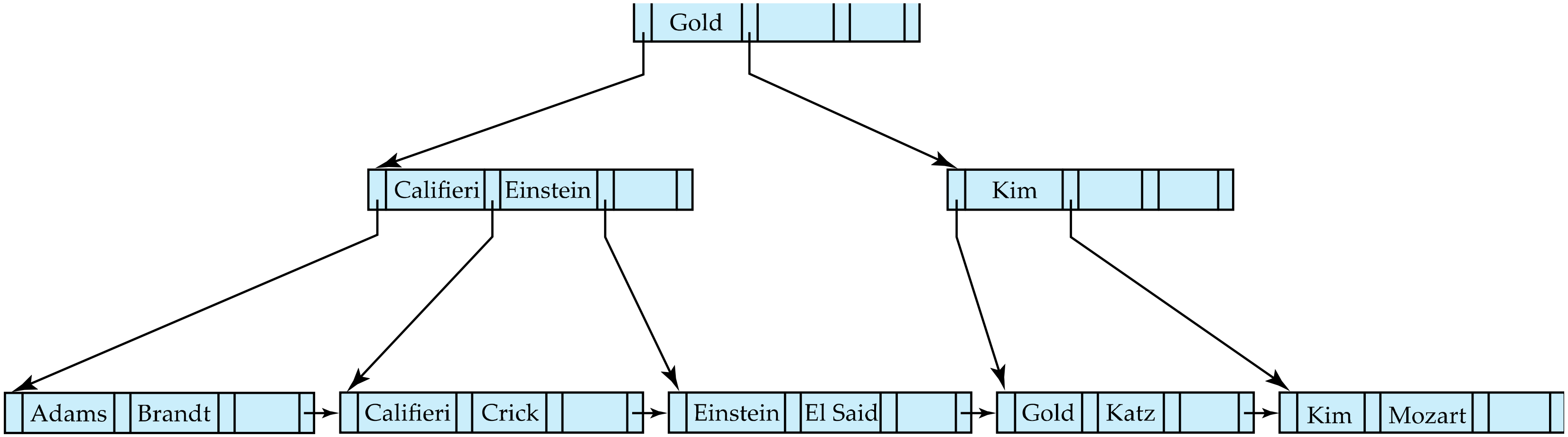


Before and after deleting "Singh" and "Wu"

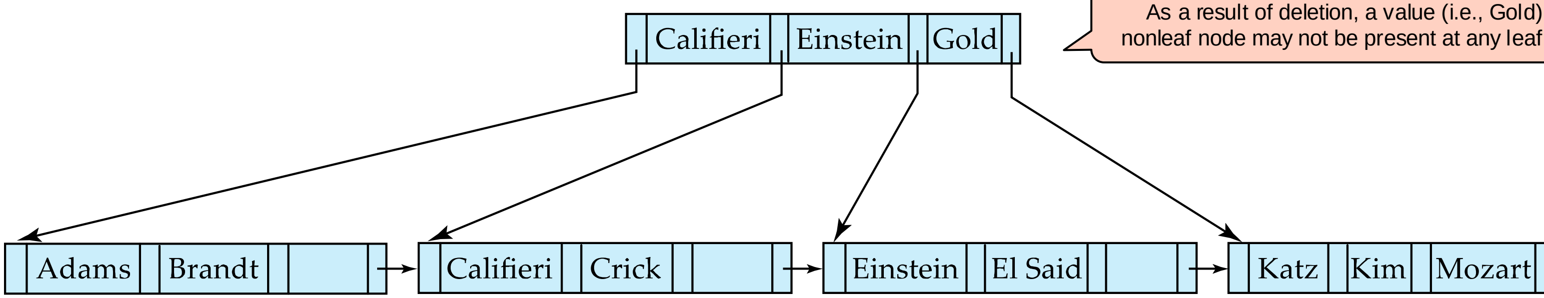


- Leaf containing Singh and Wu became underfull, and a value Kim is **borrowed** from its left sibling; the search-key value in the parent changes as a result

Example of B+-tree Deletion (Cont.)



Before and after deletion of "Gold"



As a result of deletion, a value (i.e., Gold) in a nonleaf node may not be present at any leaf nodes.

- 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.

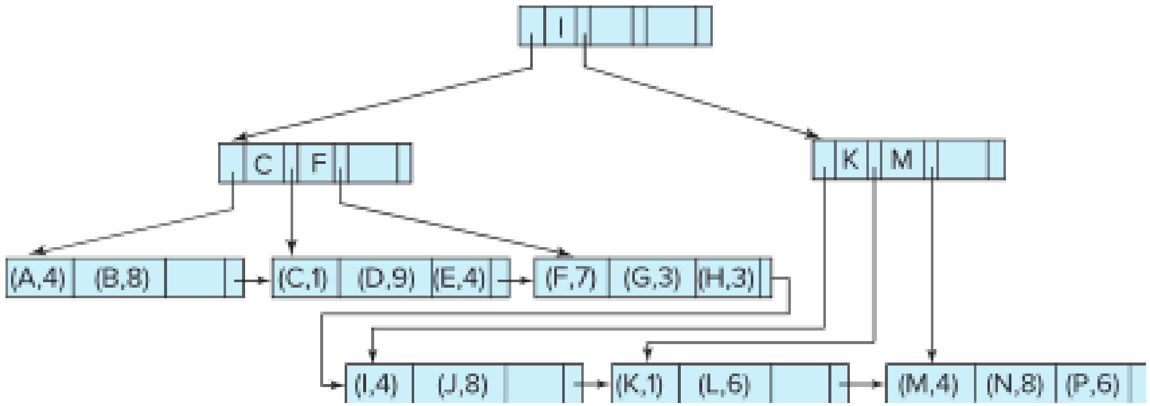
- Remove (*V, Pr*) from the leaf node
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other 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.
- 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.

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 is important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - "10, 4, 10" → "7, 7, 10" vs. "8, 8, 8"
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

B+-Tree Construction: Bulk Loading and Bottom-Up Build

- Inserting entries one-at-a-time into a B^+-tree requires $\geq$ 1 IO per entry
- Insertion of a large number of entries at a time (called **'bulk loading'**) is very inefficient, because
  - Data are not clustered in the storage (in case of secondary index) and mostly not in the buffer
- Efficient alternative 1:
  - Sort entries first (using efficient external-memory sort algorithms discussed later in Section 12.4), and 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

- 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.
- B-tree leaf node (a) and nonleaf node (b) – pointers Bi are the bucket or file record pointers.

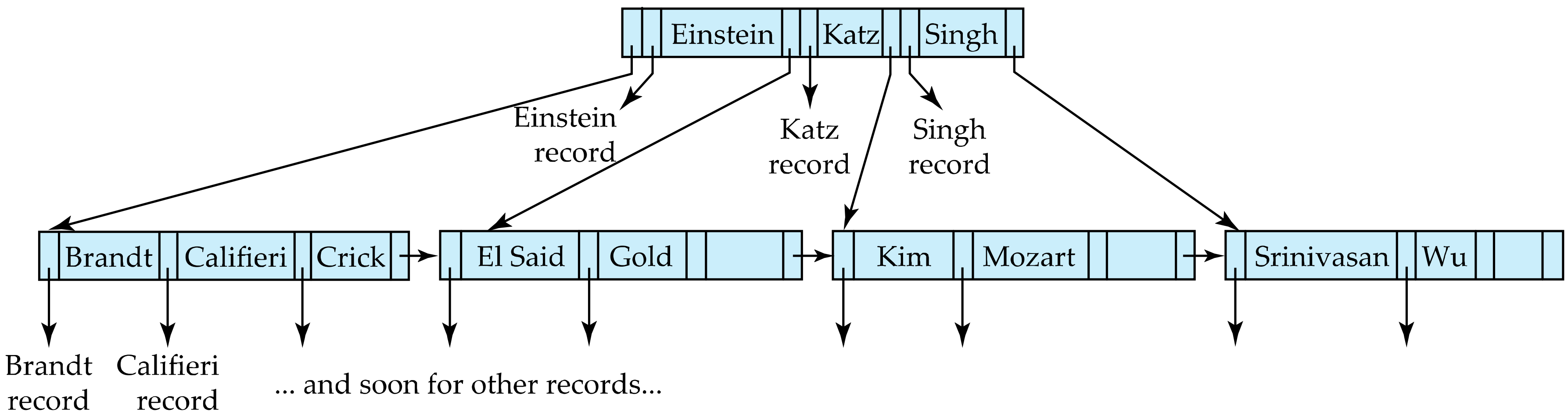
P_1	K_1	P_2	$\cdots$	P_{n-1}	K_{n-1}	P_n

(a)

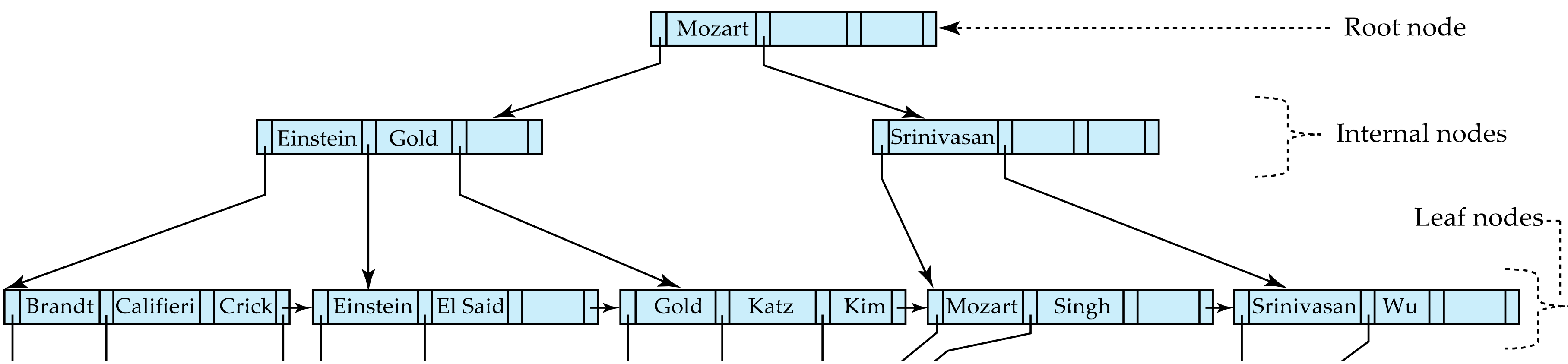
P_1	B_1	K_1	P_2	B_2	K_2	$\cdots$	P_{m-1}	B_{m-1}	K_{m-1}	P_m

(b)

B-Tree Index Example



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



B-Tree Index (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.

End of 1^{st} Half of Chapter 14