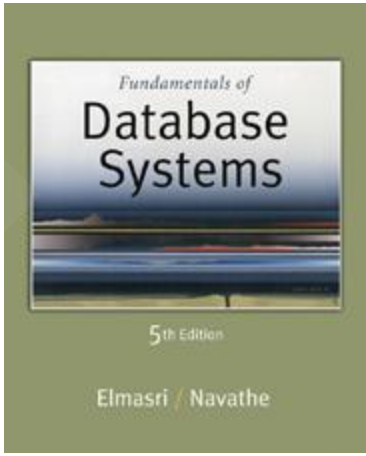
# Chapter 18

## Indexing Structures for Files



# Chapter Outline

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- Indexes on Multiple Keys

# Indexes as Access Paths

- A single-level index is an auxiliary file that makes it more efficient to search for a record in the data file.
- The index is usually specified on one field of the file (although it could be specified on several fields)
- One form of an index is a file of entries <**field value, pointer to record>**, which is ordered by field value
- The index is called an access path on the field.

# Indexes as Access Paths (contd.)

- The index file usually occupies considerably less disk blocks than the data file because its entries are much smaller
- A binary search on the index yields a pointer to the file record
- Indexes can also be characterized as dense or sparse
  - A **dense index** has an index entry for every search key value (and hence every record) in the data file.
  - A **sparse (or nondense) index**, on the other hand, has index entries for only some of the search values

# Indexes as Access Paths (contd.)

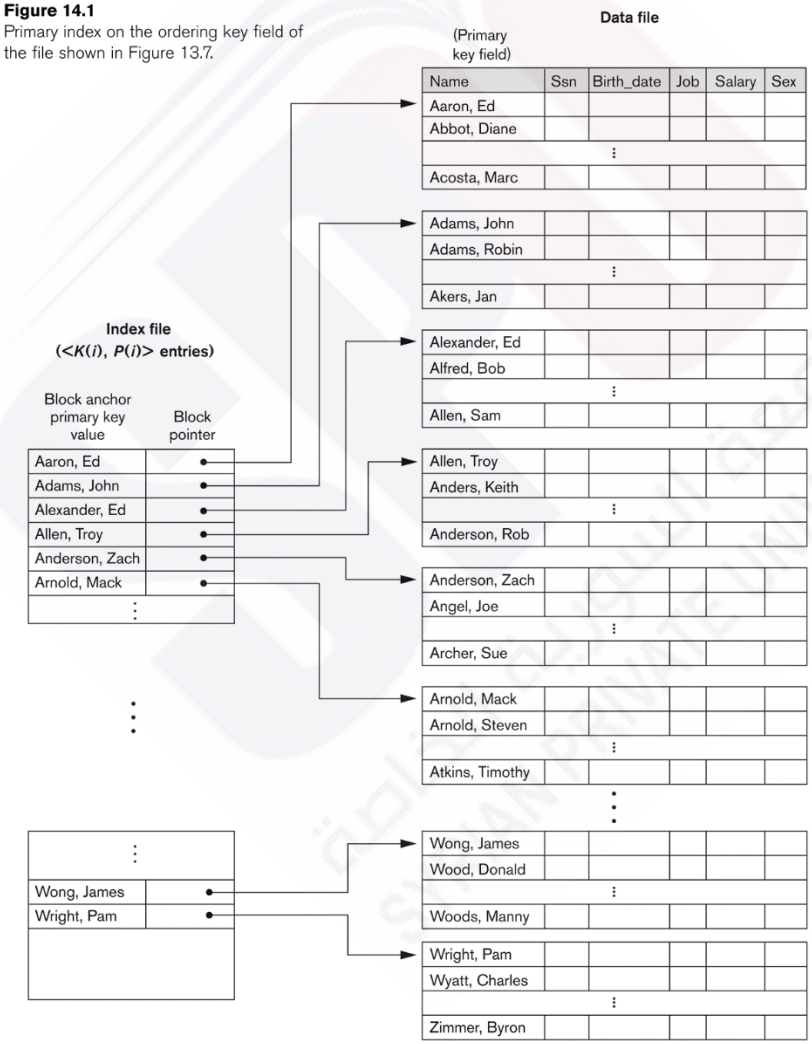
- Example: Given the following data file EMPLOYEE(NAME, SSN, ADDRESS, JOB, SAL, ... )
- Suppose that:
  - record size R=150 bytes  block size B=512 bytes  r=30000 records
- Then, we get:
  - blocking factor Bfr= B div R= 512 div 150= 3 records/block
  - number of file blocks b= (r/Bfr)= (30000/3)= 10000 blocks
- For an index on the SSN field, assume the field size $V_{SSN}$=9 bytes, assume the record pointer size $P_R$=7 bytes. Then:
  - index entry size $R_I$=($V_{SSN}$+ $P_R$)=(9+7)=16 bytes
  - index blocking factor $Bfr_I$= B div $R_I$= 512 div 16= 32 entries/block
  - number of index blocks b= (r/ $Bfr_I$)= (30000/32)= 938 blocks
  - binary search needs $\log_2 bI = \log_2 938$= 10 block accesses
  - This is compared to an average linear search cost of:
    - (b/2)= 30000/2= 15000 block accesses
  - If the file records are ordered, the binary search cost would be:
    - $\log_2 b = \log_2 30000$= 15 block accesses

# Types of Single-Level Indexes

- Primary Index
  - Defined on an ordered data file
  - The data file is ordered on a **key field**
  - Includes one index entry *for each block* in the data file; the index entry has the key field value for the *first record* in the block, which is called the *block anchor*
  - A similar scheme can use the *last record* in a block.
  - A primary index is a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value.

# Primary index on the ordering key field

**Figure 14.1**
Primary index on the ordering key field of the file shown in Figure 13.7.

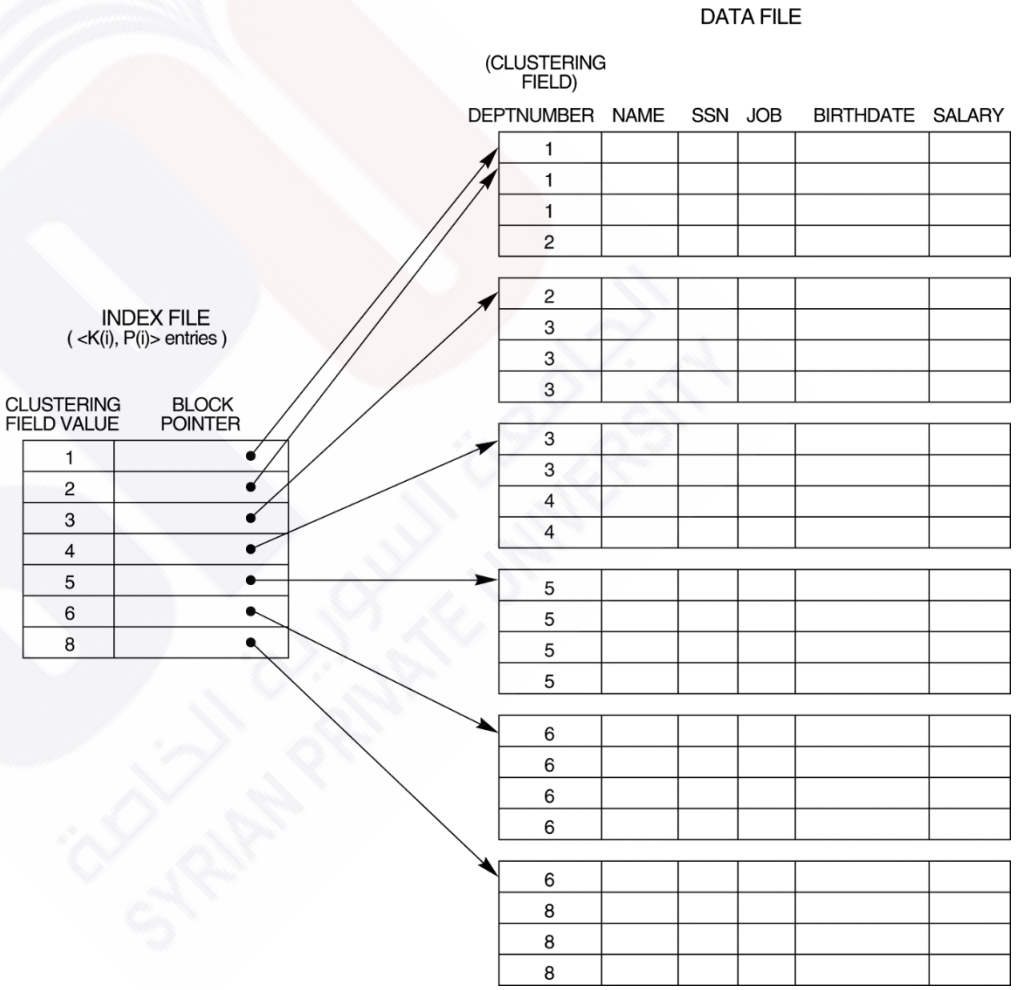


# Types of Single-Level Indexes

- Clustering Index
  - Defined on an ordered data file
  - The data file is ordered on a *non-key field* unlike primary index, which requires that the ordering field of the data file have a distinct value for each record.
  - Includes one index entry *for each distinct value* of the field; the index entry points to the first data block that contains records with that field value.
  - It is another example of *nondense* index where Insertion and Deletion is relatively straightforward with a clustering index.

# A Clustering Index Example

- FIGURE 14.2
  A clustering index on the DEPTNUMBER ordering non-key field of an EMPLOYEE file.



# Another Clustering Index Example

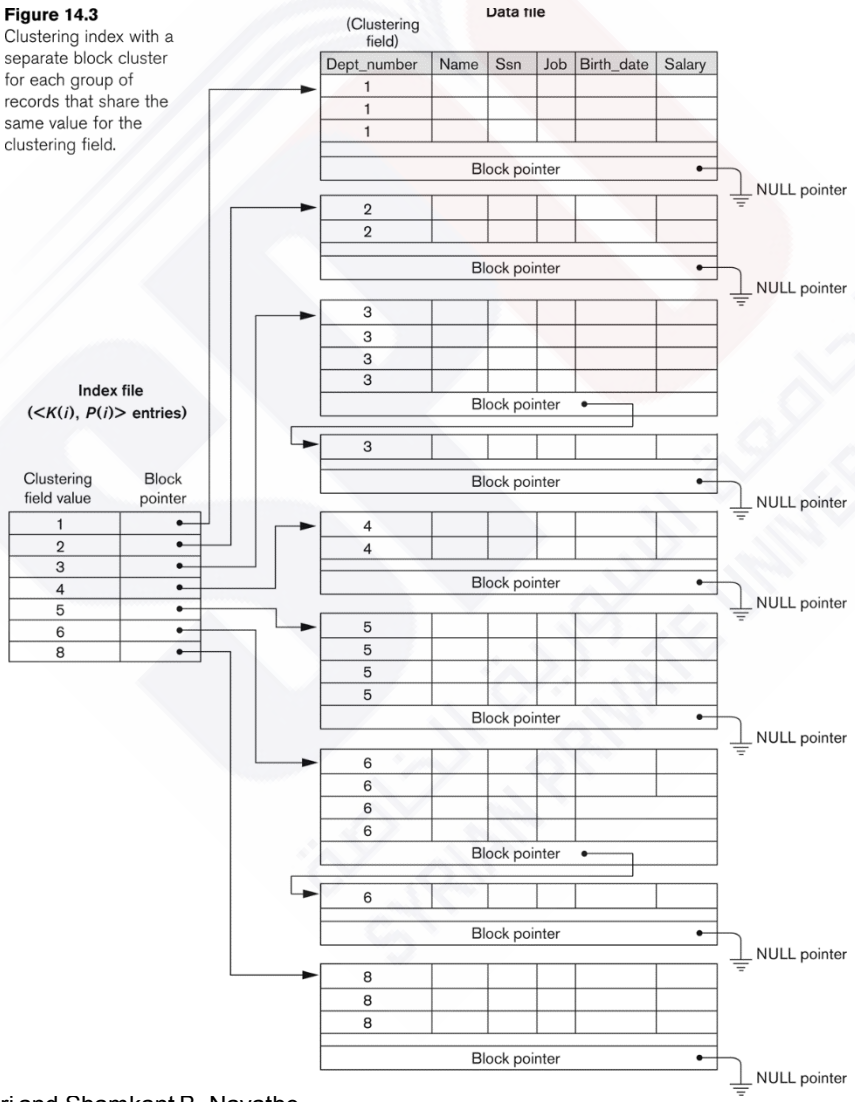



**Figure 14.3**
Clustering index with a separate block cluster for each group of records that share the same value for the clustering field.

# Types of Single-Level Indexes

- Secondary Index
  - A secondary index provides a secondary means of accessing a file for which some primary access already exists.
  - The secondary index may be on a field which is a candidate key and has a unique value in every record, or a non-key with duplicate values.
  - The index is an ordered file with two fields.
    - The first field is of the same data type as some **non-ordering field** of the data file that is an indexing field.
    - The second field is either a **block** pointer or a record pointer.
    - There can be *many* secondary indexes (and hence, indexing fields) for the same file.
  - Includes one entry *for each record* in the data file; hence, it is a *dense index*

# Example of a Dense Secondary Index

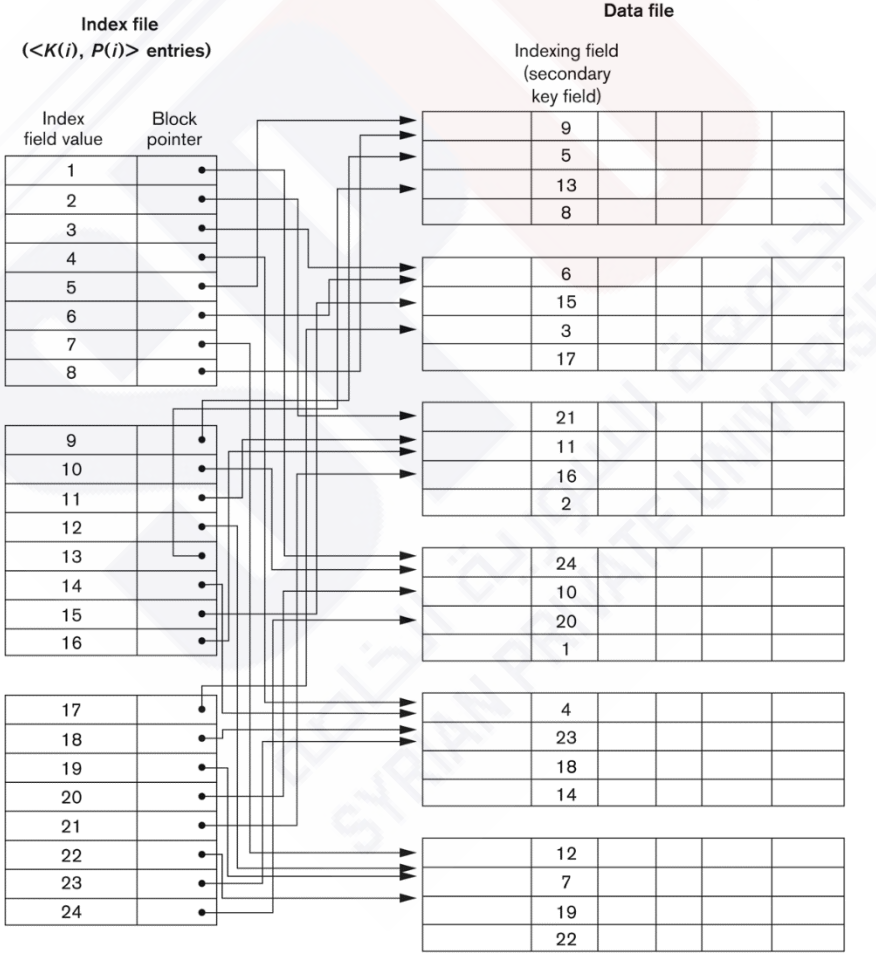



**Figure 14.4**
A dense secondary index (with block pointers) on a nonordering key field of a file.

# An Example of a Secondary Index

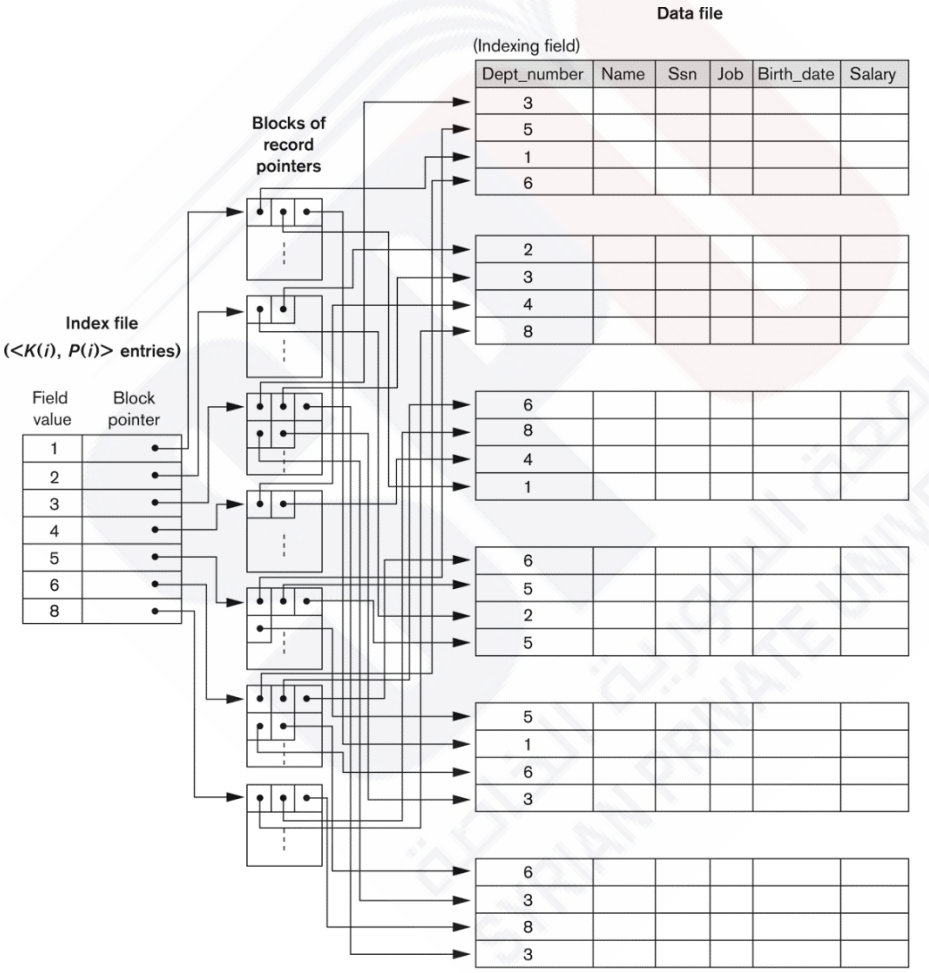



**Figure 14.5**
A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values.

# Properties of Index Types

**TABLE 14.2 PROPERTIES OF INDEX TYPES**

| TYPE OF INDEX | NUMBER OF (FIRST-LEVEL) INDEX ENTRIES | DENSE OR NONDENSE | BLOCK ANCHORING ON THE DATA FILE |
|---|---|---|---|
| Primary | Number of blocks in data file | Nondense | Yes |
| Clustering | Number of distinct index field values | Nondense | Yes/no[a] |
| Secondary (key) | Number of records in data file | Dense | No |
| Secondary (nonkey) | Number of records[b] or Number of distinct index field values[c] | Dense or Nondense | No |

[a]Yes if every distinct value of the ordering field starts a new block; no otherwise.
[b]For option 1.
[c]For options 2 and 3.

# Multi-Level Indexes

- Because a single-level index is an ordered file, we can create a primary index *to the index itself*;
  - In this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering) as long as the first-level index consists of *more than one* disk block

# A Two-level Primary Index

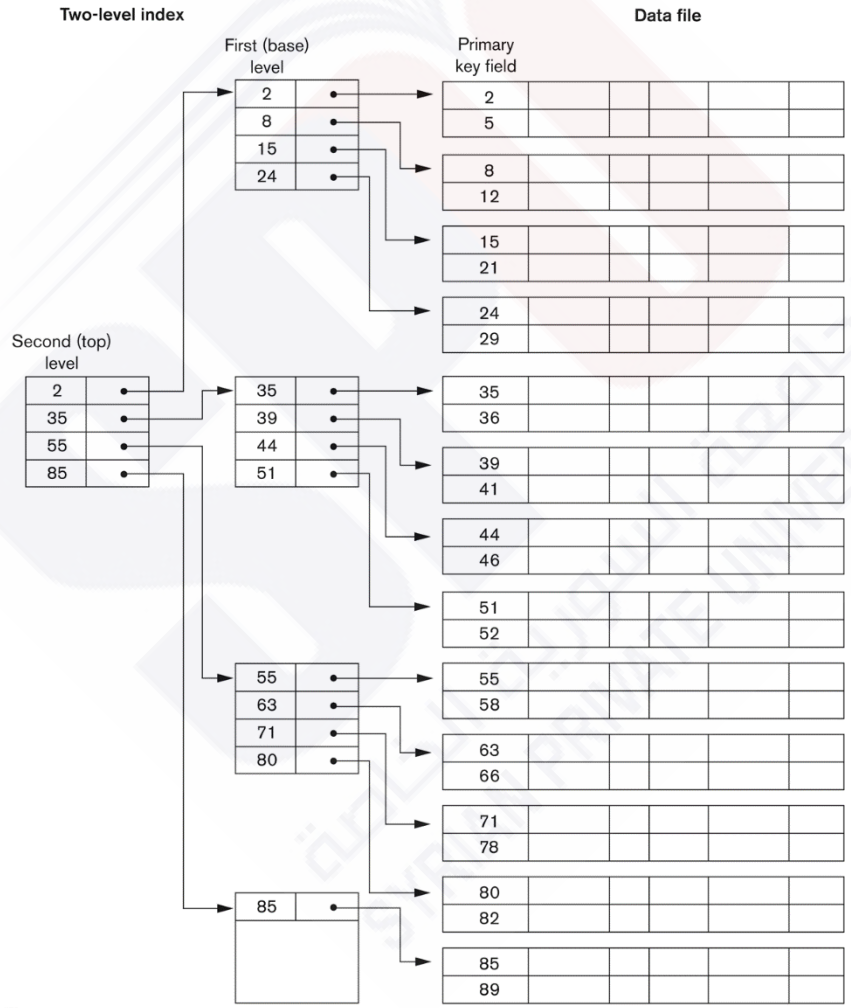



**Figure 14.6**
A two-level primary index resembling ISAM (Index Sequential Access Method) organization.

# Multi-Level Indexes

- Such a multi-level index is a form of *search tree*
  - However, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.

# Dynamic Multilevel Indexes Using B-Trees and B+-Trees

- Most multi-level indexes use B+-tree data structures because of the insertion and deletion problem
  - This leaves space in each tree node (disk block) to allow for new index entries
- These data structures are variations of search trees that allow efficient insertion and deletion of new search values.
- In B+-Tree data structures, each node corresponds to a disk block
- Each node is kept between half-full and completely full

# Dynamic Multilevel Indexes Using B-Trees and B+-Trees (contd.)

- An insertion into a node that is not full is quite efficient
  - If a node is full the insertion causes a split into two nodes
- Splitting may propagate to other tree levels
- A deletion is quite efficient if a node does not become less than half full
- If a deletion causes a node to become less than half full, it must be merged with neighboring nodes

# B+-Trees Example

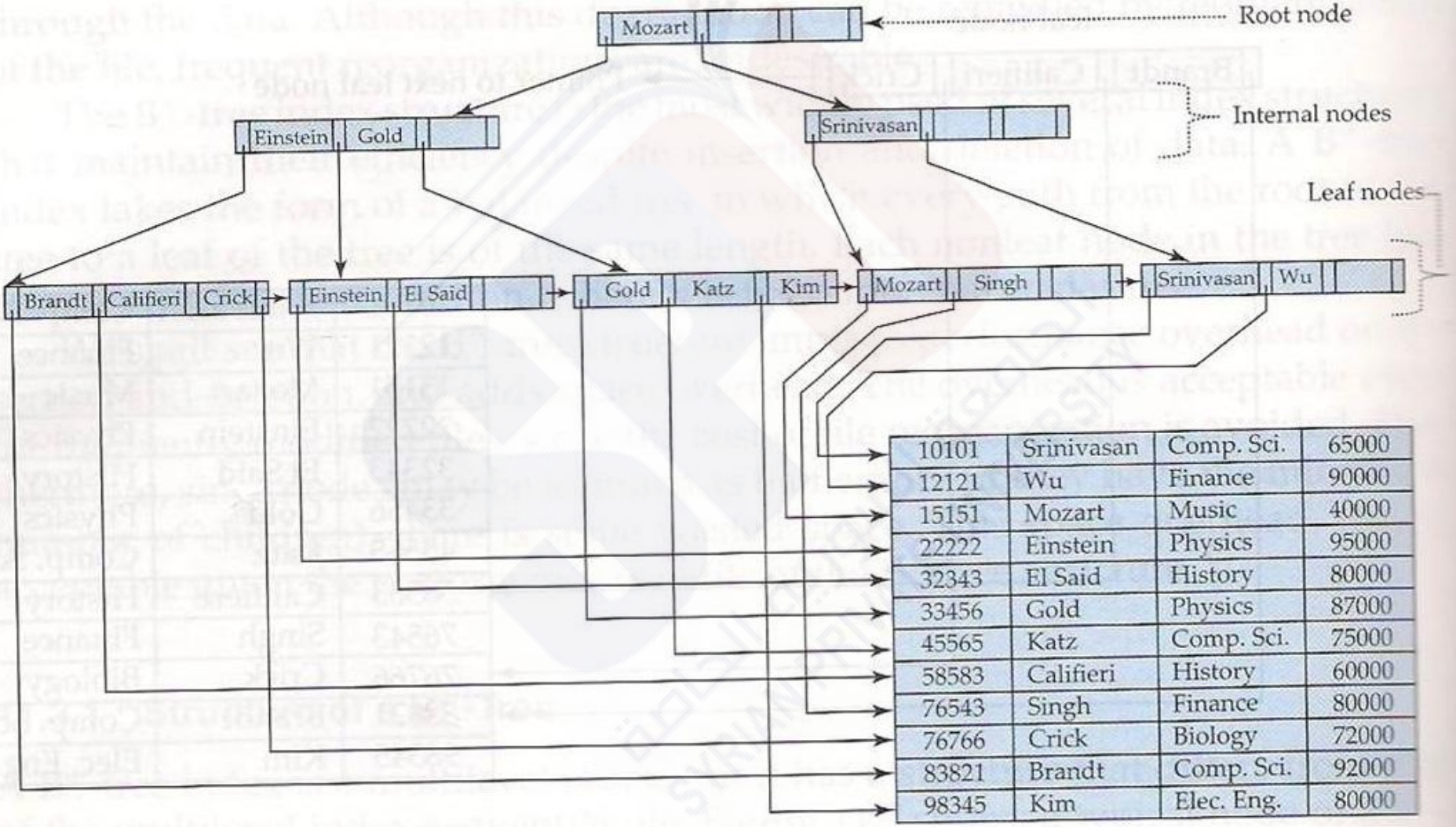
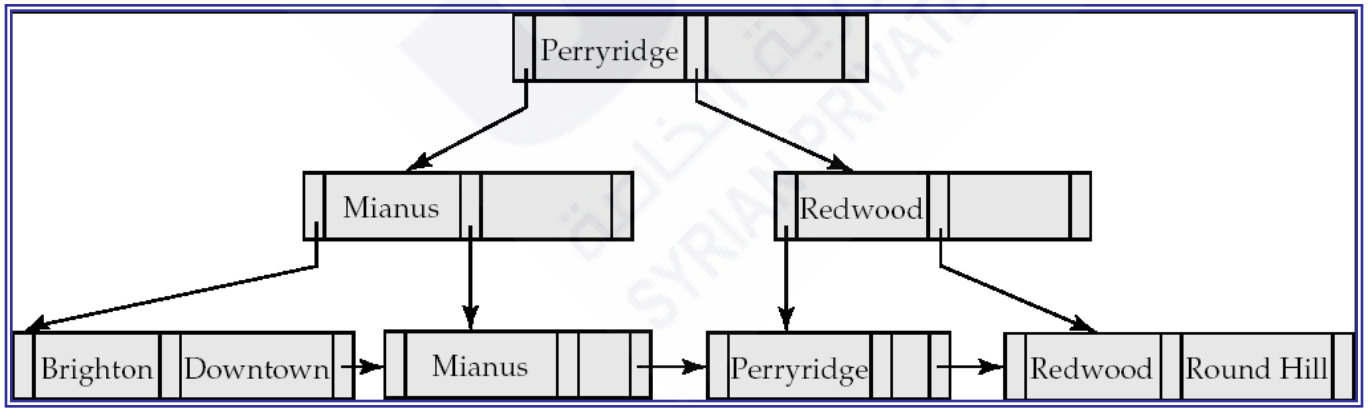



**Figure 11.9** B$^+$-tree for *instructor* file ($n = 4$).

# $B^+$-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:
  - 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

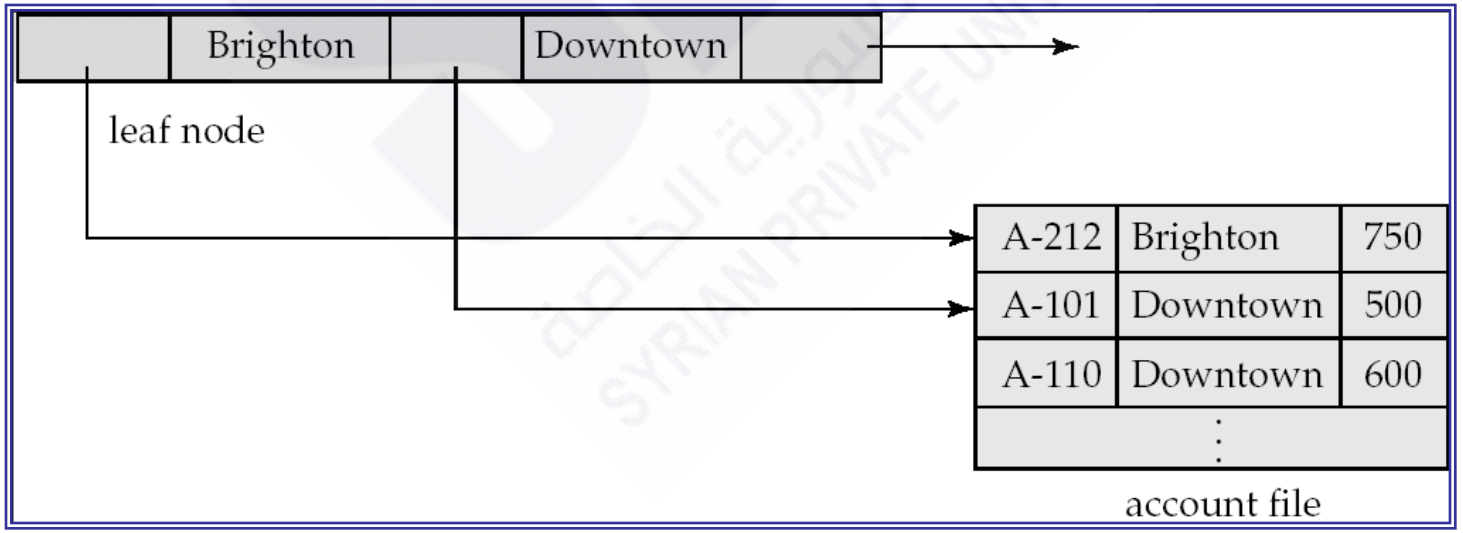

  - $K_i$ are the search-key values
  - $P_i$ are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

# Leaf Nodes in B$^+$-Trees

Properties of a leaf node:

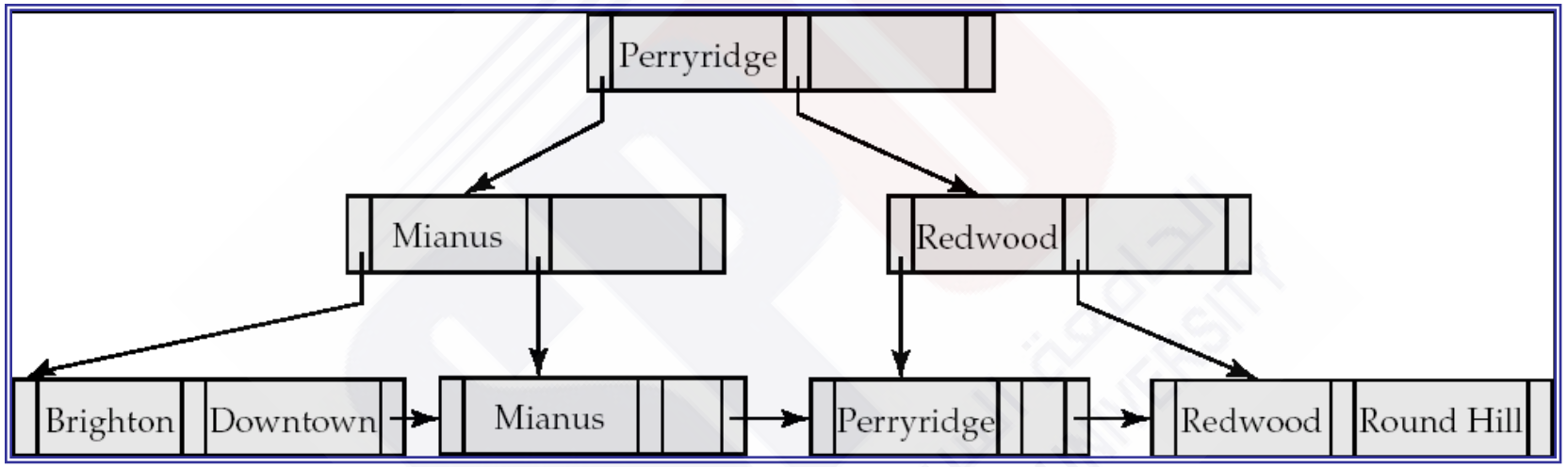
- For $i = 1, 2, \ldots, n-1$, pointer $P_i$ either points to a file record with search-key value $K_i$, or to a bucket of pointers to file records, each record having search-key value $K_i$. Only need bucket structure if search-key does not form a primary key.
- If $L_i$, $L_j$ are leaf nodes and $i < j$, $L_i$'s search-key values are less than $L_j$'s search-key values
- $P_n$ points to next leaf node in search-key order



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

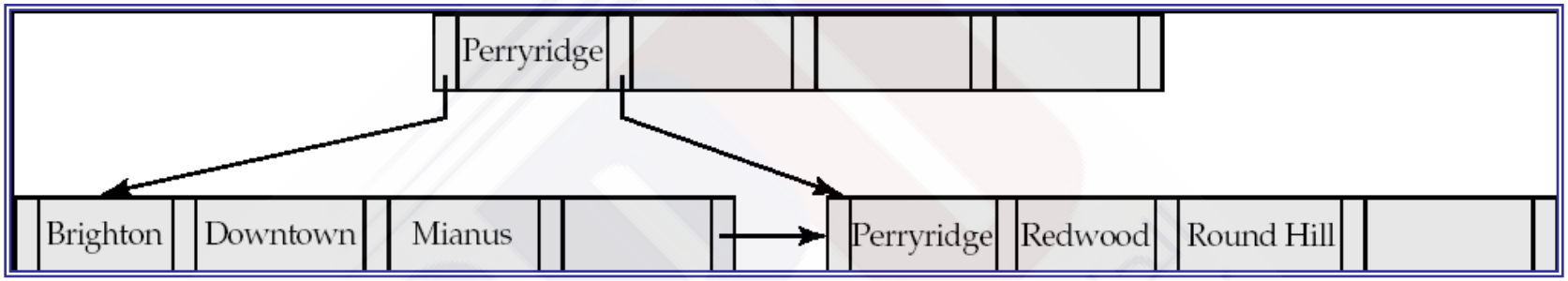
- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with *m* pointers:
  - All the search-keys in the subtree to which $P_1$ points are less than $K_1$
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which $P_i$ points have values greater than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

# Example of a B+-tree




B+-tree for *account* file (*n* = 3)

# Example of B+-tree




B+-tree for *account* file ($n$ = 5)

- Leaf nodes must have between 2 and 4 values ($\lceil (n-1)/2 \rceil$ and $n-1$, with $n$ = 5).
- Non-leaf nodes other than root must have between 3 and 5 children ($\lceil n/2 \rceil$ and $n$ with $n$ =5).
- Root must have at least 2 children.

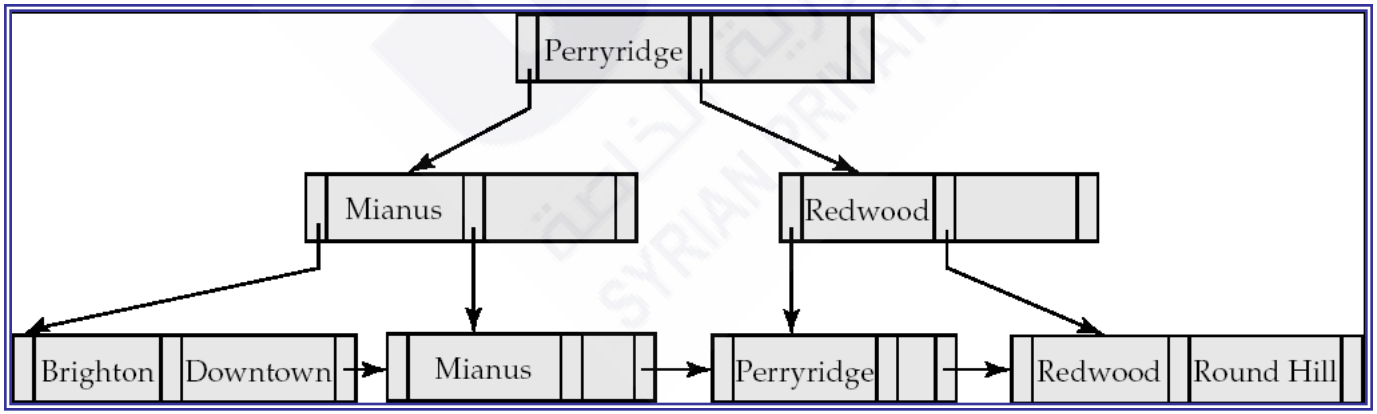
# Observations about $B^+$-trees

- 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 all records with a search-key value of *k.*
  1. *N=root*
  2. Repeat
     1. Examine *N* for the smallest search-key value > *k.*
     2. If such a value exists, assume it is $K_i$. Then set $N = P_i$
     3. Otherwise $k \geq K_{n-1}$. Set $N = P_n$

     Until *N* is a leaf node
  3. If for some *i*, key $K_i = k$ follow pointer $P_i$ to the desired record or bucket.
  4. 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

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

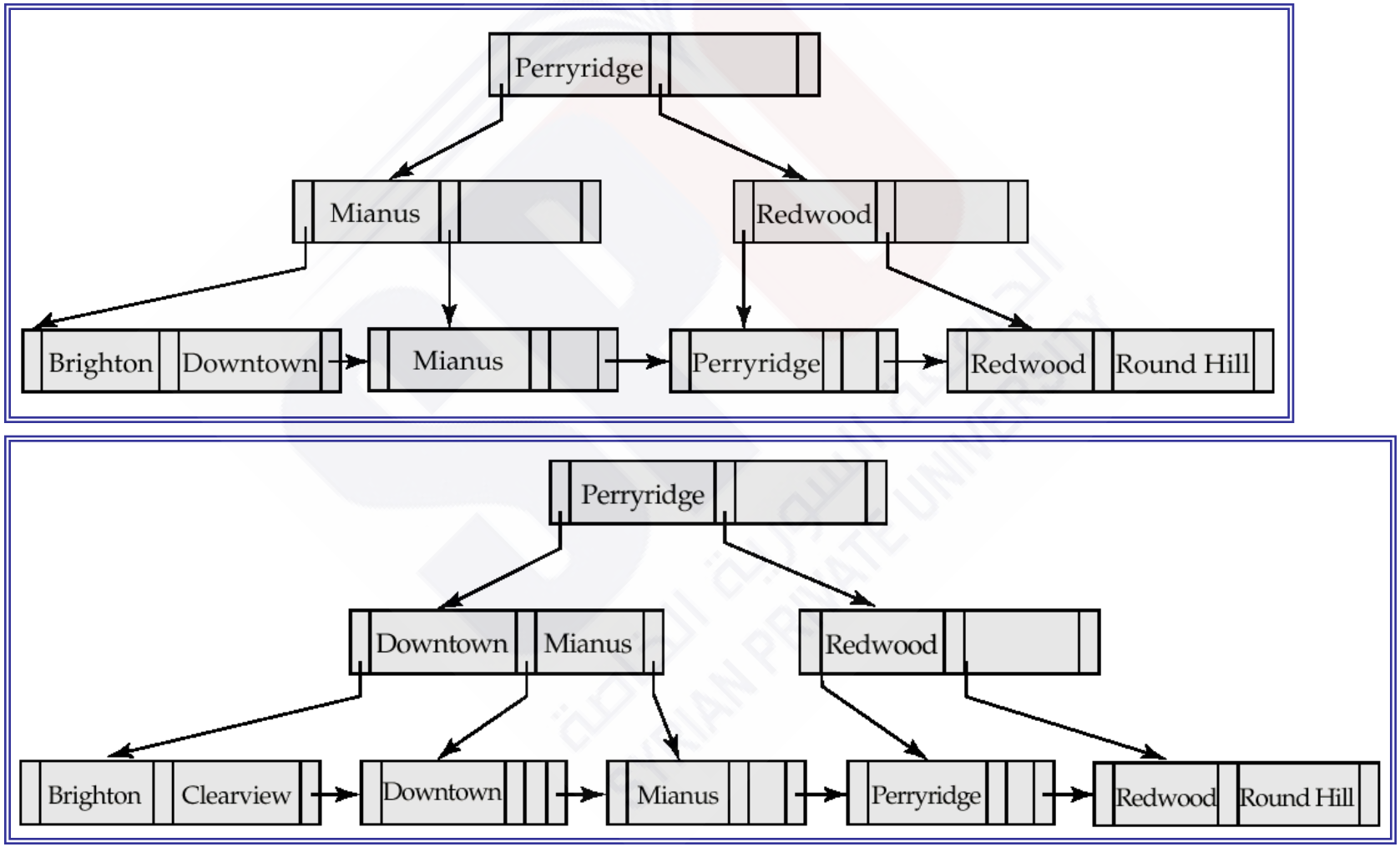
1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
    1. Add record to the file
3. If the search-key value is not present, then
    1. add the record to the main file (and create a bucket if necessary)
    2. If there is room 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 Brighton and Downtown on inserting Clearview
Next step: insert entry with (Downtown,pointer-to-new-node) into parent

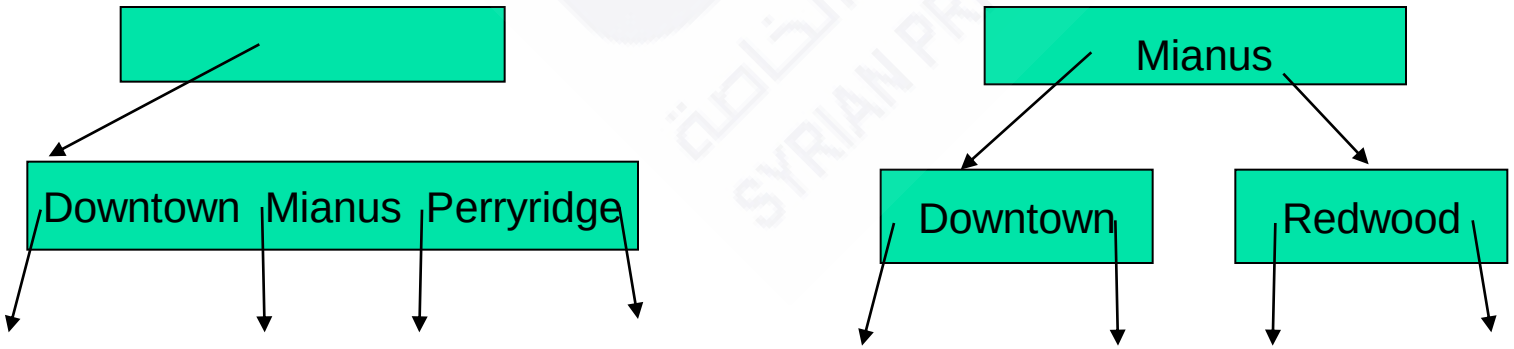
# Updates on B+-Trees: Insertion (Cont.)




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

# 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



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

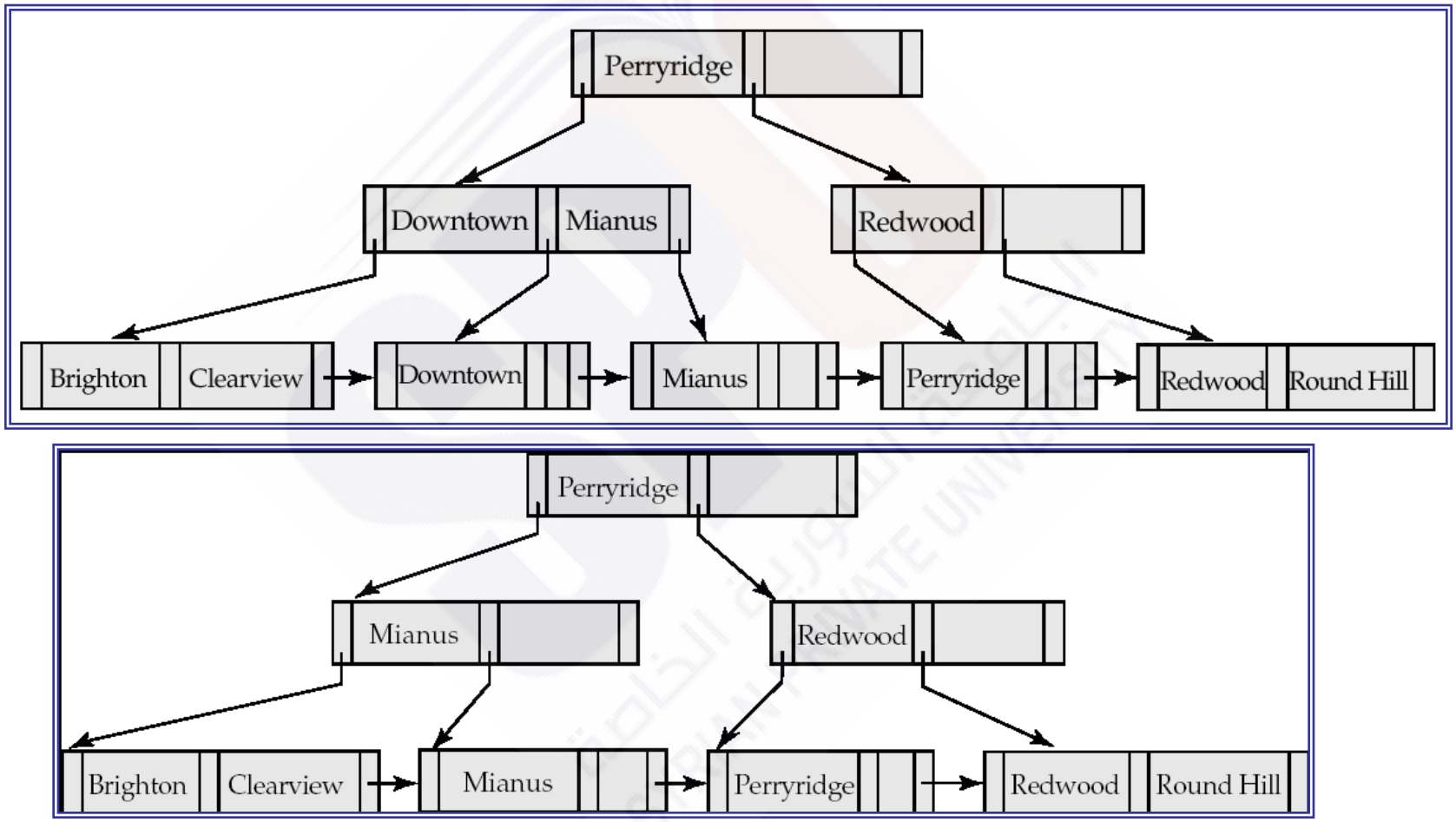
- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling 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.

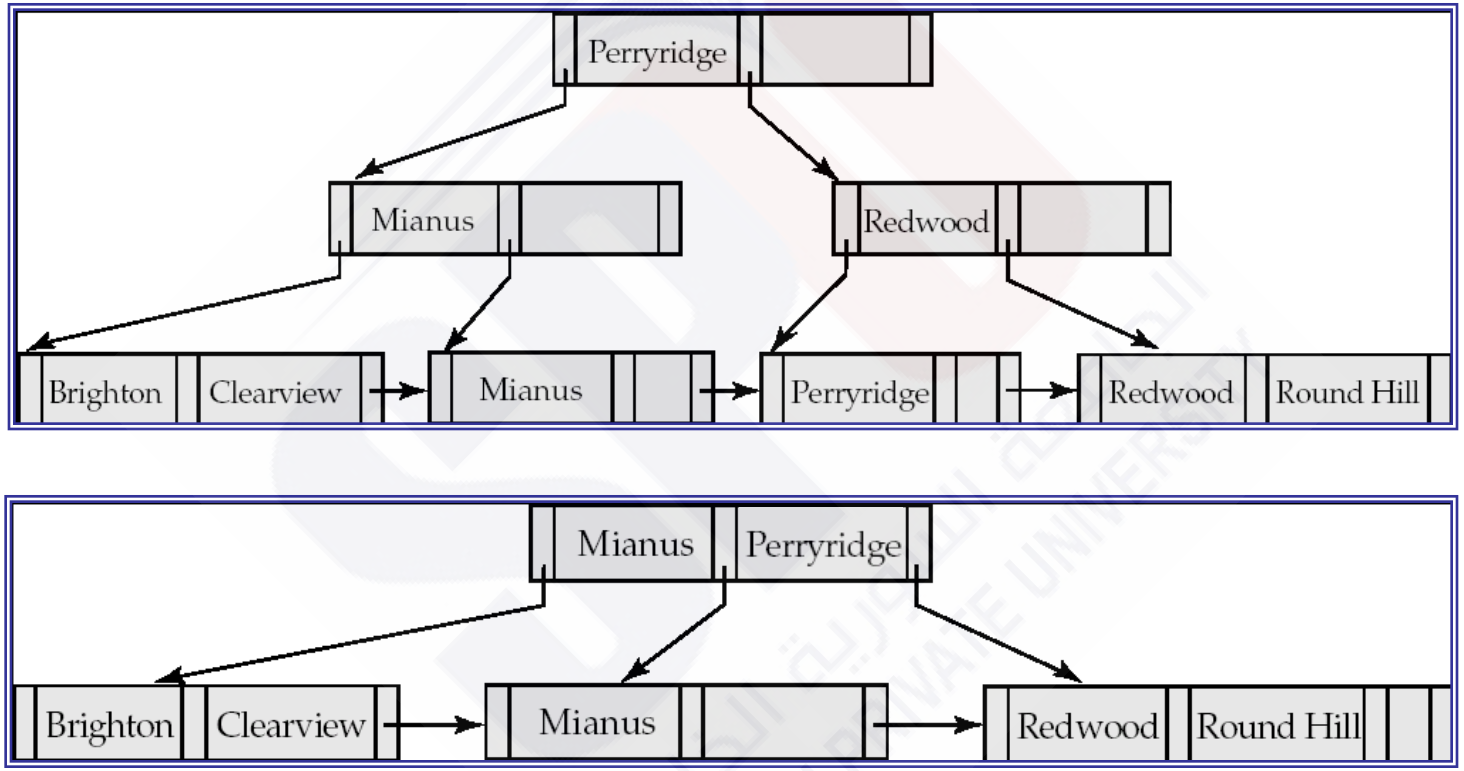
# Examples of B+-Tree Deletion

Before and after deleting “Downtown”



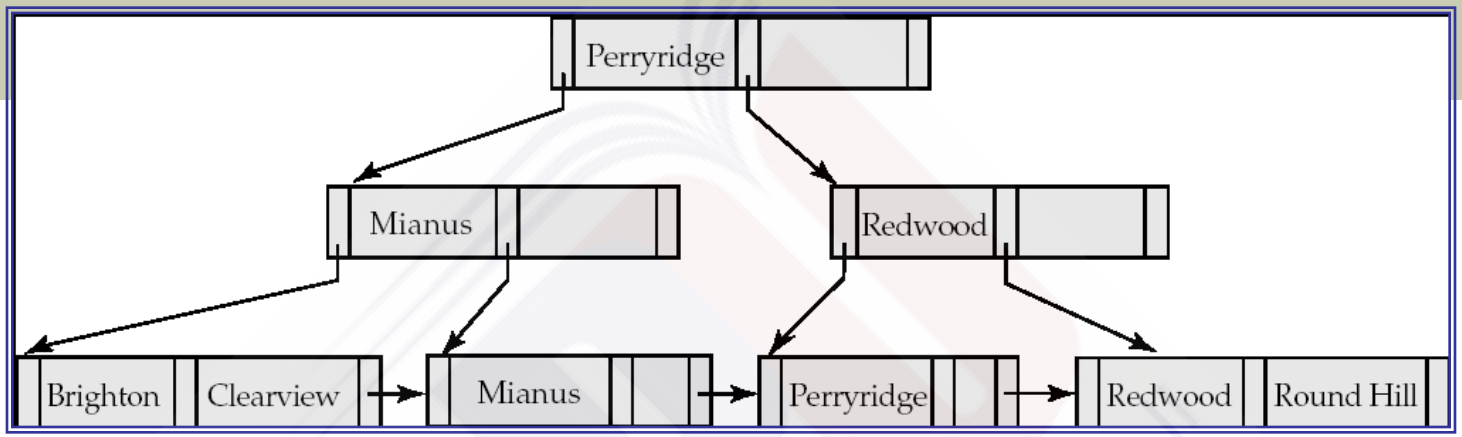

- Deleting “Downtown” causes merging of under-full leaves
  - leaf node can become empty only for n=3!

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

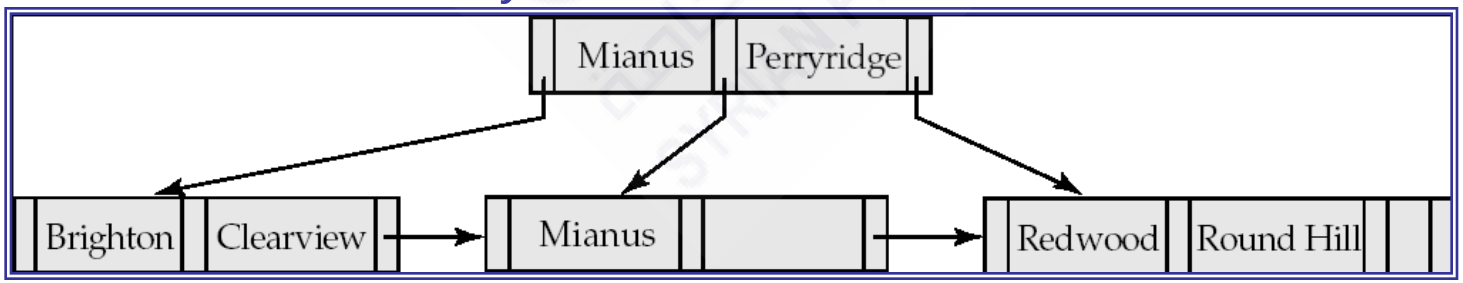



Before and After deletion of “Perryridge” from result of previous example

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

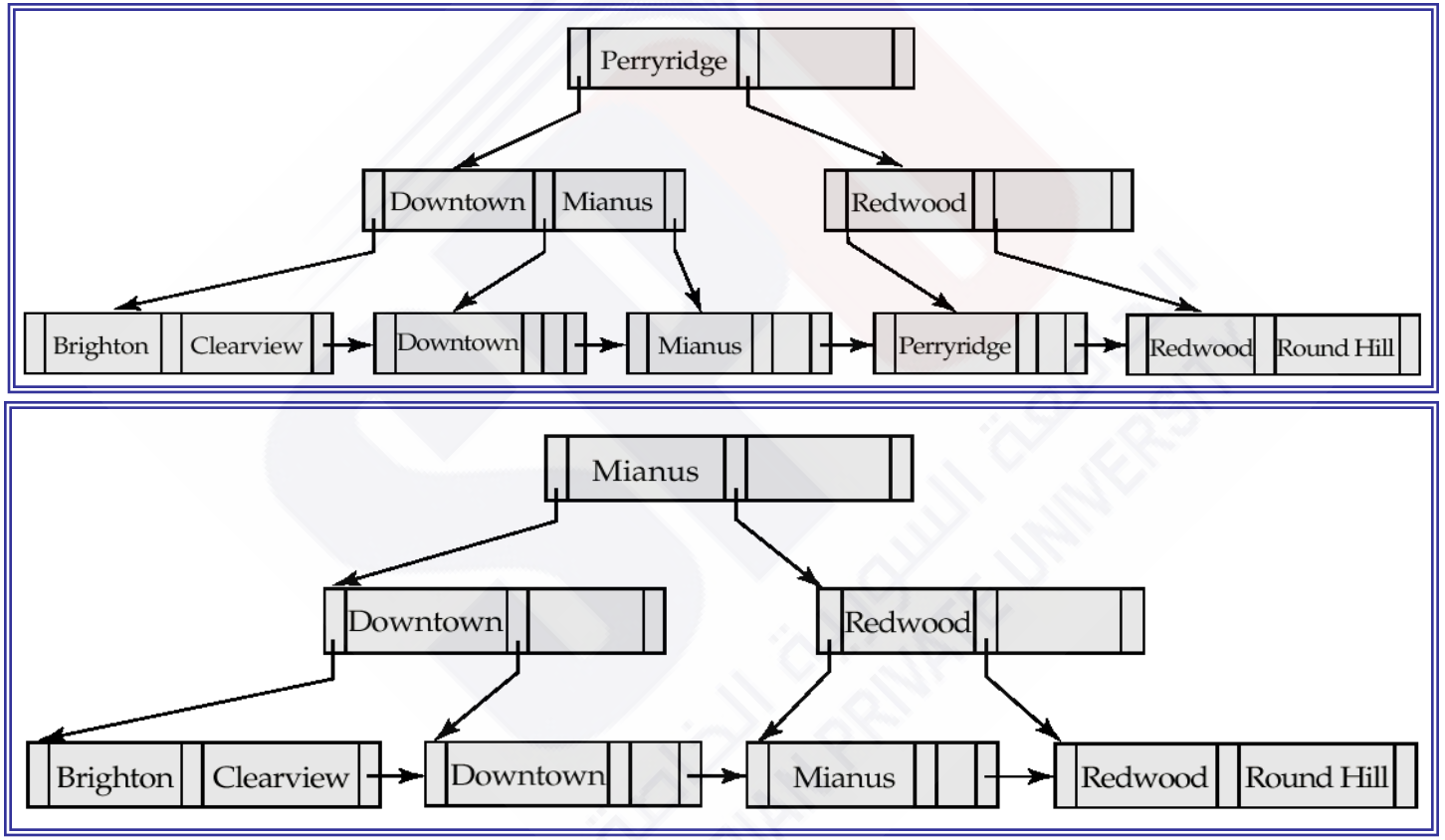



- Leaf with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling.
- As a result "Perryridge" node's parent became underfull, and was merged with its sibling
  - Value separating two nodes (at parent) moves into merged node
  - Entry deleted from parent
- Root node then has only one child, and is deleted



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

Before and after deletion of “Perryridge” from earlier example




- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- Search-key value in the parent’s parent changes as a result

# Summary

- Types of Single-level Ordered Indexes
  - Primary Indexes
  - Clustering Indexes
  - Secondary Indexes
- Multilevel Indexes
- Dynamic Multilevel Indexes Using B-Trees and B+-Trees
- Indexes on Multiple Keys