Chapter 9: 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
- B-Tree Index Files
- Hashing
- Spatio-Temporal Indexing

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in a 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".

©Silberschatz, Korth and Sudarshan

Index Evaluation Metrics

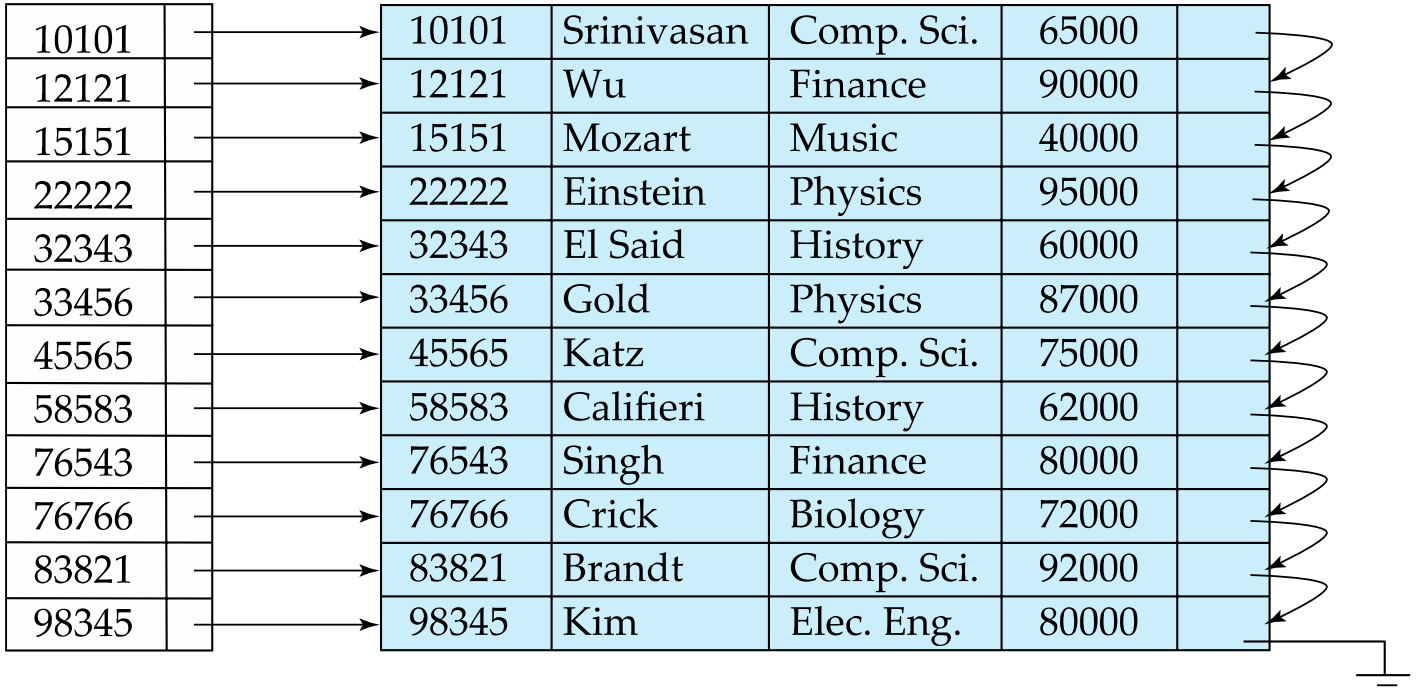
- Access types supported efficiently. e.g.,
  - Records with a specified value in the attribute
  - Records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index,** index entries are stored 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

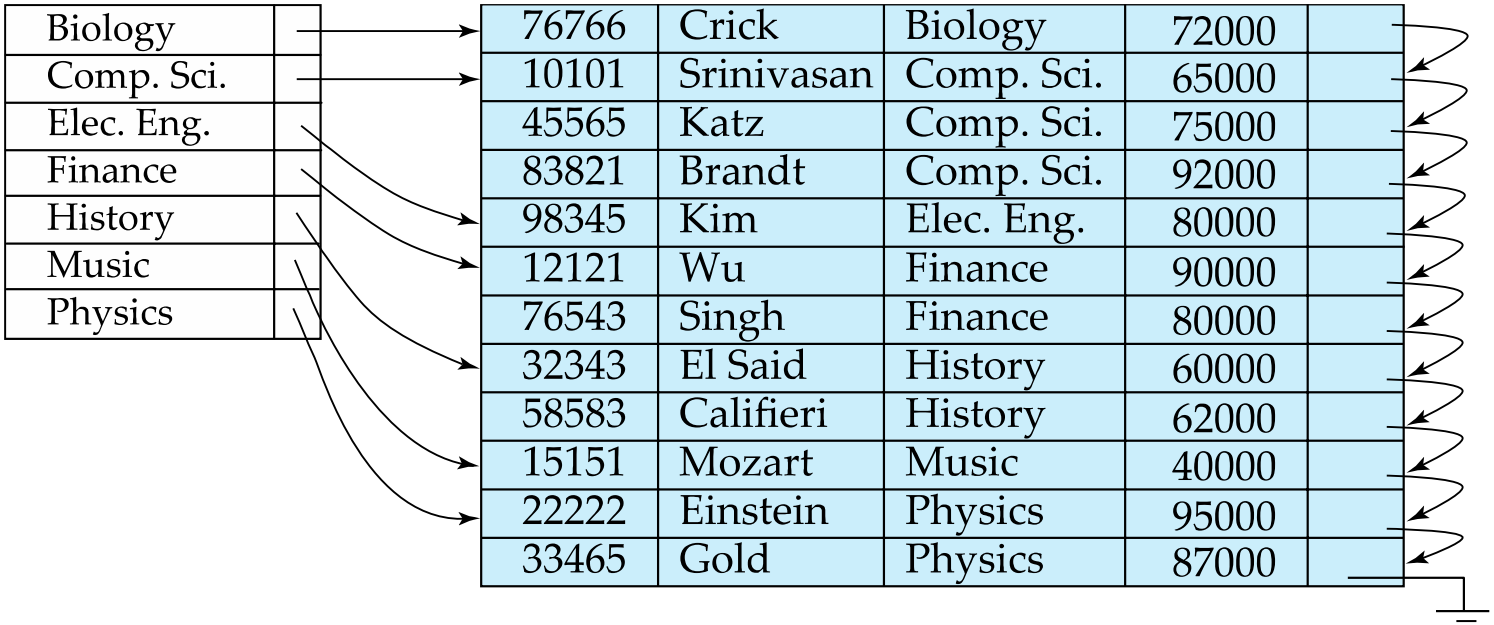
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation



 ©Silberschatz, Korth and Sudarshan

Dense Index Files (Cont.)

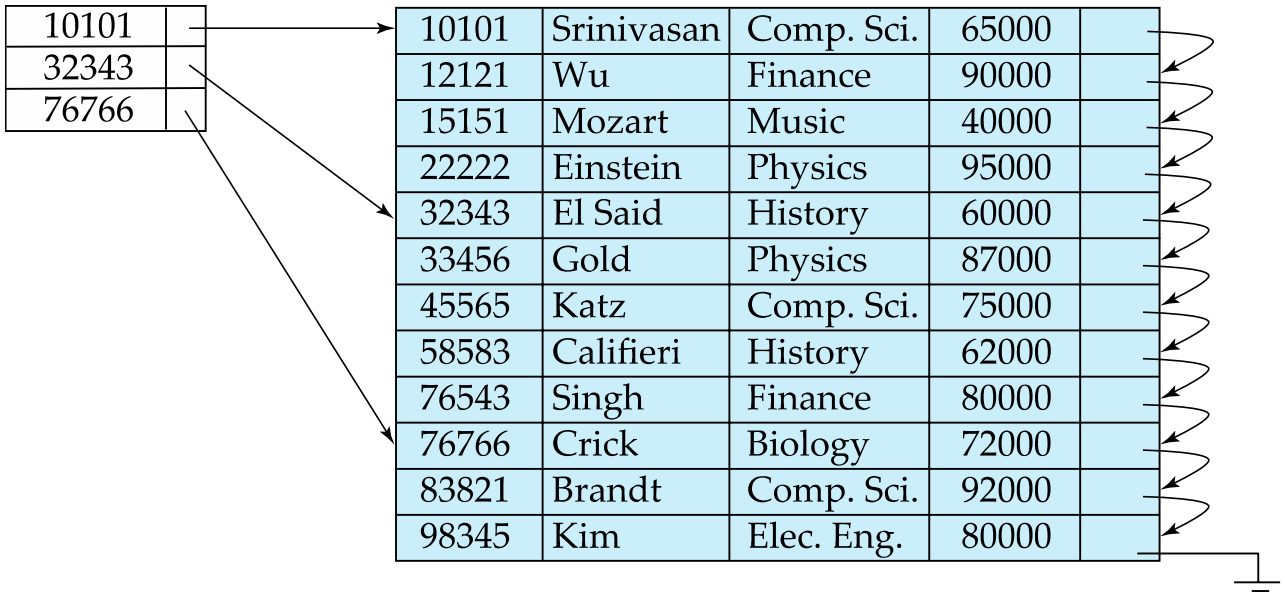
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*



 ©Silberschatz, Korth and Sudarshan

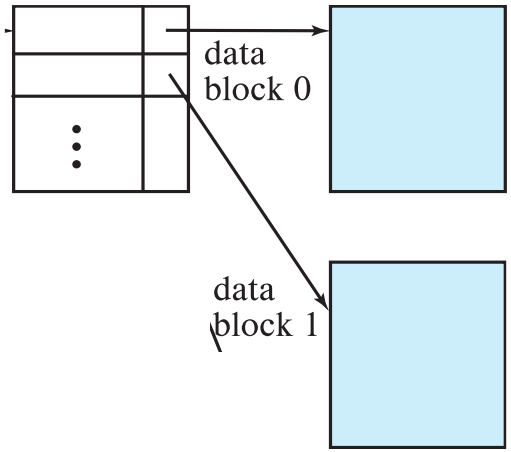
Sparse Index Files

- **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 the index record with the largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record points



Sparse Index Files (Cont.)

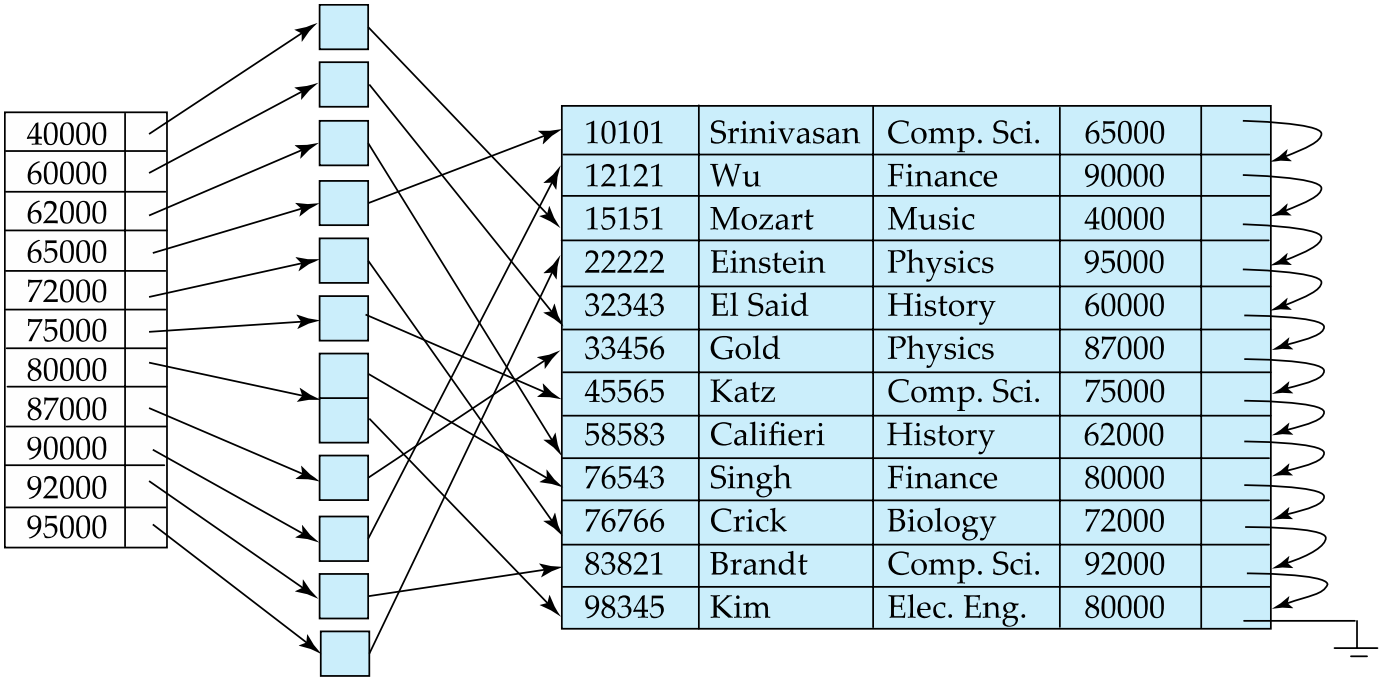
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than the dense index for locating records.
- **Good tradeoff**:
  - For clustered index: sparse index with an index entry for every block in the file, corresponding to the least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



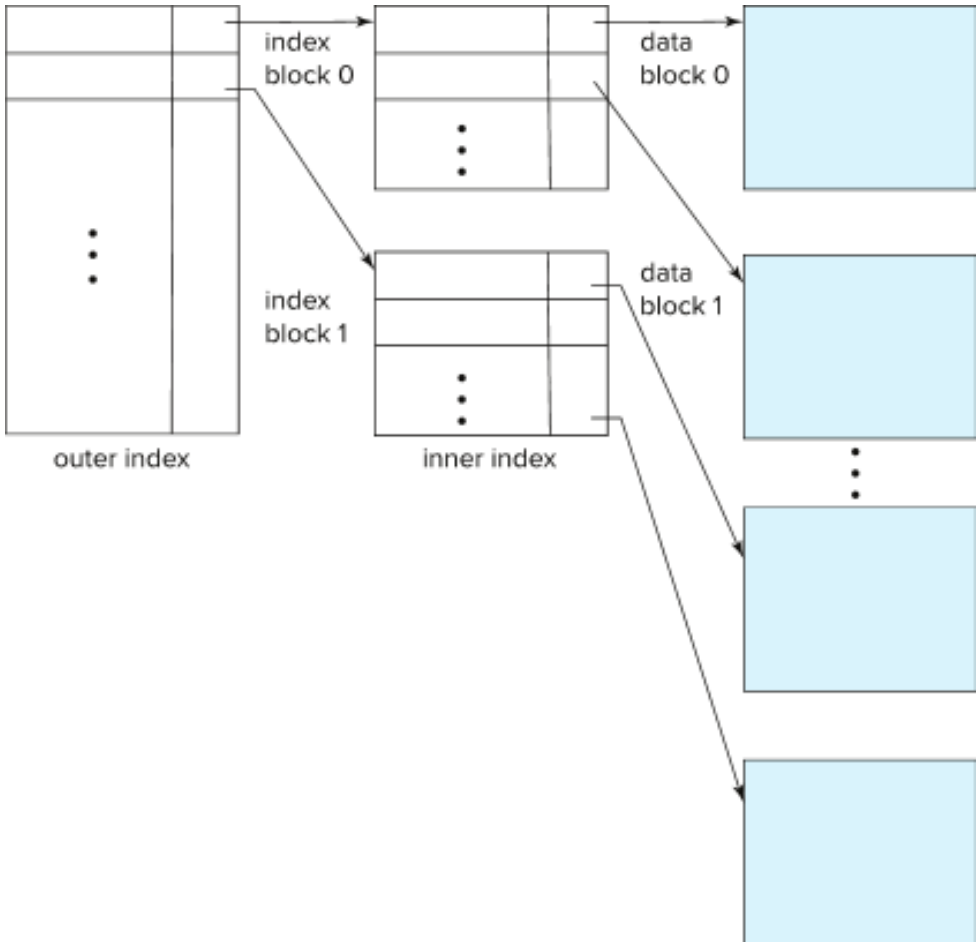
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense

Multilevel Index

- If the index does not fit in memory, access becomes expensive.
- Solution: treat the 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 the outer index is too large to fit in the main memory, yet another level of the index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

©Silberschatz, Korth and Sudarshan

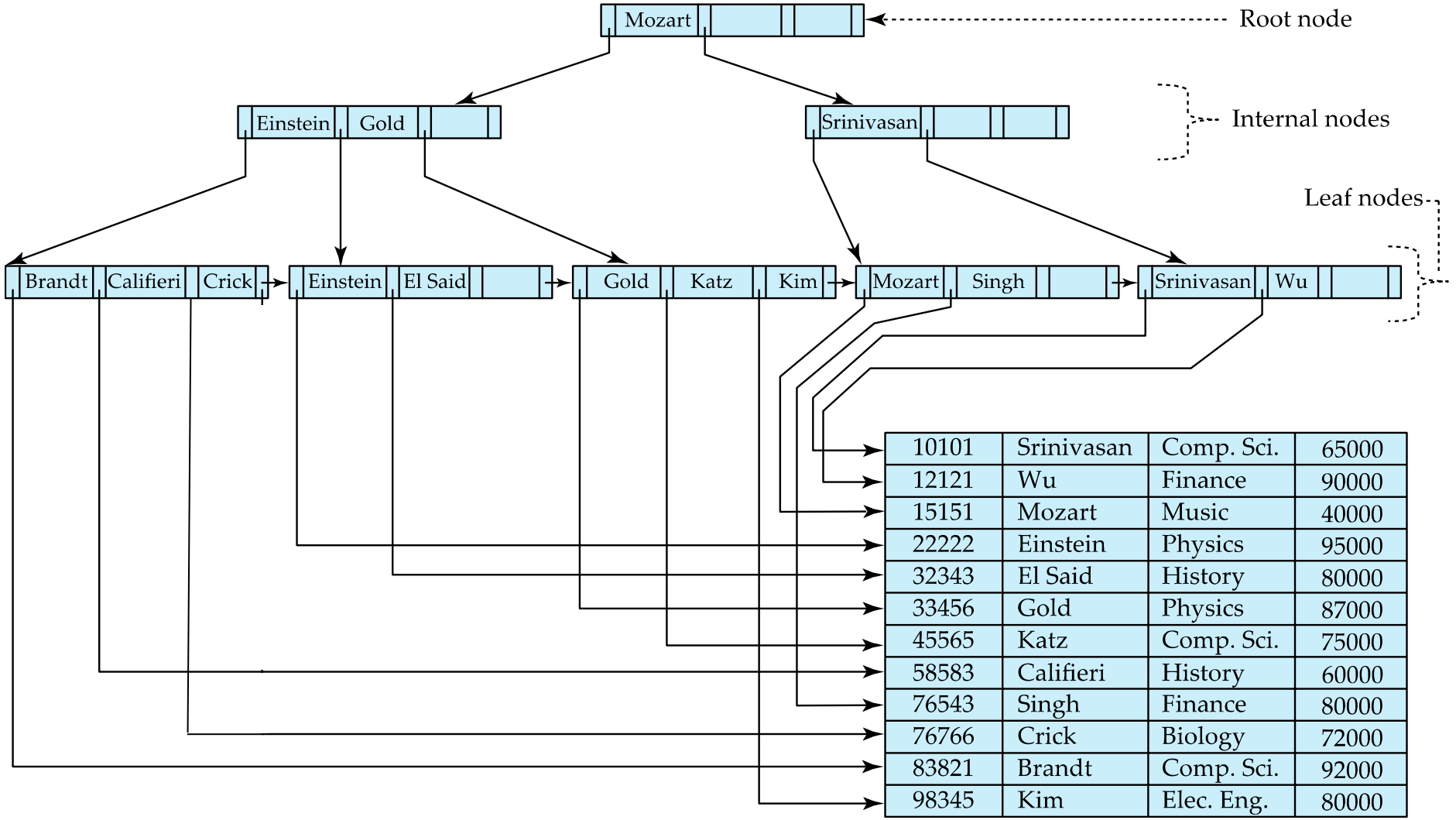
Multilevel Index (Cont.)



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*)

Example of B+-Tree



 ©Silberschatz, Korth and Sudarshan

B^+-Tree Index Files (Cont.)

A B^+-tree is a rooted tree satisfying the following properties:

- All paths from the root to a 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.

©Silberschatz, Korth and Sudarshan

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

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

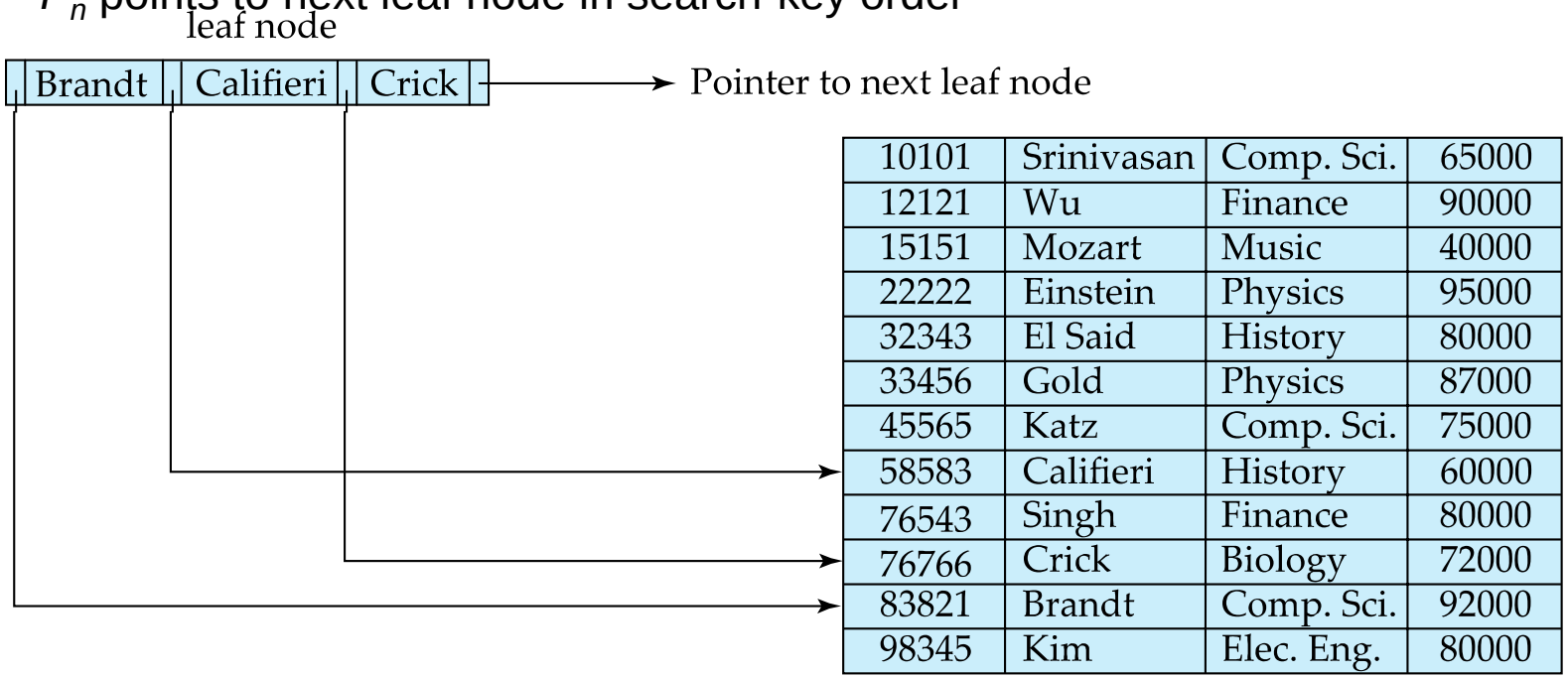
(Initially assume no duplicate keys, address duplicates later)

©Silberschatz, Korth and Sudarshan

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

©Silberschatz, Korth and Sudarshan

Non-Leaf Nodes in B+-Trees

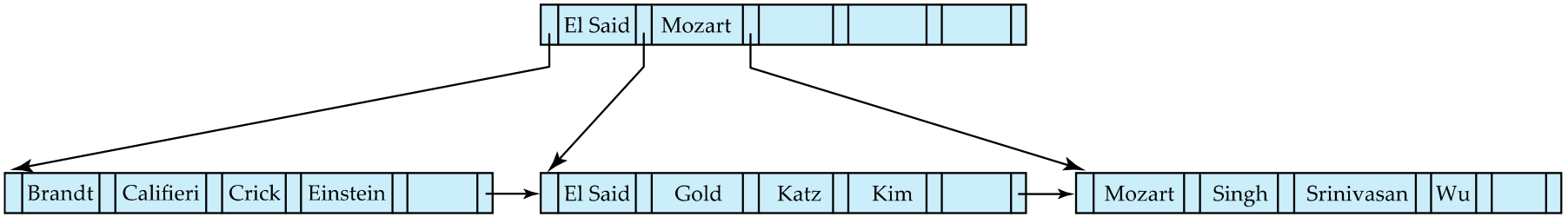
- Non-leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with n pointers:
  - All the search keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search keys in the subtree to which P_i points have values greater than or equal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

©Silberschatz, Korth and Sudarshan

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.

©Silberschatz, Korth and Sudarshan

Observations about B^+-trees

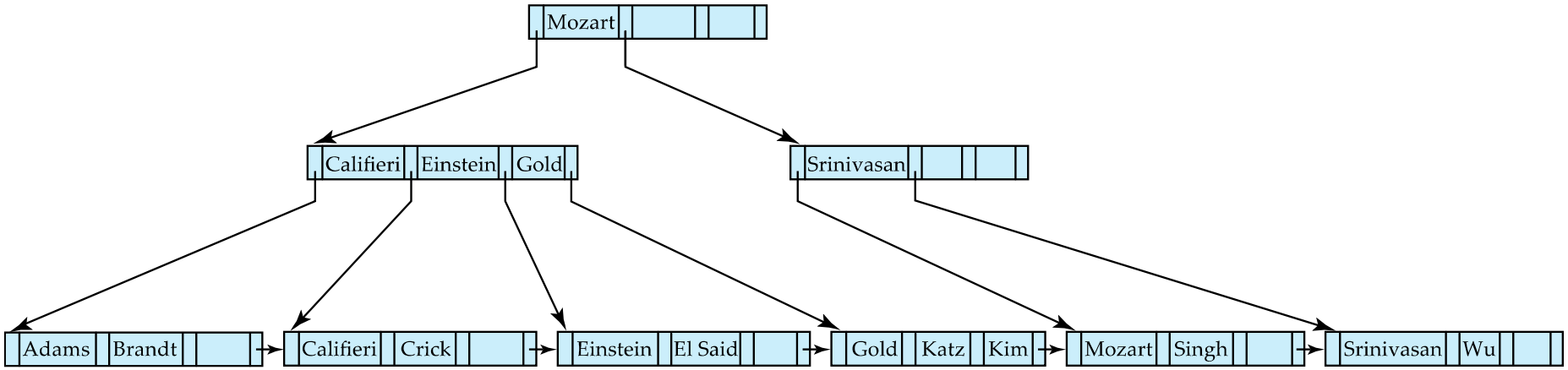
- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
  - Level below root has at least $2* \lceil n/2 \rceil$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

©Silberschatz, Korth and Sudarshan

Queries on B+-Trees

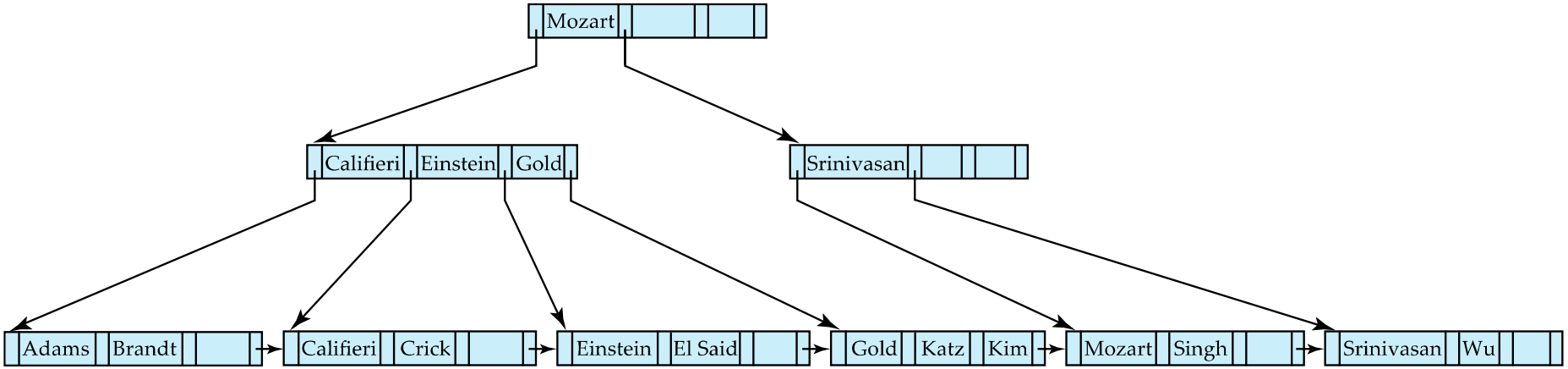
function *find*(*v*)

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

©Silberschatz, Korth and Sudarshan

Non-Unique Keys

- If a search key a_i is not unique, create instead an index on a composite key (a_i , A_p), which is unique
  - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on a composite key, with range (v, - ∞) to (v, + ∞)
- But more I/O operations are needed to fetch the actual records
  - If the index is clustering, all accesses are sequential+
  - If the index is non-clustering, each record access may need an I/O operation

©Silberschatz, Korth and Sudarshan

Updates on B+-Trees: Insertion

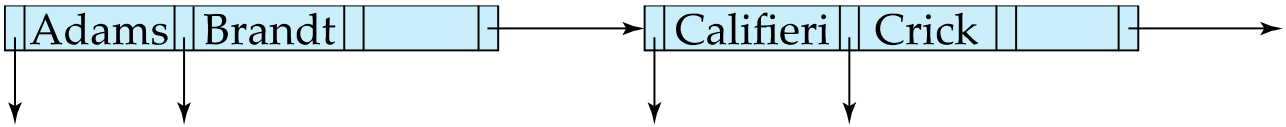
Assume a record is already added to the file. Let

- *pr* be a 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.

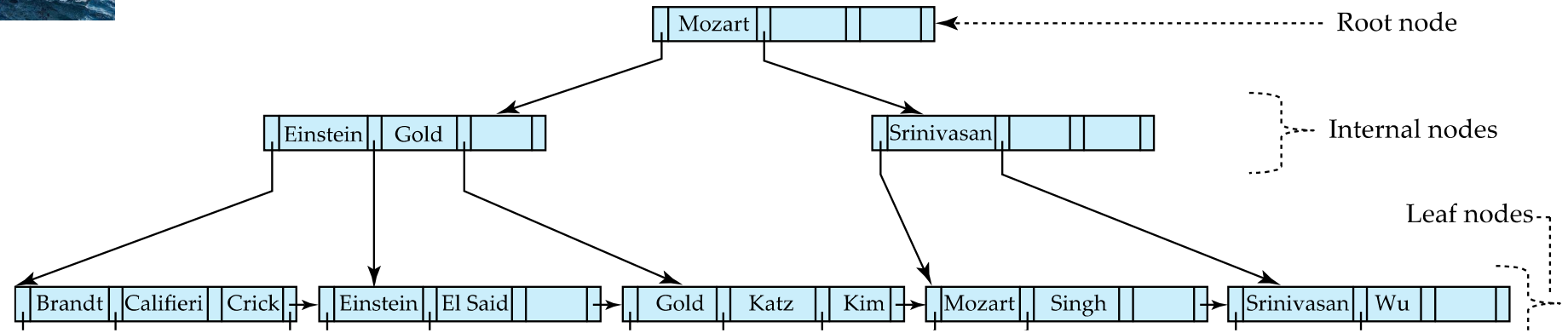
Updates on B+-Trees: Insertion (Cont.)

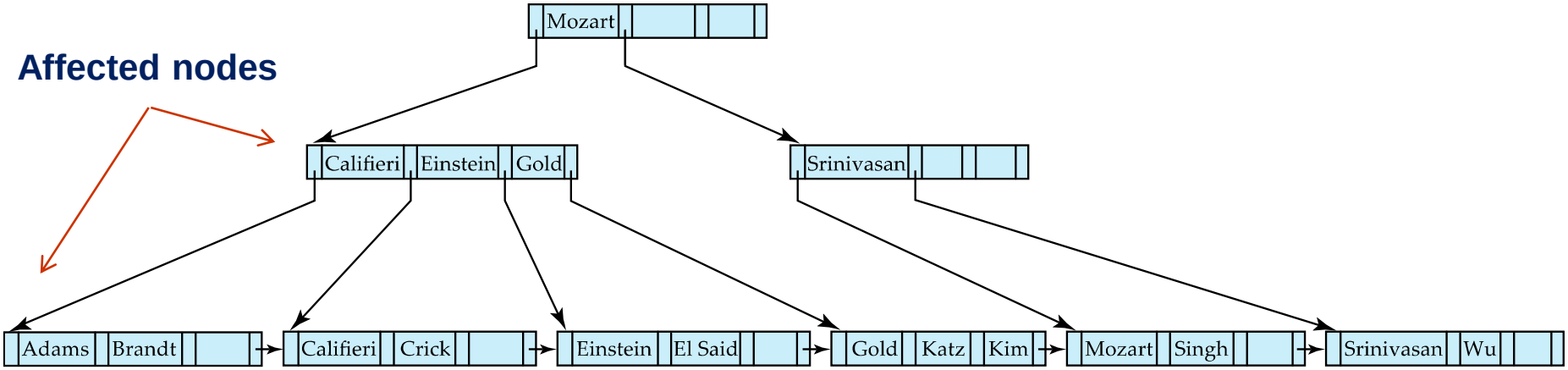
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*)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

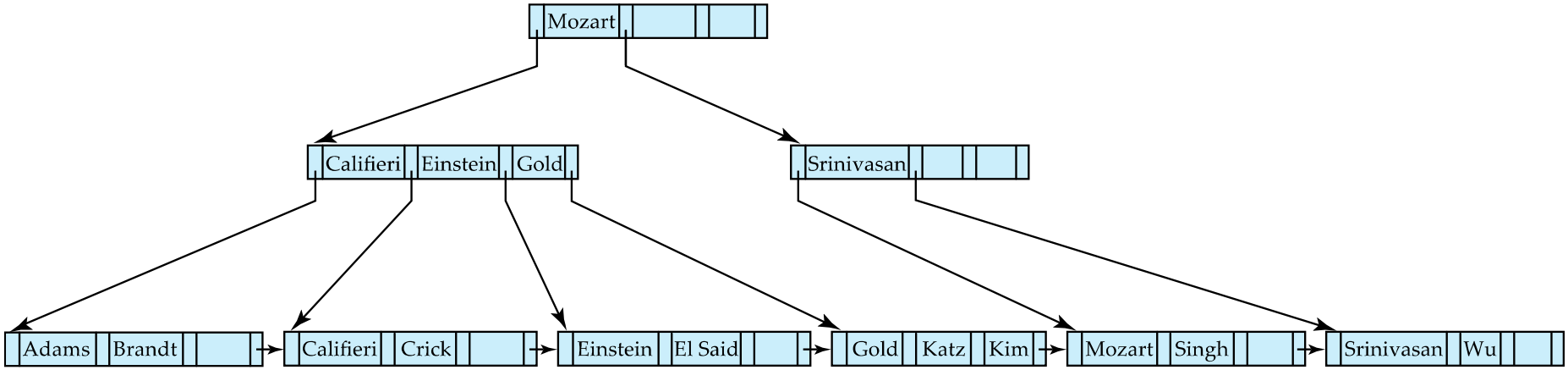
B+-Tree Insertion





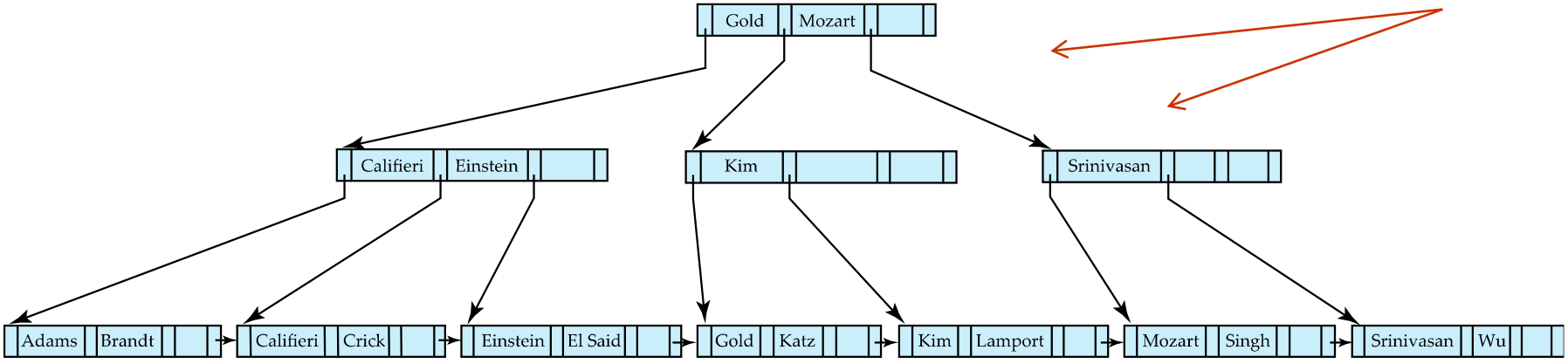
B+-Tree before and after insertion of “Adams”

B+-Tree Insertion



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

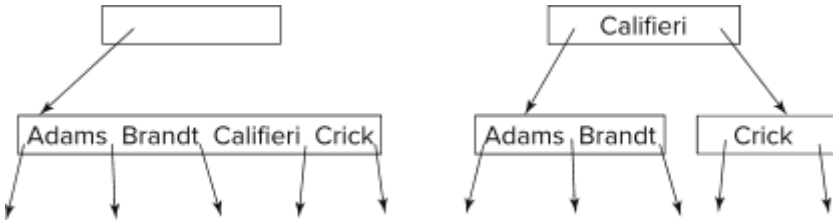
Affected nodes



Affected nodes

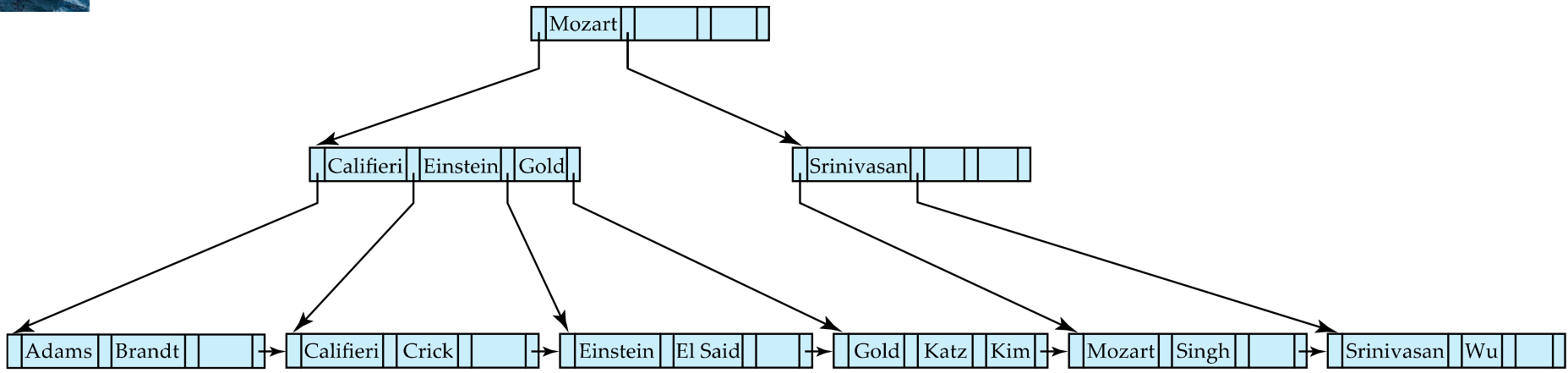
Insertion in B+-Trees (Cont.)

- 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

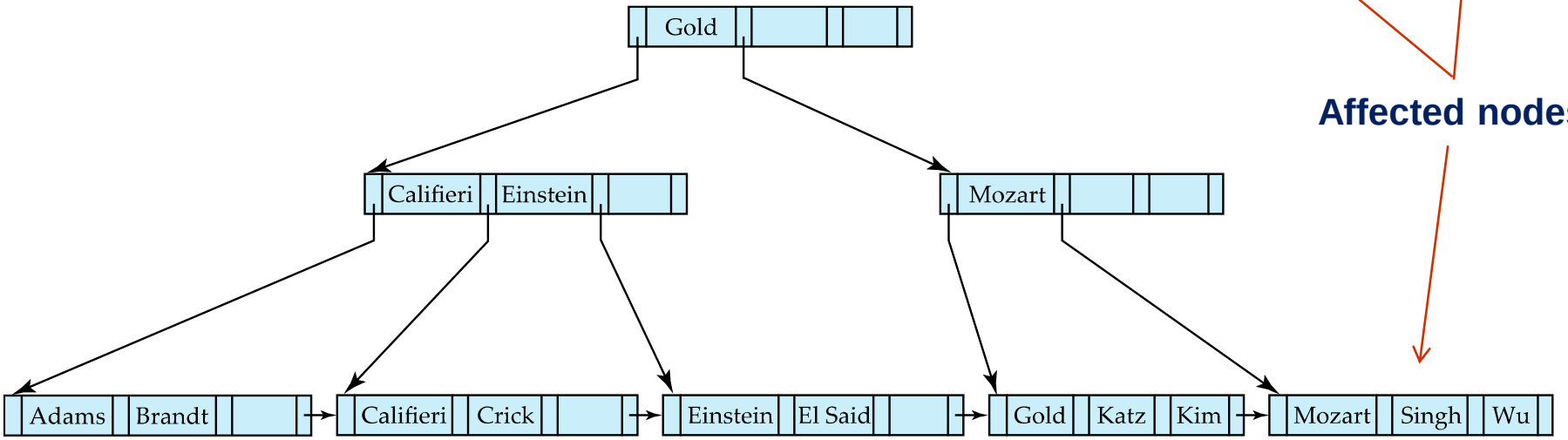


- **Read pseudocode in book!**

Examples of B+-Tree Deletion



Before and after deleting "Srinivasan"

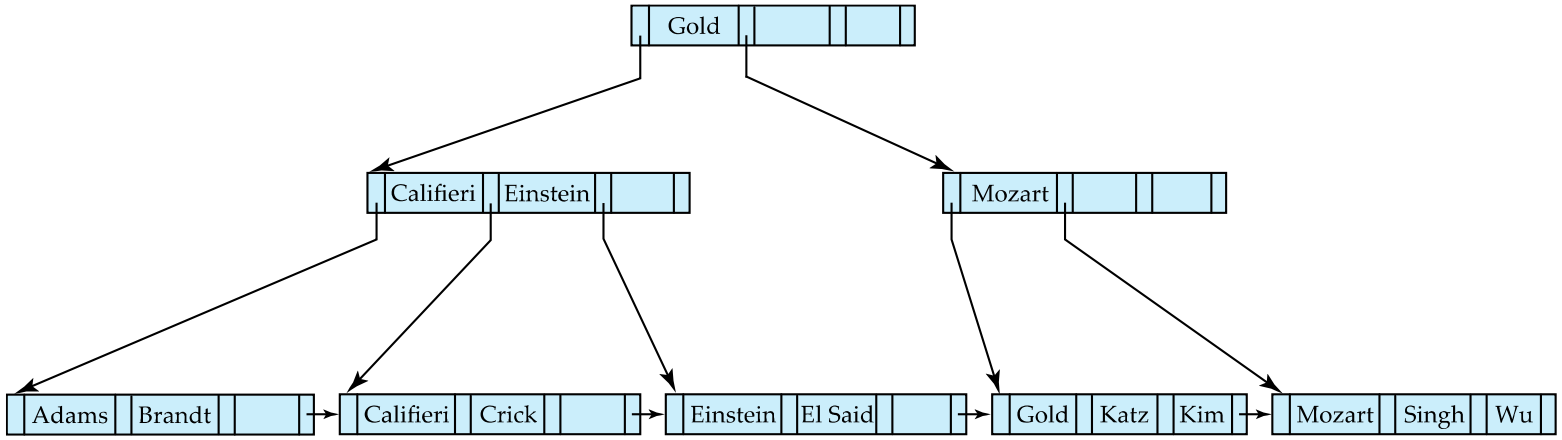


Affected nodes

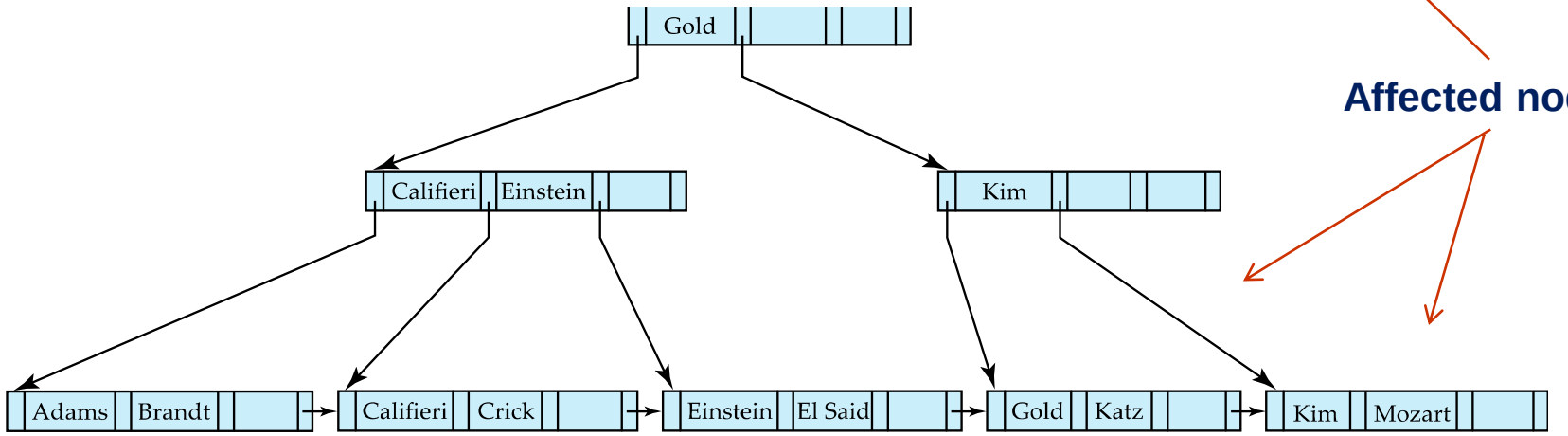
- Deleting "Srinivasan" causes **merging** of under-full leaves

©Silberschatz, Korth and Sudarshan

Examples of B+-Tree Deletion (Cont.)

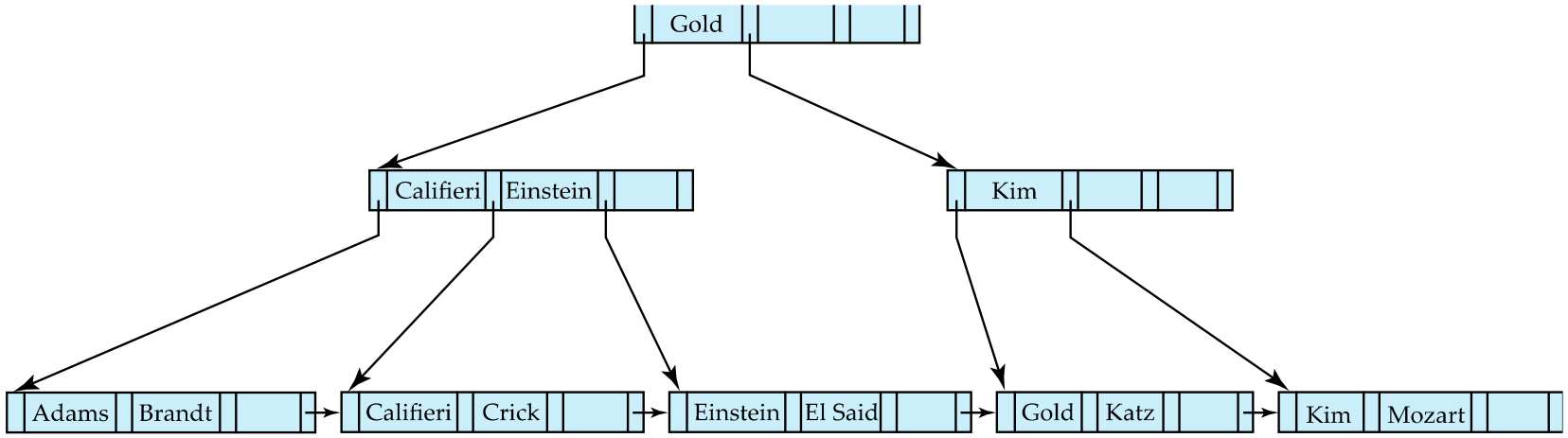


Before and after deleting "Singh" and "Wu"

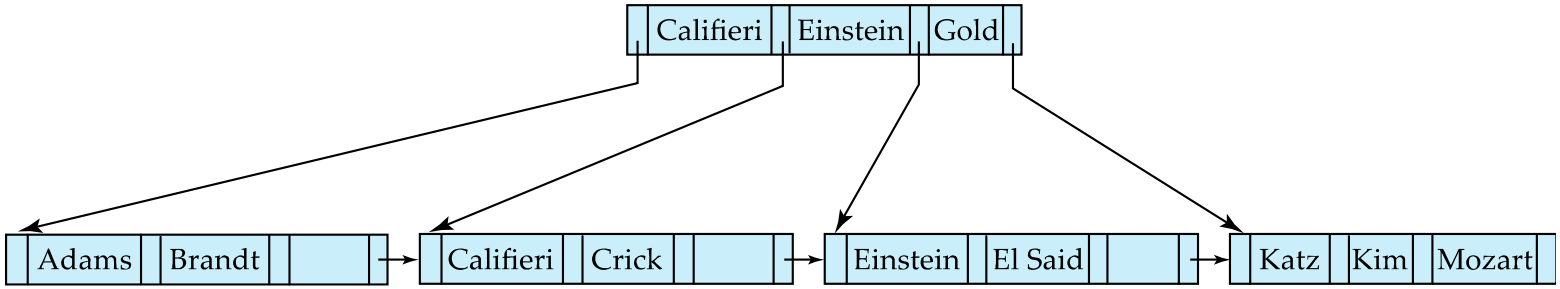


- Leaf containing Singh and Wu became underfull and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result

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

 ©Silberschatz, Korth and Sudarshan

Updates on B$^+$-Trees: Deletion

Assume a record is already deleted from the 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.

©Silberschatz, Korth and Sudarshan

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 that 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.

©Silberschatz, Korth and Sudarshan

Complexity of Updates

- Cost (in terms of the number of I/O operations) of insertion and deletion of a single entry proportional to the 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 a buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on the insertion order
  - 2/3rds with random, much more with insertion in sorted order

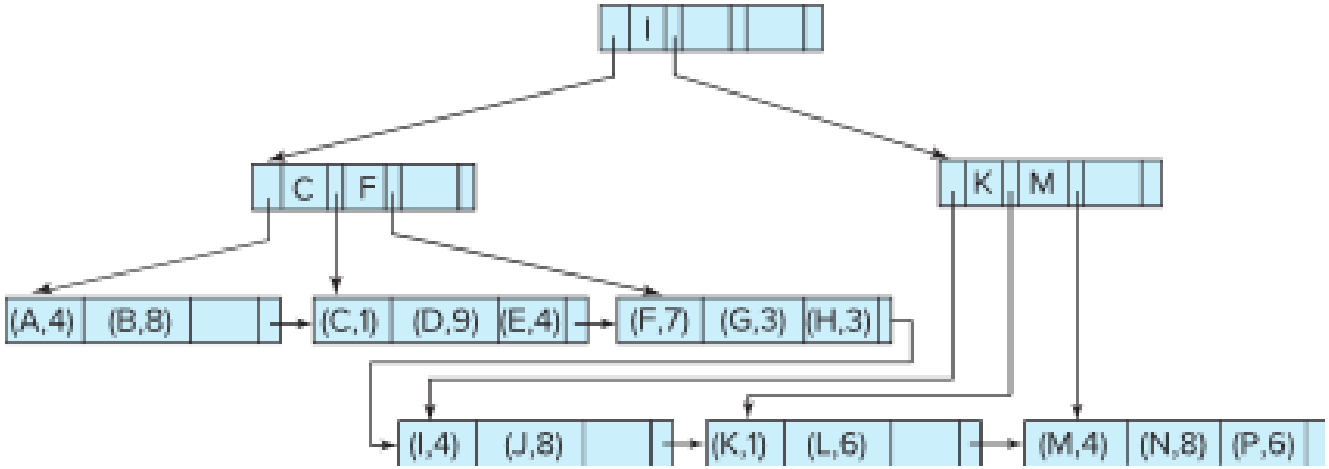
©Silberschatz, Korth and Sudarshan

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 non-leaf 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
  - Involving 2 siblings in redistribution (to avoid split/merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

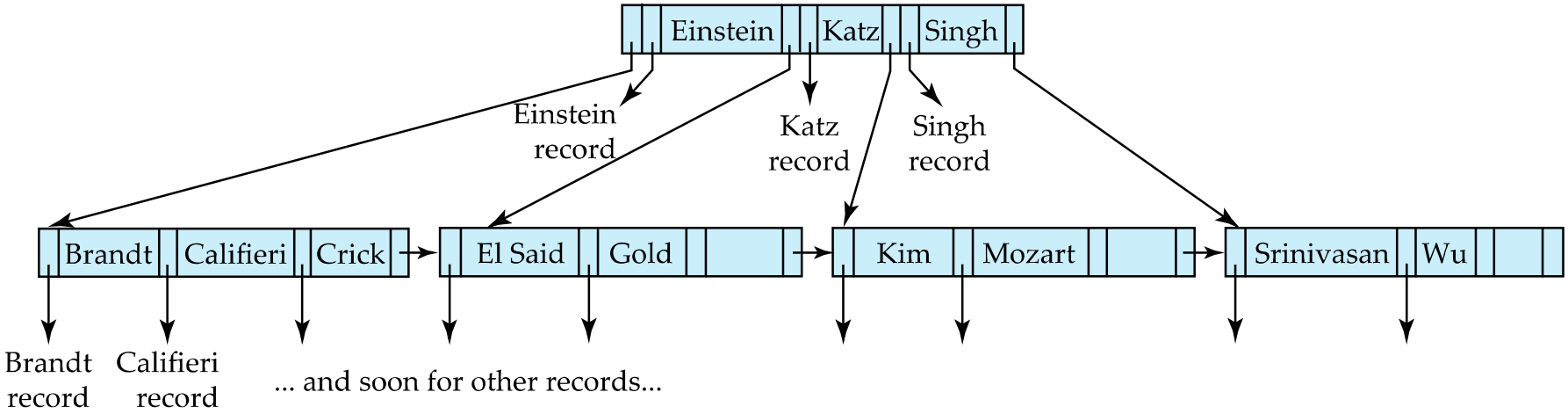
©Silberschatz, Korth and Sudarshan

Bulk Loading and Bottom-Up Build

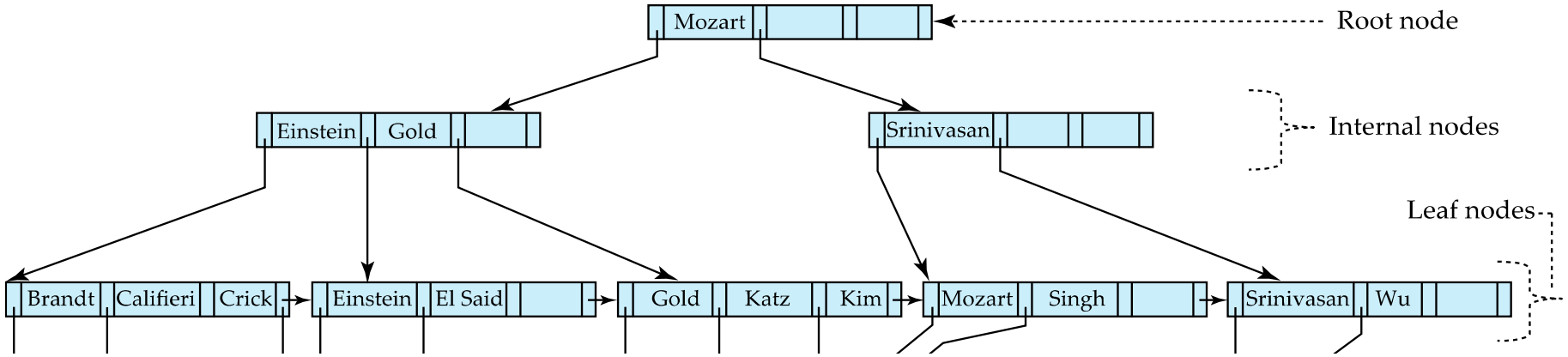
- Inserting entries one-at-a-time into a B^+-tree requires $\geq$ 1 IO per entry
  - assuming leaf level does not fit in memory
  - can be very inefficient for loading a large number of entries at a time (**bulk loading**)
- Efficient alternative 1:
  - sort entries first (using efficient external-memory sort algorithms to be discussed later)
  - insert in sorted order
    - insertion will go to an 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

©Silberschatz, Korth and Sudarshan

B-Tree Index File Example



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



Hashing

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

©Silberschatz, Korth and Sudarshan

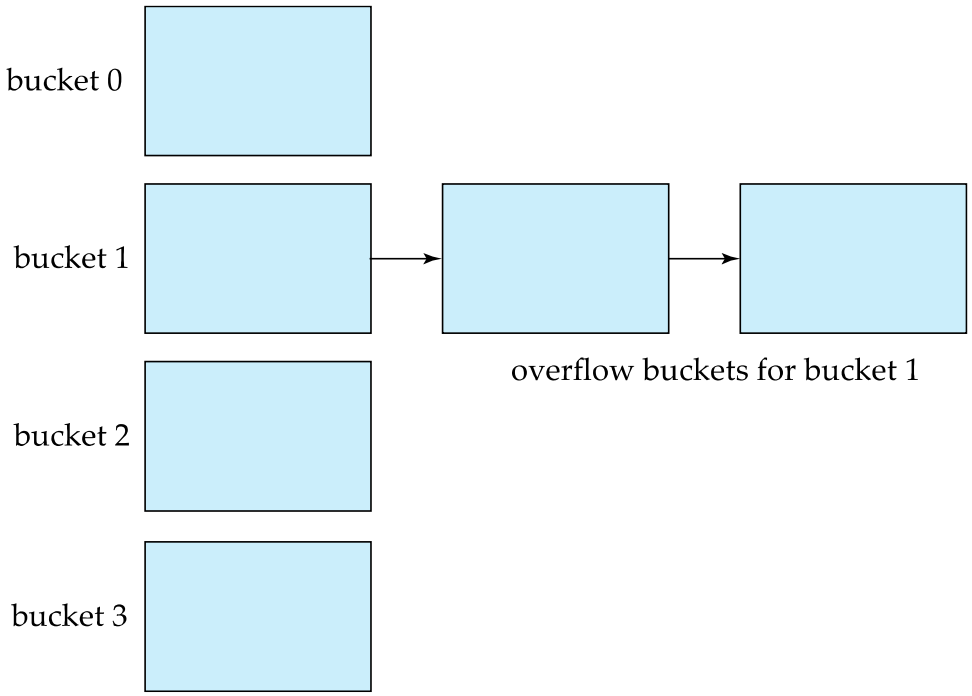
Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient bucket capacity
  - Skew in the distribution of records. This can occur due to two reasons:
    - Multiple records have the same search-key value
    - Chosen hash function produces a 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***.

©Silberschatz, Korth and Sudarshan

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 over-flow buckets, is not suitable for database applications.



 ©Silberschatz, Korth and Sudarshan

Example of Hash File Organization

Hash file organization of *instructor* file, using *dept_name* as key.

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If the initial number of buckets is too small, and the file grows, performance will degrade due to too many overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If the database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

©Silberschatz, Korth and Sudarshan

Dynamic Hashing

- Periodic rehashing
  - If the number of entries in a hash table becomes (say) 1.5 times the size of the hash table,
    - Create a new hash table of size (say) 2 times the size of the previous hash table
    - Rehash all entries to the new table
- Linear Hashing
  - Do rehashing in an incremental manner
- Extendable Hashing
  - Tailored to disk-based hashing, with buckets shared by multiple hash values
  - Doubling of # of entries in the hash table, without doubling # of buckets

©Silberschatz, Korth and Sudarshan

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

©Silberschatz, Korth and Sudarshan

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 queries 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 the intersection of both sets of pointers obtained.

©Silberschatz, Korth and Sudarshan

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

Creation of Indices

- Example
  create index *takes_pk* **on** *takes* (*ID,course_ID, year, semester, section*)
  drop index *takes_pk*
- Most database systems allow specification of type of index, and clustering.
- Indices on primary key created automatically by all databases
  - Why?
- Some databases also create indices on foreign key attributes
  - Why might such an index be useful for this query:
    - $takes \bowtie \sigma_{name='Shankar'}(student)$
- Indices can greatly speed up lookups, but impose cost on updates
  - Index tuning assistants/wizards supported on several databases to help choose indices, based on query and update workload

©Silberschatz, Korth and Sudarshan

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.
  - 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.

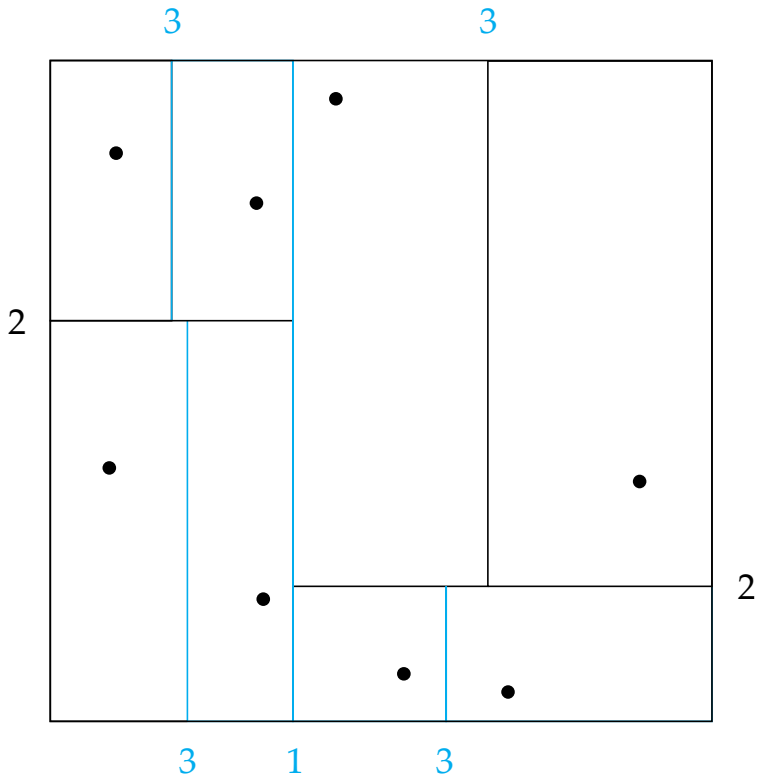
Spatial and Temporal Indices

Spatial Data

- Databases can store data types such as lines, polygons, in addition to raster images
  - allows relational databases to store and retrieve spatial information
  - Queries can use spatial conditions (e.g. contains or overlaps).
  - queries can mix spatial and nonspatial conditions
- **Nearest neighbor queries**, given a point or an object, find the nearest object that satisfies given conditions.
- **Range queries** deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.
- **Spatial join** of two spatial relations with the location playing the role of join attribute.

Indexing of Spatial Data

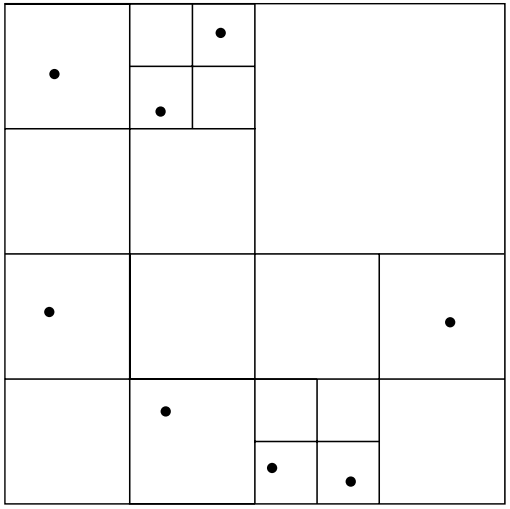
- **k-d tree** - early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - Choose one dimension for partitioning at the root level of the tree.
  - Choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.
- Partitioning stops when a node has less than a given number of points.



- The **k-d-B tree** extends the *k-d* tree to allow multiple child nodes for each internal node; well-suited for secondary storage.

Division of Space by Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have between zero and some fixed maximum number of points (set to 1 in example).

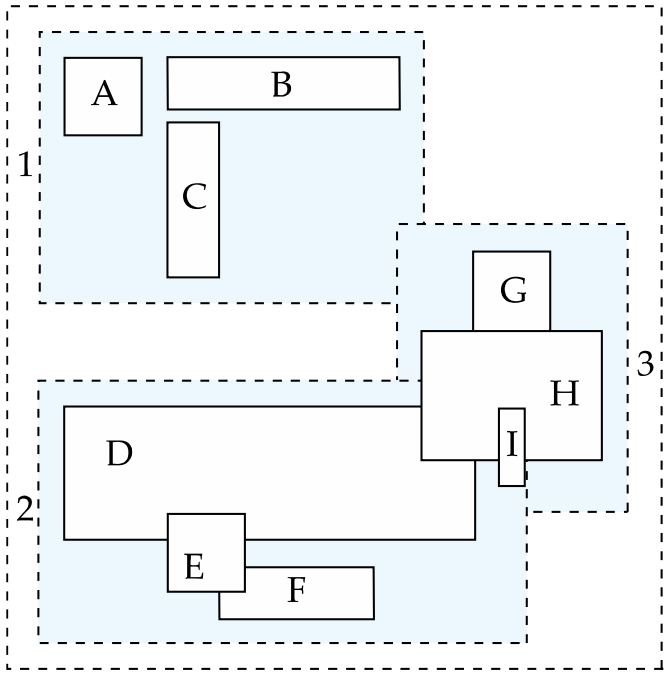


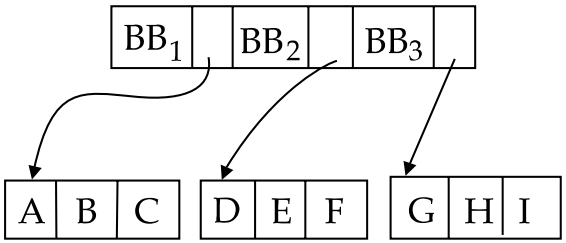
R-Trees

- **R-trees** are a N-dimensional extension of B^+-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R^+ -trees and R*-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an
  N-dimensional interval, that is, an N-dimensional rectangle.
- Will consider only the two-dimensional case ($N = 2$)
  - generalization for $N > 2$ is straightforward, although R-trees work well only for relatively small N
- The **bounding box** of a node is a minimum sized rectangle that contains all the rectangles/polygons associated with the node
  - *Bounding boxes of children of a node are allowed to overlap*

Example R-Tree

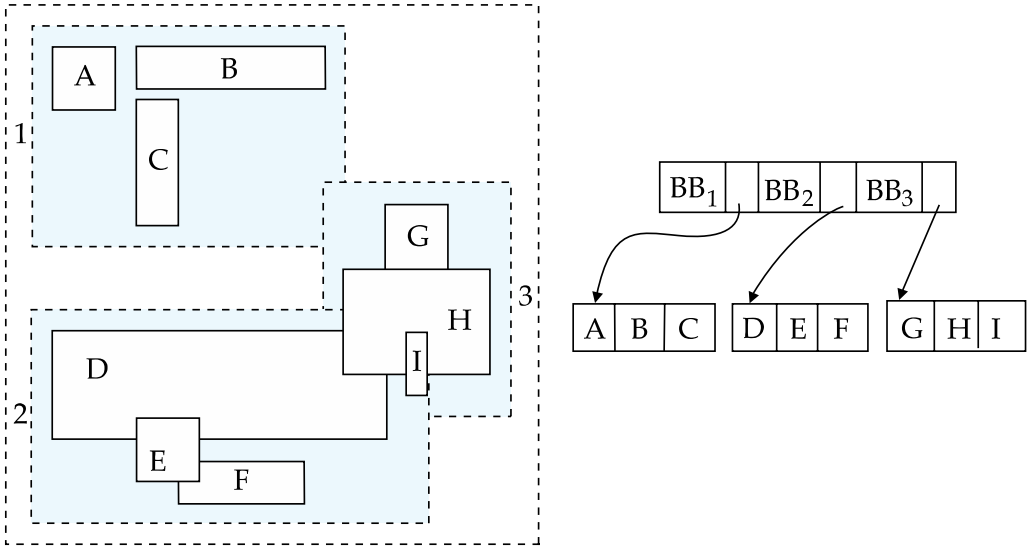
- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of an R-tree for the rectangles.
- The R-tree is shown on the right.





Search in R-Trees

- To find data items intersecting a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box intersects the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be searched, but works acceptably in practice.



Indexing Temporal Data

- Temporal data refers to data that has an associated time period (interval)
  - Example: a temporal version of the *course* relation

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

- Time interval has a start and end time
  - End time set to infinity (or large date such as 9999-12-31) if a tuple is currently valid and its validity end time is not currently known
- Query may ask for all tuples that are valid at a point in time or during a time interval
  - Index on valid time period speeds up this task

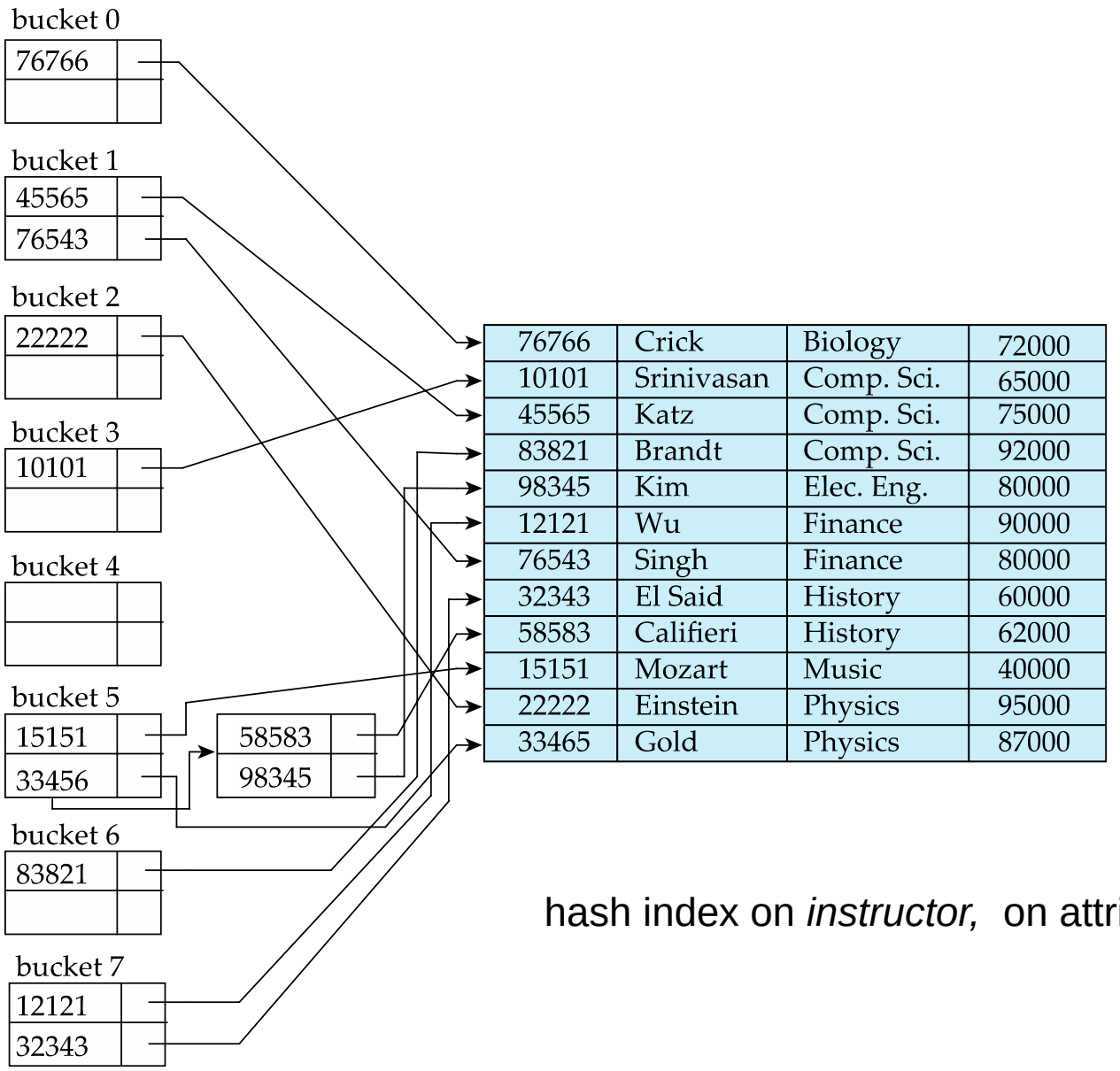
Indexing Temporal Data (Cont.)

- To create a temporal index on attribute *a*:
  - Use spatial index, such as R-tree, with attribute *a* as one dimension, and time as another dimension
    - Valid time forms an interval in the time dimension
  - Tuples that are currently valid cause problems, since the value is infinite or very large
    - Solution: store all current tuples (with the end time as infinity) in a separate index, indexed on (*a, start-time*)
      - To find tuples valid at a point in time *t* in the current tuple index, search for tuples in the range (*a, 0*) to (*a,t*)
- Temporal index on primary key can help enforce the temporal primary key constraint

course_id	*title*	*dept_name*	*credits*	*start*	*end*
BIO-101	Intro. to Biology	Biology	4	1985-01-01	9999-12-31
CS-201	Intro. to C	Comp. Sci.	4	1985-01-01	1999-01-01
CS-201	Intro. to Java	Comp. Sci.	4	1999-01-01	2010-01-01
CS-201	Intro. to Python	Comp. Sci.	4	2010-01-01	9999-12-31

End of Chapter 9

Example of Hash Index



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