# Chapter 11: Indexing and Hashing

**Database System Concepts, 6th Ed.**

# Basic Concepts

- Indexing mechanisms used to speed up access to desired data
  - e.g., author catalog in library, term index at the end of a book
- **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

| | |
|---|---|
| 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
  - **Point query**: records with a specified value in the attribute
  - **Range query**: records with an attribute value falling in a specified range of values

- Access time
- Insertion time
- Deletion time

- Space overhead

# Ordered Indices

- **Ordered index**: index entries are stored sorted on the search key value
  - e.g., author catalog in library
- **Primary index**: an 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

| | |
|---|---|
| 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 | |

# Primary Indices Example

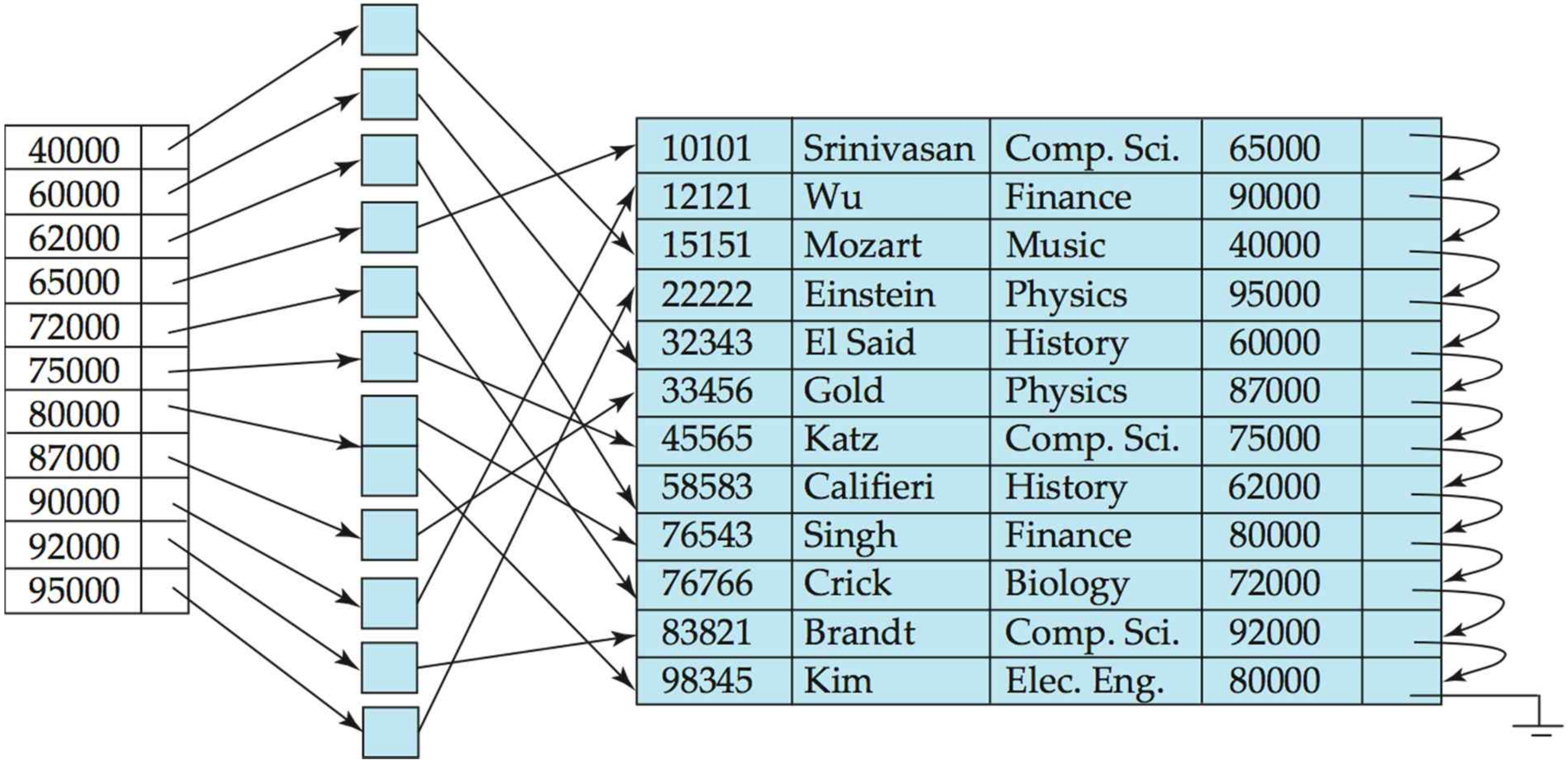
- Example: dense index on *dept_name*, with *instructor* file sorted on *dept_name*

| | |
|---|---|
| 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 | |

# Secondary Indices Example

- 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




**Secondary index on *salary* field of *instructor***

# Sparse Index Files

- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions
  - Generally slower than dense index for locating records

| | |
|---|---|
| 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:  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.
- **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,
    - Replace 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
    - Delete the entry

| | |
|---|---|
| 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

- Perform a lookup using the search-key value appearing in the record to be inserted

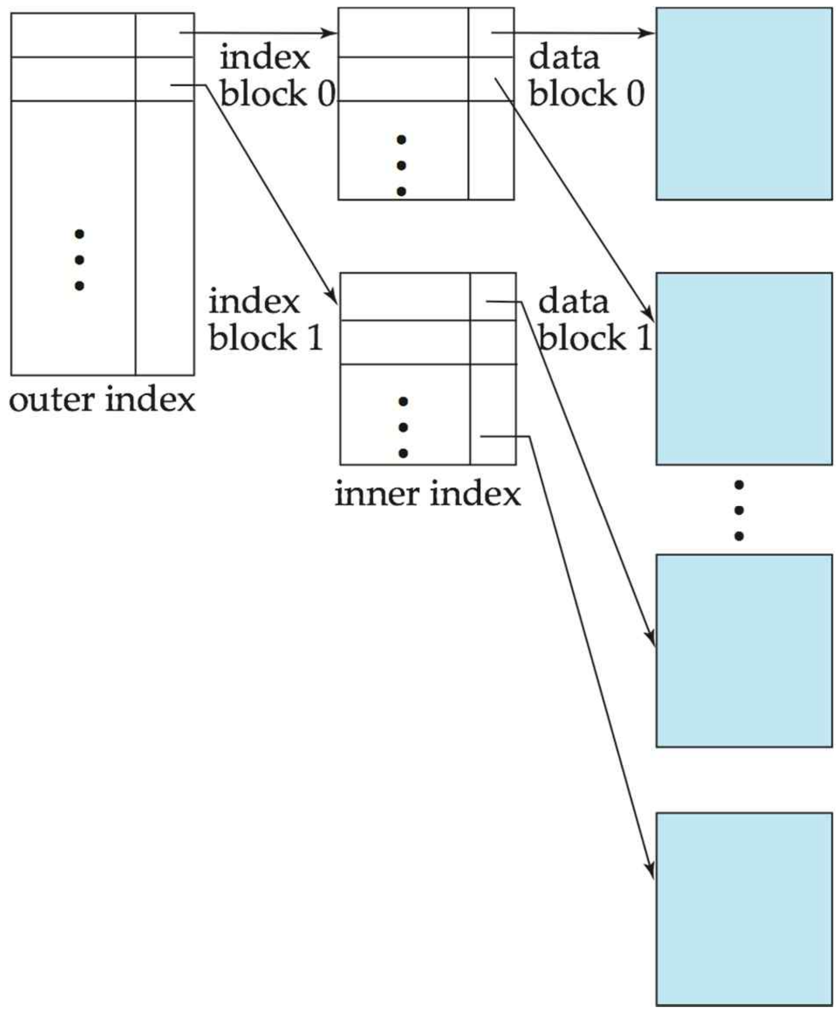
- **Dense indices**
  - If the search-key value does not appear in the index, insert it.
- **Sparse indices**
  - If index stores an entry for each block of the file
    - No change needs to be made to the index, unless a new block is created
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index

# Primary and Secondary Indices

- Indices offer substantial benefits when searching for records
- When a file is modified, every index on the file must be updated
  - Updating indices imposes overhead on database modification
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

# Multilevel Index

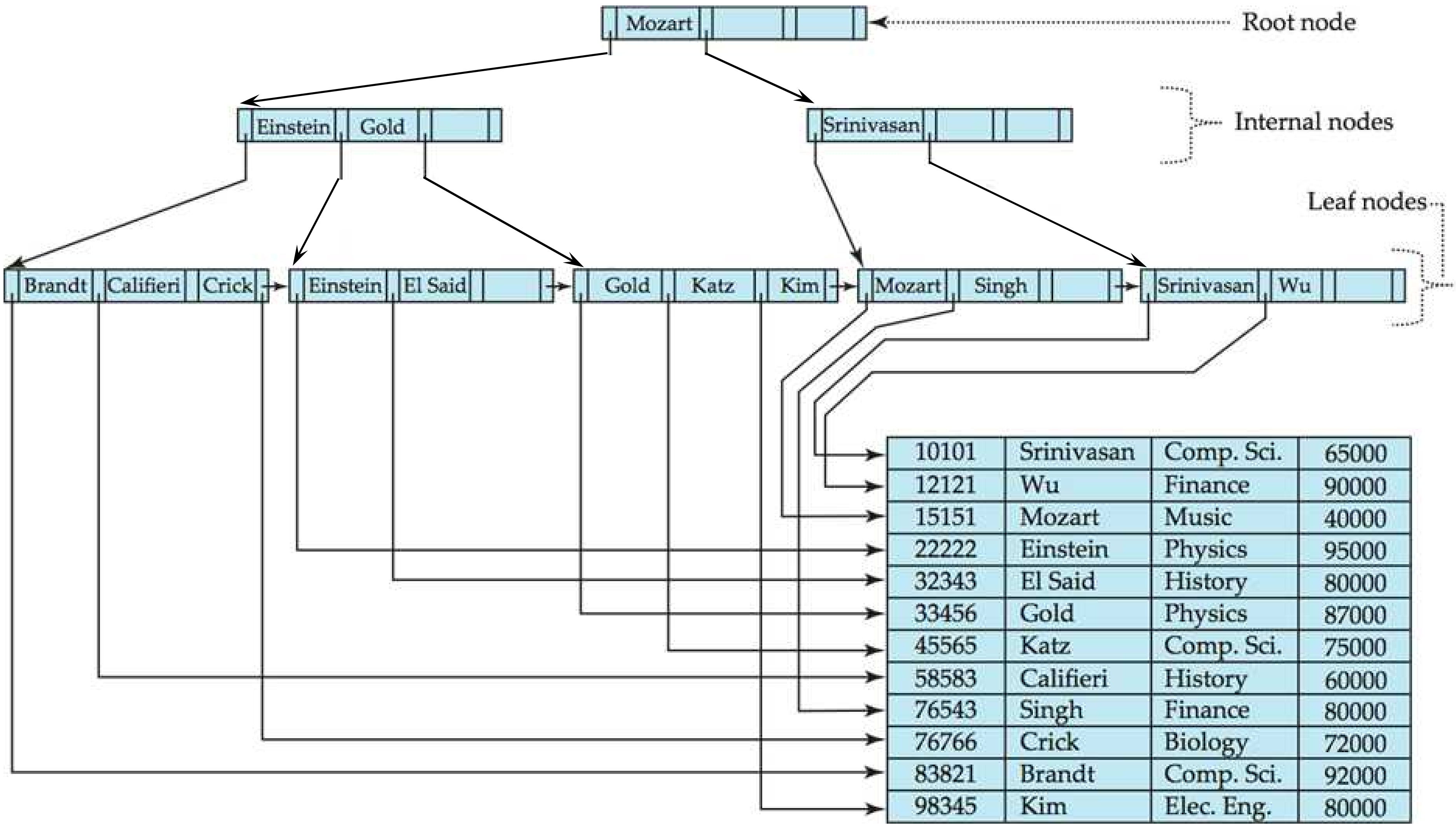
- If primary index does not fit in memory, access becomes expensive
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit 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



# $B^+$-Tree Index Files

- 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
- $B^+$-trees are used extensively
  - Advantages of $B^+$-trees outweigh disadvantages

# Example of B+-Tree




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

# $B^+$-Tree Index Files (Cont.)

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

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and $n$ children
- A leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Special cases: root node
  - 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 (non-leaf 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 (for leaf nodes)

- The search-keys in a node are ordered

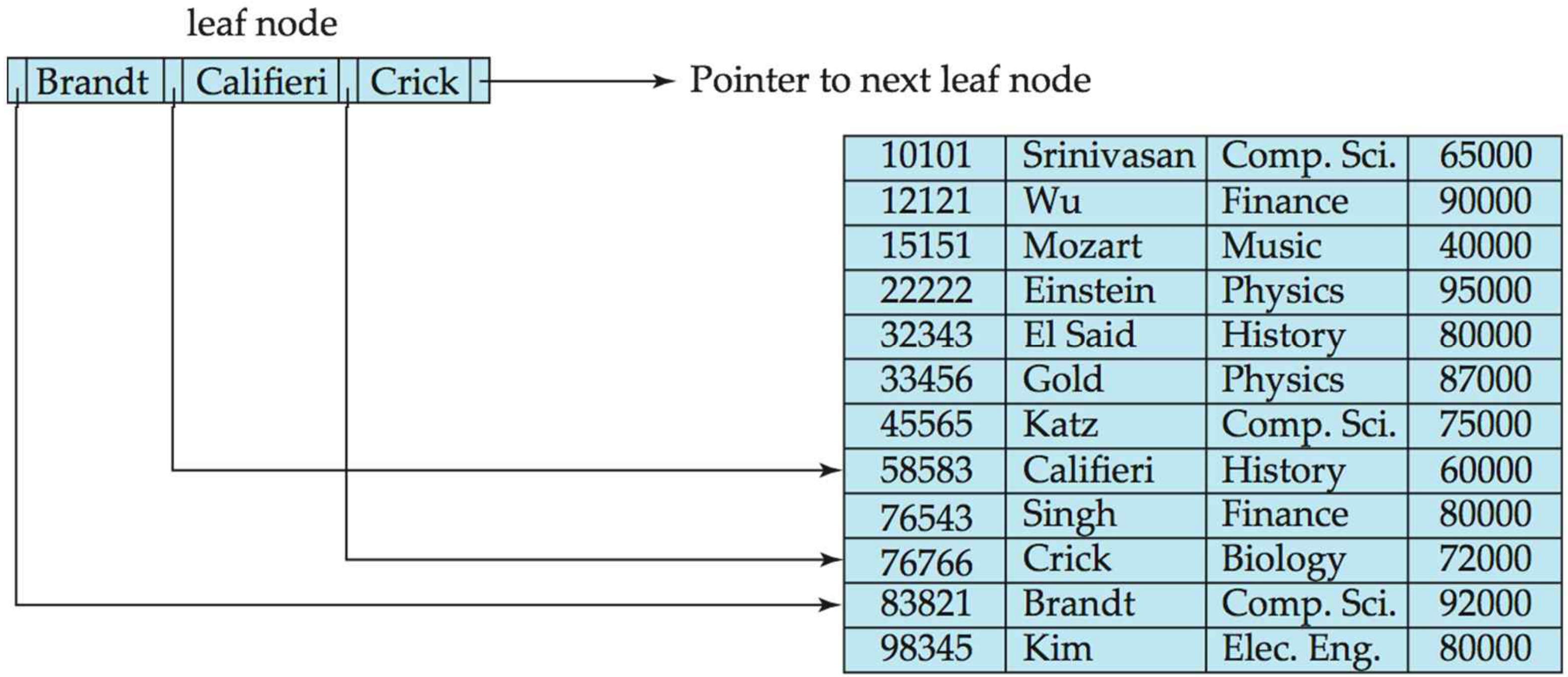
$$K_1 < K_2 < K_3 < . . . < K_{n-1}$$

  - We assume no duplicate keys

# Leaf Nodes in B$^+$-Trees

- For $i$ = 1, 2, . . ., $n$–1, pointer $P_i$ points to a file record with search-key value $K_i$
- $P_n$ points to next leaf node in search-key order

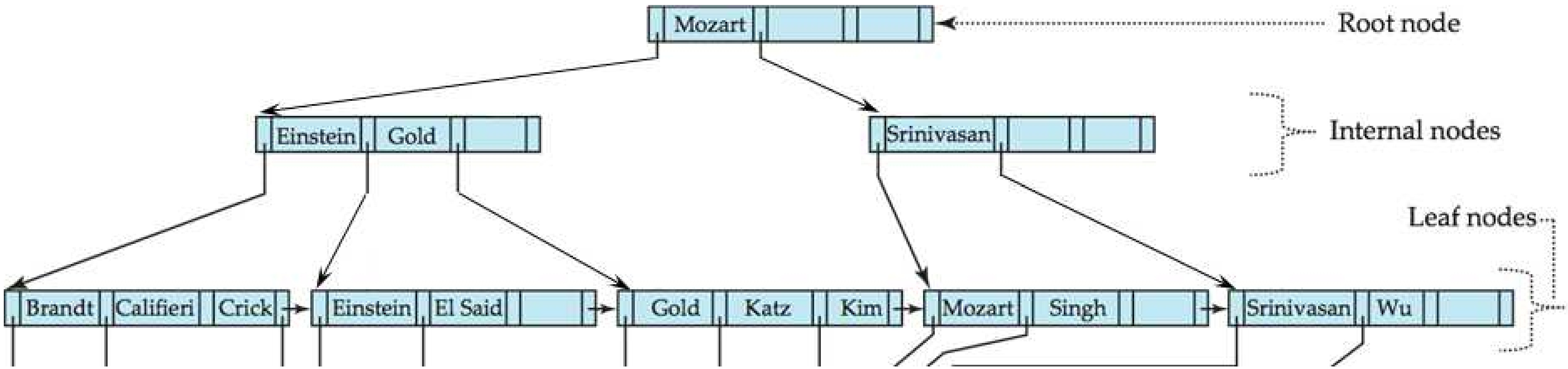
- If $L_i$, $L_j$ are leaf nodes and $i < j$, $L_i$'s search-key values are less than or equal to $L_j$'s search-key values



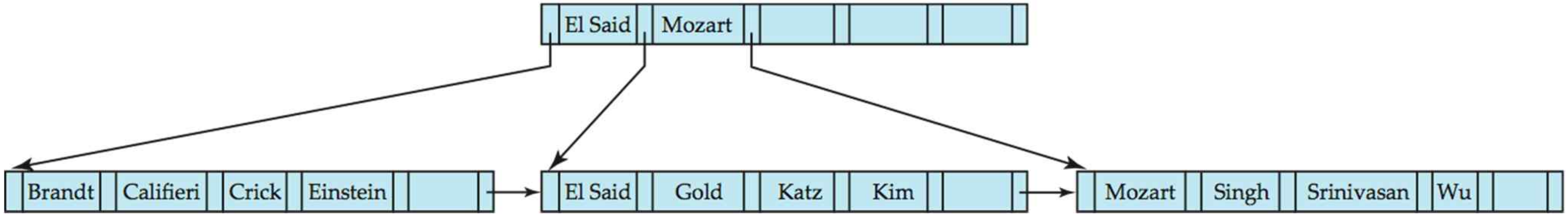

| | | | |
|---|---|---|---|
| 10101 | Srinivasan | Comp. Sci. | 65000 |
| 12121 | Wu | Finance | 90000 |
| 15151 | Mozart | Music | 40000 |
| 22222 | Einstein | Physics | 95000 |
| 32343 | El Said | History | 80000 |
| 33456 | Gold | Physics | 87000 |
| 45565 | Katz | Comp. Sci. | 75000 |
| 58583 | Califieri | 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 ($m \leq n$):
  - All the search-keys in the subtree to which $P_1$ points are less than $K_1$
  - For $2 \leq i \leq m - 1$, 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 $P_m$ points have values greater than or equal to $K_{m-1}$



# 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

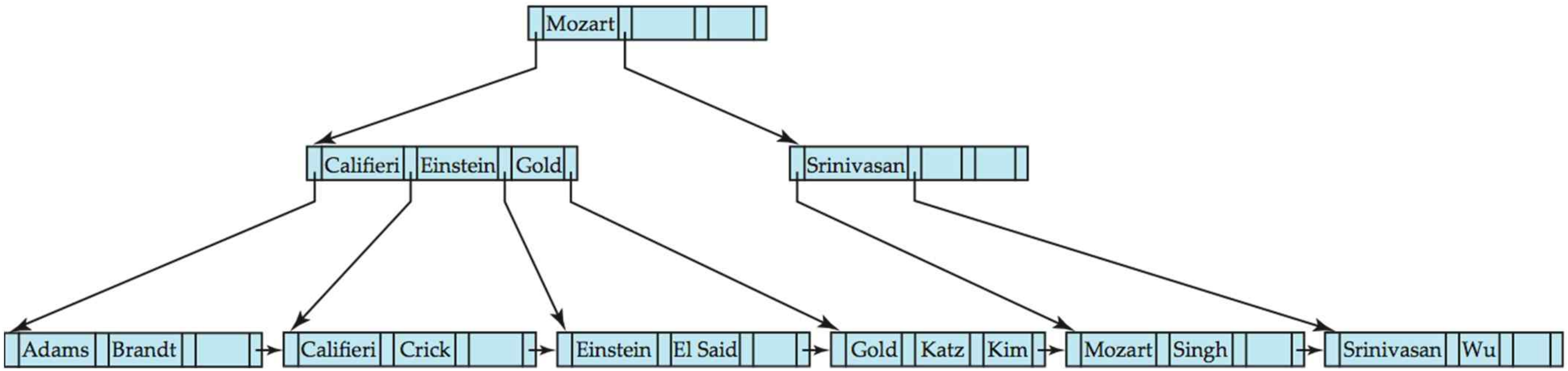
# Observations about $B^+$-trees

- 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. $C = root$
  2. While C is not a leaf node {
     1. Let $i$ be least value s.t. $V \le K_i$.
     2. If no such exists, set $C$ = *last non-null pointer in C*
     3. Else { if ($V = K_i$ ) set C = $P_{i+1}$
        else set $C = P_i$ }

     }
  3. Let $i$ be least value s.t. $K_i = V$
  4. If there is such a value $i$, follow pointer $P_i$ to the desired record
  5. Else no record with search-key value $k$ exists



# Queries on B⁺-Trees (Cont.)

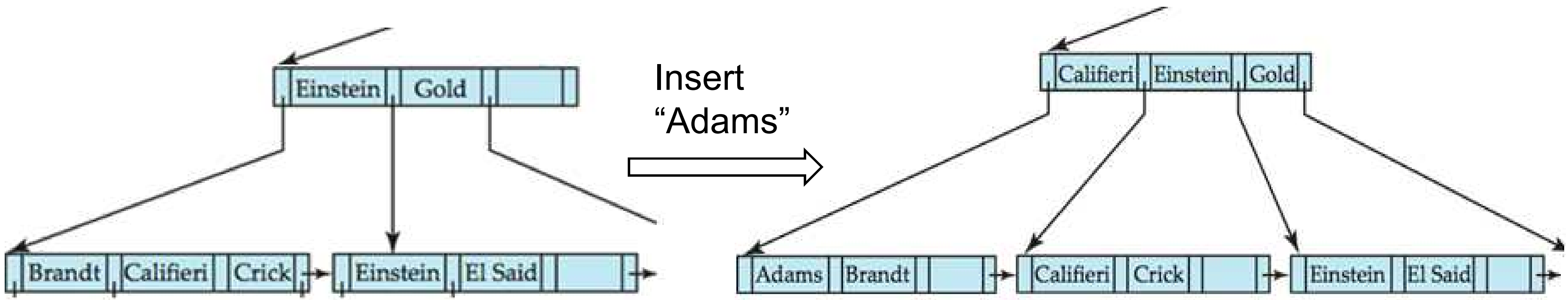
- If there are $K$ search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
- A node is generally the same size as a disk block, typically 4 kilobytes
  - $n$ is typically around 100 (40 bytes per index entry)
- With 1 million search key values 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
  - 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. Add the record to the file
3. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
4. Otherwise, split the node (along with the new (key-value, pointer) entry)

# Leaf Node Split in B$^+$-Trees

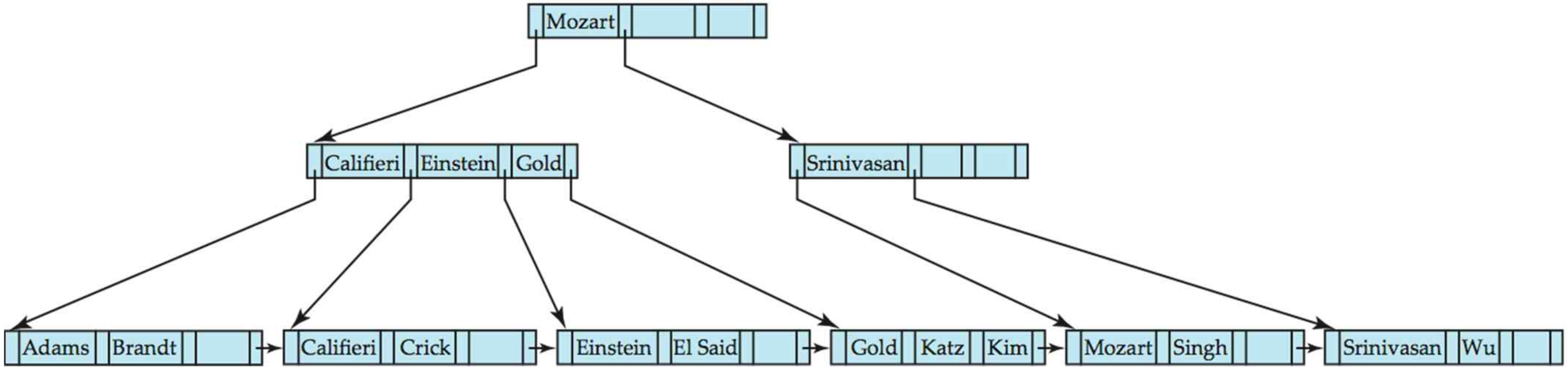
- Splitting a leaf node:
  - Take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order
    - Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node
    - Make the original node point to the new node
  - Let the new node be *p,* and 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



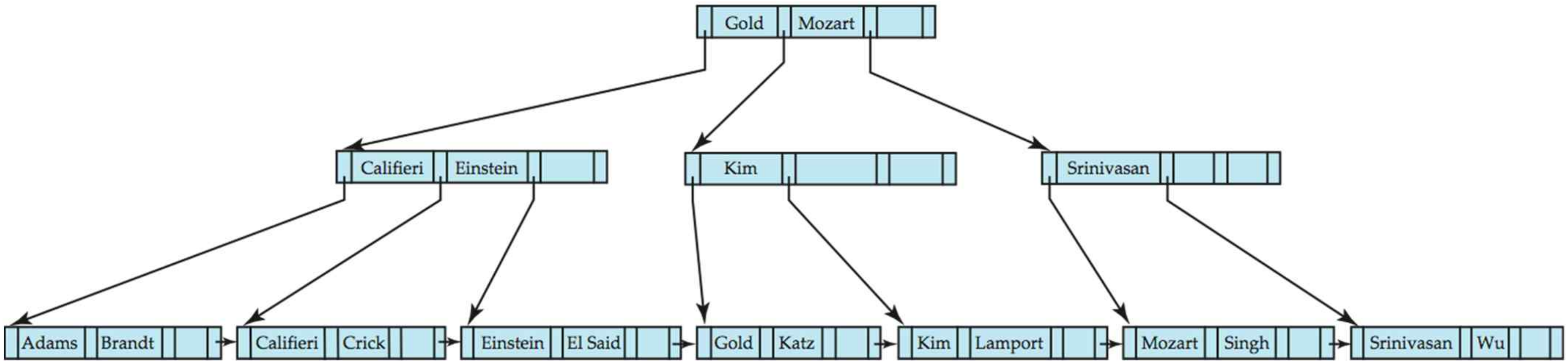
# Non-Leaf Node Split in B$^+$-Trees

- 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

# B$^+$-Tree Non-Leaf Node Split Example




After insertion of “Lamport”



# Updates on B$^+$-Trees: Deletion

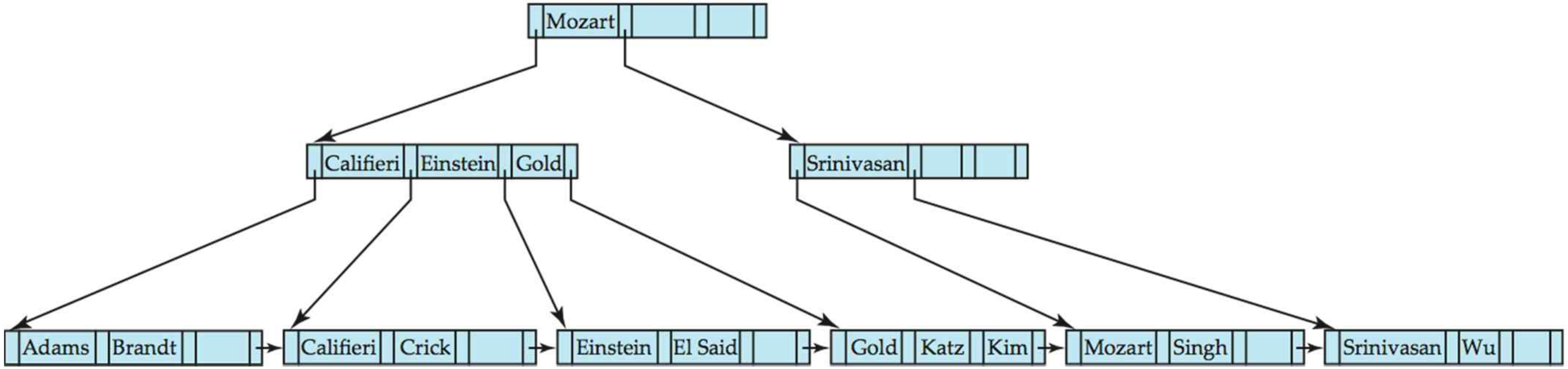
1. Find the record to be deleted, and remove it from the main file
2. Remove (search-key value, pointer) from the leaf node
3. If the node has too few entries due to the removal, **merge siblings** or **redistributed pointers** (next slide)
4. The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found
5. If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root

# Updates on B$^{+}$-Trees: Deletion (Cont.)

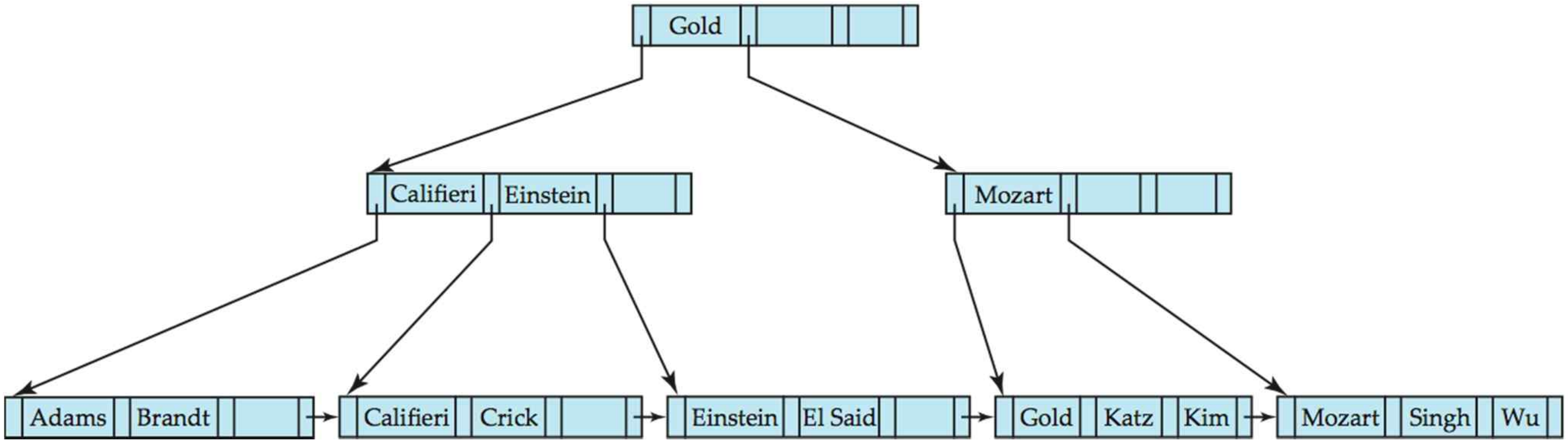
3. If the node has too few entries due to the removal,
   1. The entries in the node and a sibling fit into a single node – ***merge siblings:***
      - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node
      - Delete the pair ($K_{i-1}$, $P_i$), where $P_i$ is the pointer to the deleted node, from its parent, recursively using the above procedure
   2. The entries in the node and a sibling do not fit into a single node – **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

- Order to check an adjacent sibling
  - **Right sibling first**
  - If not possible, then, left sibling

# Examples of B$^+$-Tree Deletion

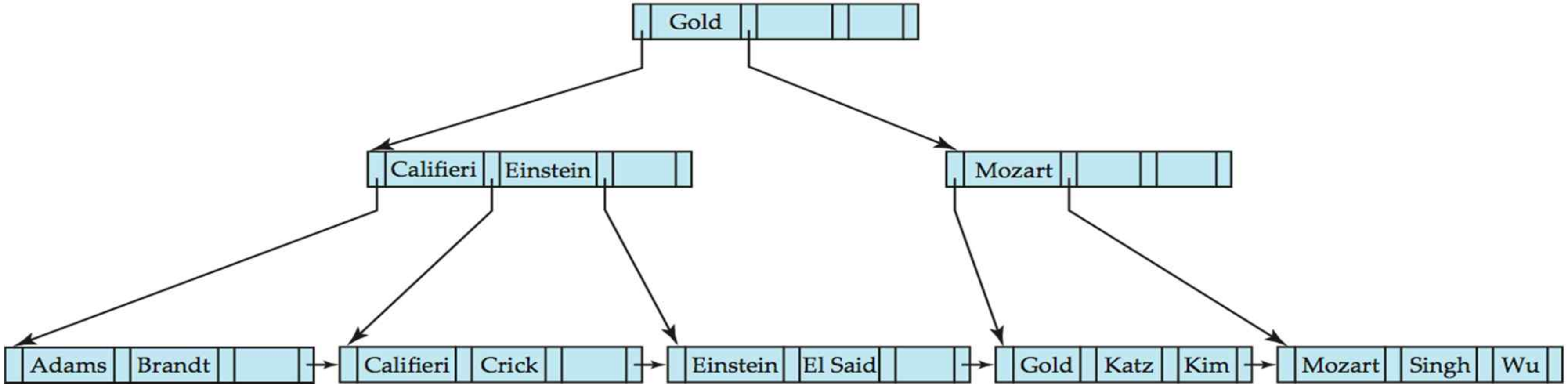



After deletion of "Srinivasan"

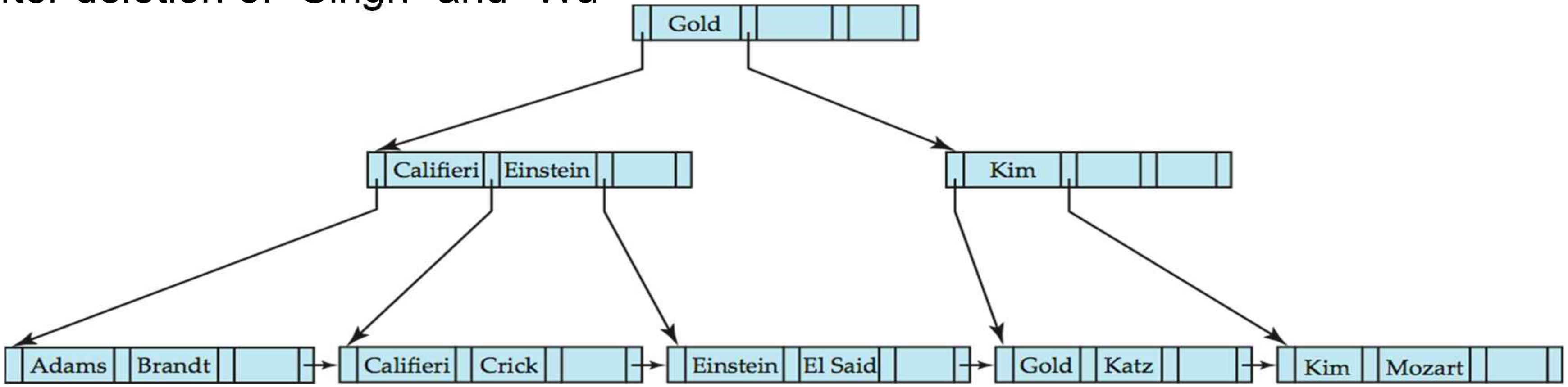



- Deleting "Srinivasan" causes merging of under-full leaves

# Examples of B⁺-Tree Deletion (Cont.)

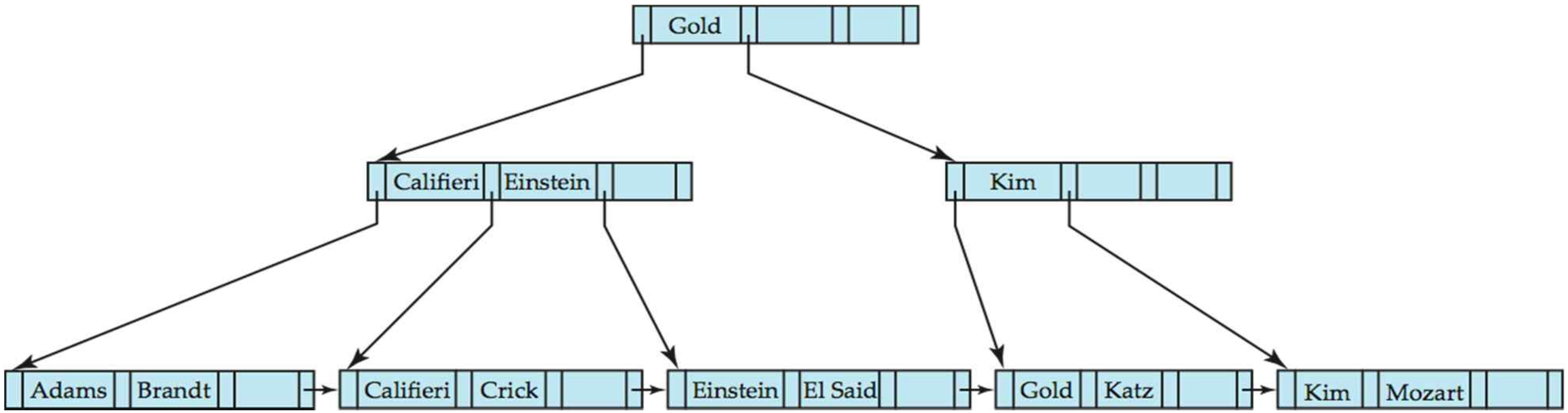



After deletion of “Singh” and “Wu”

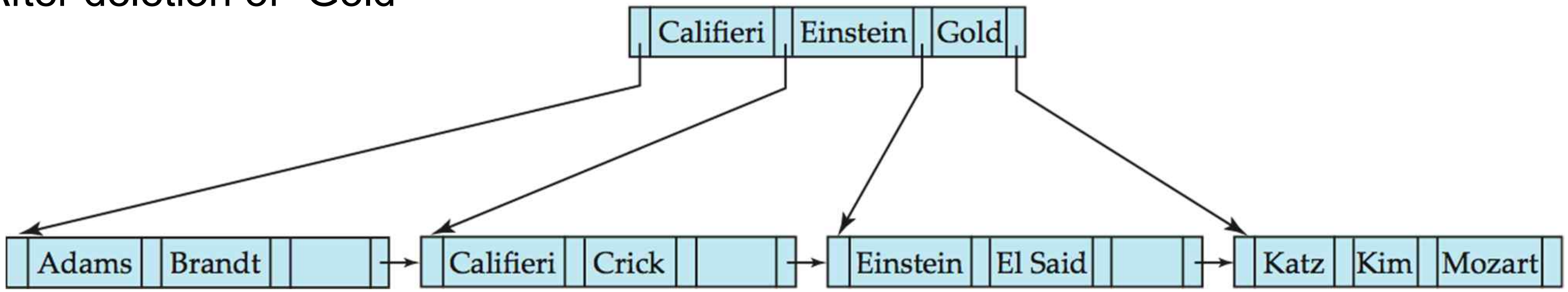



- Leaf containing Singh and Wu became underfull, and borrowed a value Kim from its left sibling
- Search-key value in the parent changes as a result

# Example of B⁺-tree Deletion (Cont.)




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

# Handling Non-Unique Search Keys

- 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
  - 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

# 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 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 $a_2 < b_2$

- Suppose we have an index on combined search-key (*dept_name, salary*).
- Can efficiently handle

  **where** *dept_name* = "Finance" **and** *salary* = 80000
  - Fetch only records that satisfy both 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

# Static Hashing

- **Bucket**
  - A unit of storage containing one or more records
  - Typically a disk block
- **Hash file organization**
  - The bucket of a record is directly obtained from its search-key value using a **hash function**
- Hash function *h*
  - A function from the set of all search-key values *K* to the set of all bucket addresses *B*
  - is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket
  - Thus entire bucket has to be searched sequentially to locate a record

# Example of Hash File Organization

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

- # of buckets = 10
- The binary representation of the *i*th character is assumed to be the integer *i*
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Music) = 1
    h(History) = 2
    h(Physics) = 3
    h(Elec. Eng.) = 3

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

| | | | |
|---|---|---|---|
| | | | |
| | | | |
| | | | |
| | | | |

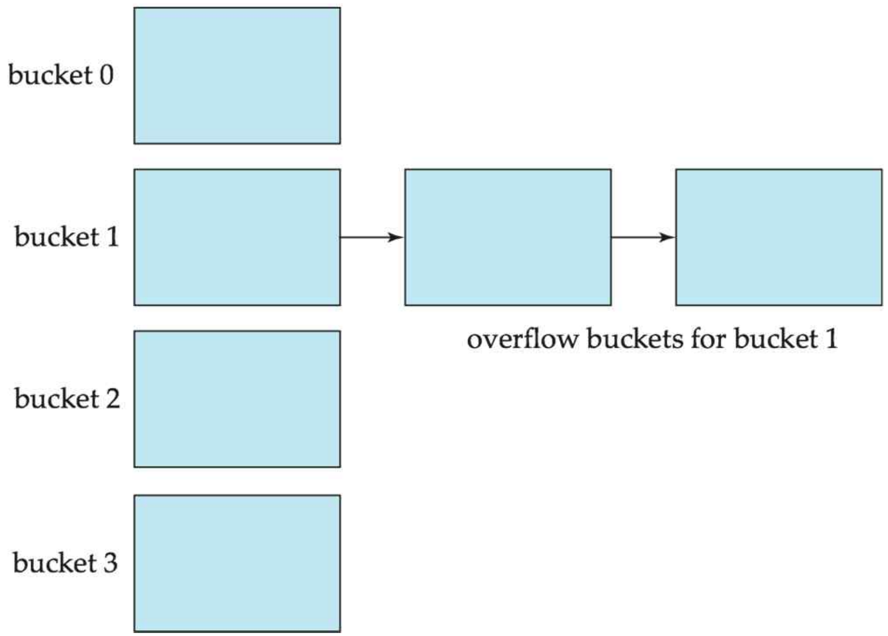
# Hash Functions

- Worst hash function
  - All search-key values are mapped to the same bucket
  - Access time is proportional to the number of search-key values in the file
- Ideal hash function
  - **Uniform**: each bucket is assigned the same number of search-key values from the set of *all* possible values
  - **Random**: each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file

- Typical hash functions perform computation on the internal binary representation of the search-key

# Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using ***overflow buckets.***
- **Overflow chaining**
  - The overflow buckets of a given bucket are chained together in a linked list

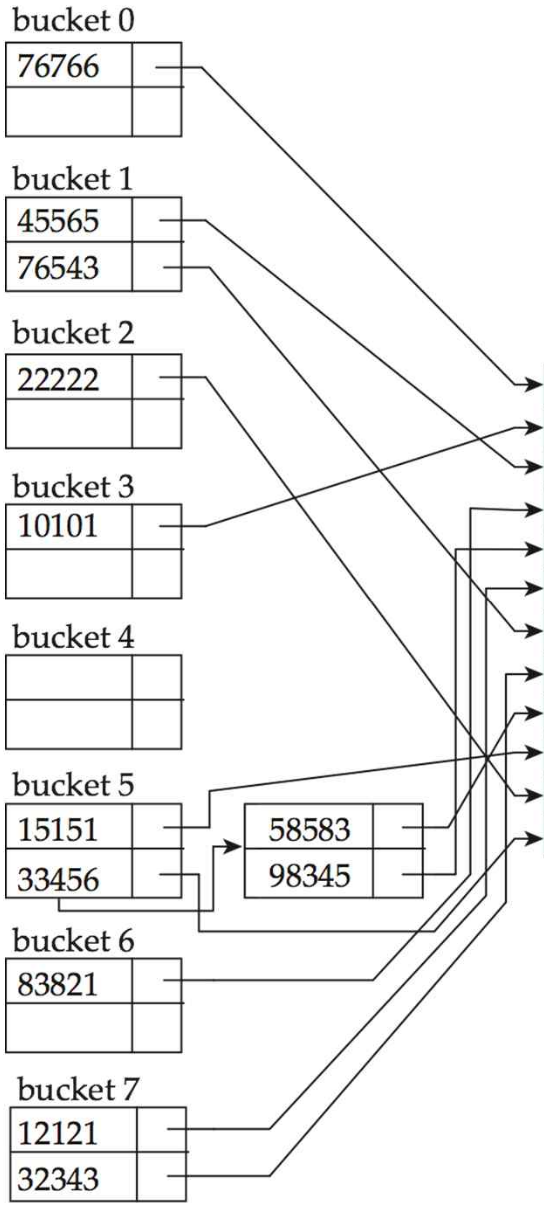


# Hash Indices

- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary
  - However, we use the term hash index to refer to both secondary index structures and hash organized files

# Example of Hash Index




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

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

# 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 initial number of buckets is too small, and file grows, performance will degrade due to too much overflows
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull)
  - If 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
  - Dynamic hashing is not covered in this class

# 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 (point query)
  - 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

# Bitmap Indices

- **Bitmap**: simply an array of bits
- **Bitmap index**: a specialized type of index designed for efficient querying on multiple keys
- In the simplest form, a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

| record number | *ID* | *gender* | *income_level* |
|---|---|---|---|
| 0 | 76766 | m | L1 |
| 1 | 22222 | f | L2 |
| 2 | 12121 | f | L1 |
| 3 | 15151 | m | L4 |
| 4 | 58583 | f | L3 |

Bitmaps for *gender*

| | |
|---|---|
| m | 10010 |
| f | 01101 |

Bitmaps for *income_level*

| | |
|---|---|
| L1 | 10100 |
| L2 | 01000 |
| L3 | 00001 |
| L4 | 00010 |
| L5 | 00000 |

# Bitmap Indices (Cont.)

- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, …
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation
    - If # of distinct attribute values is 8, bitmap is only 1% of relation size

# Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and): e.g., 100110 AND 110011 = 100010
  - Union (or): e.g., 100110 OR 110011 = 110111
  - Complementation (not): e.g., 100110 NOT 100110 = 011001
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g. males with income level L1: 10010 AND 10100 = 10000
    - Can then retrieve required tuples
    - Counting number of matching tuples is even faster

# Index Definition in SQL

- Create an index

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

  E.g.: **create index** *dept_index* **on** *instructor (dept_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

# End of Chapter 11