# 21121350
# Database System

## Lecture 8: Indexing and Hashing

Lu Chen (陈璐)
College of Computer Science
Zhejiang University
Spring & Summer 2023
luchen@zju.edu.cn/18868818726

# Outline

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

# Basic Concepts

- ❑ Why we need an index ?

- ❑ 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.

- ❑ 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.

# Basic Concepts (Cont.)

- Two basic kinds of indices: (index文件中索引记录如何组织？取决于索引类型)
  - Ordered indices(顺序索引): search keys (index entries) are stored in sorted order
  - Hash indices(散列索引): search keys (index entries) are distributed uniformly across “buckets” using a “hash function”.

# Index Evaluation Metrics

- Access types which are supported efficiently, e.g.,
  - Records with a specified value in the attribute. (where Col=v )
  - Or records with an attribute value falling in a specified range of values. E.g. where Col Between v1 and v2 .

  如: Hash index is not good for ‘Between’ query condition, but ordered index is good.
- Access time
- Insertion time
- Deletion time
- Space overhead (空间开销)
- 时间效率和空间效率的衡量索引技术最主要的指标，也是数据库系统组织和管理技术的关注的焦点之一。

# Outline

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

# Ordered Indices

Some basic concepts on indexing :

- In an ordered index, index entries are stored sorted on the search key value, e.g., author catalog in library.
- Sequentially ordered file (顺序排序文件): The records in the file (data file) are ordered by a search-key. (chapter 14)
- Primary index: an index whose search key equals to the search key of the sequentially ordered data file, on which the index is made.

  (与对应的数据文件本身的排列顺序相同的索引称为主索引.)
  - Also called **clustering index**（聚集索引 ）
    - The search key of a primary index is usually but not necessarily the primary key (主索引的搜索键通常是但并非一定是主码).
    - Non-sequential files don't have primary index, but the relations can have primary key.
  - Index-sequential file（索引顺序文件）: sequentially ordered file with a primary index.

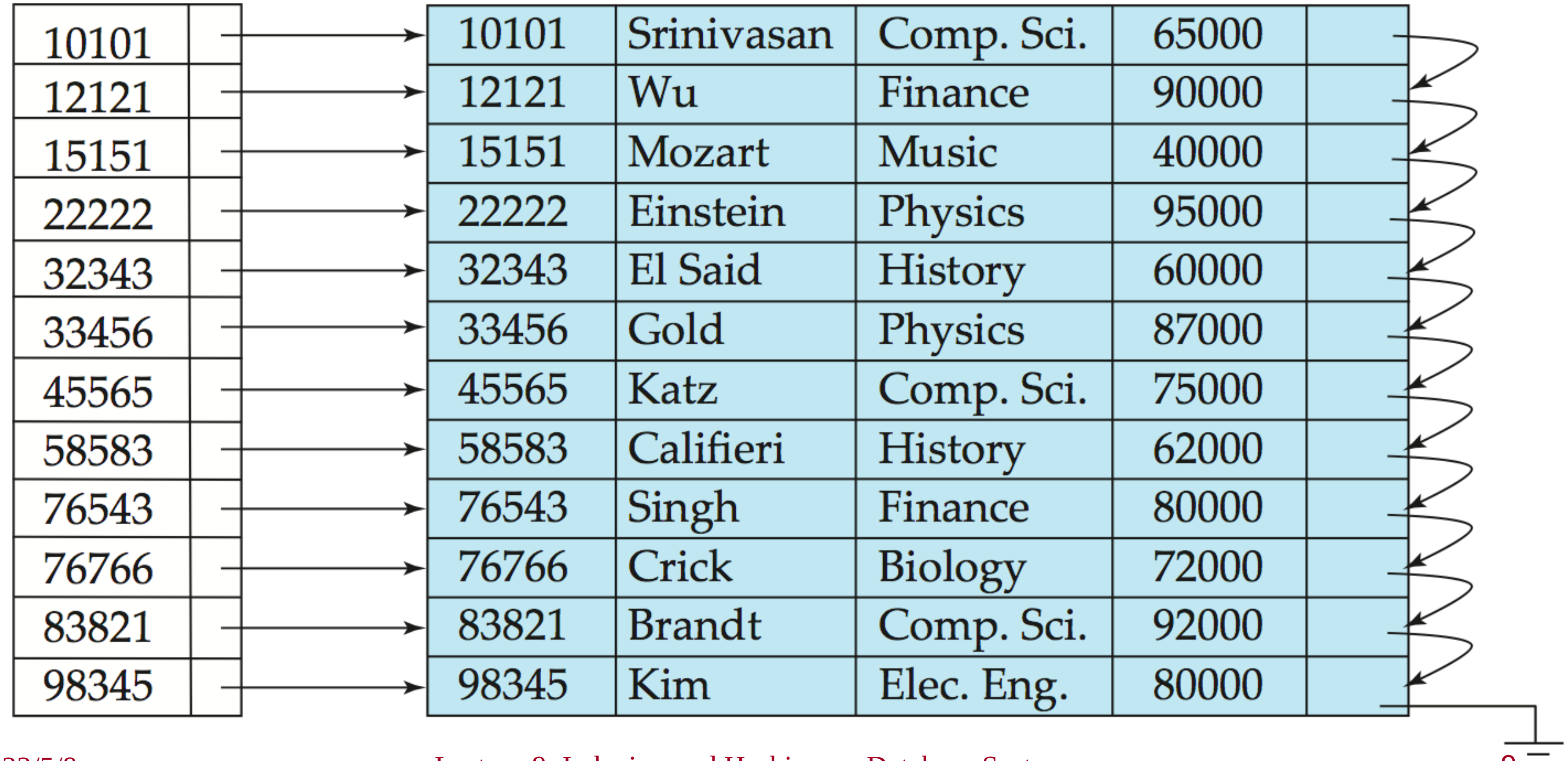
# Ordered Indices (Cont.)

- Secondary index(辅助索引): an index whose search key specifies an order different from the sequential order of the file.  Also called non-clustering index.

- Two types of ordered indices:
  - Dense index (稠密索引)
  - Sparse index (稀疏索引)

# (1) Dense Index Files (稠密索引文件)

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

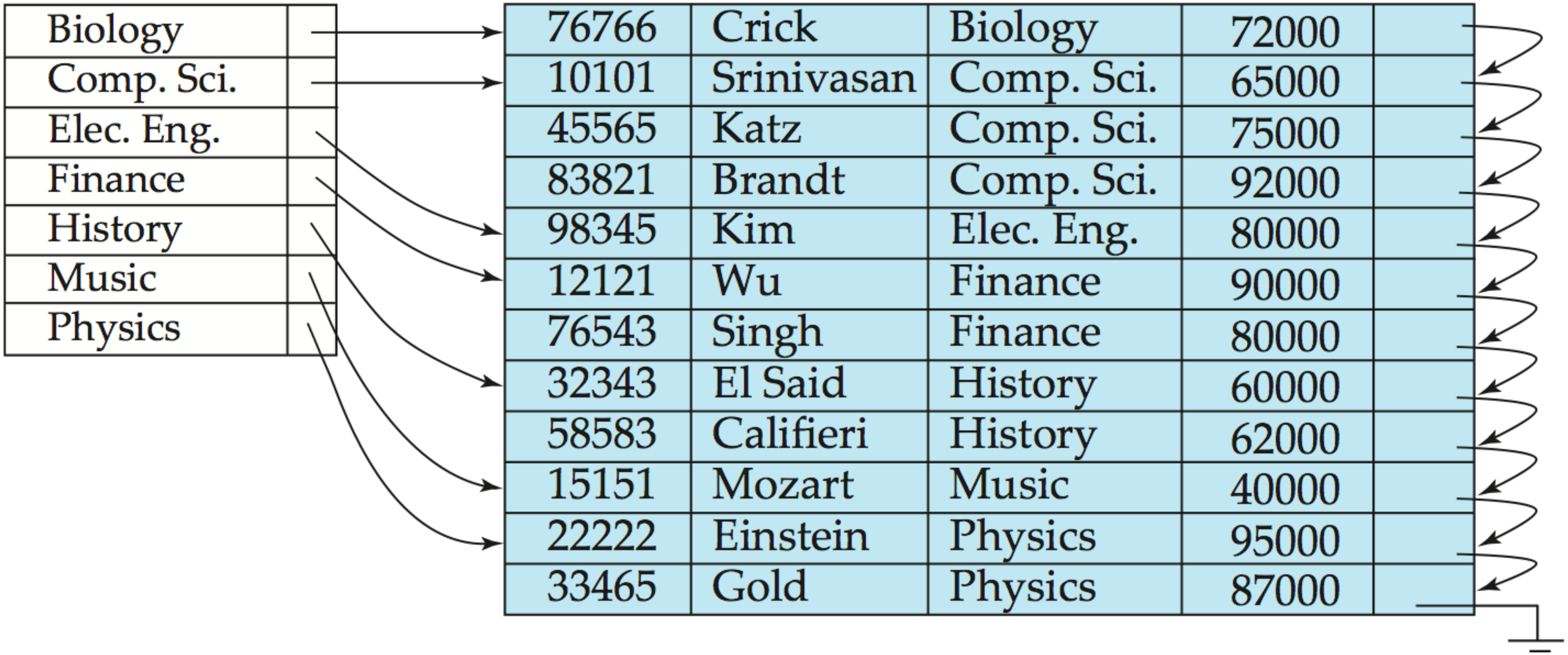


| | |
|---|---|
| 10101 | |
| 12121 | |
| 15151 | |
| 22222 | |
| 32343 | |
| 33456 | |
| 45565 | |
| 58583 | |
| 76543 | |
| 76766 | |
| 83821 | |
| 98345 | |

| | | | | |
|---|---|---|---|---|
| 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*

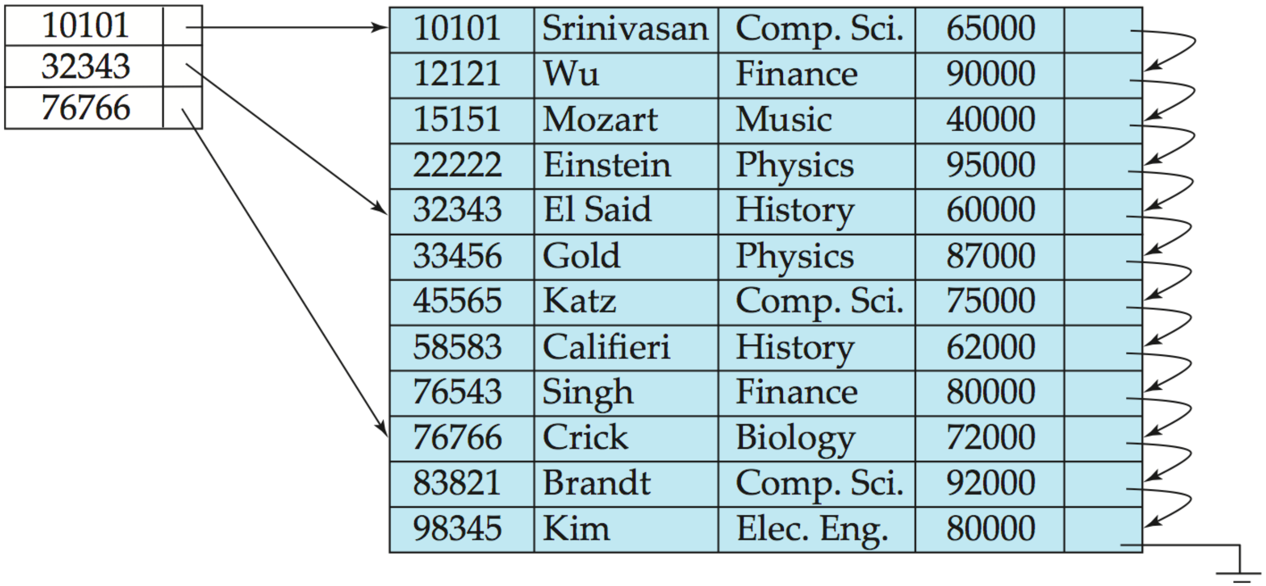


| | |
|---|---|
| Biology | |
| Comp. Sci. | |
| Elec. Eng. | |
| Finance | |
| History | |
| Music | |
| Physics | |

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

# (2) Sparse Index Files

- Sparse Index:  contains index entries for only some search-key values. ( Usually, one block of data gives an index entry, a block contains a number of ordered data records)
  - Applicable only when data file records are sequentially ordered on search-key
- To locate a record with search-key value *K* , (搜索方法)
  - Step1: Find index record with largest search-key value < *K*
  - Step2: Search file sequentially starting at the record to which the index entry 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.

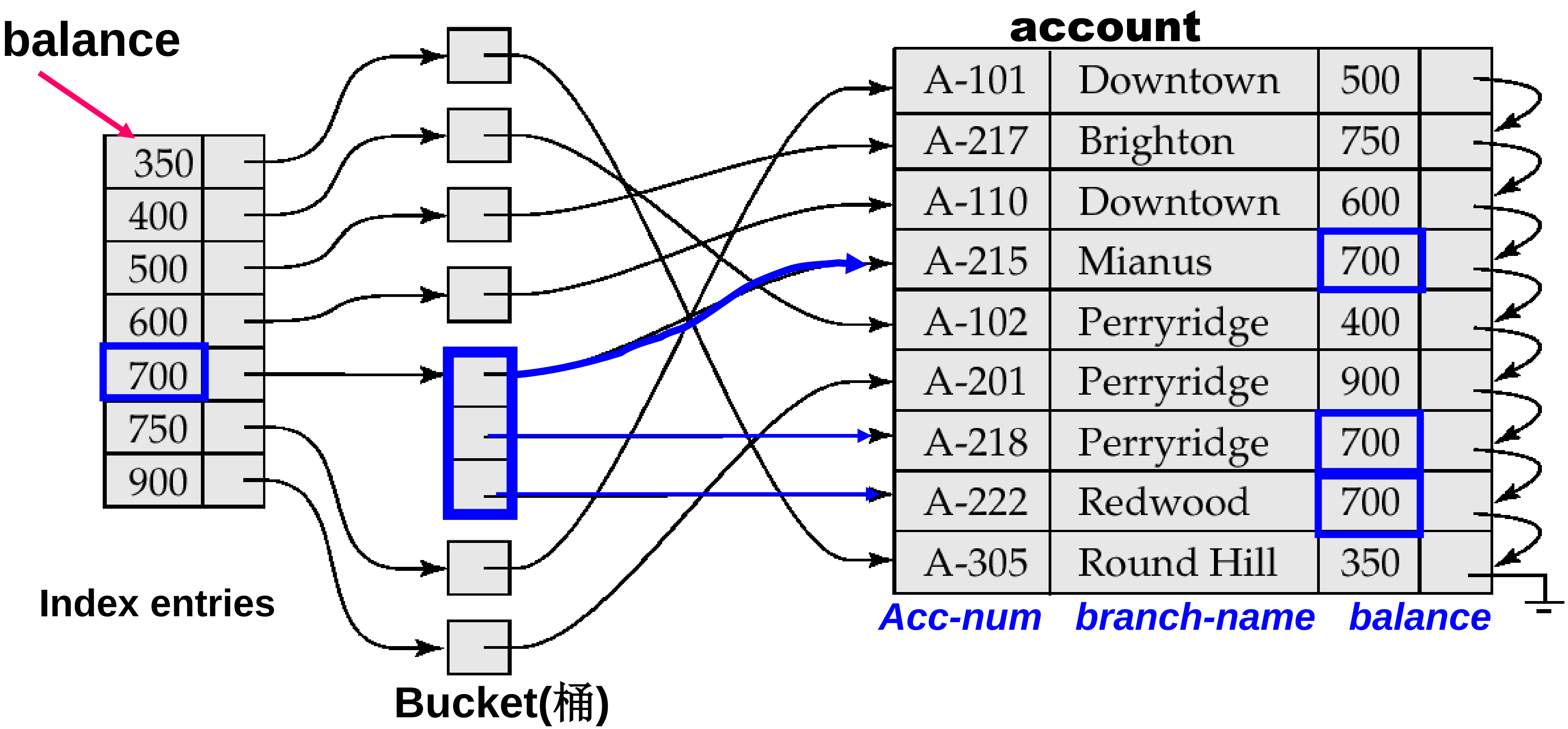
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block. (a block中通常包含多个数据记录，每块中最小的搜索键值放到索引项中。)

- Sparse index只能用于顺序文件, 而dense index可以用于顺序和非顺序文件，如构成索引无序文件。

# (3) Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field *which is not the search-key of the primary index* satisfy some condition.（实际应用中常有多种属性作为查询条件）
  - Example : In the *account* database stored sequentially by account number, we may want to find all accounts with a specified balance or range of balances. *(see next slide)*
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that one particular search-key value.

# Secondary Index on *balance* field of *account*



| Acc-num | branch-name | balance |
|---|---|---|
| A-101 | Downtown | 500 |
| A-217 | Brighton | 750 |
| A-110 | Downtown | 600 |
| A-215 | Mianus | 700 |
| A-102 | Perryridge | 400 |
| A-201 | Perryridge | 900 |
| A-218 | Perryridge | 700 |
| A-222 | Redwood | 700 |
| A-305 | Round Hill | 350 |

（辅助索引不能使用稀疏索引，每条记录都必须有指针指向。但**search-key**常存在重复项---如**700**，而**index entry**不能有重复，否则查找算法复杂化**,** 为此，使用**bucket**结构。）

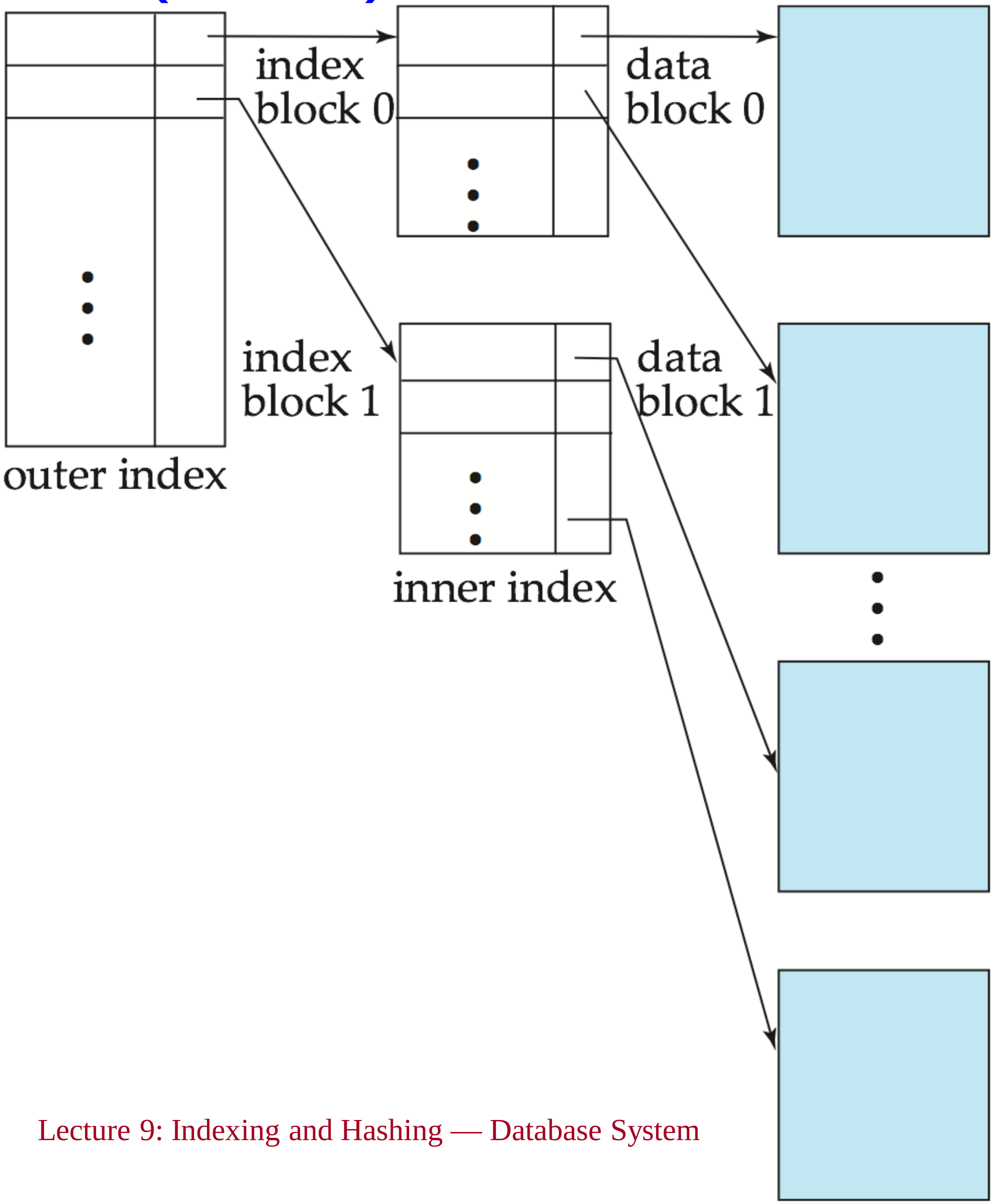
# (4) Multilevel Index

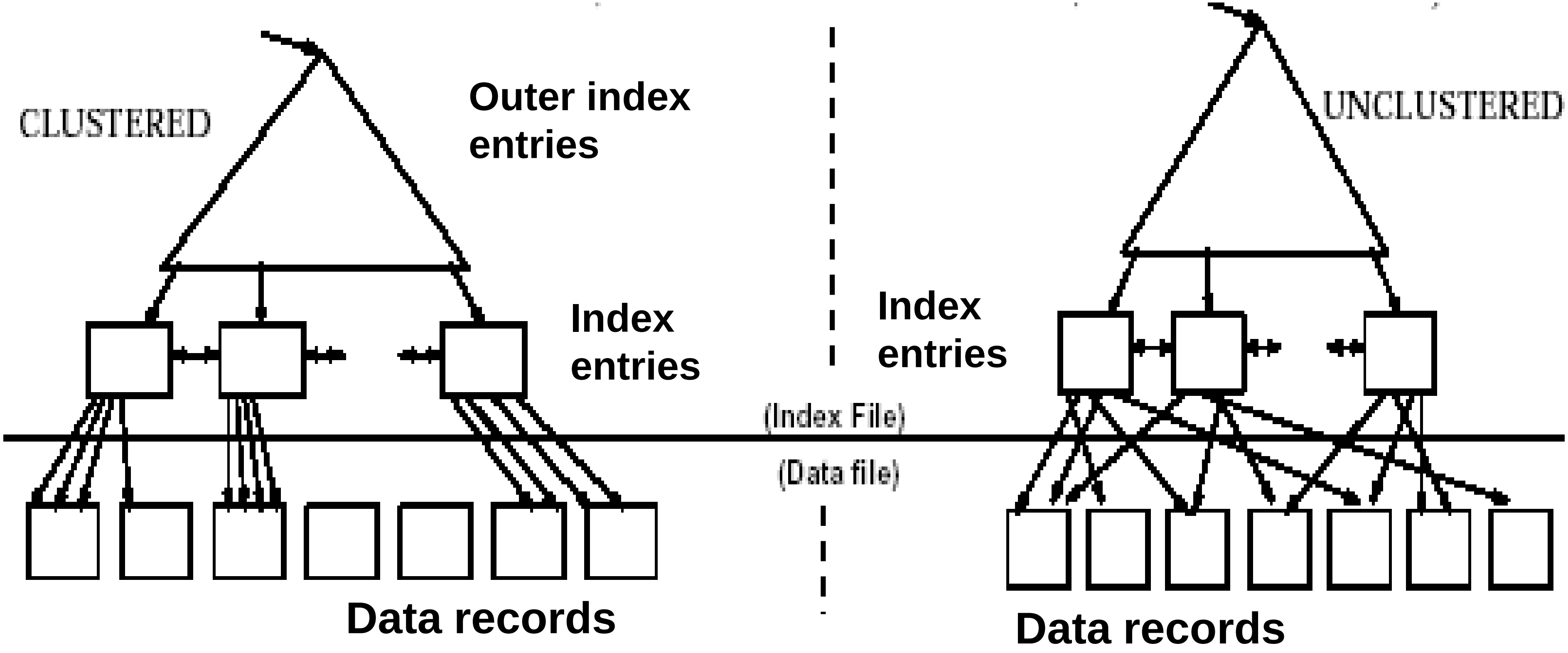
- If primary index is too big to fit in memory, access becomes expensive.

  E.g: 1,000,000records / 10records per block = 100,000blocks = 100,000 index entries, (sparse index ) / 100entries per block = 1000blocks (size of sparse index file).

  Binary search: ⌈$\log_2(1000)$⌉ = 9 block reads, 10*15ms = 150ms
- To reduce number of disk accesses to index records, treat primary index kept on disk as a sequential file and construct a sparse index on it. (把内层索引文件看作顺序数据文件一样，在其上建立外层的稀疏索引)
  - 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.(可以推广到任意多层索引)
- Indices at all levels must be updated on insertion or deletion from the file.

# Multilevel Index (Cont.)



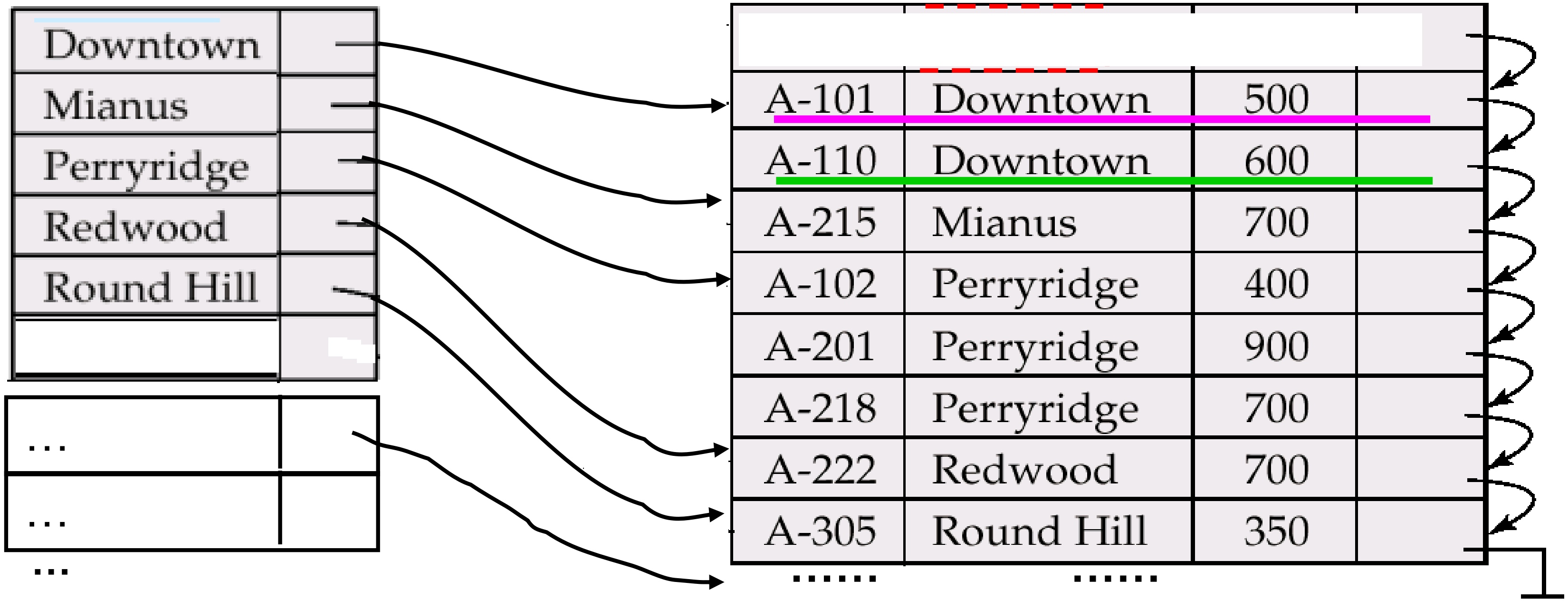


多级索引技术可应用于聚集索引和非聚集索引

# (5) Index Update: Deletion

- Step1: The system finds the record in the data file, then deletes the record.
- Step2: updates index file: (for single-level index deletion):
  - Case1: Dense indices –
    - If the deleted record was the only record with its particular search-key value, then deletes the corresponding index entry from index file. (单记录, 即search-key有唯一性)
    - Otherwise, (多条记录，即search-key无唯一性)
      - If have multi pointers to all records with the same search-key value, then deletes the pointer to the deleted record from the index entry. ( 对应辅助索引的情况)
      - Otherwise, ( 对应主索引)
        - » If the deleted record was the first record to be pointed, then change the pointer to the next record.
        - » Otherwise, nothing to do with the index entry.

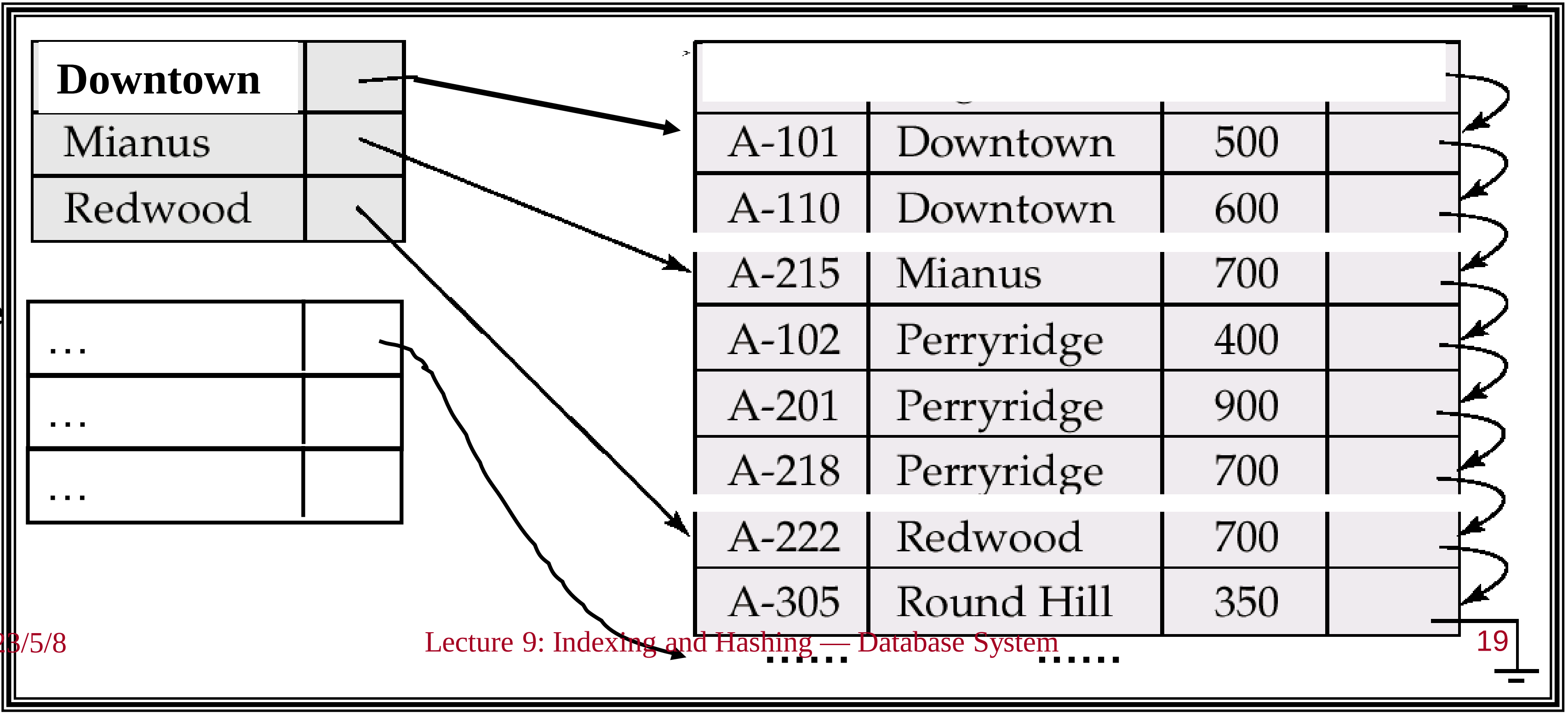
**Dense index**



| |
|---|
| Downtown |
| Mianus |
| Perryridge |
| Redwood |
| Round Hill |
| |
| … |
| … |
| … |

| | | |
|---|---|---|
| A-101 | Downtown | 500 |
| A-110 | Downtown | 600 |
| A-215 | Mianus | 700 |
| A-102 | Perryridge | 400 |
| A-201 | Perryridge | 900 |
| A-218 | Perryridge | 700 |
| A-222 | Redwood | 700 |
| A-305 | Round Hill | 350 |
| …… | …… | |

**Sparse index**



| |
|---|
| Downtown |
| Mianus |
| Redwood |
| … |
| … |
| … |

| | | |
|---|---|---|
| A-101 | Downtown | 500 |
| A-110 | Downtown | 600 |
| A-215 | Mianus | 700 |
| A-102 | Perryridge | 400 |
| A-201 | Perryridge | 900 |
| A-218 | Perryridge | 700 |
| A-222 | Redwood | 700 |
| A-305 | Round Hill | 350 |
| …… | …… | |

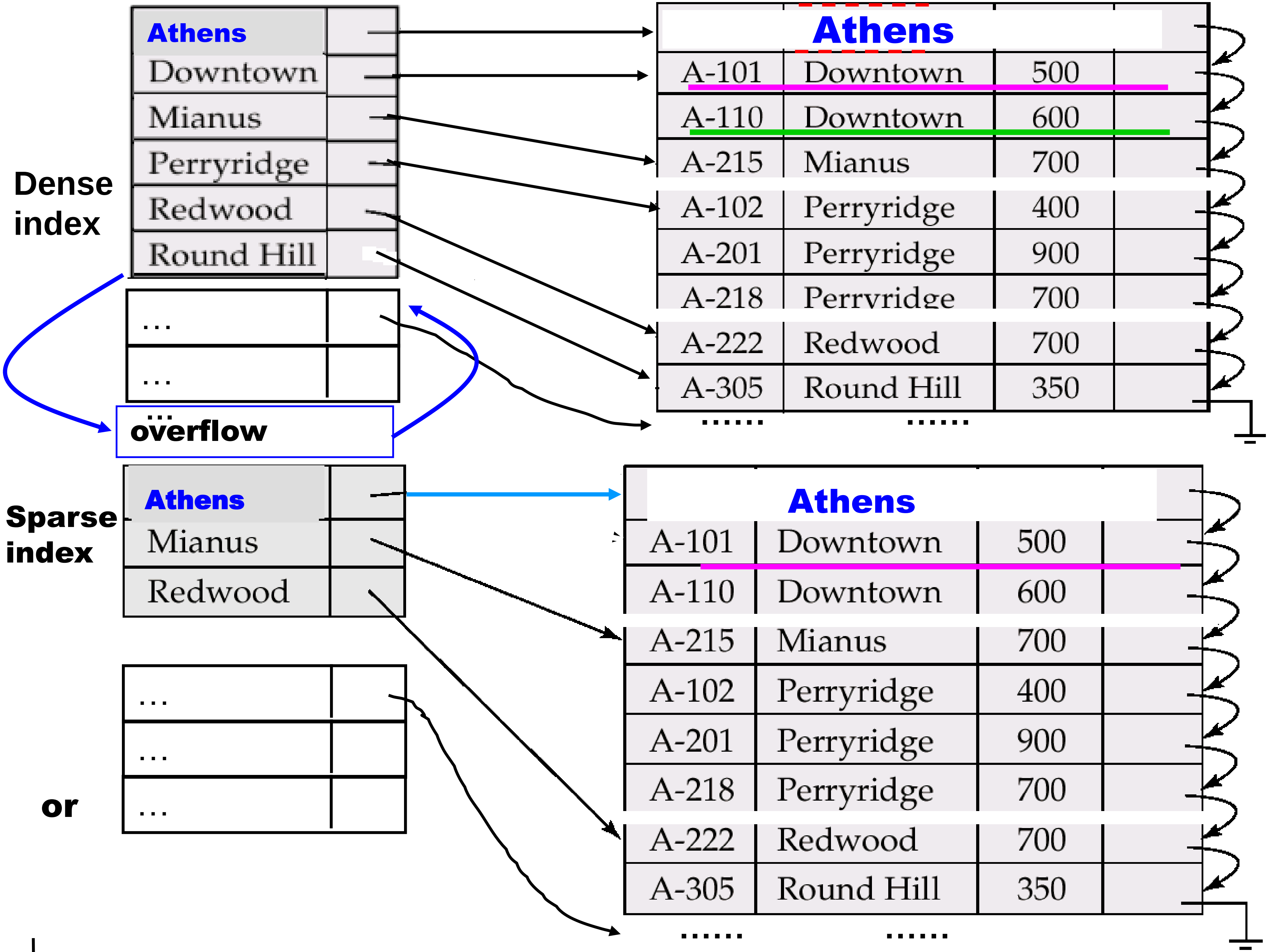
# (5) Index Update: Deletion

- Case2: Sparse indices –
  - If the search-key values of the deleted record does not appear in the index, then nothing to be done to the index.
  - Otherwise, if an index entry for the search key exists in the index file, it is deleted by replacing the entry with the next search-key value in the data file (in search-key order).
    - If the next search-key value already has an index entry, the index entry is deleted instead of being replaced.

- For multilevel indices :由底层逐级向上层扩展，每一层的处理过程与上述单层索引情况下类似。

# (5) Index Update: Insertion

- Perform a lookup using the search-key value appearing in the record to be inserted.(利用索引找到插入位置，在数据文件中插入记录，然后分别根据情况来修改索引)
- Single-level index insertion:
  - Dense indices –
    - if the search-key value does not appear in the index, inserts an index entry with the search-key value .
    - Otherwise,
      - If with multi pointers, adds a pointer (in the index entry) that points to the new record.
      - Otherwise, nothing to do with the index entry.
  - Sparse indices – (assume an index entry for each block)
    - If a new block is created, insert the first search-key value appearing in the new block into the index.
    - if the new record has the least search-key value in its block, update the index entry.
    - otherwise, no change to the index
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms.



Dense index

| | |
|---|---|
| Athens | |
| Downtown | |
| Mianus | |
| Perryridge | |
| Redwood | |
| Round Hill | |
| … | |
| … | |

…

overflow

| | | | |
|---|---|---|---|
| Athens | | | |
| A-101 | Downtown | 500 | |
| A-110 | Downtown | 600 | |
| A-215 | Mianus | 700 | |
| A-102 | Perryridge | 400 | |
| A-201 | Perryridge | 900 | |
| A-218 | Perryridge | 700 | |
| A-222 | Redwood | 700 | |
| A-305 | Round Hill | 350 | |

……  ……

Sparse index

| | |
|---|---|
| Athens | |
| Mianus | |
| Redwood | |

or

| | |
|---|---|
| … | |
| … | |
| … | |

| | | | |
|---|---|---|---|
| Athens | | | |
| A-101 | Downtown | 500 | |
| A-110 | Downtown | 600 | |
| A-215 | Mianus | 700 | |
| A-102 | Perryridge | 400 | |
| A-201 | Perryridge | 900 | |
| A-218 | Perryridge | 700 | |
| A-222 | Redwood | 700 | |
| A-305 | Round Hill | 350 | |

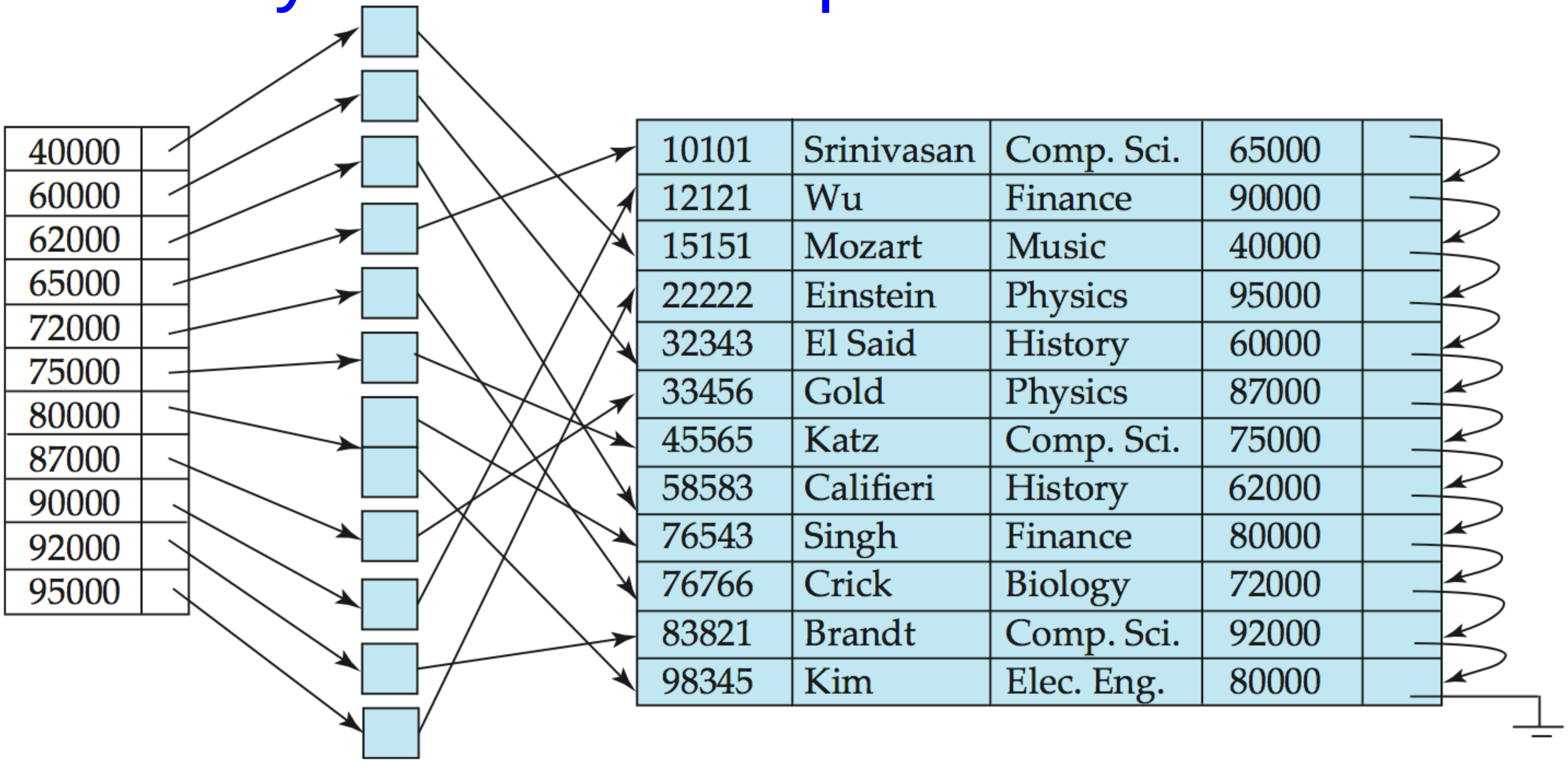
……  ……

# (6) Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *instructor* relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values

- We can have a secondary index with an index record for each search-key value

# Secondary Indices Example



|  |
|---|
| 40000 |
| 60000 |
| 62000 |
| 65000 |
| 72000 |
| 75000 |
| 80000 |
| 87000 |
| 90000 |
| 92000 |
| 95000 |

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

**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

# (7) Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.

- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,

- 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

# Outline

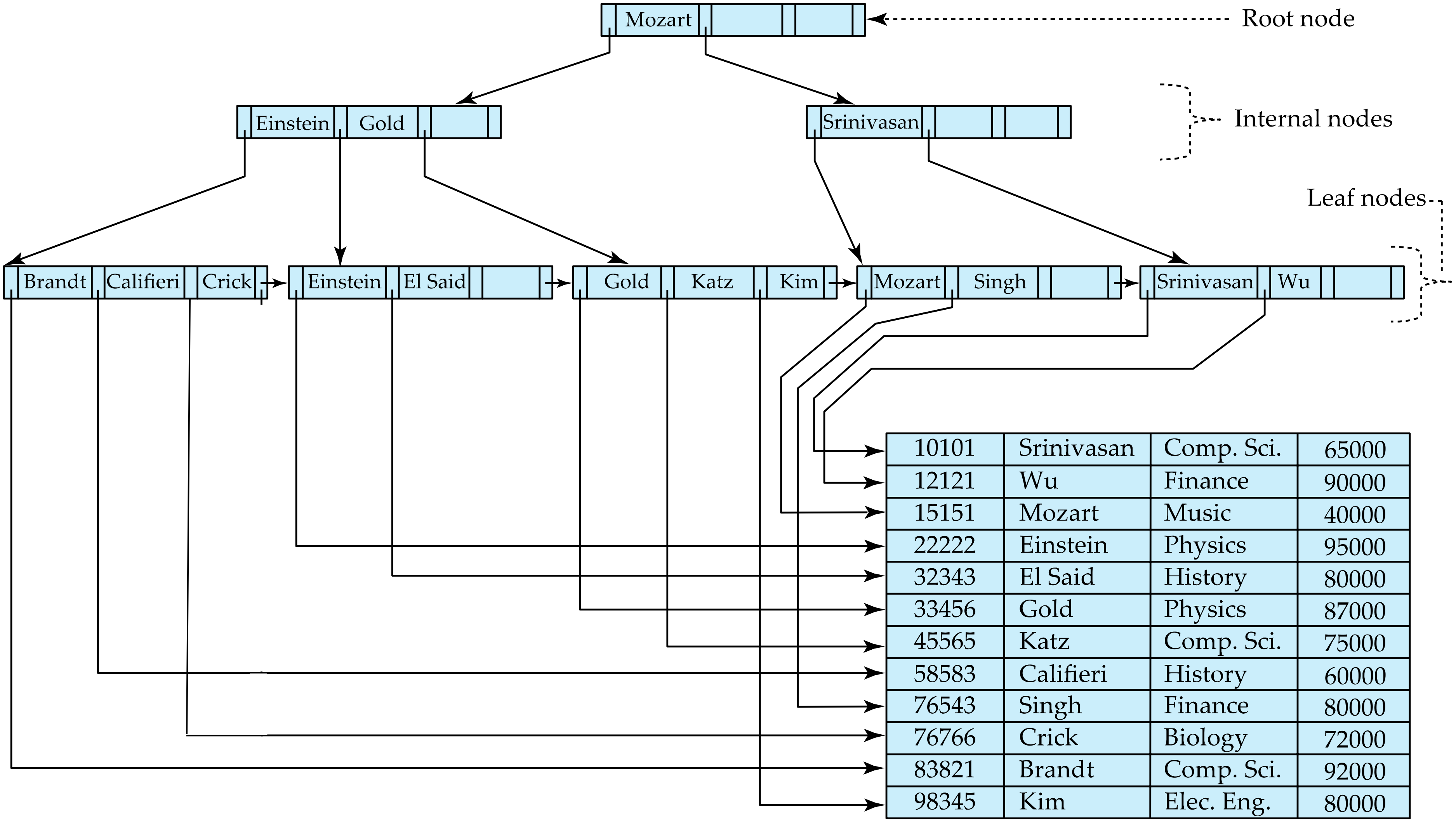
- ❑ Basic Concepts
- ❑ Ordered Indices
- ❑ B+-Tree Index Files
- ❑ * B-Tree Index Files
- ❑ * Static Hashing
- ❑ * Dynamic Hashing
- ❑ * Comparison of Ordered Indexing and Hashing
- ❑ Write-optimized Indices
- ❑ * Multiple-Key Access
- ❑ Index Definition in SQL

# $B^+$-Tree Index Files

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

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

# Example of B$^+$-Tree



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

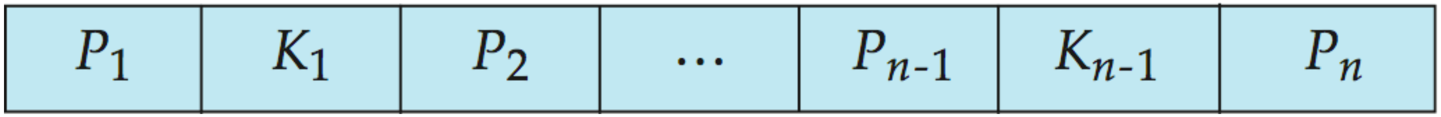
# $B^+$-Tree Index Files (Cont.)

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

- All paths from root to leaf are of the same length – 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



| | | | | | | |
|---|---|---|---|---|---|---|
| $P_1$ | $K_1$ | $P_2$ | … | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |

  - $K_i$ 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)
  - 通常，一个节点对应一个block
- The search-keys in a node are ordered

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

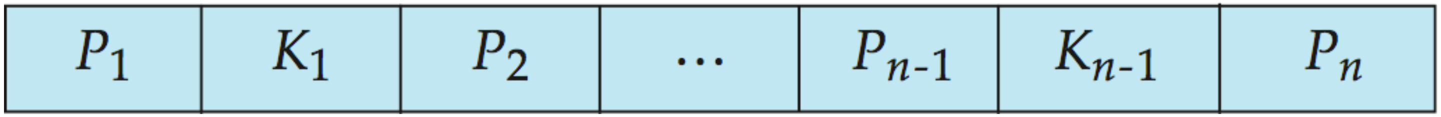
(Initially assume no duplicate keys, address duplicates later)

# Leaf Nodes in B$^+$-Trees

- For $i$ = 1, 2, . . ., $n$–1, pointer $P_i$ either points to a file record with search-key value $K_i$, or to a bucket of pointers to file records, each record having search-key value $K_i$. Only need bucket structure if search-key does not form a primary key. (like dense index, every search key appears in leaf nodes)
- If $L_i$, $L_j$ are leaf nodes and $i < j$, all search-key values in $L_i$ are less than $L_j$'s search-key values. (叶节点间的search-key不重叠, 且所有左节点中的search-key值一定小于右节点中的search-key值)
- Must have between $\lceil(n–1)/2\rceil$ and $n$ –1 search-keys. With $n$ = 5, 2 ≤ number of search-keys ≤ 4
- $P_n$ points to next leaf node in search-key order, which is convenient for sequential processing of the file.

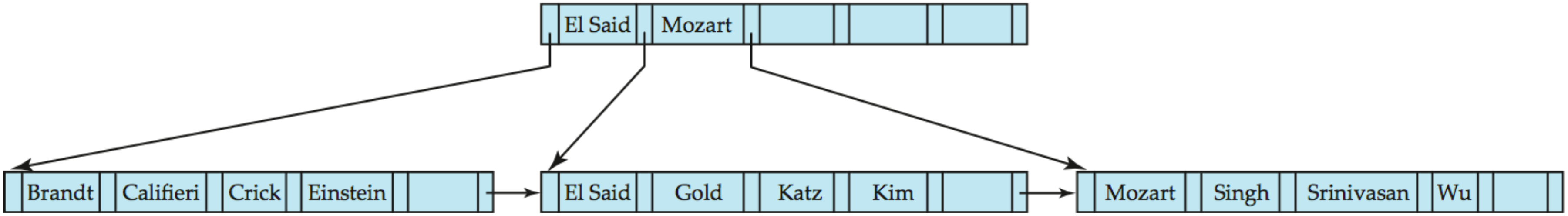
# Non-Leaf Nodes in B$^+$-Trees

- have pointers (subtrees) between $\lceil(n/2\rceil$ and n. Fanout (扇出数) = the number of pointers in a node
- 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 $K_1$。($P_1$ 所指的子树中的所有search-keys皆小于$K_1$ )
  - 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 $K_i$ **($P_i$ 所指的子树中的所有search-keys皆大于或等于$K_{i-1}$ 但小于$K_i$ )**
  - All the search-keys in the subtree to which $P_n$ points have values greater than or equal to $K_{n-1}$



| | | | | | | |
|---|---|---|---|---|---|---|
| $P_1$ | $K_1$ | $P_2$ | … | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |

# Example of B⁺-tree



B⁺-tree for *instructor* file (*n* = 6)

- Leaf nodes must have between 3 and 5 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2) \rceil$ and $n$ with $n = 6$).
- Root must have at least 2 children.

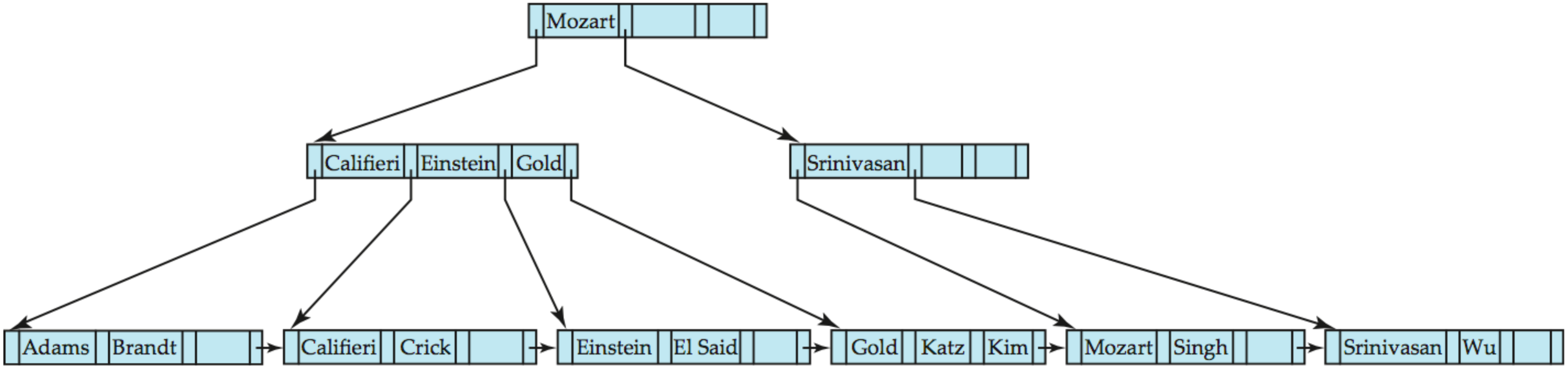
# *Observations about B$^+$-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B$^+$-tree form a hierarchy of sparse indices. (非叶节点构成了多层的稀疏索引)
- The B$^+$-tree contains a relatively small number of **levels** (logarithmic in the size of the main file), 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. $V \leq K_i$.
     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
  - and $n$ is typically around 100 (40 bytes per index entry).
- With 1 million search key values and $n$ = 100
  - at most $log_{50}(1{,}000{,}000)$ = 4 nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

# Handling Duplicates

- With duplicate search keys
  - In both leaf and internal nodes,
    - we cannot guarantee that $K_1 < K_2 < K_3 < . . . < K_{n–1}$
    - but can guarantee $K_1 \leq K_2 \leq K_3 \leq . . . \leq K_{n–1}$
  - Search-keys in the subtree to which $P_i$ points
    - are $\leq K_{i,}$, but not necessarily $< K_{i,}$
    - To see why, suppose same search key value $V$ is present in two leaf node $L_i$ and $L_{i+1}$. Then in parent node $K_i$ must be equal to V
- We modify find procedure as follows
  - traverse $P_i$ even if $V = K_i$
  - As soon as we reach a leaf node $C$ check if $C$ has only search key values less than $V$
    - if so set $C$ = right sibling of $C$ before checking whether $C$ contains $V$
- Procedure printAll
  - uses modified find procedure to find first occurrence of $V$
  - Traverse through consecutive leaves to find all occurrences of $V$

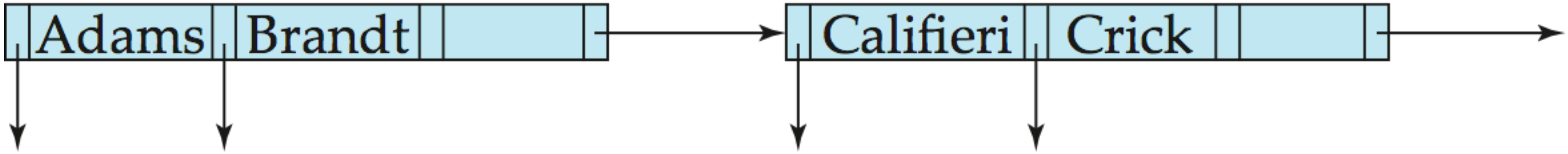
**** Errata note: modified find procedure missing in first printing of 6th edition**

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

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

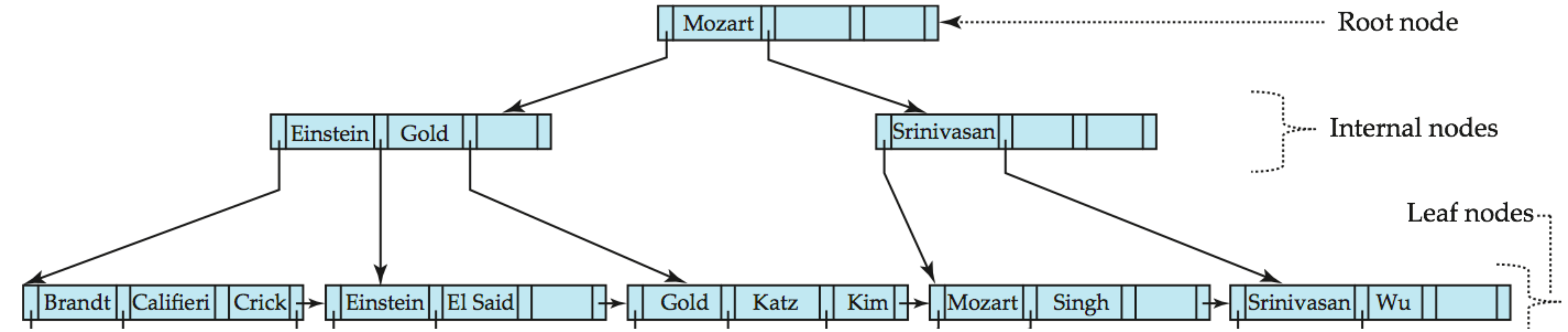
# Updates on $B^+$-Trees: Insertion (Cont.)

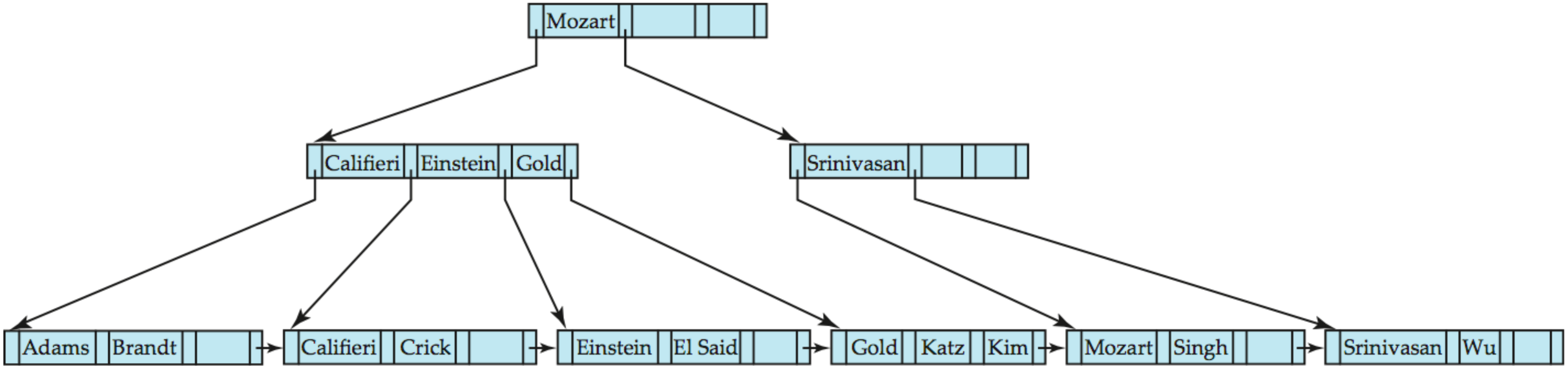
- Splitting a leaf node:
  - take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be *p,* and let *k* be the least key value in *p.* Insert (*k,p*) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.



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

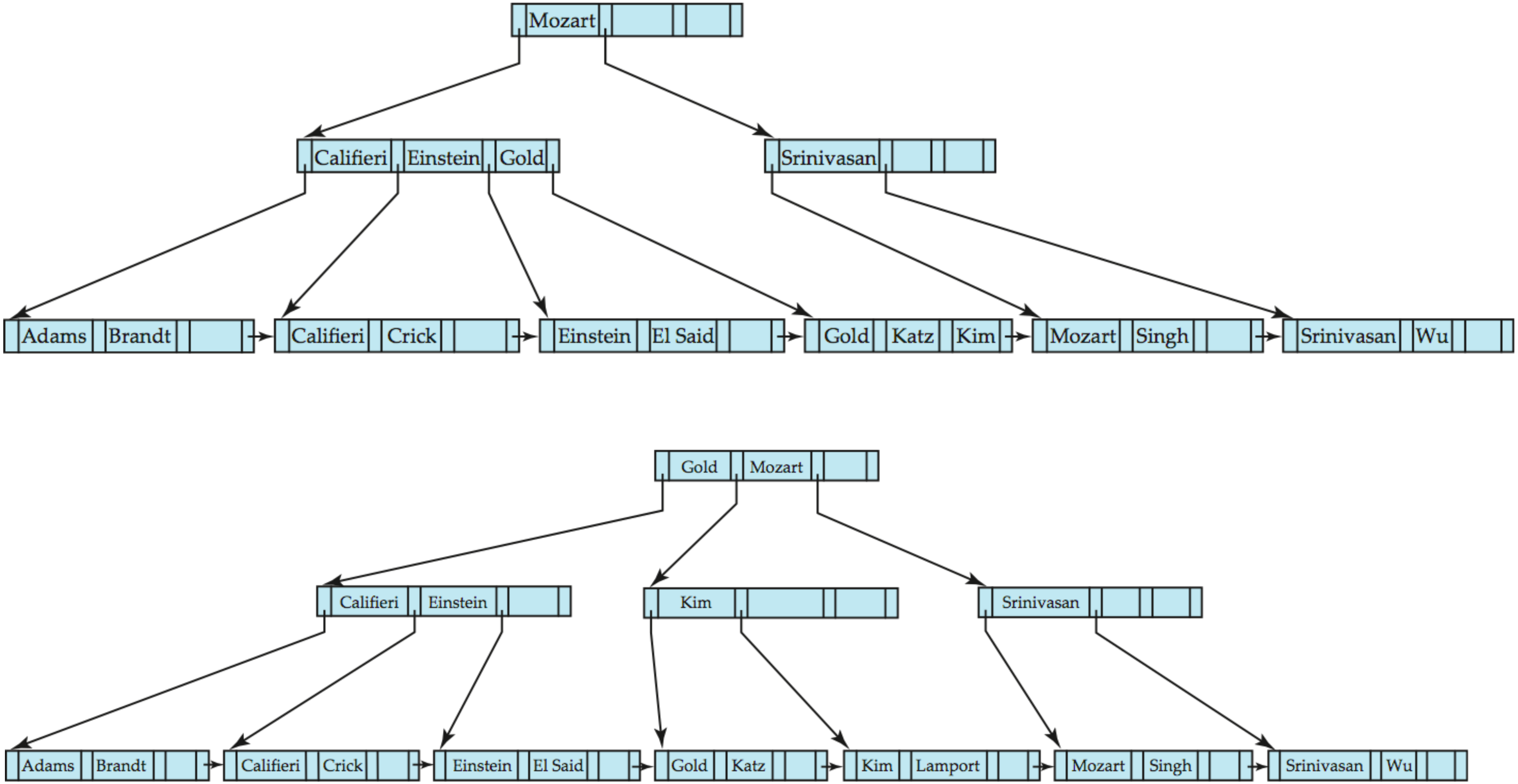
# B$^+$-Tree  Insertion





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

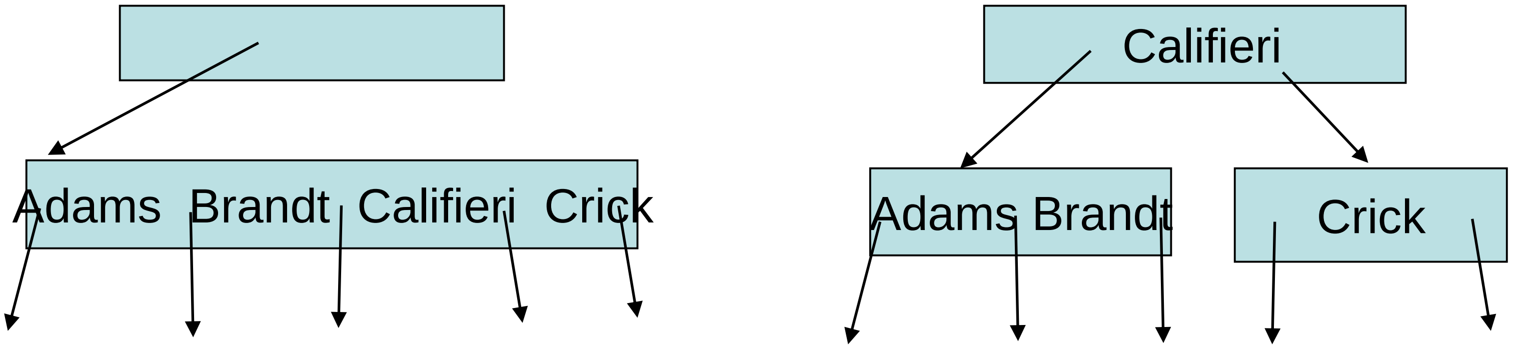
# B$^+$-Tree Insertion



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

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

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ 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
- **Read pseudocode in book!**



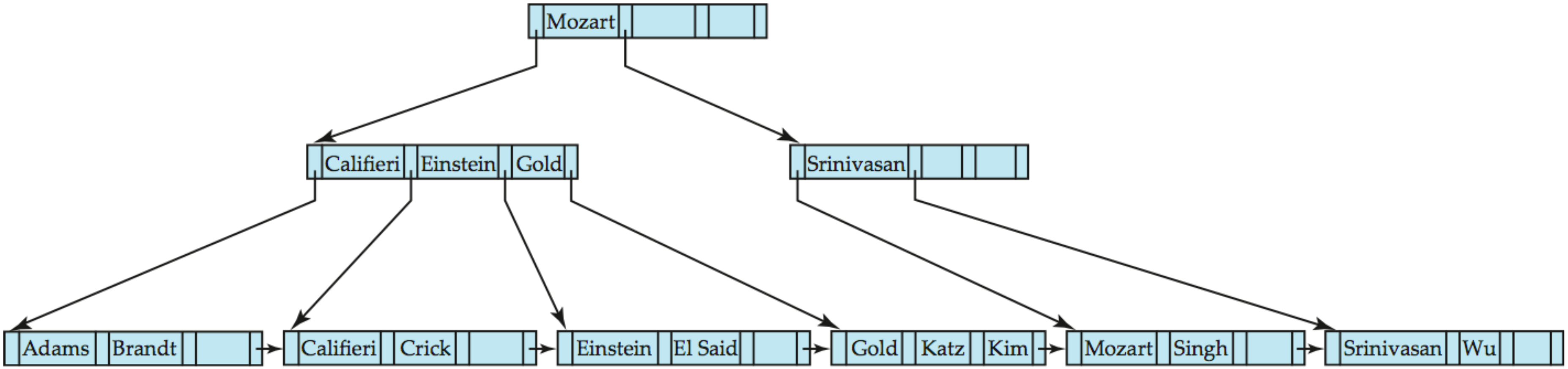
# Updates on B$^{+}$-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings*:**
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair ($K_{i-1}$, $P_i$), where $P_i$ is the pointer to the deleted node, from its parent, recursively using the above procedure.

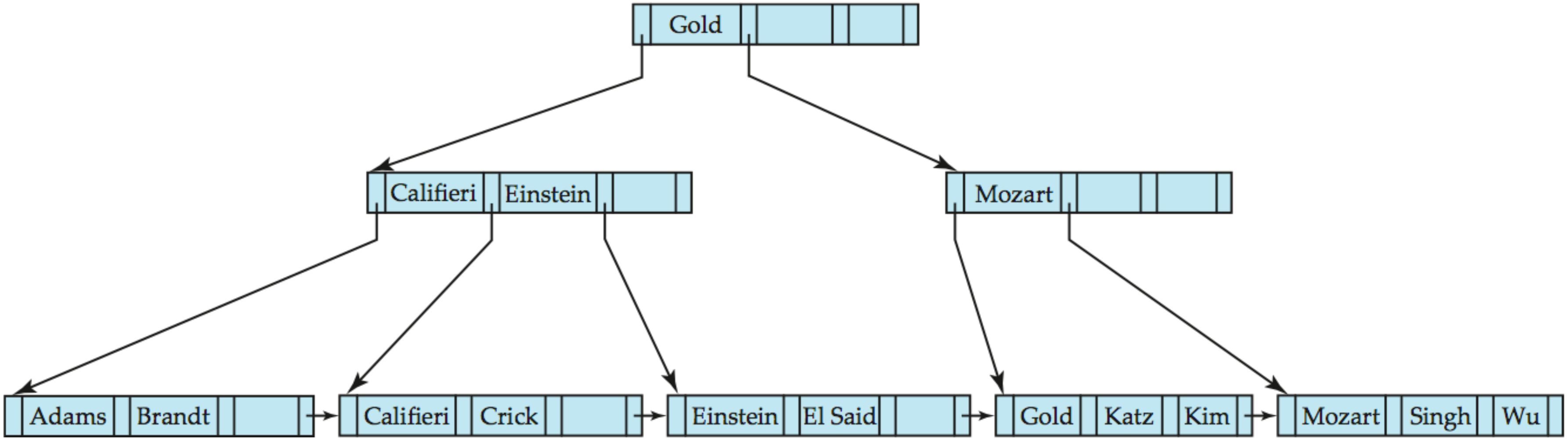
# Updates on B$^+$-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

# Examples of B$^+$-Tree Deletion

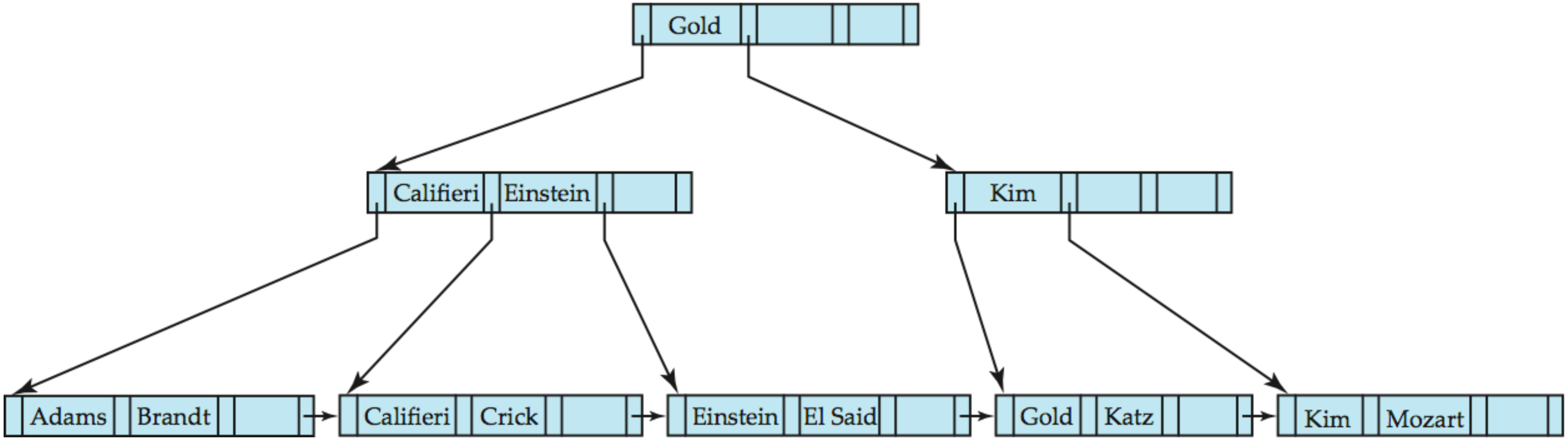


Before and after deleting “Srinivasan”



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

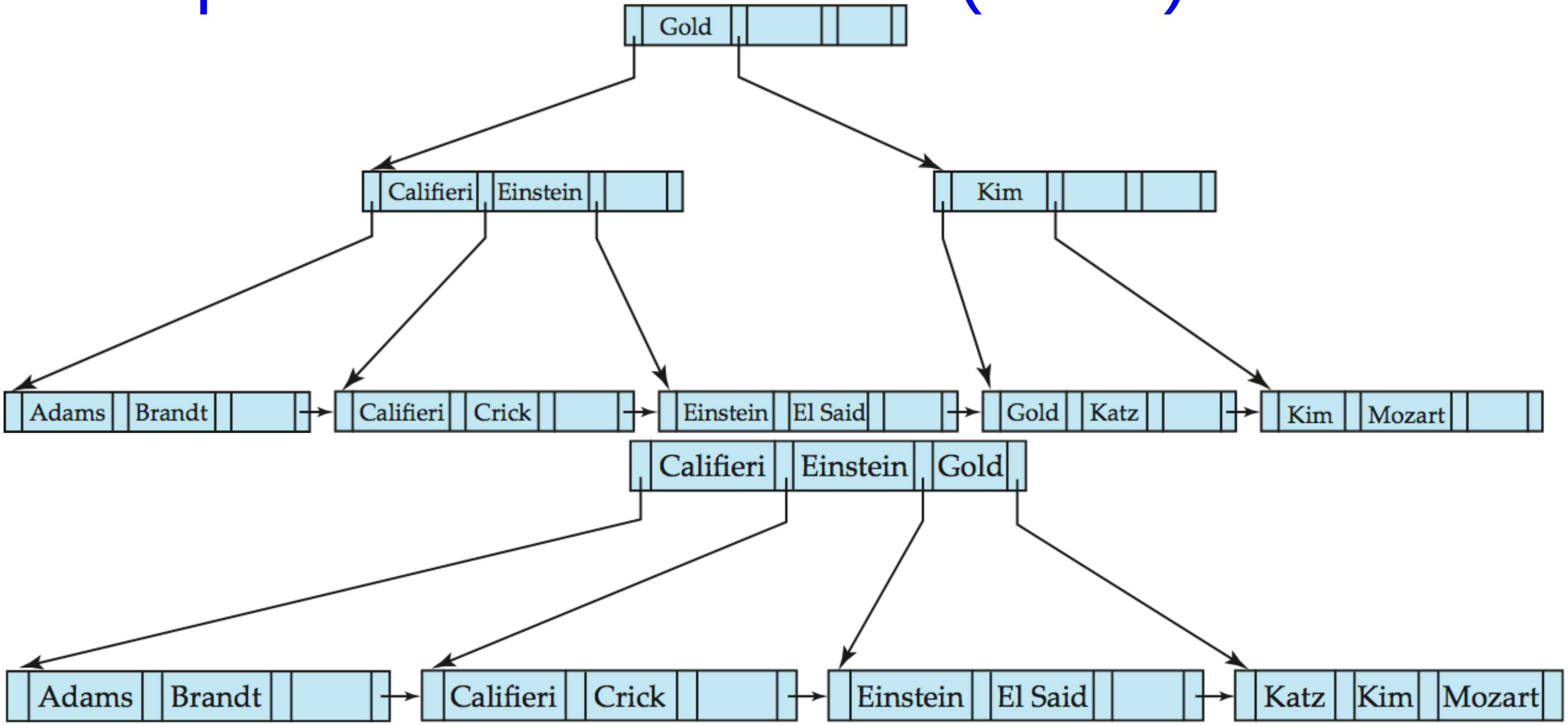
# Examples of B⁺-Tree Deletion (Cont.)



Deletion of "Singh" and "Wu" from result of previous example

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

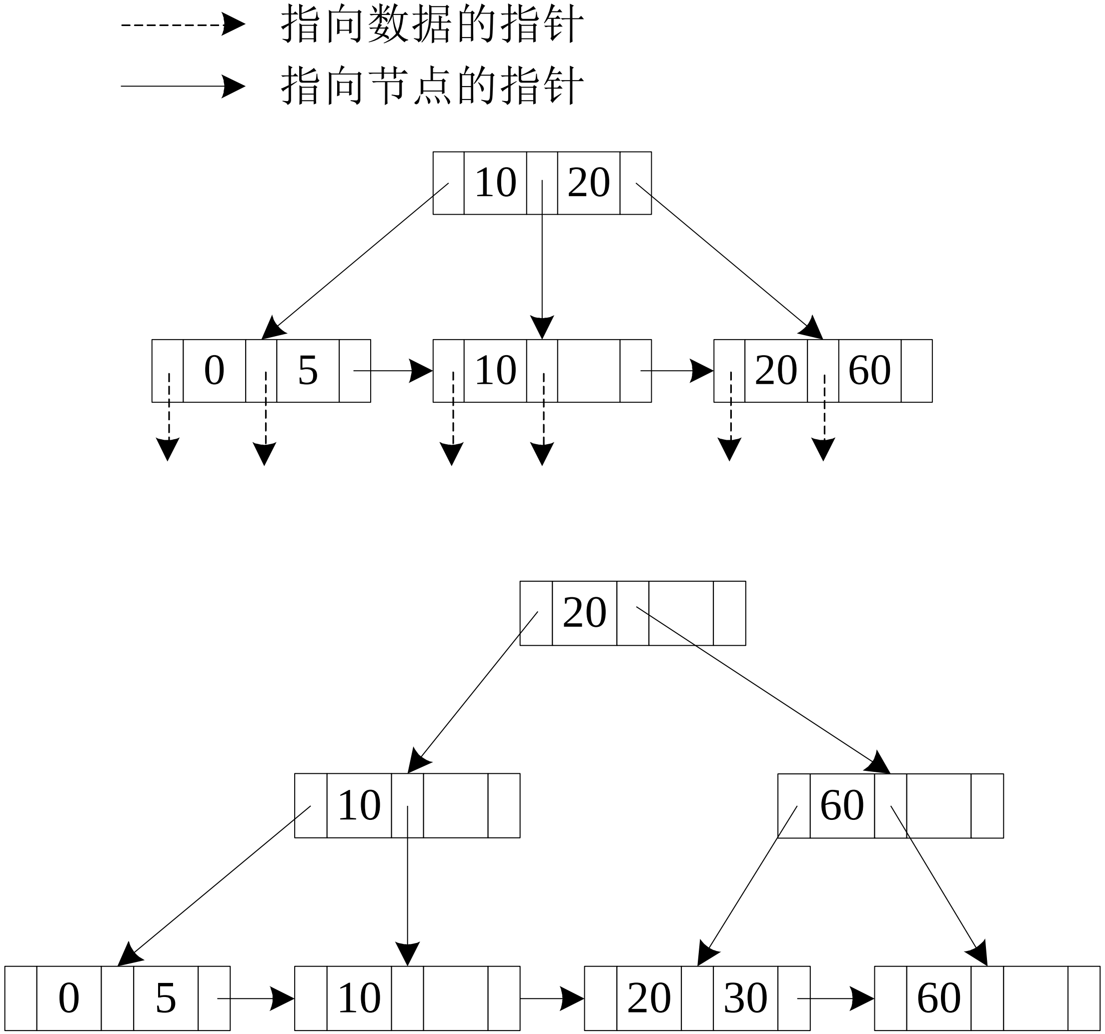
# Example of B$^+$-tree Deletion (Cont.)



Before and after deletion of “Gold” from earlier example

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

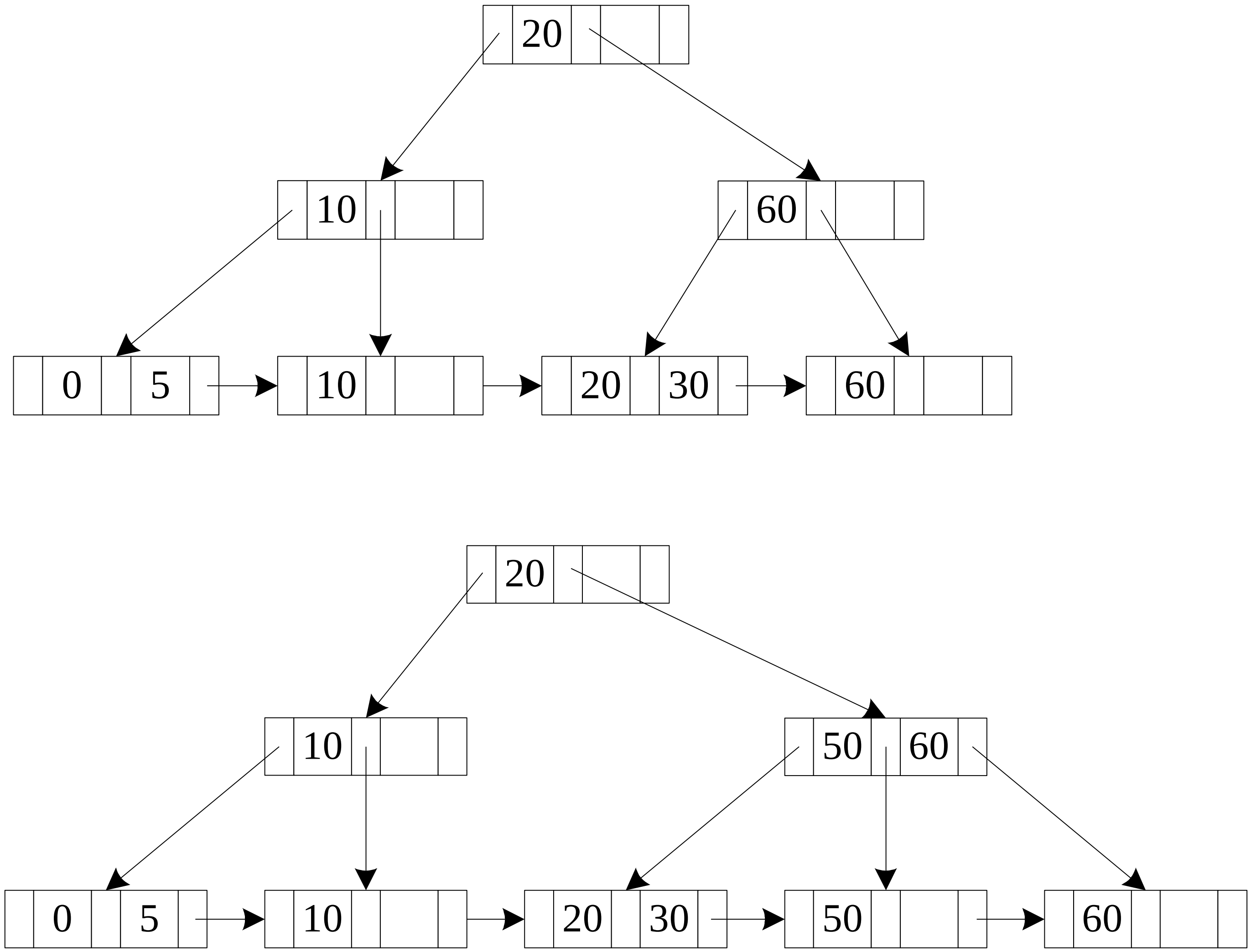
# Example of B+ Tree: insert 30



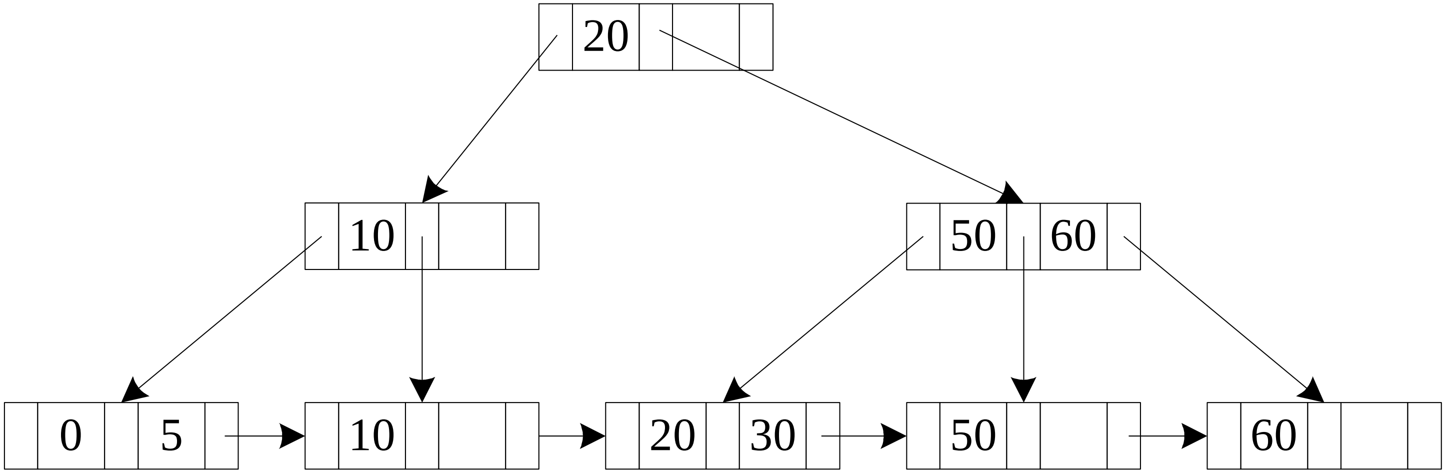
指向数据的指针

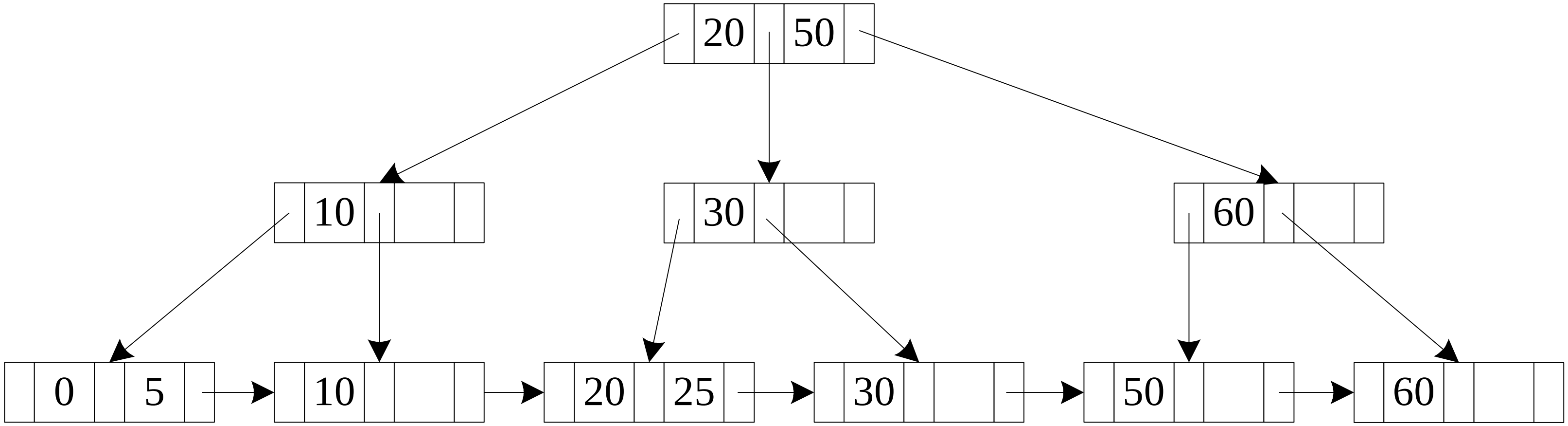
指向节点的指针

# Example of B+ Tree: insert 50

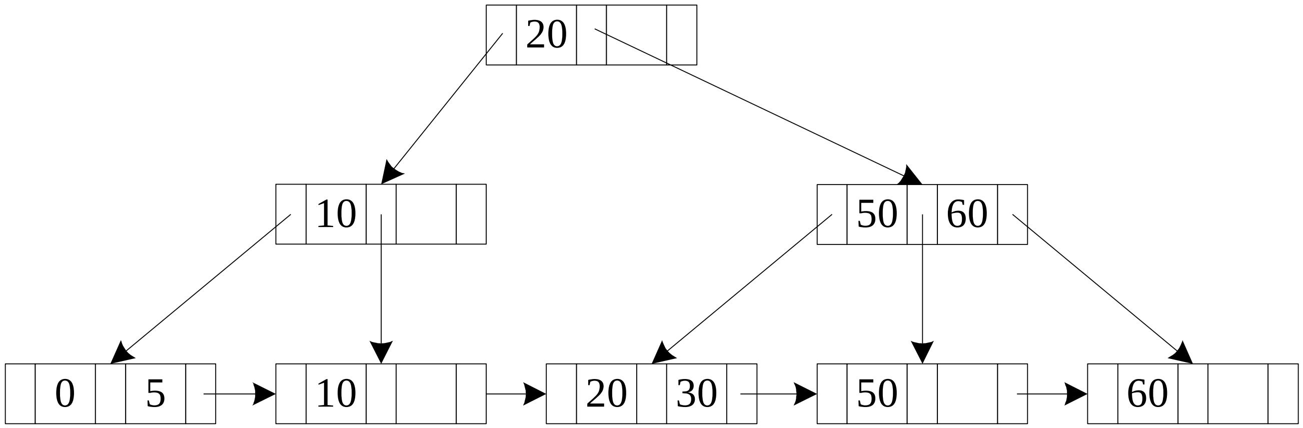


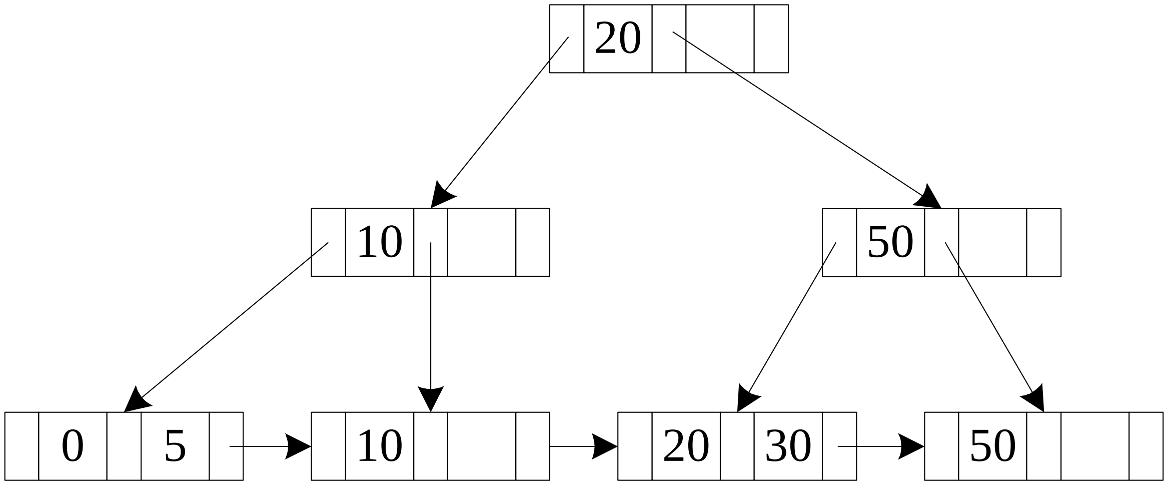
# Example of B+ Tree: insert 25



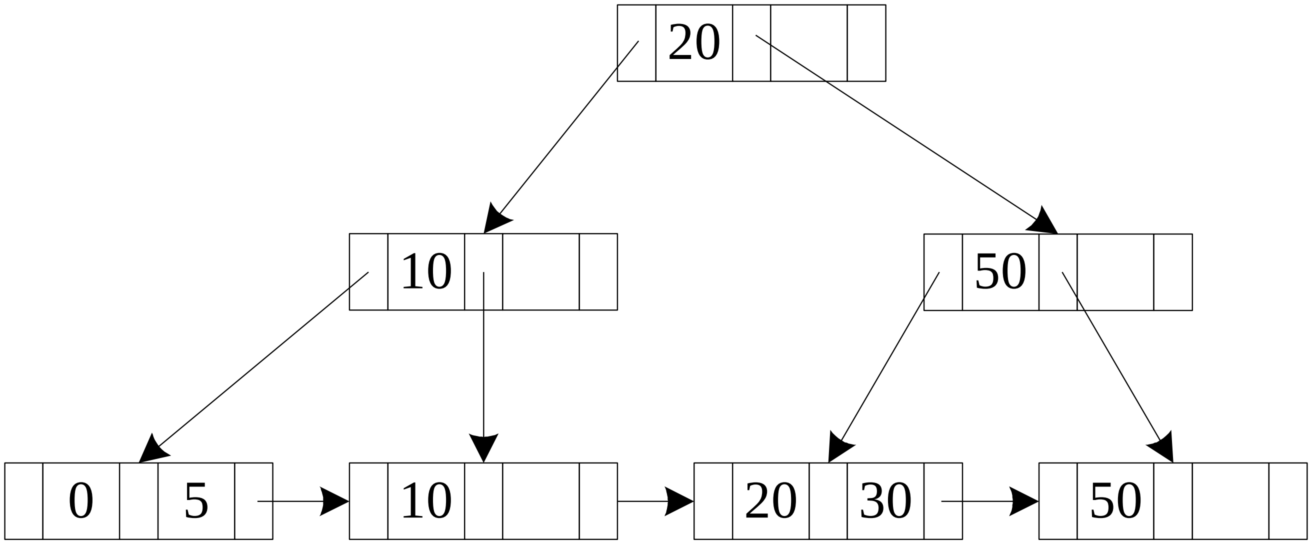


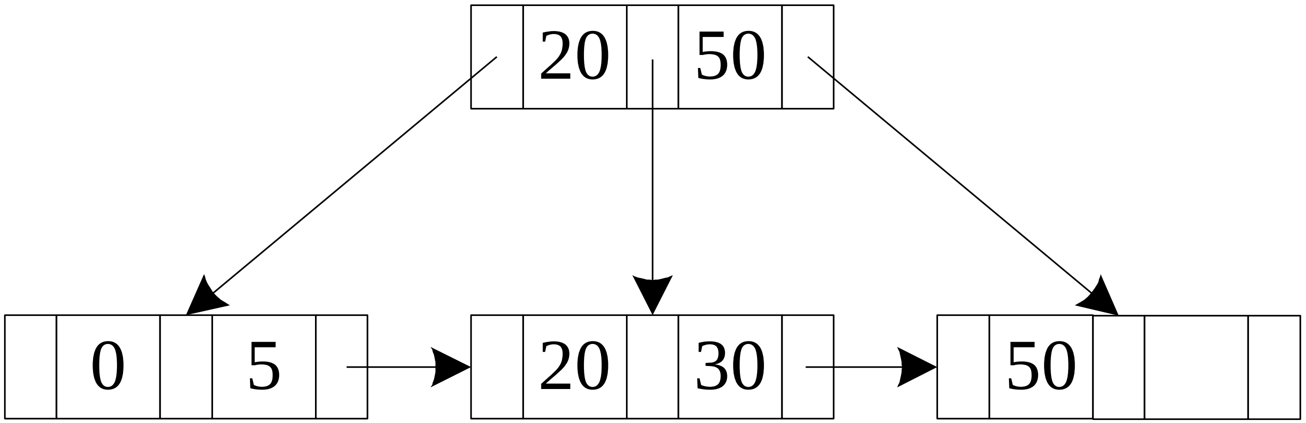
# Example of B+ Tree: delete 60





# Example of B+ Tree: delete 10

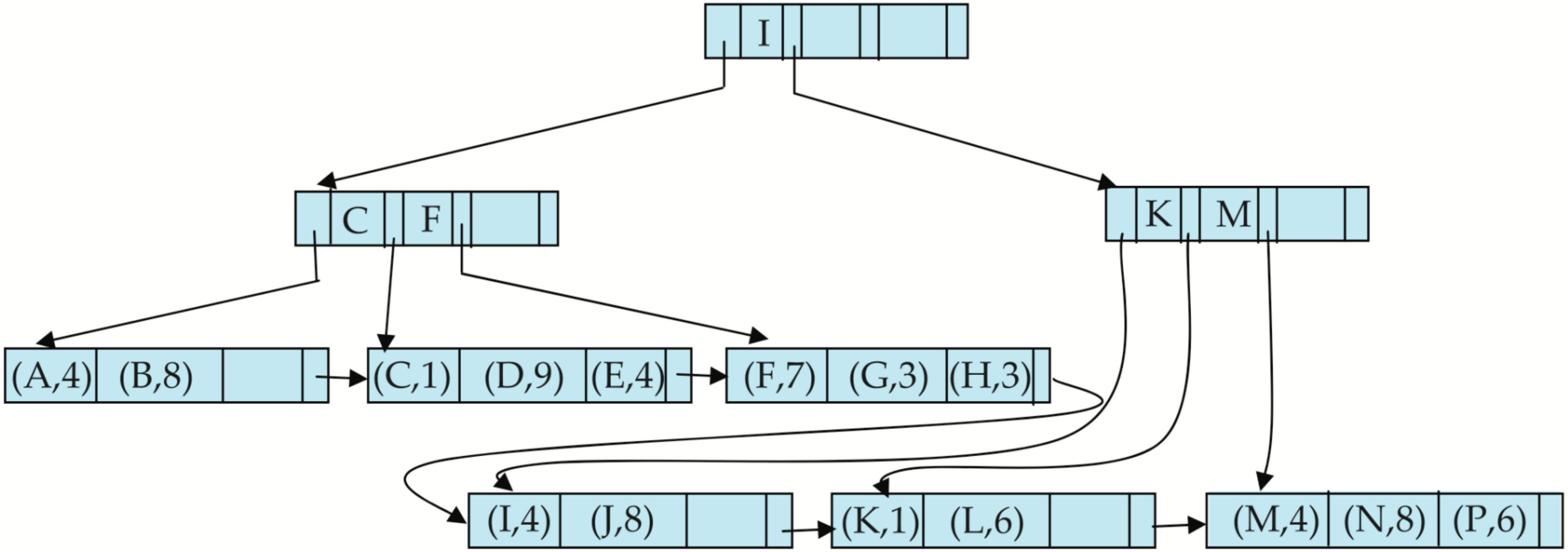




# Non-Unique Search Keys

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

# B$^{+}$-Tree File Organization (Cont.)



Example of B$^{+}$-tree File Organization

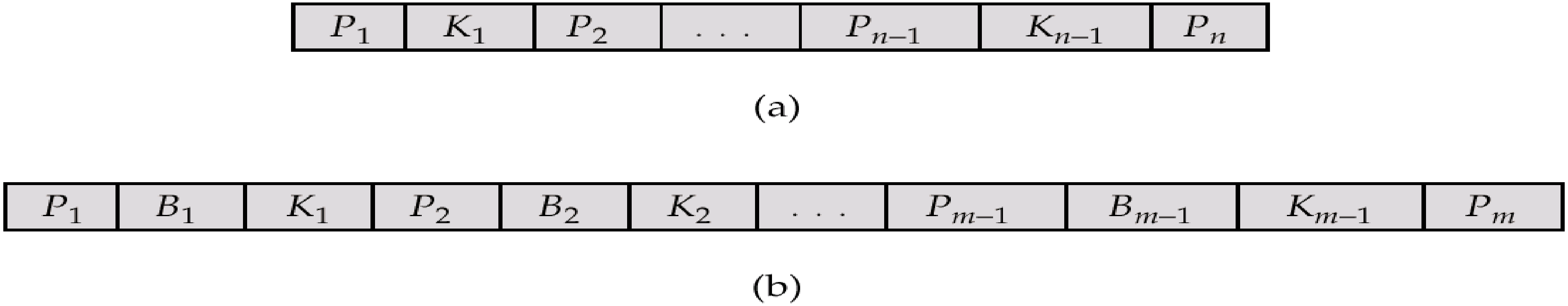
- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least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

# Outline

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

# * 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 non-leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- All nodes are the generalized leaf nodes in the B-tree



| | | | | | | |
|---|---|---|---|---|---|---|
| $P_1$ | $K_1$ | $P_2$ | . . . | $P_{n-1}$ | $K_{n-1}$ | $P_n$ |

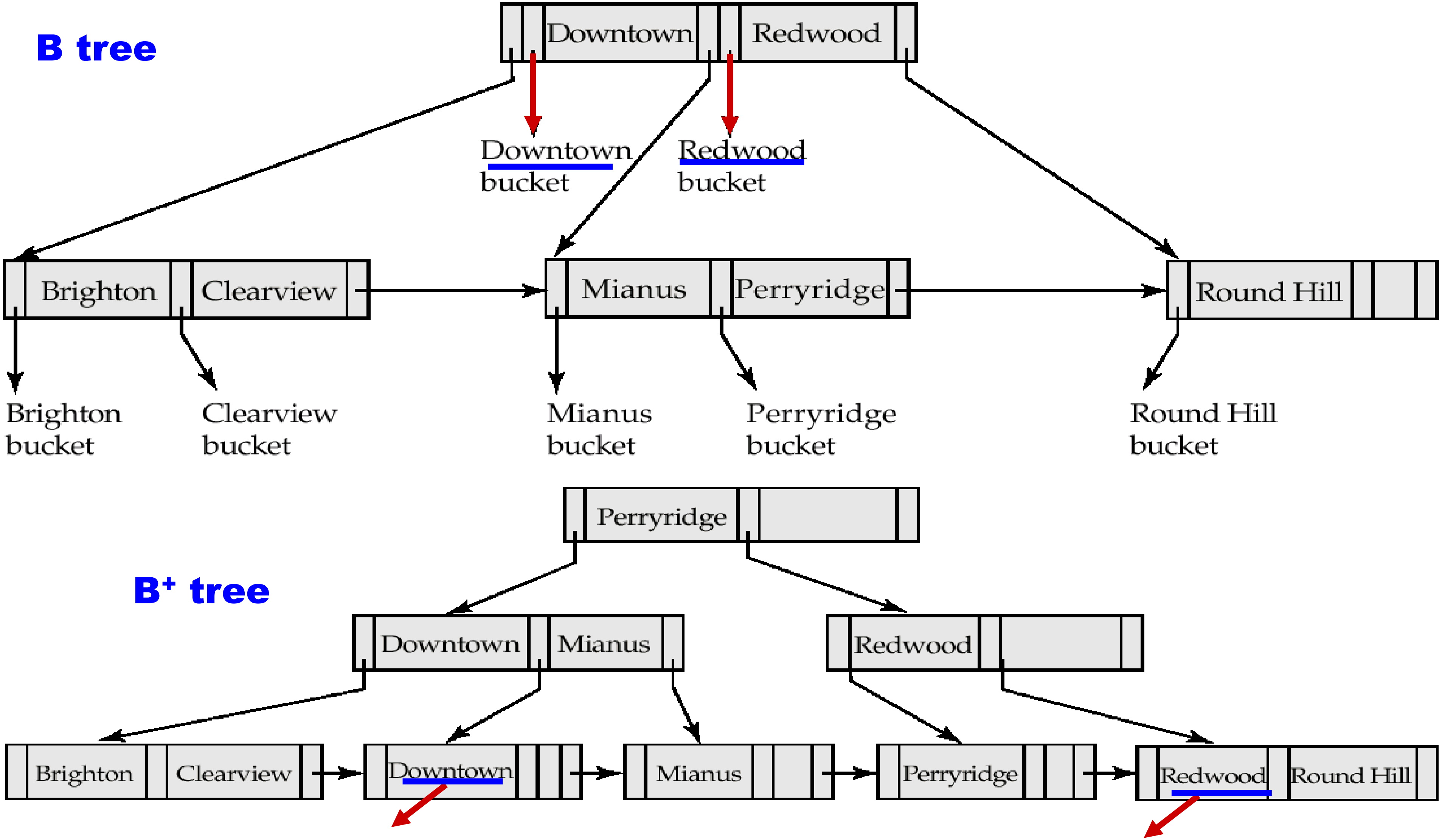
(a)

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

(b)

- Nonleaf node – pointer $B_i$ points the bucket or file record

# 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 (because of duplicate).
  - 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 out weight disadvantages.

# Outline

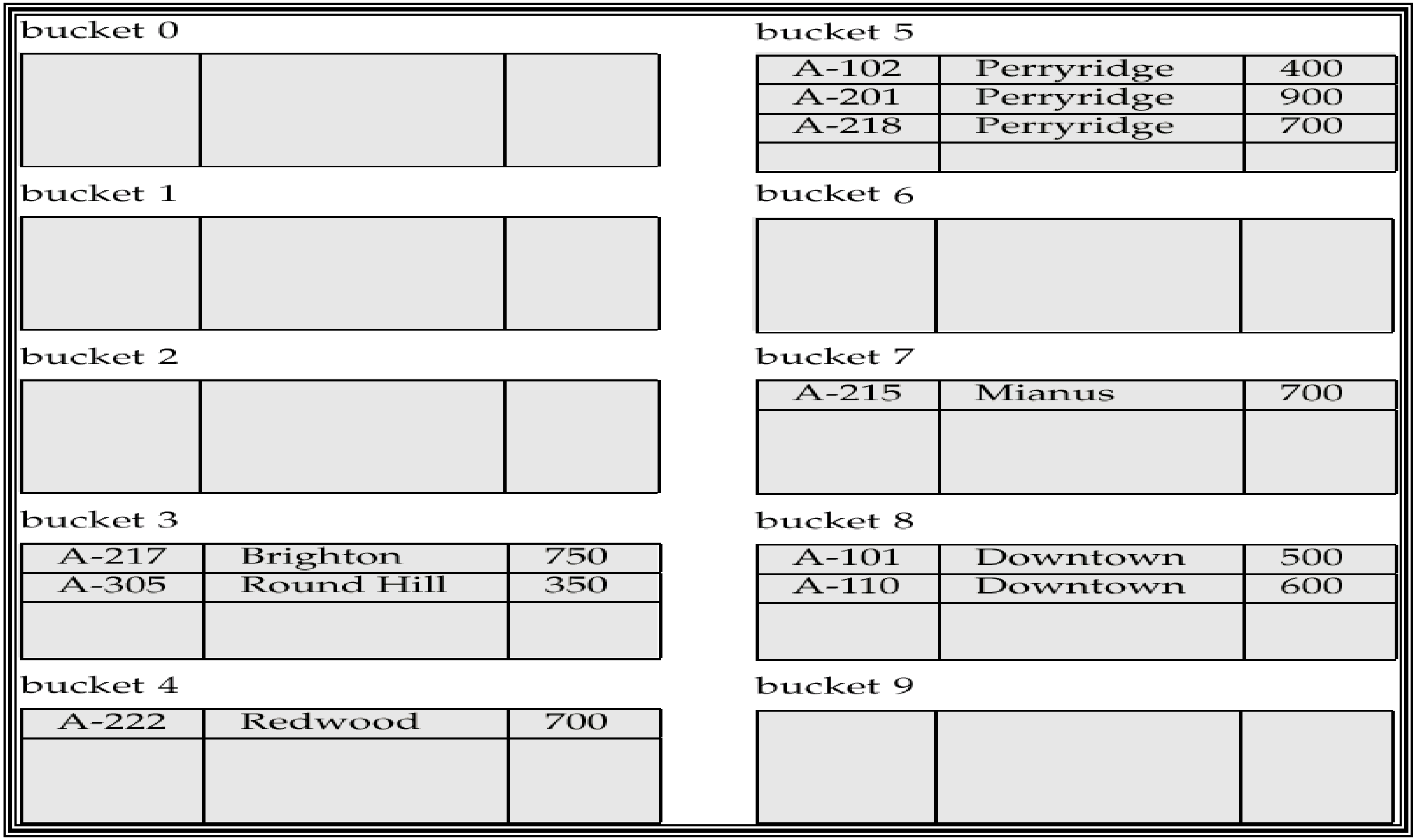
- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- * B-Tree Index Files
- * Static Hashing
- * Dynamic Hashing
- * Comparison of Ordered Indexing and Hashing
- Write-optimized Indices
- * Multiple-Key Access
- Index Definition in SQL

# *Static Hashing

- A bucket is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a hash file organization we obtain the bucket of a record directly 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 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 *account* file, using *branch-name* as key (see previous slide for details).



bucket 0

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

bucket 1

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

bucket 2

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

bucket 3

| | | |
|---|---|---|
| A-217 | Brighton | 750 |
| A-305 | Round Hill | 350 |
| | | |

bucket 4

| | | |
|---|---|---|
| A-222 | Redwood | 700 |
| | | |

bucket 5

| | | |
|---|---|---|
| A-102 | Perryridge | 400 |
| A-201 | Perryridge | 900 |
| A-218 | Perryridge | 700 |
| | | |

bucket 6

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

bucket 7

| | | |
|---|---|---|
| A-215 | Mianus | 700 |
| | | |

bucket 8

| | | |
|---|---|---|
| A-101 | Downtown | 500 |
| A-110 | Downtown | 600 |
| | | |

bucket 9

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

# Hash Functions

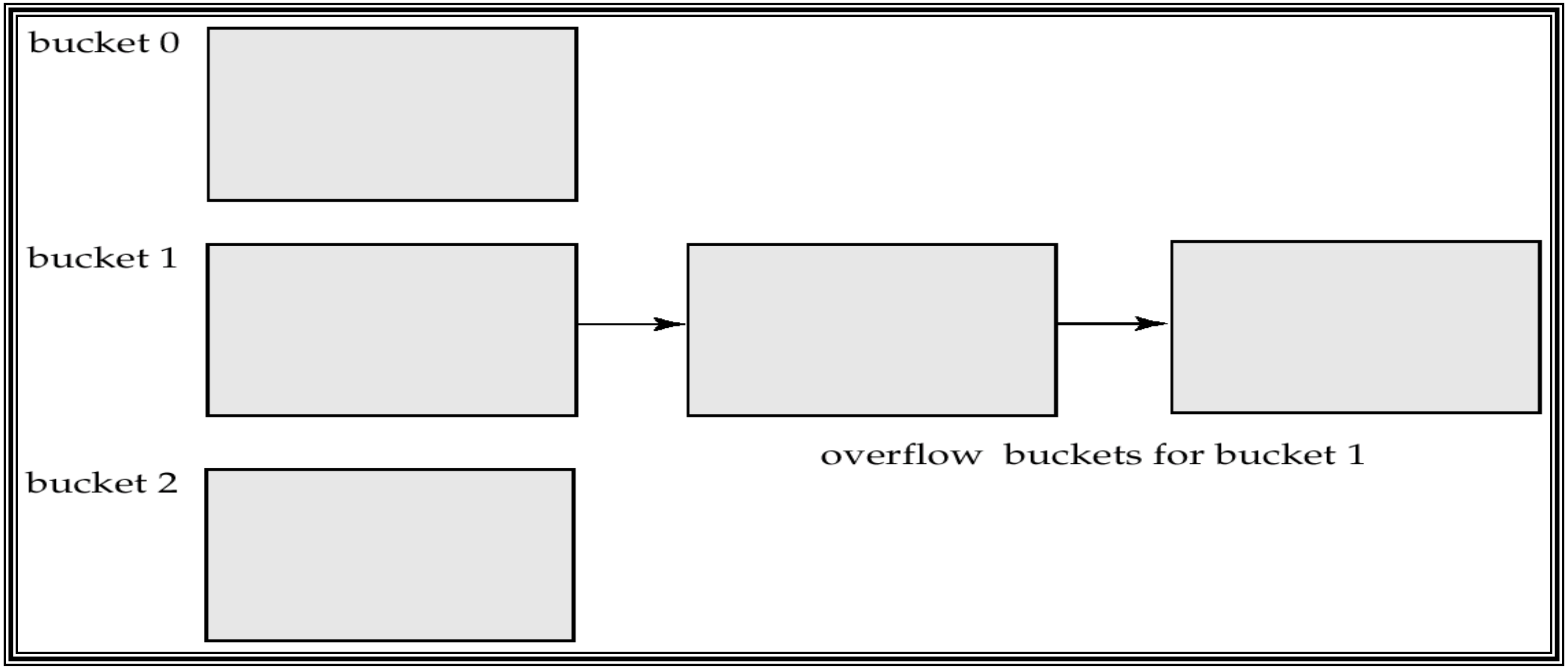
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is random, so 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.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.

# 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.*

# Handling of Bucket Overflows (Cont.)

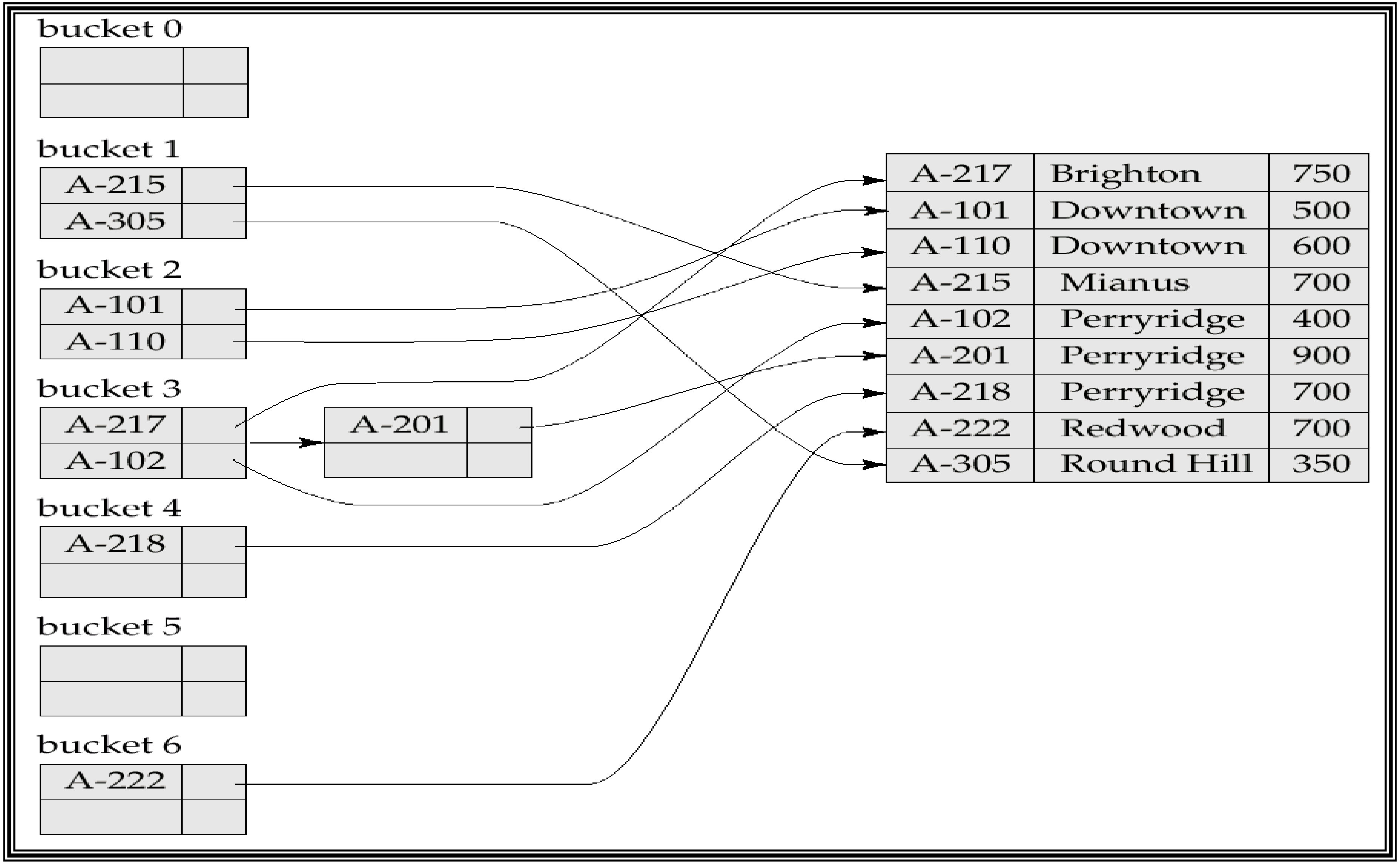
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called closed hashing.
  - An alternative, called open hashing, which does not use overflow buckets,  is not suitable for database applications.



# Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- 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



bucket 0

bucket 1
A-215
A-305

bucket 2
A-101
A-110

bucket 3
A-217
A-102
A-201

bucket 4
A-218

bucket 5

bucket 6
A-222

| | | |
|---|---|---|
| A-217 | Brighton | 750 |
| A-101 | Downtown | 500 |
| A-110 | Downtown | 600 |
| A-215 | Mianus | 700 |
| A-102 | Perryridge | 400 |
| A-201 | Perryridge | 900 |
| A-218 | Perryridge | 700 |
| A-222 | Redwood | 700 |
| A-305 | Round Hill | 350 |

# Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses.
  - Databases grow with time.  If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very expensive.

- These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

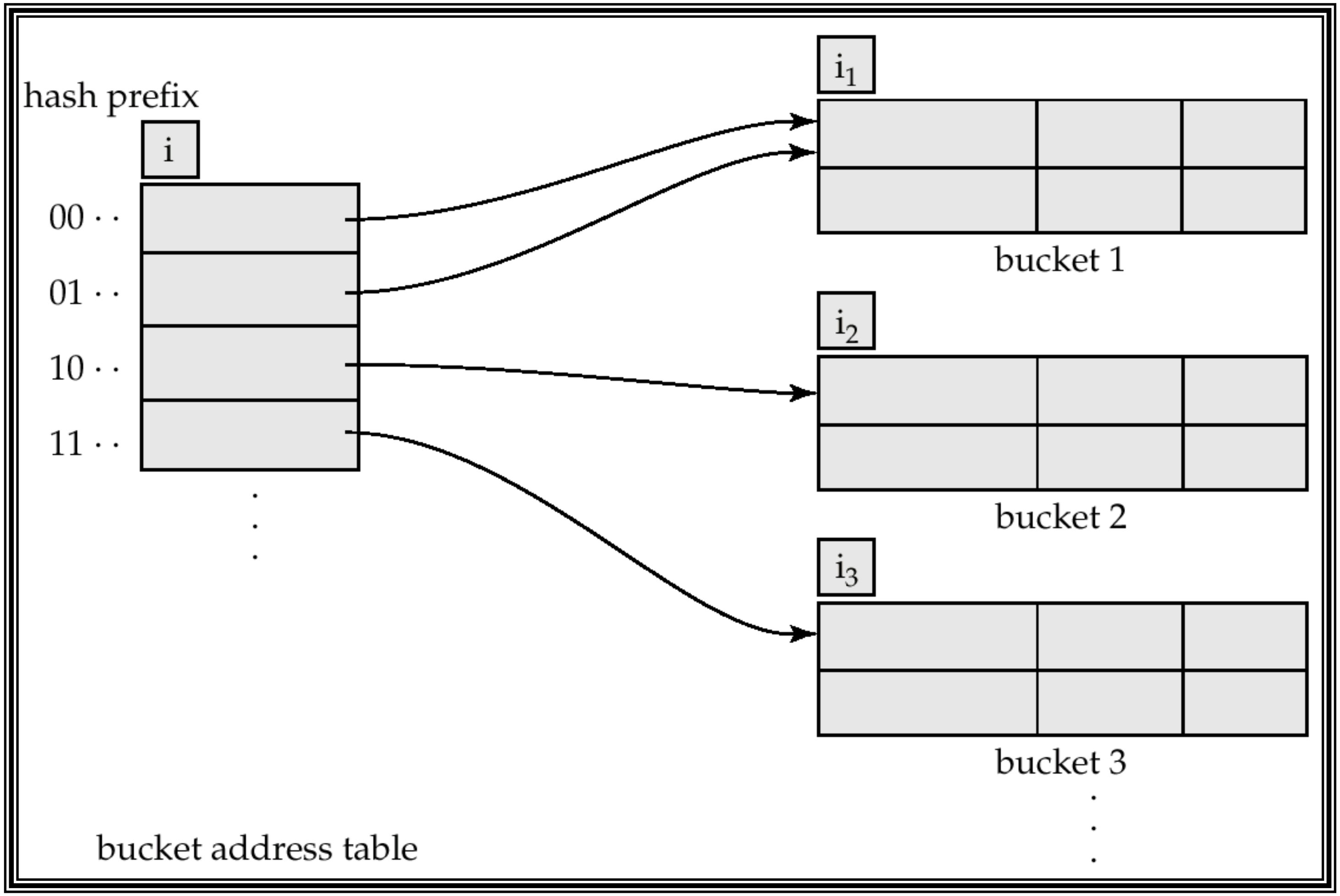
# Outline

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

# *Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- Extendable hashing – one form of dynamic hashing
  - Hash function generates values over a large range — typically *b*-bit integers, with *b* = 32.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be *i* bits, $0 \leq i \leq 32$.
  - Bucket address table size = $2^{i.}$ Initially *i* = 0
  - Value of *i* grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket.
  - Thus, actual number of buckets is < $2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

# General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

# Use of Extendable Hash Structure

- Each bucket *j* stores a value $i_j$; all the entries that point to the same bucket have the same values on the first $i_j$ bits.
- To locate the bucket containing search-key $K_j$:
  1. Compute $h(K_j) = X$
  2. Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value $K_j$
  - follow same procedure as look-up and locate the bucket, say *j*.
  - If there is room in the bucket *j*, insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)
    - Overflow buckets used instead in some cases (will see shortly)

# Updates in Extendable Hash Structure

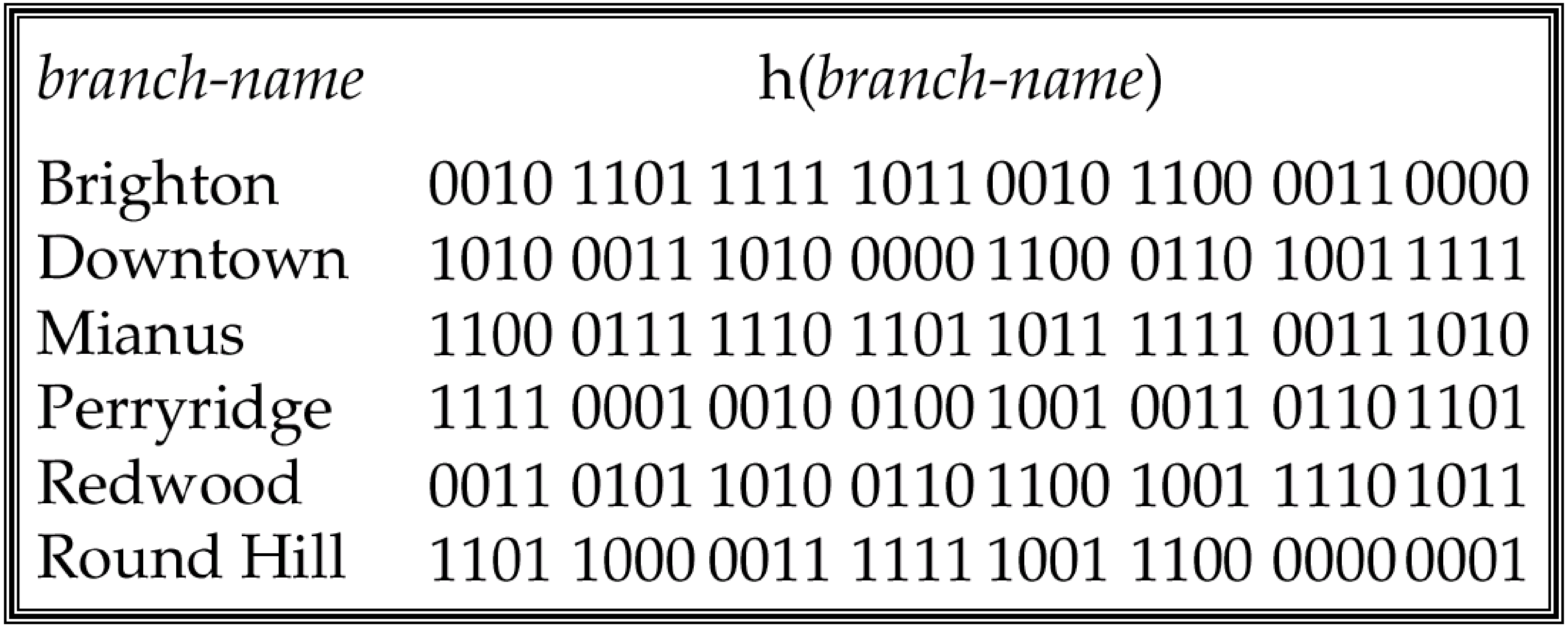
To split a bucket *j* when inserting record with search-key value $K_j$:

- ❑ If $i > i_j$ (more than one pointer to bucket *j*)
  - ➢ allocate a new bucket *z*, and set $i_j$ and $i_z$ to the old $i_j$ -+ 1.
  - ➢ make the second half of the bucket address table entries pointing to *j* to point to *z*
  - ➢ remove and reinsert each record in bucket *j.*
  - ➢ recompute new bucket for $K_j$ and insert record in the bucket (further splitting is required if the bucket is still full)
- ❑ If $i = i_j$ (only one pointer to bucket *j*)
  - ➢ increment *i* and double the size of the bucket address table.
  - ➢ replace each entry in the table by two entries that point to the same bucket.
  - ➢ recompute new bucket address table entry for $K_j$<br>Now $i > i_j$ so use the first case above.

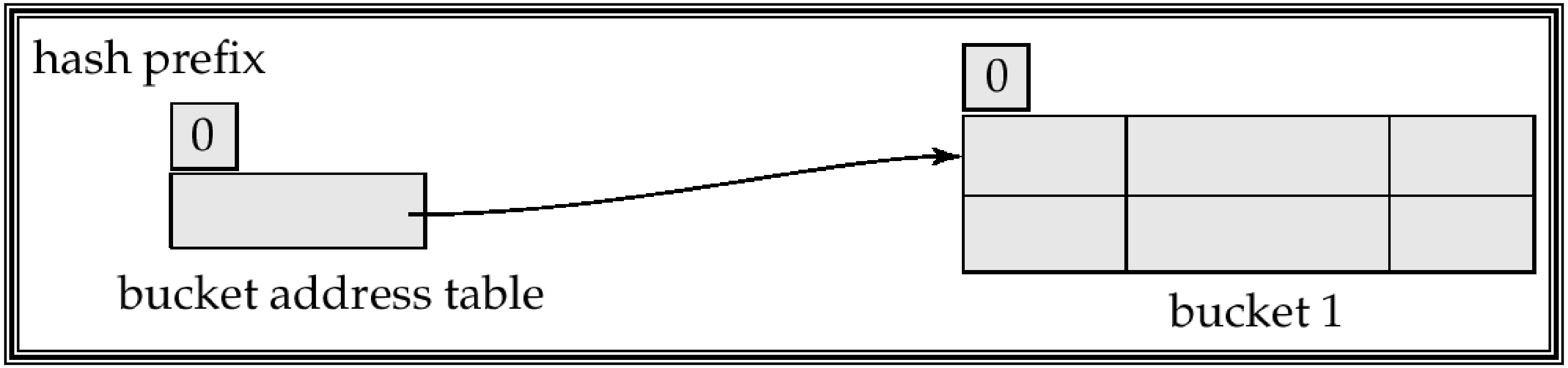
# Updates in Extendable Hash Structure (Cont.)

- When inserting a value, if the bucket is full after several splits (that is, *i* reaches some limit *b*) create an overflow bucket instead of splitting bucket entry table further.
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "buddy" bucket having same value of $i_j$ and same $i_j$ –1 prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

# Use of Extendable Hash Structure:  Example



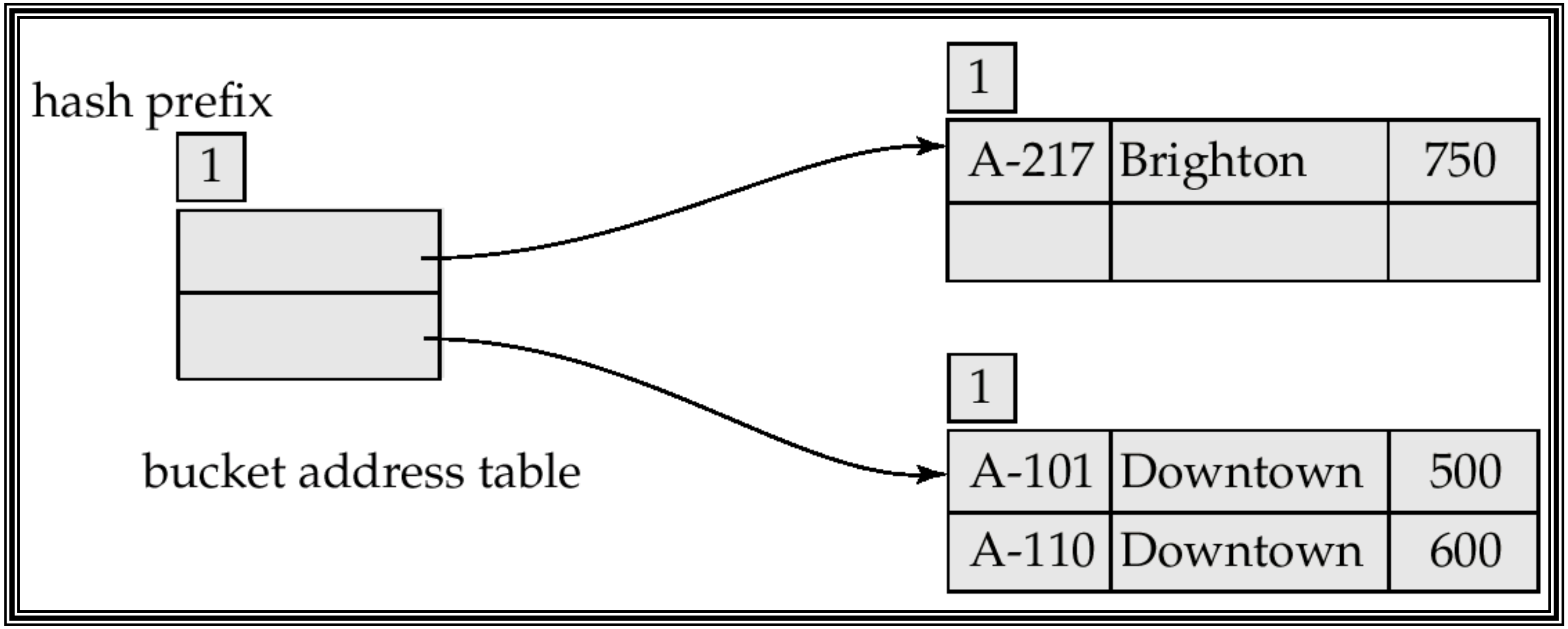
| *branch-name* | h(*branch-name*) |
|---|---|
| Brighton | 0010 1101 1111 1011 0010 1100 0011 0000 |
| Downtown | 1010 0011 1010 0000 1100 0110 1001 1111 |
| Mianus | 1100 0111 1110 1101 1011 1111 0011 1010 |
| Perryridge | 1111 0001 0010 0100 1001 0011 0110 1101 |
| Redwood | 0011 0101 1010 0110 1100 1001 1110 1011 |
| Round Hill | 1101 1000 0011 1111 1001 1100 0000 0001 |



Initial Hash structure, bucket size = 2

# Example (Cont.)

- Hash structure after  insertion of one Brighton and two Downtown records

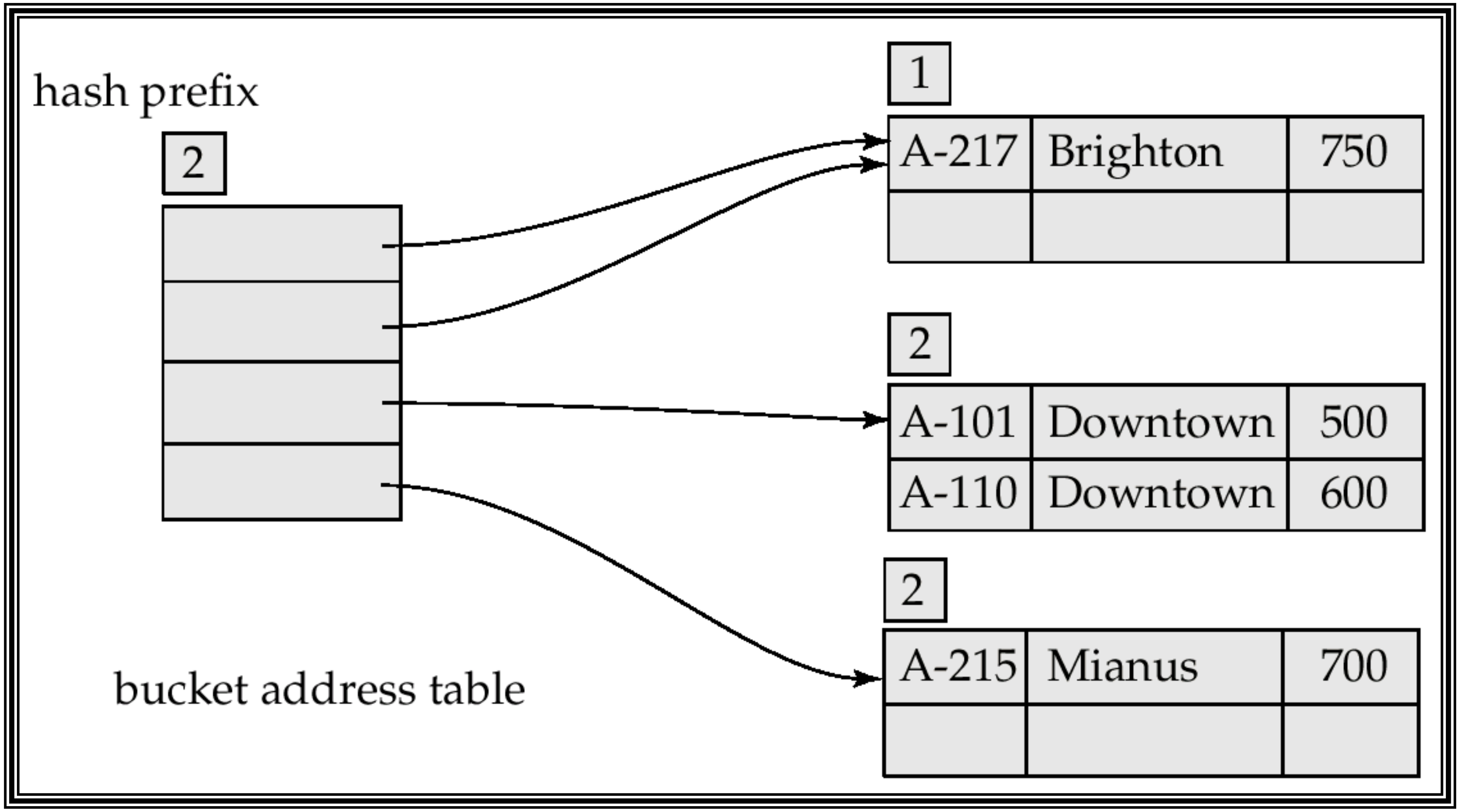


| | | |
|---|---|---|
| A-217 | Brighton | 750 |
| | | |

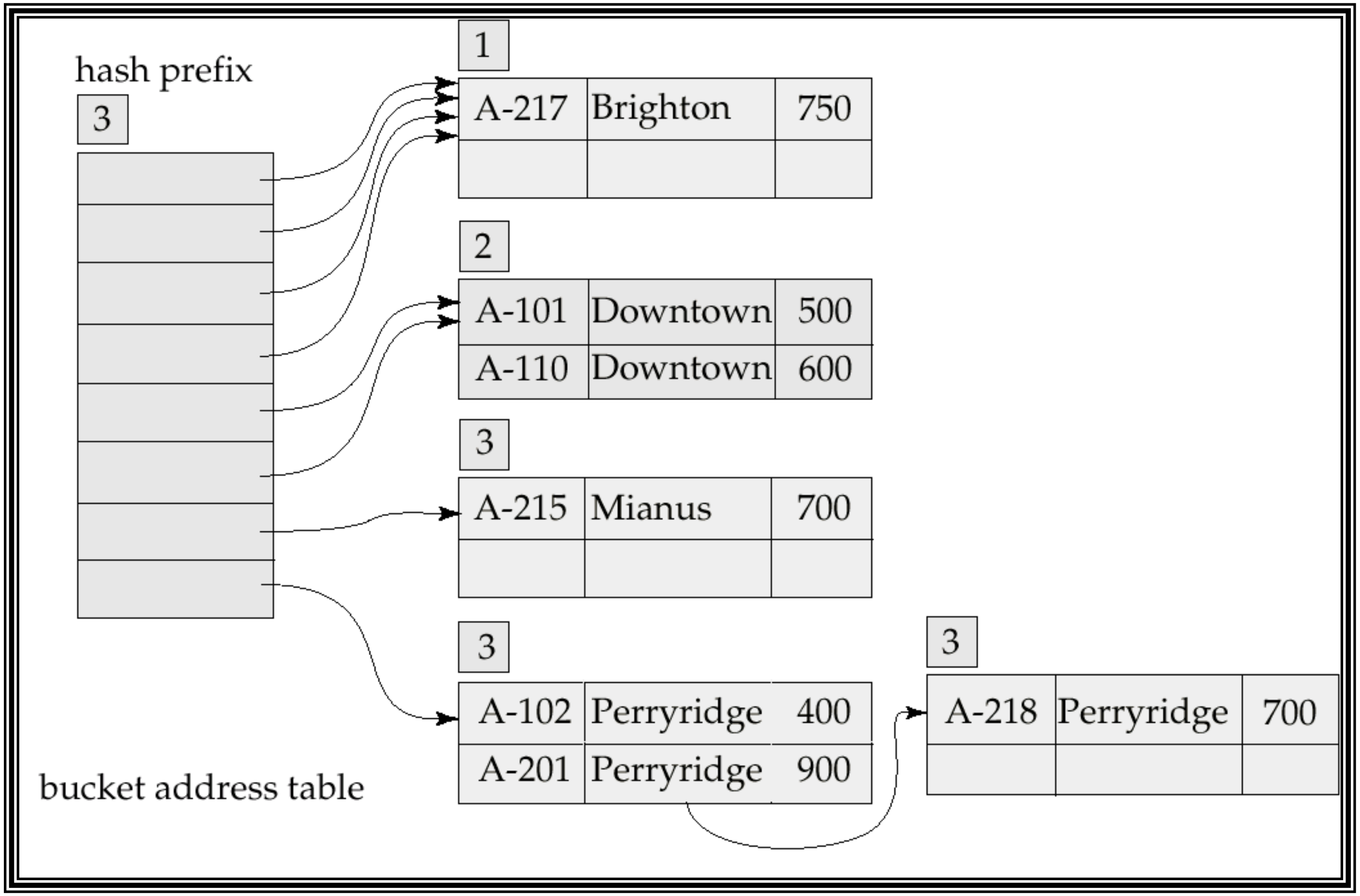
| | | |
|---|---|---|
| A-101 | Downtown | 500 |
| A-110 | Downtown | 600 |

# Example (Cont.)

Hash structure after insertion of Mianus record



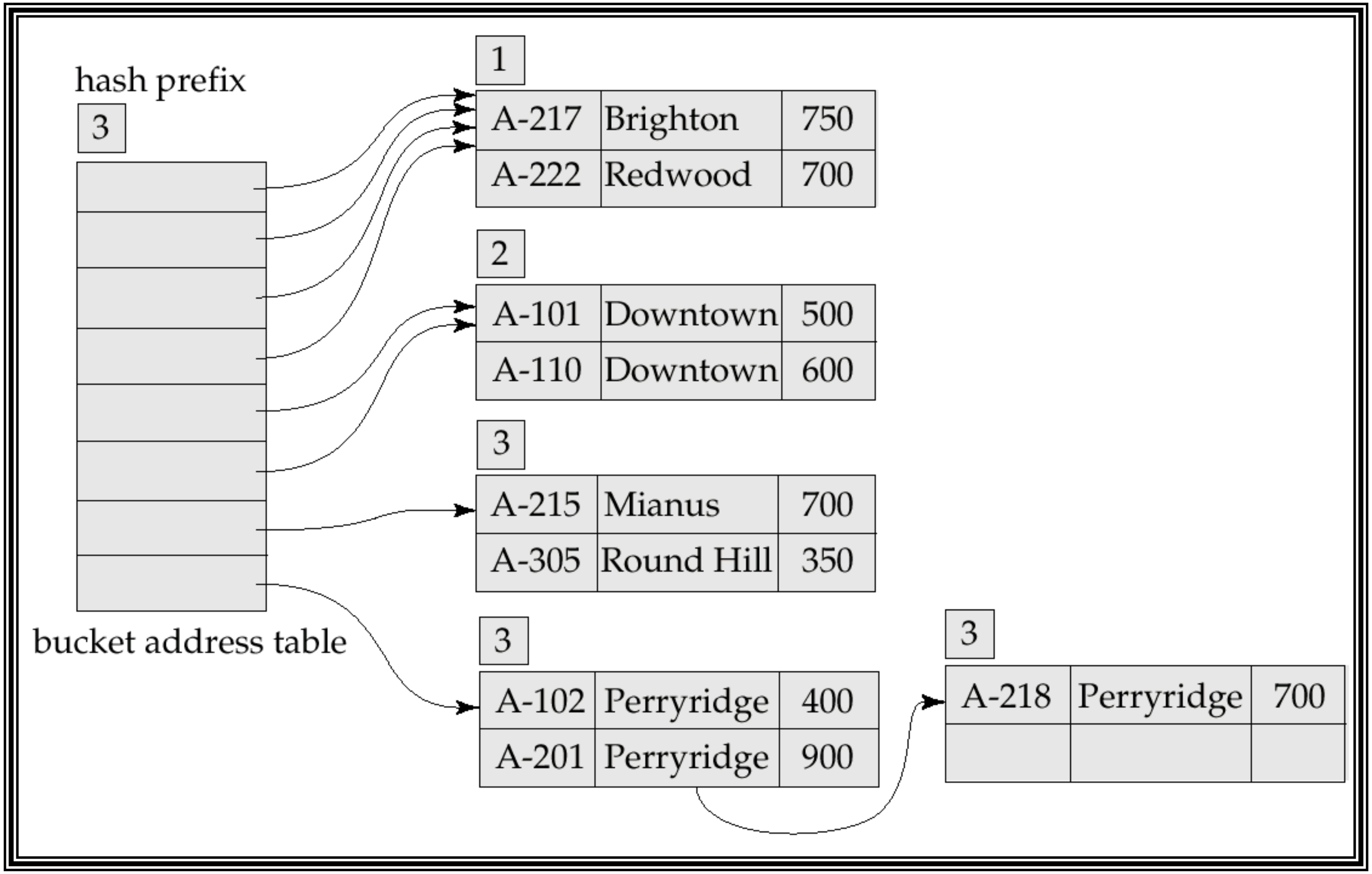
# Example (Cont.)



Hash structure after insertion of three Perryridge records

# Example (Cont.)

- Hash structure after insertion of Redwood and Round Hill records



# Outline

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

# *Comparison of Ordered Indexing and Hashing

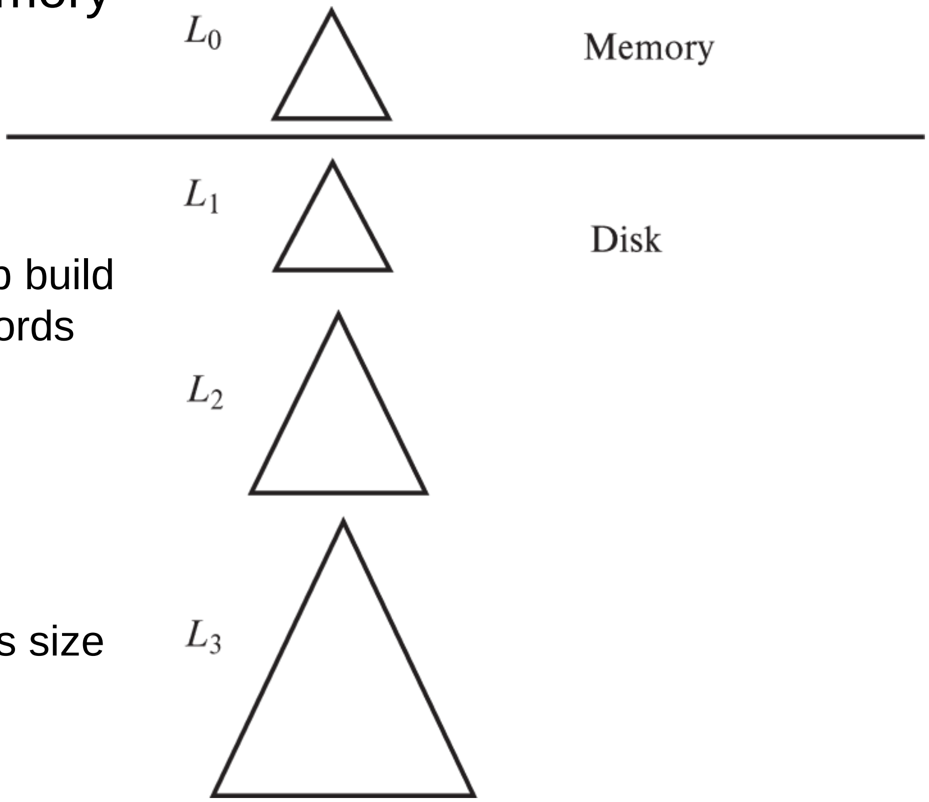
- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred

# Outline

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

# Log Structured Merge (LSM) Tree

- Consider only inserts/queries for now
- Records inserted first into in-memory tree ($L_0$ tree)
- When in-memory tree is full, records moved to disk ($L_1$ tree)
  - $B^+$-tree constructed using bottom-up build by merging existing $L_1$ tree with records from $L_0$ tree
- When $L_1$ tree exceeds some threshold, merge into $L_2$ tree
  - And so on for more levels
  - Size threshold for $L_{i+1}$ tree is *k* times size threshold for $L_i$ tree



# LSM Tree (Cont.)

- Benefits of LSM approach
  - Inserts are done using only sequential I/O operations
  - Leaves are full, avoiding space wastage
  - Reduced number of I/O operations per record inserted as compared to normal $B^+$-tree (up to some size)
- Drawback of LSM approach
  - Queries have to search multiple trees
  - Entire content of each level copied multiple times
- Stepped-merge index
  - Variant of LSM tree with multiple trees at each level
  - Reduces write cost compared to LSM tree
  - But queries are even more expensive
    - Bloom filters to avoid lookups in most trees
- Details are covered in Chapter 24

# LSM Trees (Cont.)

- Deletion handled by adding special “delete” entries
  - Lookups will find both original entry and the delete entry, and must return only those entries that do not have matching delete entry
  - When trees are merged, if we find a delete entry matching an original entry, both are dropped.

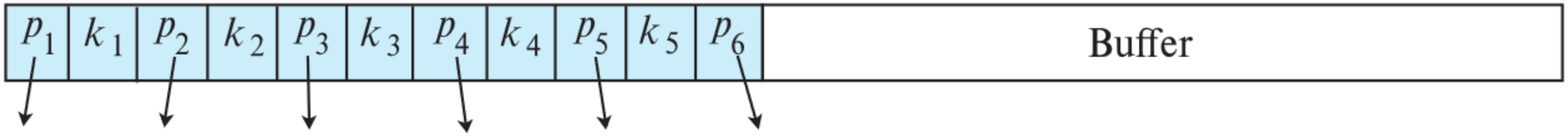
- Update handled using insert+delete

- LSM trees were introduced for disk-based indices
  - But useful to minimize erases with flash-based indices
  - The stepped-merge variant of LSM trees is used in many BigData storage systems
    - Google BigTable, Apache Cassandra, MongoDB
    - And more recently in SQLite4, LevelDB, and MyRocks storage engine of MySQL

# Buffer Tree

- Alternative to LSM tree
- Key idea: each internal node of $B^+$-tree has a buffer to store inserts
  - Inserts are moved to lower levels when buffer is full
  - With a large buffer, many records are moved to lower level each time
  - Per record I/O decreases correspondingly
- Benefits
  - Less overhead on queries
  - Can be used with any tree index structure
  - Used in PostgreSQL Generalized Search Tree (GiST) indices
- Drawback: more random I/O than LSM tree

Internal node



# Outline

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

# Index Definition in SQL

- Create an index:

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

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

  create index *cust-strt-city-index* on *customer (customer-city, customer-street);*

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

  E.g.: create unique index *uni-acnt-index* on *account(account-number);*

  - Not really required if SQL unique integrity constraint is supported

- To drop an index

  drop index <index-name>

# Outline

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

# *Multiple-Key Access

- Use multiple indices for certain types of queries.

- Example:

  select *account-number*
  from *account*
  where *branch-name* = “Perryridge” and *balance* = 1000

- Possible strategies for processing query using indices on single attributes:

  1. Use index on *branch-name* to find accounts with *branch-name* = “Perryridge” , and test if balances = $1000;.
  2. Use index on *balance* to find accounts with balances of $1000; test *branch-name* = “Perryridge”.
  3. Use *branch-name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained.

# (1) Indices on Multiple Attributes

Suppose we have an index on combined search-key
(*branch-name, balance*).

- With the where clause
  where *branch-name* = "Perryridge" and *balance* = 1000
  the index on the combined search-key will fetch only records that satisfy both conditions.
  Using separate indices in less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
  where *branch-name* = "Perryridge" and *balance* < 1000
- But cannot efficiently handle
  where *branch-name* < "Perryridge" and *balance* = 1000
  May fetch many records that satisfy the first but not the second condition.

# (2) Grid Files

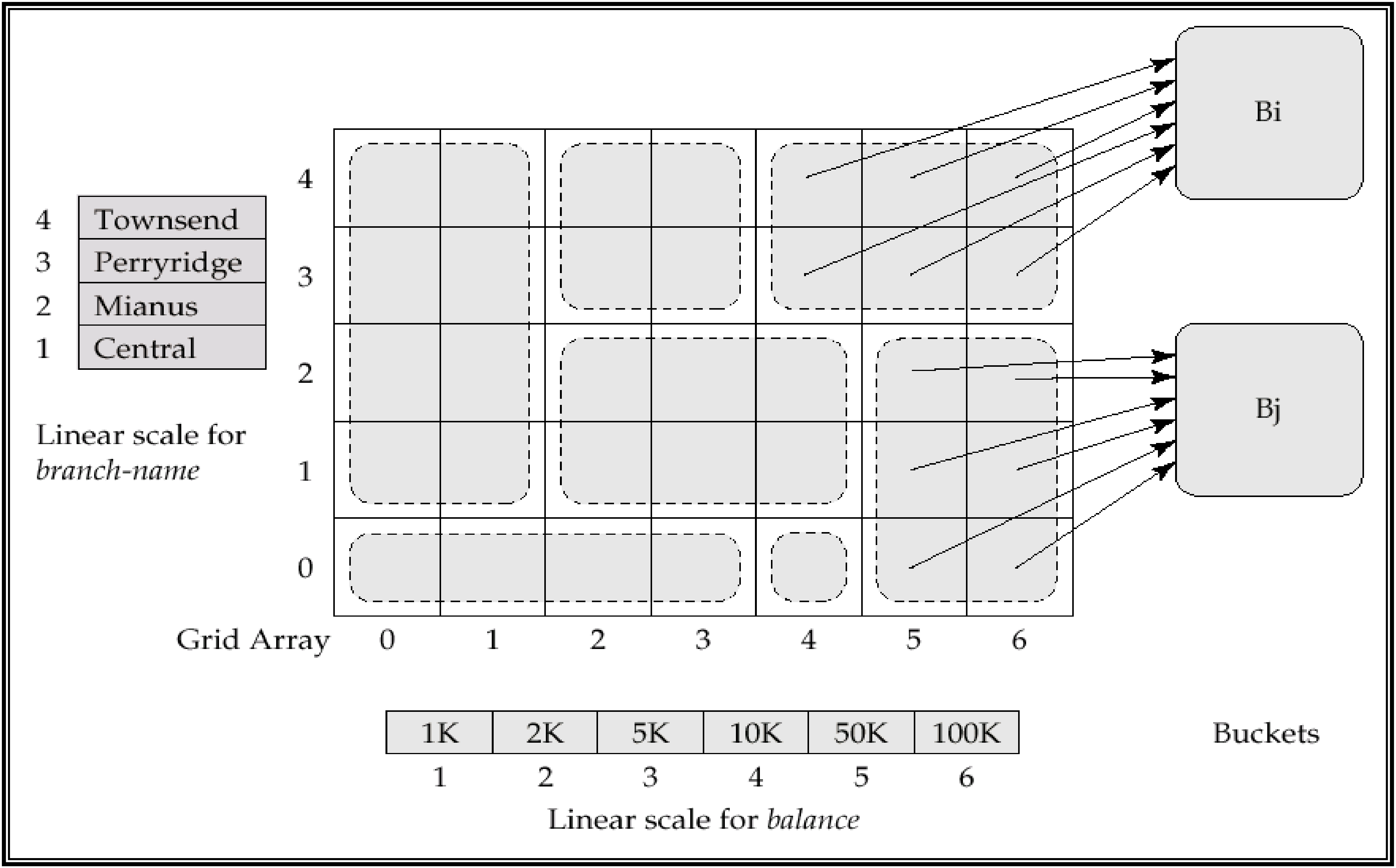
- Structure used to speed the processing of general multiple search-key queries involving one or more comparison operators

- The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes

- Multiple cells of grid array can point to same bucket

- To find the bucket for a search-key value, locate the row and column of its cell using the linear scales and follow pointer

# Example Grid File for *account*



|  |  |
|---|---|
| 4 | Townsend |
| 3 | Perryridge |
| 2 | Mianus |
| 1 | Central |

Linear scale for *branch-name*

| 1K | 2K | 5K | 10K | 50K | 100K |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

Linear scale for *balance*

# Grid Files (Cont.)

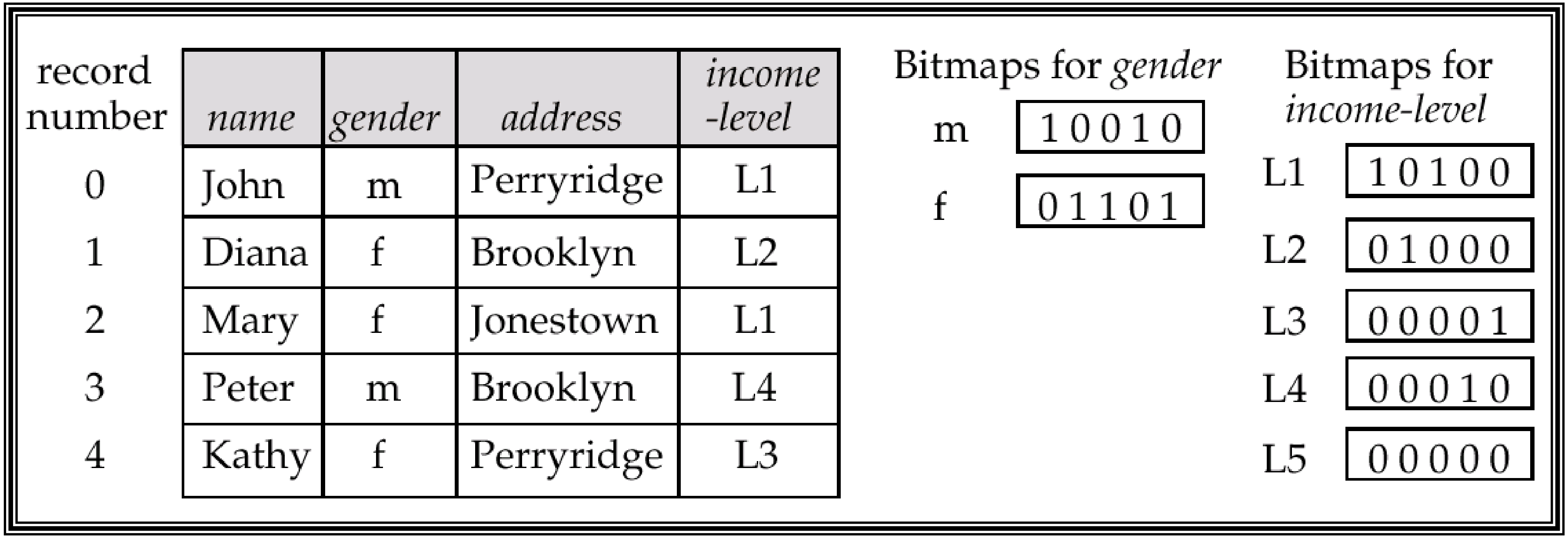
- During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it.
  - Idea similar to extendable hashing, but on multiple dimensions
  - If only one cell points to it, either an overflow bucket must be created or the grid size must be increased
- Linear scales must be chosen to uniformly distribute records across cells.
  - Otherwise there will be too many overflow buckets.
- Periodic re-organization to increase grid size will help.
  - But reorganization can be very expensive.
- Space overhead of grid array can be high.
- R-trees (Chapter 23) are an alternative

# (3) Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- 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)
- A bitmap is simply an array of bits

# Bitmap Indices (Cont.)

- In its 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 | *name* | *gender* | *address* | *income -level* |
|---|---|---|---|---|
| 0 | John | m | Perryridge | L1 |
| 1 | Diana | f | Brooklyn | L2 |
| 2 | Mary | f | Jonestown | L1 |
| 3 | Peter | m | Brooklyn | L4 |
| 4 | Kathy | f | Perryridge | 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.)

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

# Bitmap Indices (Cont.)

- 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 number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complementation
    - not(*A*=*v*): *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for NOT(*A*=*v*):
    - intersect above result with (NOT *bitmap-A-Null*)

# Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
  - E.g. 1-million-bit maps can be anded with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of $B^+$-trees, for values that have a large number of matching records
  - Worthwhile if > 1/64 of the records have that value, assuming a tuple-id is 64 bits
  - Above technique merges benefits of bitmap and $B^+$-tree indices

# Q & A

Your questions and suggestions are expected for me.

Thanks a lot!

Exercises: 14.3[a][c], 14.4, and 14.17 (see Pages 679-682)

Questions?

Questions?