CMSC424: Storage and Indexes

Instructor: Amol Deshpande
amol@cs.umd.edu

Today's Class

- Storage and Query Processing
  - Indexes; B+-Tree

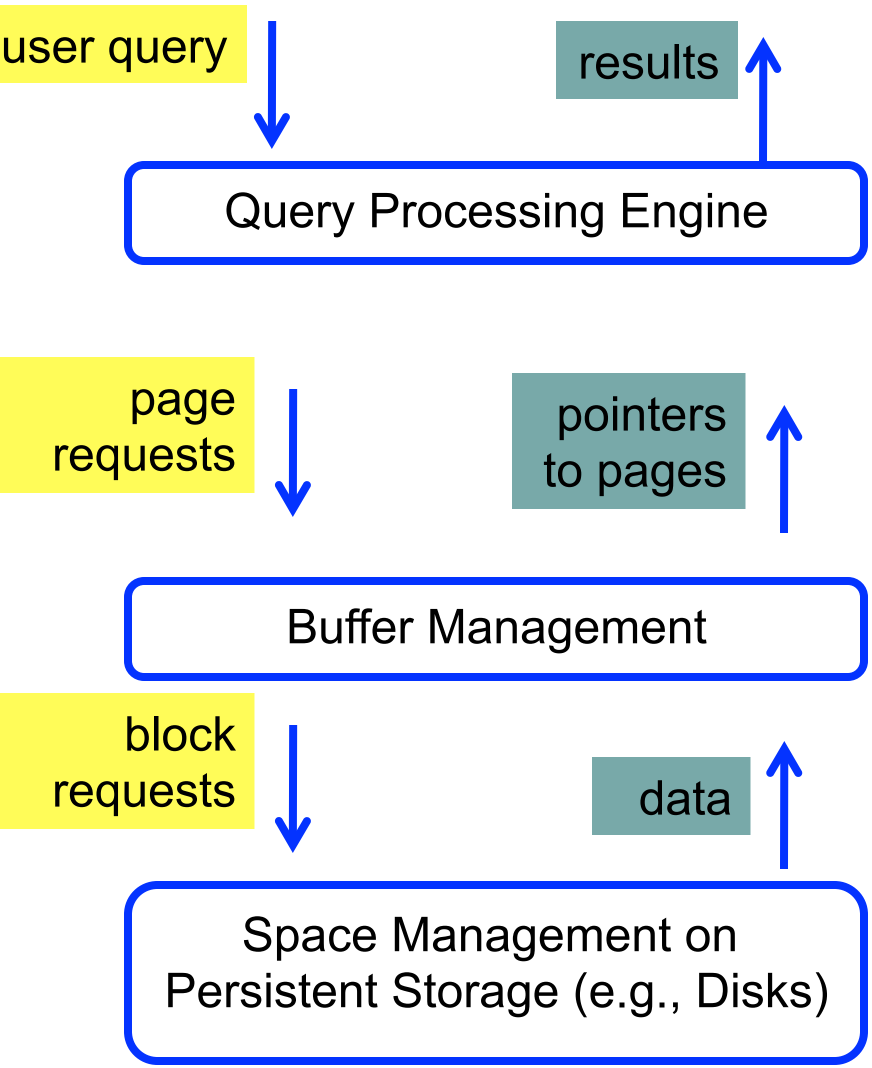
- Other things
  - Project 4: due next Friday
    - Make sure to go through the posted Notebooks

  - No laptop use in class (without permission) !!

Databases

- Data Models
  - Conceptual representation of the data
- Data Retrieval
  - How to ask questions of the database
  - How to answer those questions
- Data Storage
  - How/where to store data, how to access it
- Data Integrity
  - Manage crashes, concurrency
  - Manage semantic inconsistencies

Query Processing/Storage



- Given a input user query, decide how to “execute” it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

- Bringing pages from disk to memory
- Managing the limited memory

- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

Outline

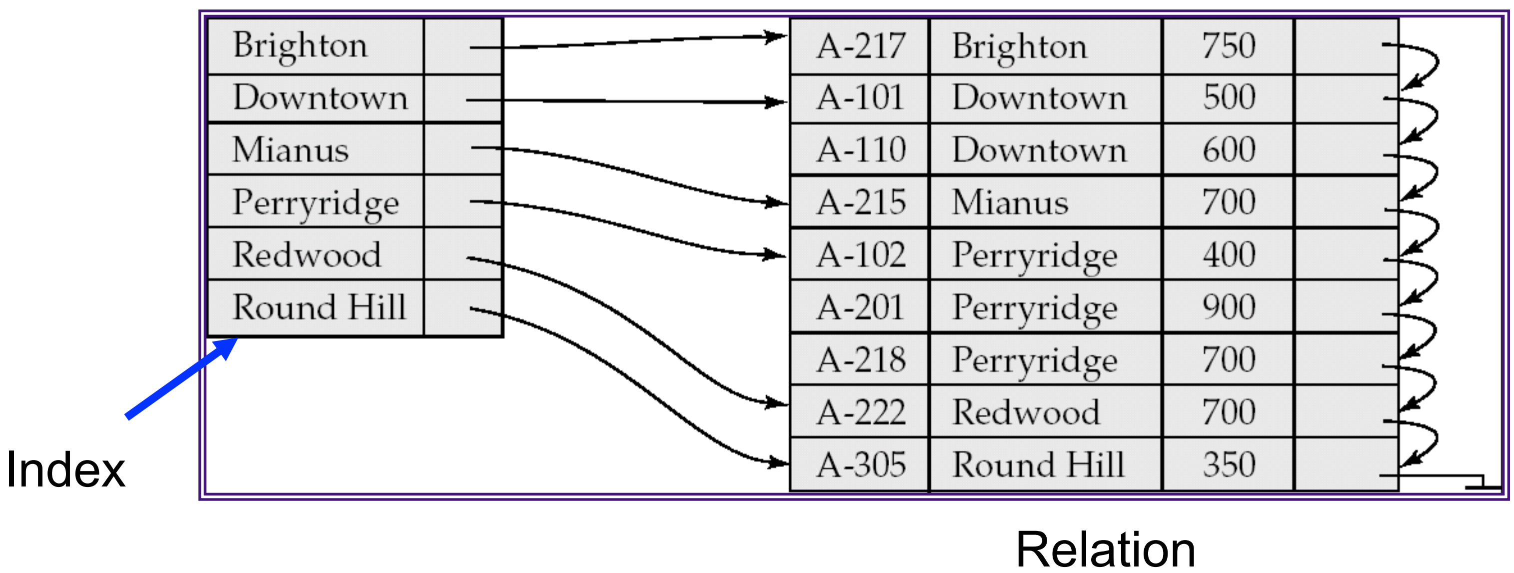
- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Indexes
- Etc…

Index

- A data structure for efficient search through large databaess
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
    - Sorted, or hash-based
  - Auxiliary data structures are maintained that allow quick search
- Think library index/catalogue
- Search key:
  - Attribute or set of attributes used to look up records
  - E.g. SSN for a persons table
- Two types of indexes
  - Ordered indexes
  - Hash-based indexes

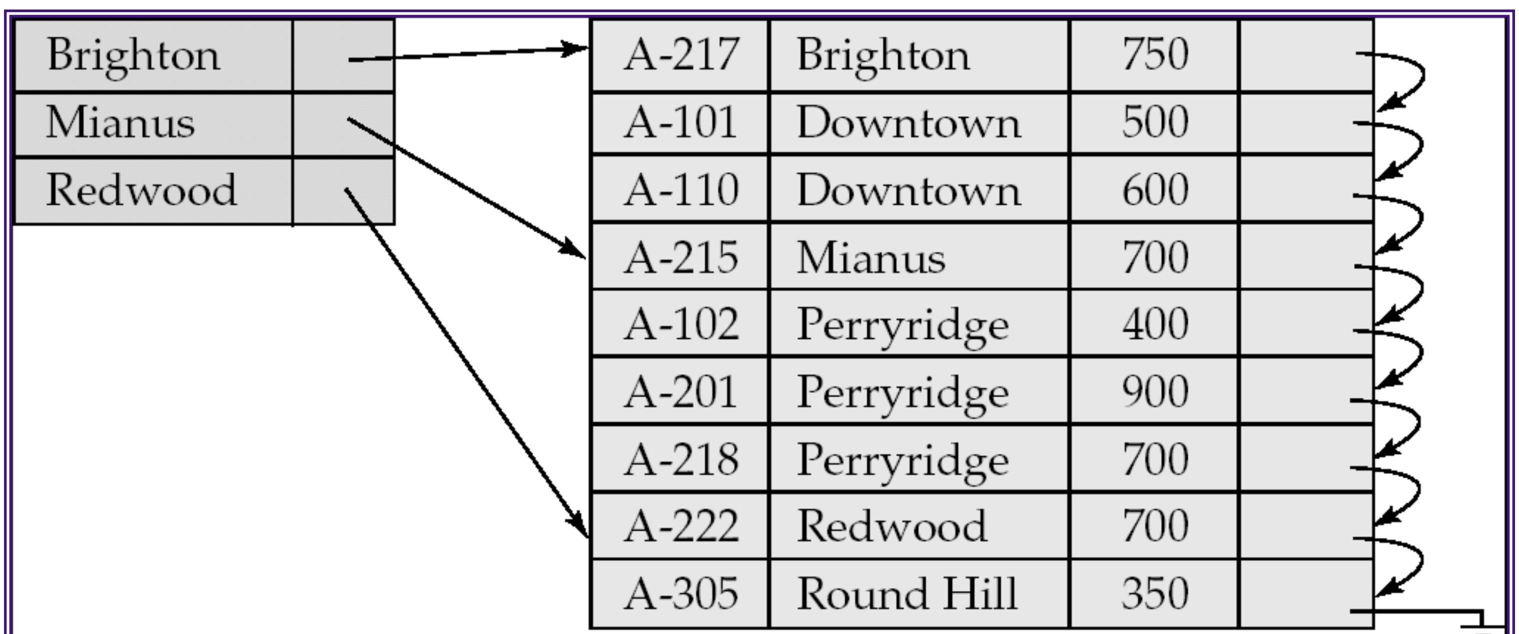
Ordered Indexes

- Primary index
  - The relation is sorted on the search key of the index
- Secondary index
  - It is not
- Can have only one primary index on a relation



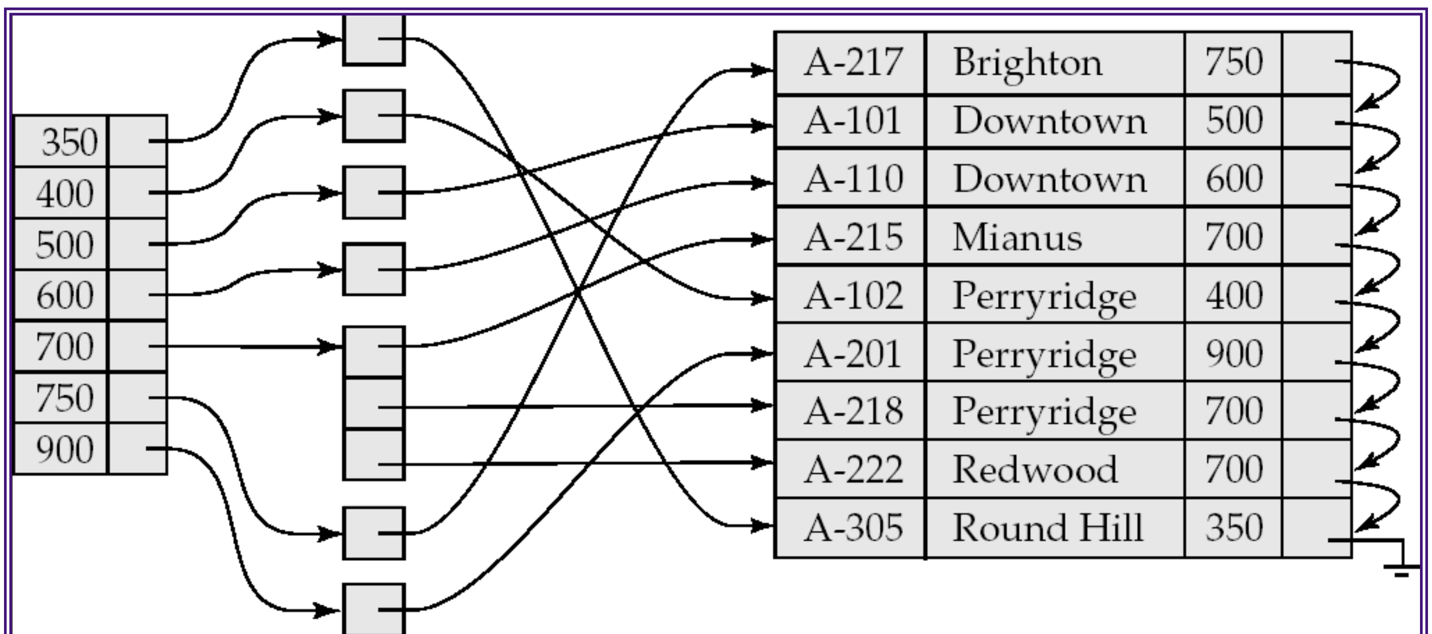
Primary *Sparse* Index

- Every key doesn't have to appear in the index
- Allows for very small indexes
  - Better chance of fitting in memory
  - Tradeoff: Must access the relation file even if the record is not present



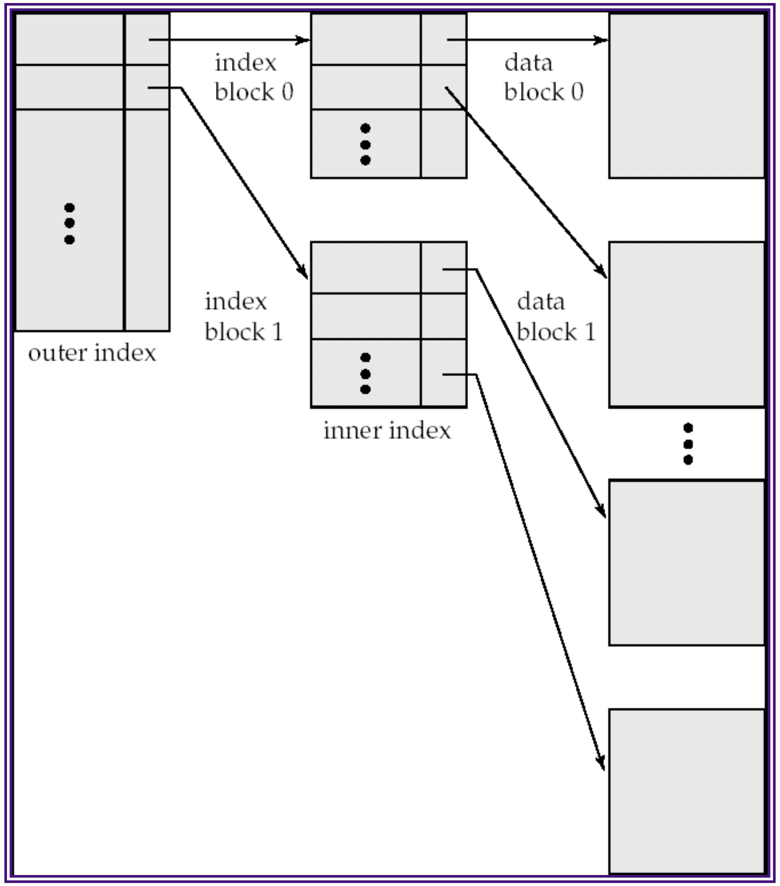
Secondary Index

- Relation sorted on *branch*
- But we want an index on *balance*
- Must be dense
  - Every search key must appear in the index



Multi-level Indexes

- What if the index itself is too big for memory ?
- Relation size = n = 1,000,000,000
- Block size = 100 tuples per block
- So, number of pages = 10,000,000
- Keeping one entry per page takes too much space
- Solution
  - Build an index on the index itself



Multi-level Indexes

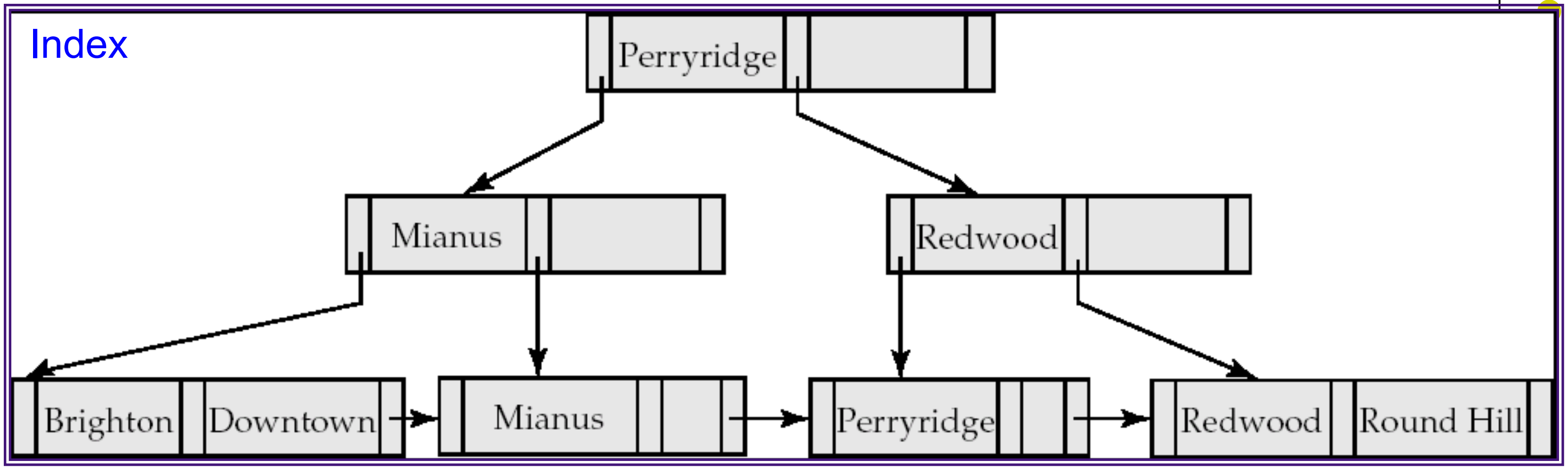
- How do you search through a multi-level index ?

- What about keeping the index up-to-date ?
  - Tuple insertions and deletions
    - This is a static structure
    - Need overflow pages to deal with insertions
  - Works well if no inserts/deletes
  - Not so good when inserts and deletes are common

Outline

- Storage hierarchy
- Disks
- RAID
- Buffer Manager
- File Organization
- Indexes
- B+-Tree Indexes
- Etc..

Example B+-Tree Index



B+-Tree Node Structure

- Typical node

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

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

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

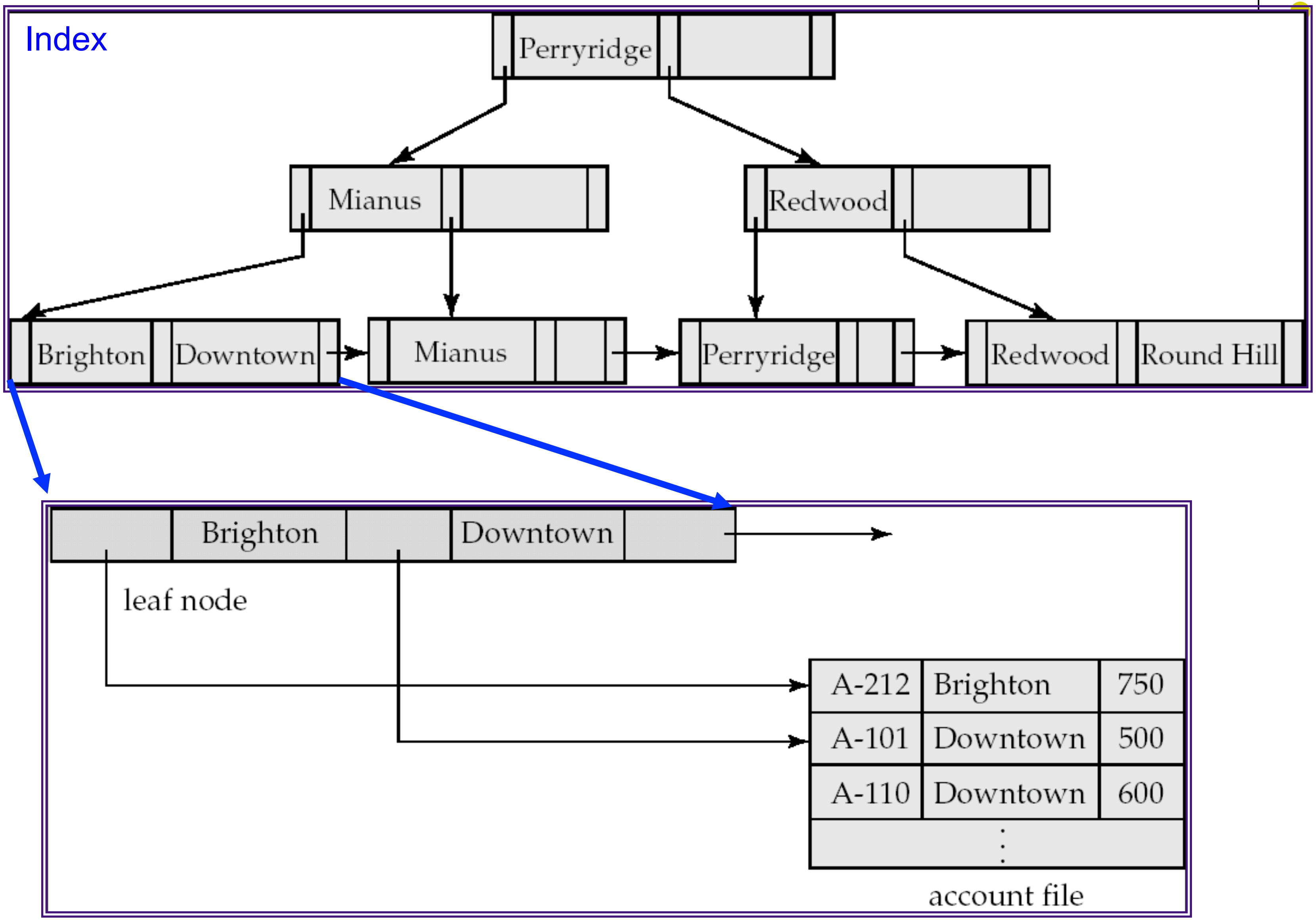
Properties of B+-Trees

- It is balanced
  - Every path from the root to a leaf is same length

- Leaf nodes (at the bottom)
  - *P1* contains the pointers to tuple(s) with key *K1*
  - …
  - *Pn* is a pointer to the *next* leaf node
  - Must contain at least n/2 entries

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

Example B+-Tree Index



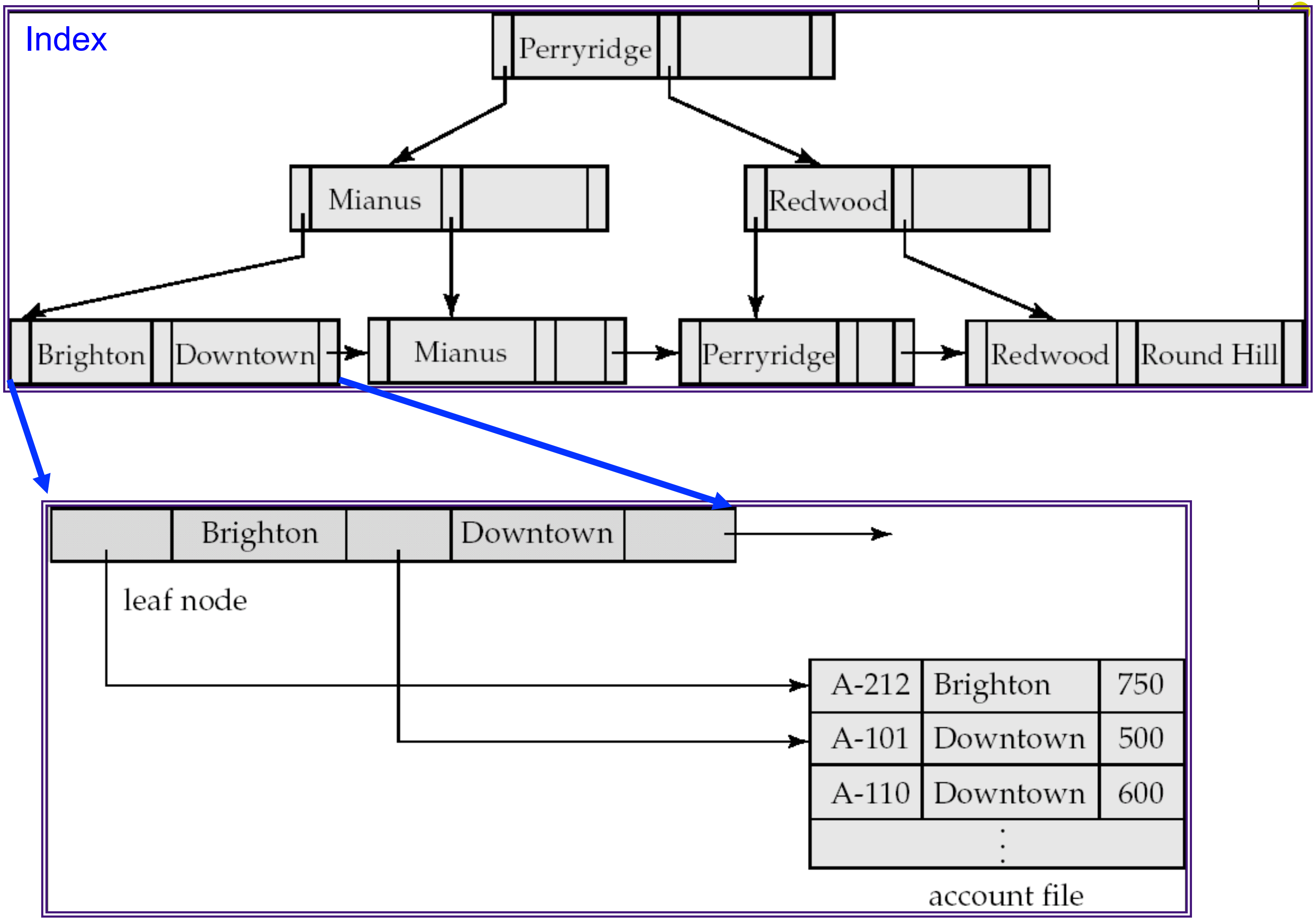
Properties

- Interior nodes

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

  - All tuples in the subtree pointed to by *P1,* have search key < *K1*
  - To find a tuple with key *K1’* < *K1,* follow *P1*
  - …
  - Finally, search keys in the tuples contained in the subtree pointed to by P_n, are all larger than K_{n-1}
  - Must contain at least n/2 entries (unless root)

Example B+-Tree Index



B+-Trees - Searching

- How to search ?
  - Follow the pointers
- Logarithmic
  - *$log_{B/2}(N)$*, where *B = Number of entries per block*
  - *B* is also called the order of the B+-Tree Index
    - Typically 100 or so
- If a relation contains1,000,000,000 entries, takes only 4 random accesses
- The top levels are typically in memory
  - So only requires 1 or 2 random accesses per request

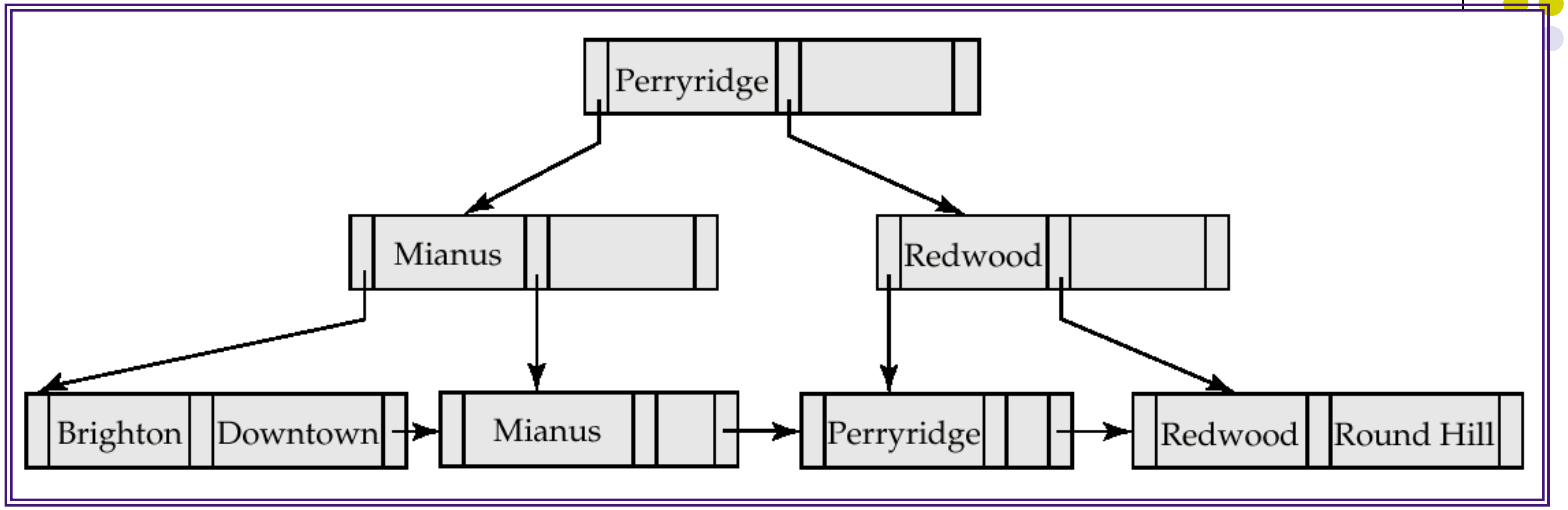
Tuple Insertion

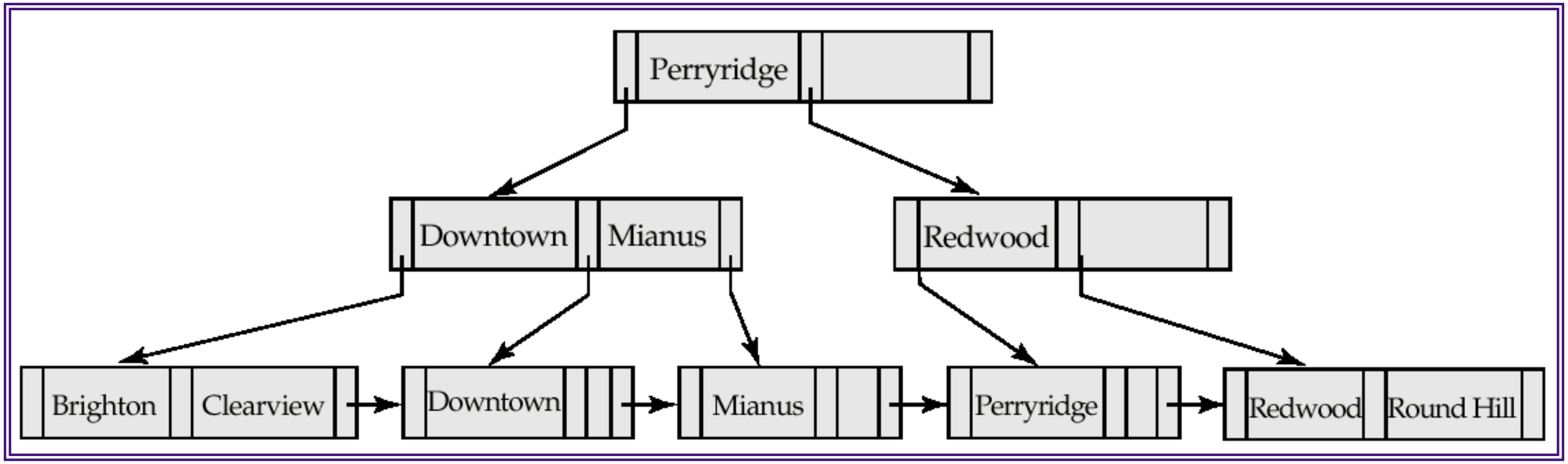
- Find the leaf node where the search key should go
- If already present
  - Insert record in the file. Update the bucket if necessary
    - This would be needed for secondary indexes
- If not present
  - Insert the record in the file
  - Adjust the index
    - Add a new *(Ki, Pi)* pair to the leaf node
    - Recall the keys in the nodes are sorted
  - What if there is no space ?

Tuple Insertion

- Splitting a node
  - Node has too many key-pointer pairs
    - Needs to store *n,* only has space for *n-1*
  - Split the node into two nodes
    - Put about half in each
  - Recursively go up the tree
    - May result in splitting all the way to the root
    - In fact, may end up adding a *level* to the tree
  - Pseudocode in the book !!

B+-Trees: Insertion



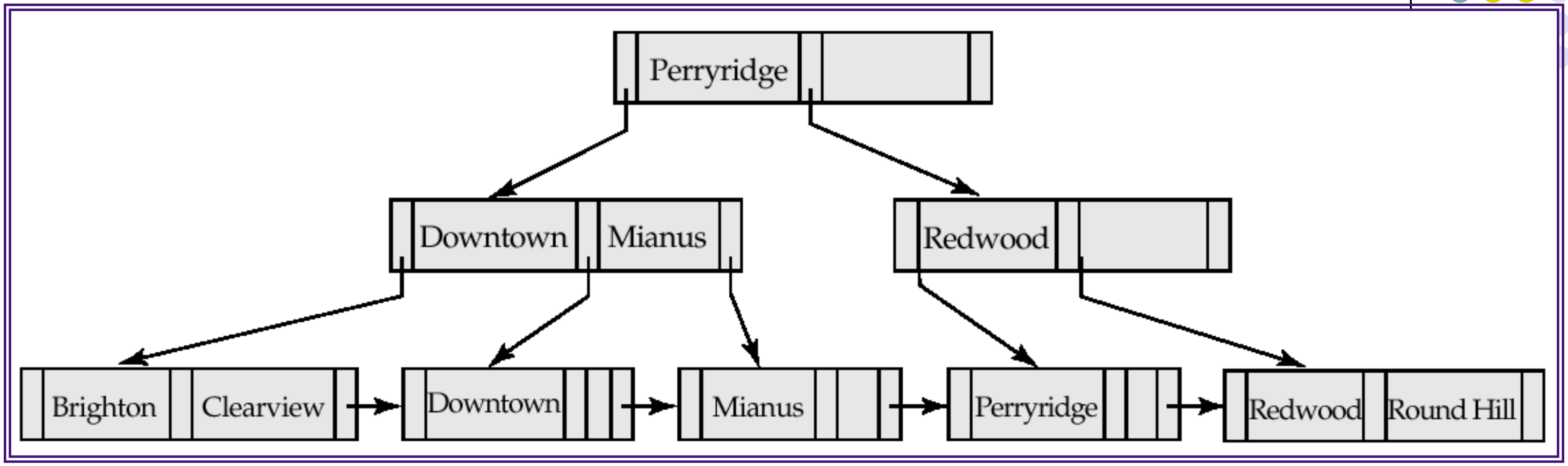


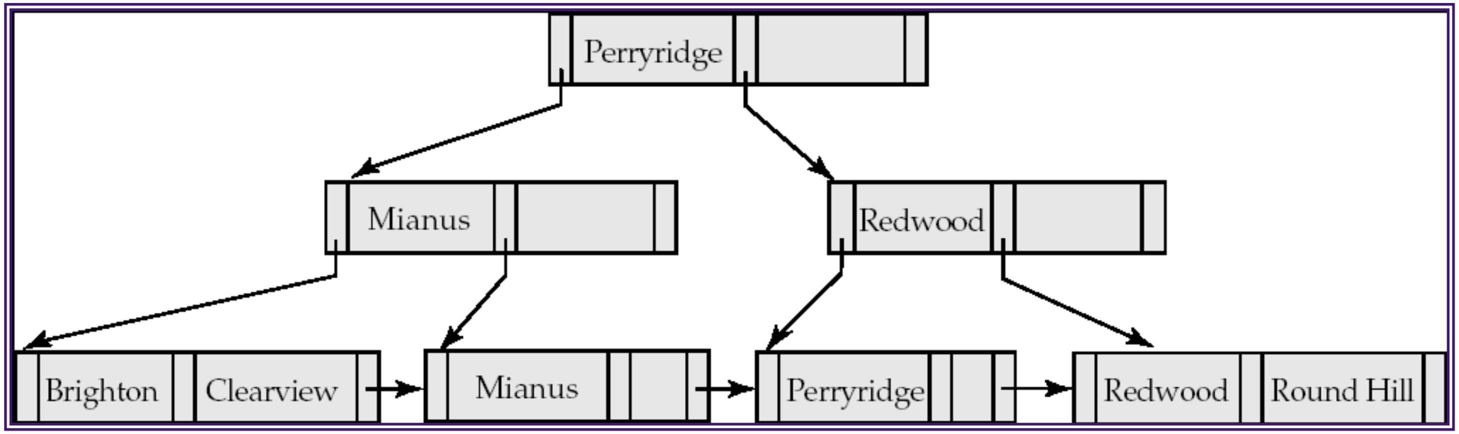
B+-Tree before and after insertion of "Clearview"

Updates on B^+-Trees: Deletion

- Find the record, delete it.
- Remove the corresponding (search-key, pointer) pair from a leaf node
  - Note that there might be another tuple with the same search-key
  - In that case, this is not needed
- Issue:
  - The leaf node now may contain too few entries
    - Why do we care ?
  - Solution:
    1. See if you can borrow some entries from a sibling
    2. If all the siblings are also just barely full, then *merge (opposite of split)*
  - May end up merging all the way to the root
  - In fact, may reduce the height of the tree by one

Examples of B+-Tree Deletion

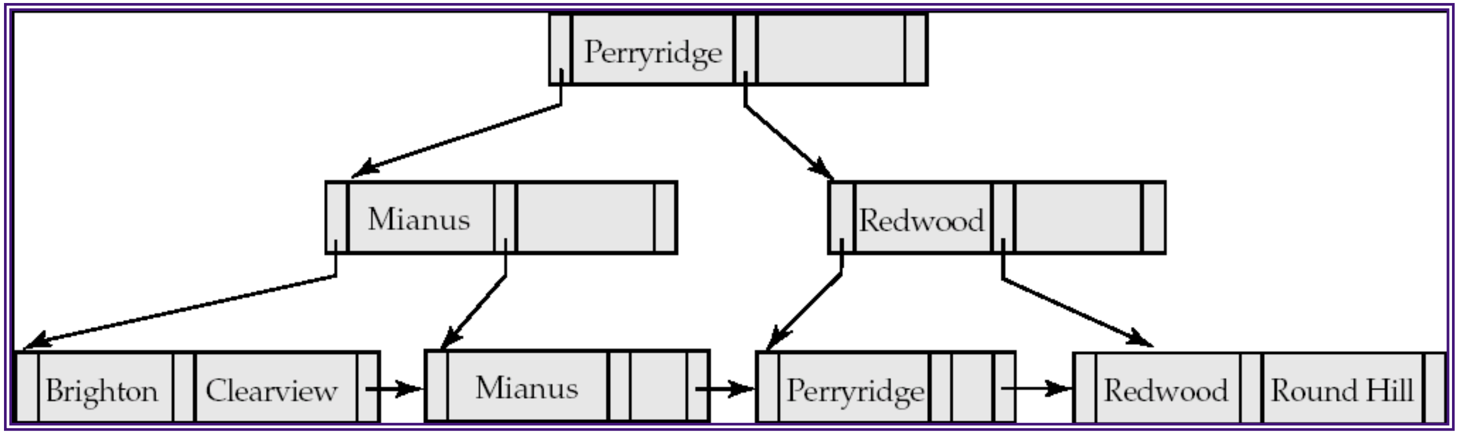


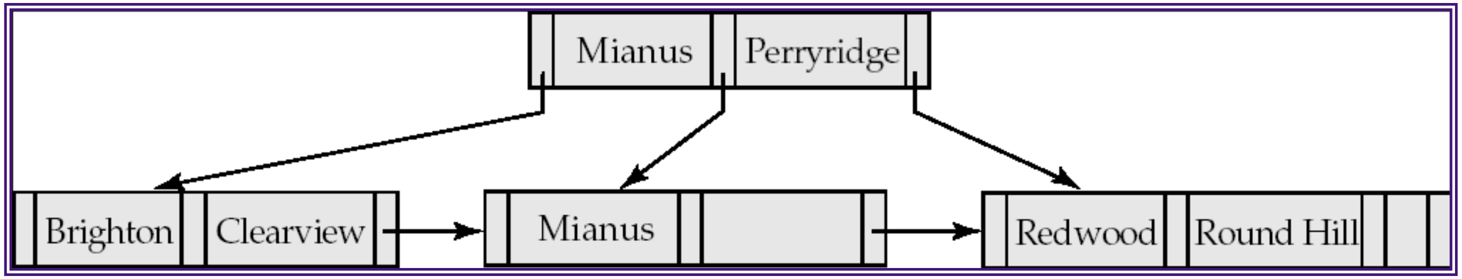


Before and after deleting "Downtown"

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

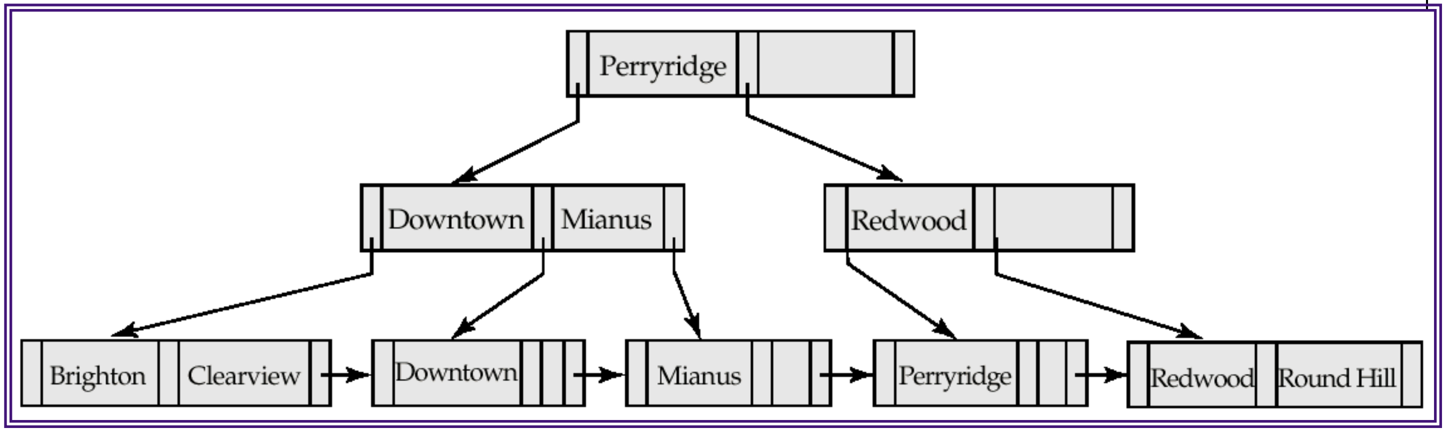
Examples of B+-Tree Deletion

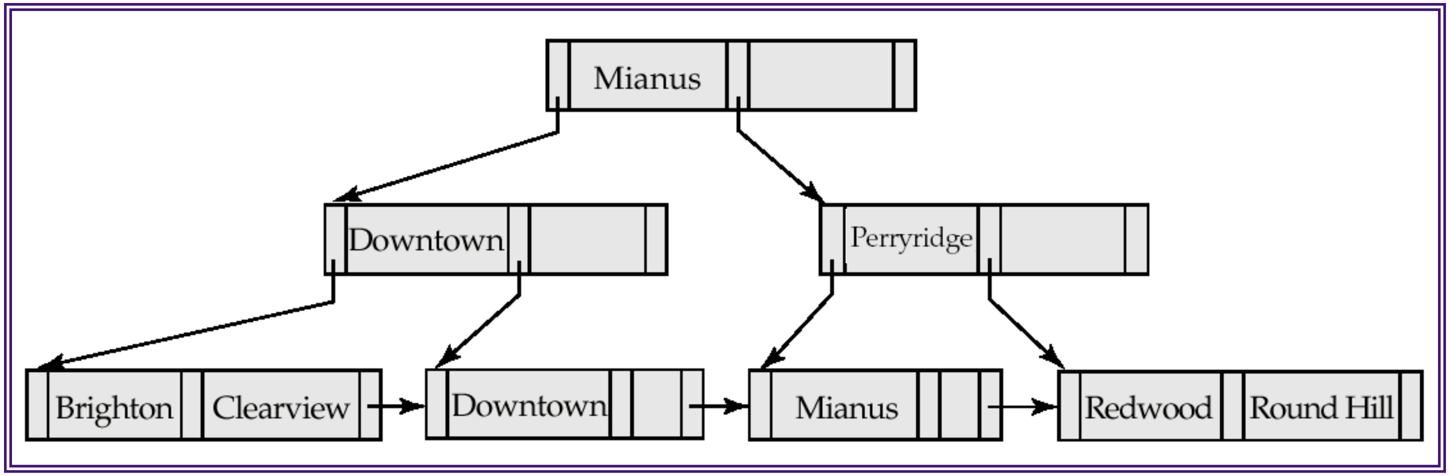




Deletion of "Perryridge" from result of previous example

Example of B^{+}-tree Deletion

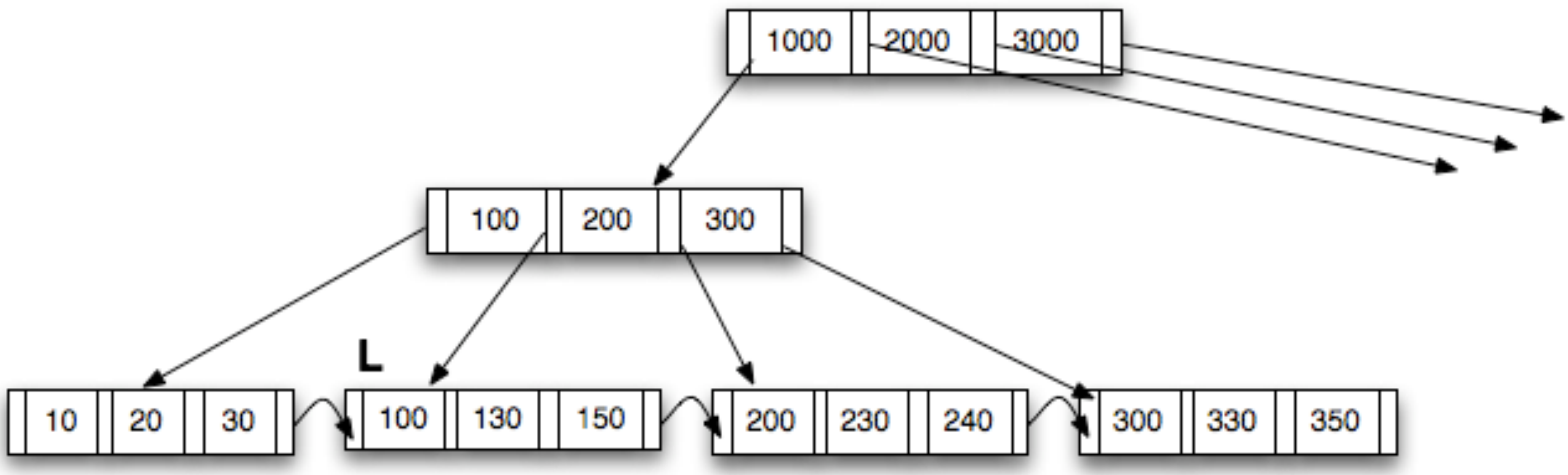




Before and after deletion of "Perryridge" from earlier example

Another B+Tree Insertion Example

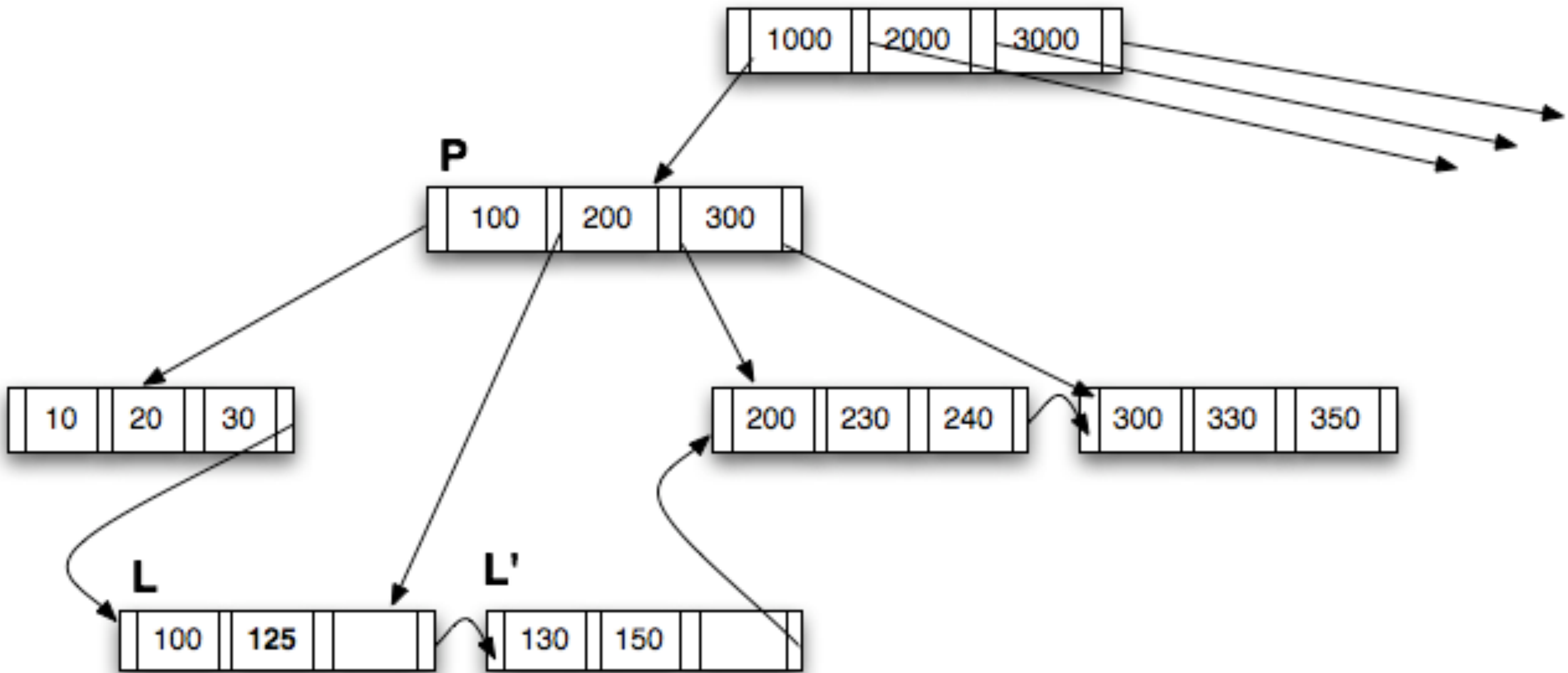
INITIAL TREE



Next slides show the insertion of (125) into this tree
According to the Algorithm in Figure 12.13, Page 495

Another Example: INSERT (125)

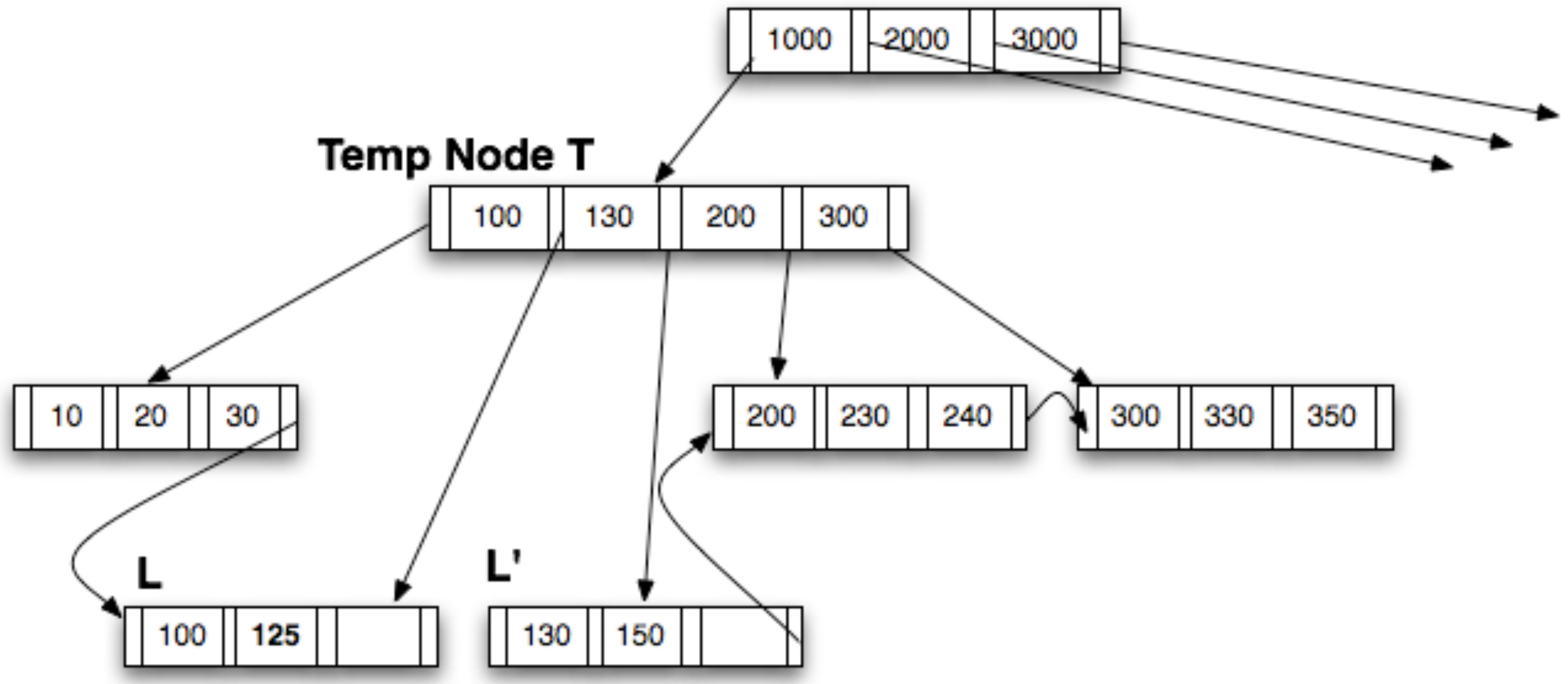
Step 1: Split L to create L’



Insert the lowest value in L’ (130) upward into the parent P

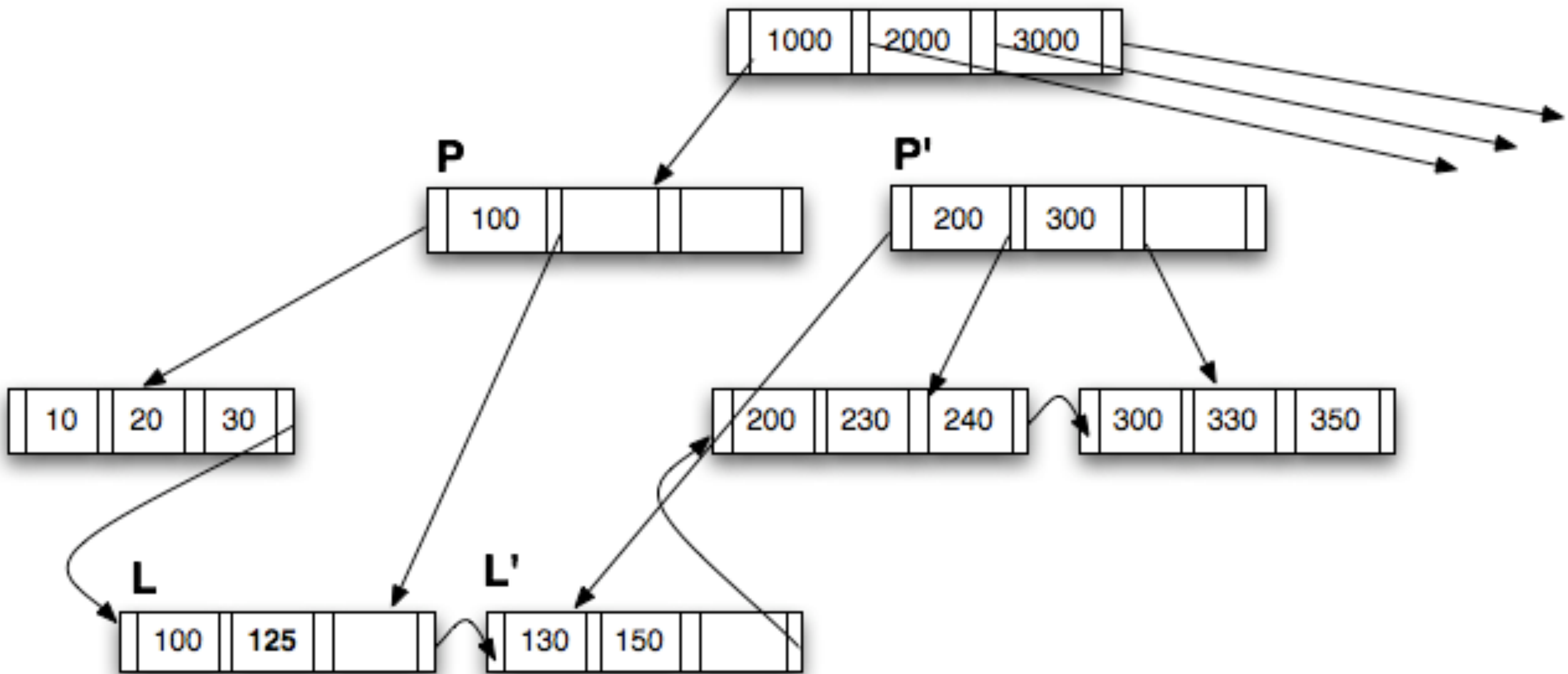
Another Example: INSERT (125)

Step 2: Insert (130) into P by creating a temp node T



Another Example: INSERT (125)

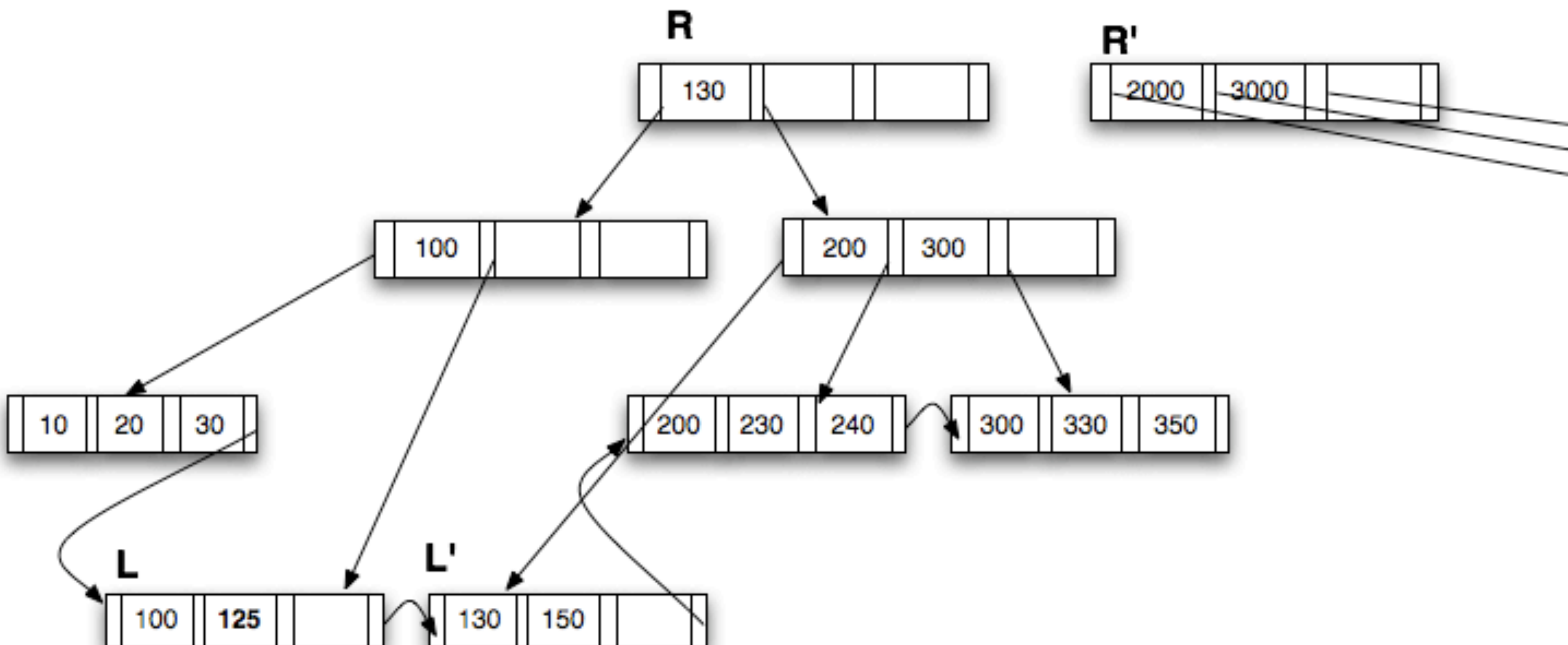
Step 3: Create P'; distribute from T into P and P'



New P has only 1 key, but two pointers so it is OKAY.
This follows the last 4 lines of Figure 12.13 (note that "n" = 4)
K" = 130. Insert upward into the root

Another Example: INSERT (125)

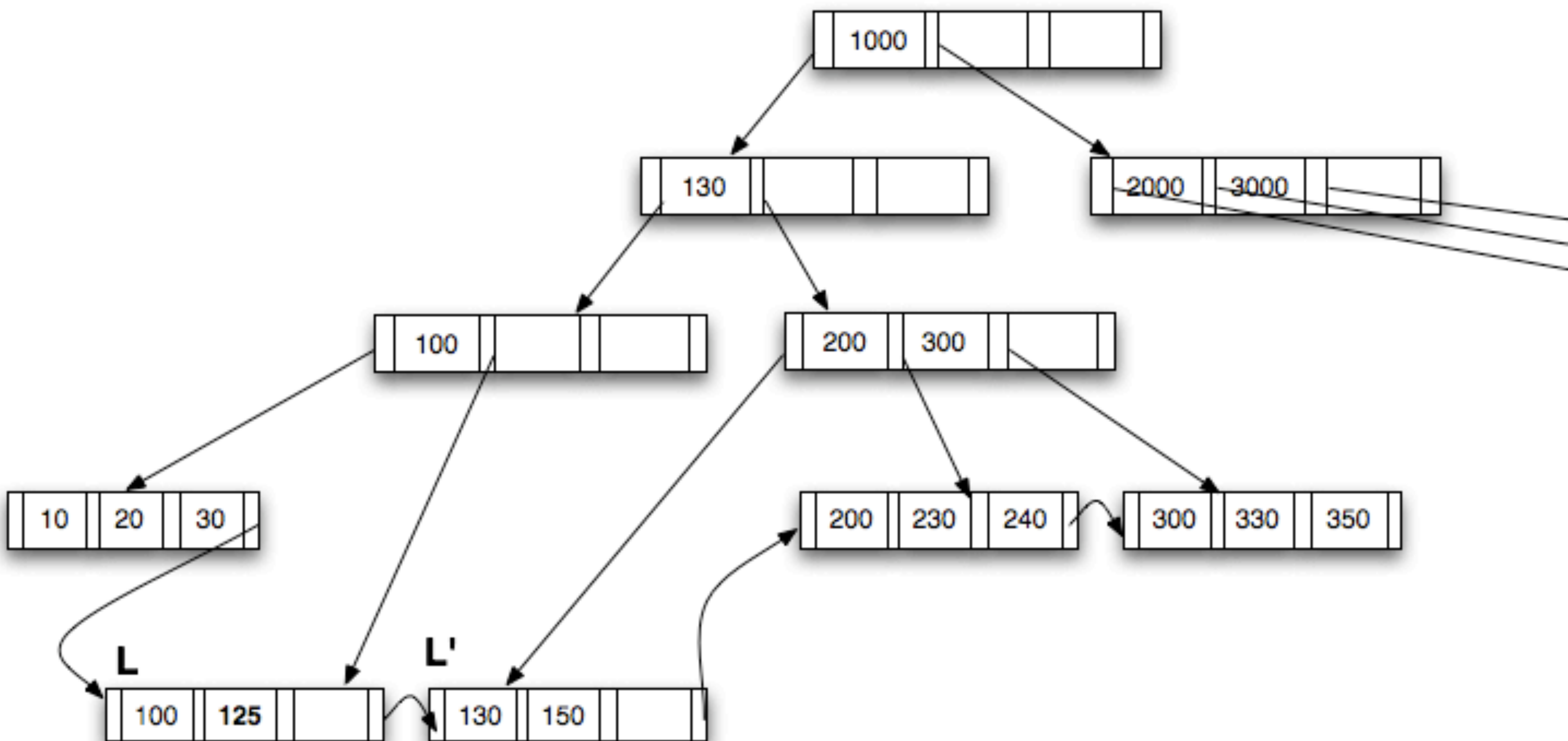
Step 4: Insert (130) into the parent (R); create R’



Once again following the insert_in_parent() procedure, K” = 1000

Another Example: INSERT (125)

Step 5: Create a new root



B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 entries
  - Height 4: 133^4 = 312,900,700 entries
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

B+ Trees: Summary

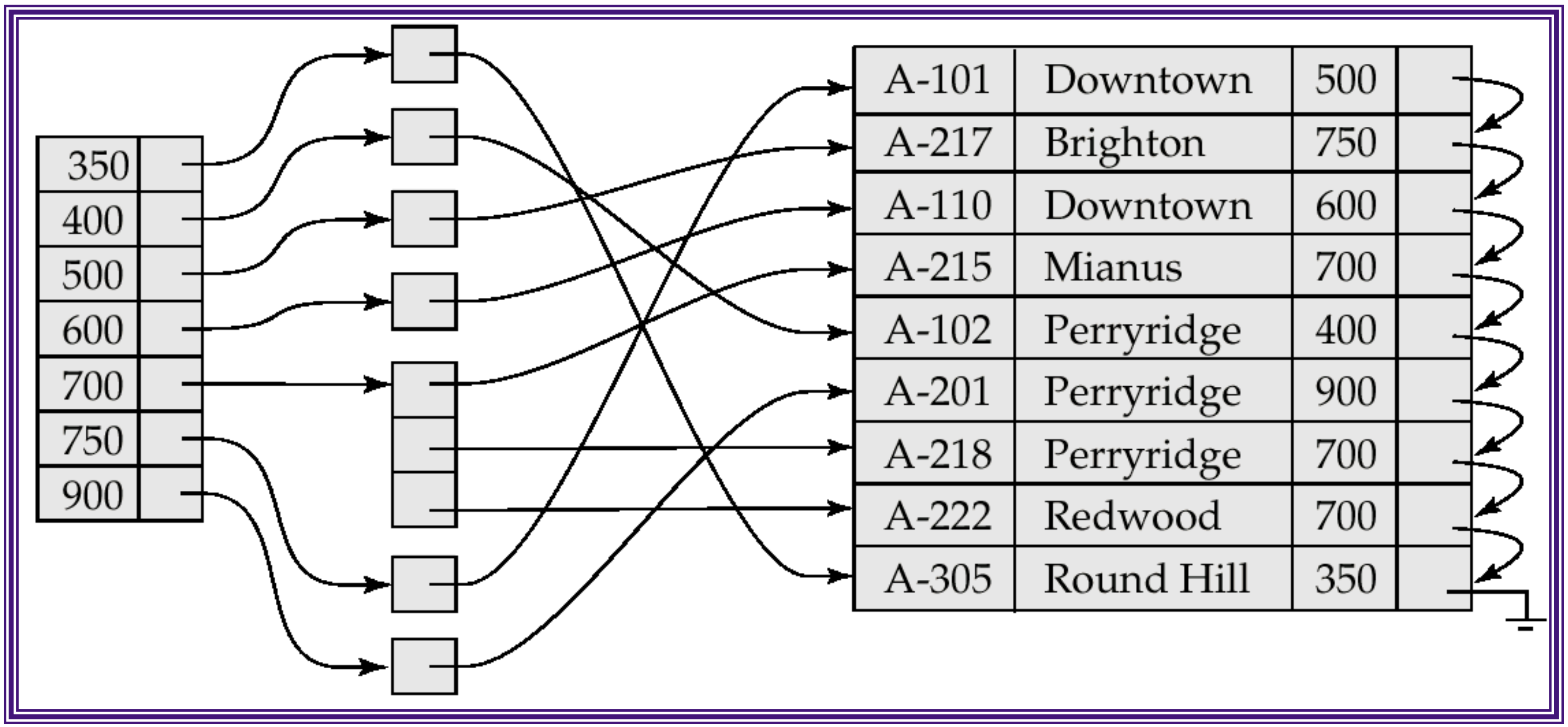
- Searching:
  - $\log_d(n)$ – Where *d* is the order, and *n* is the number of entries
- Insertion:
  - Find the leaf to insert into
  - If full, split the node, and adjust index accordingly
  - Similar cost as searching
- Deletion
  - Find the leaf node
  - Delete
  - May not remain half-full; must adjust the index accordingly

More…

- Primary vs Secondary Indexes
- More B+-Trees
- Hash-based Indexes
  - Static Hashing
  - Extendible Hashing
  - Linear Hashing
- Grid-files
- R-Trees
- etc…

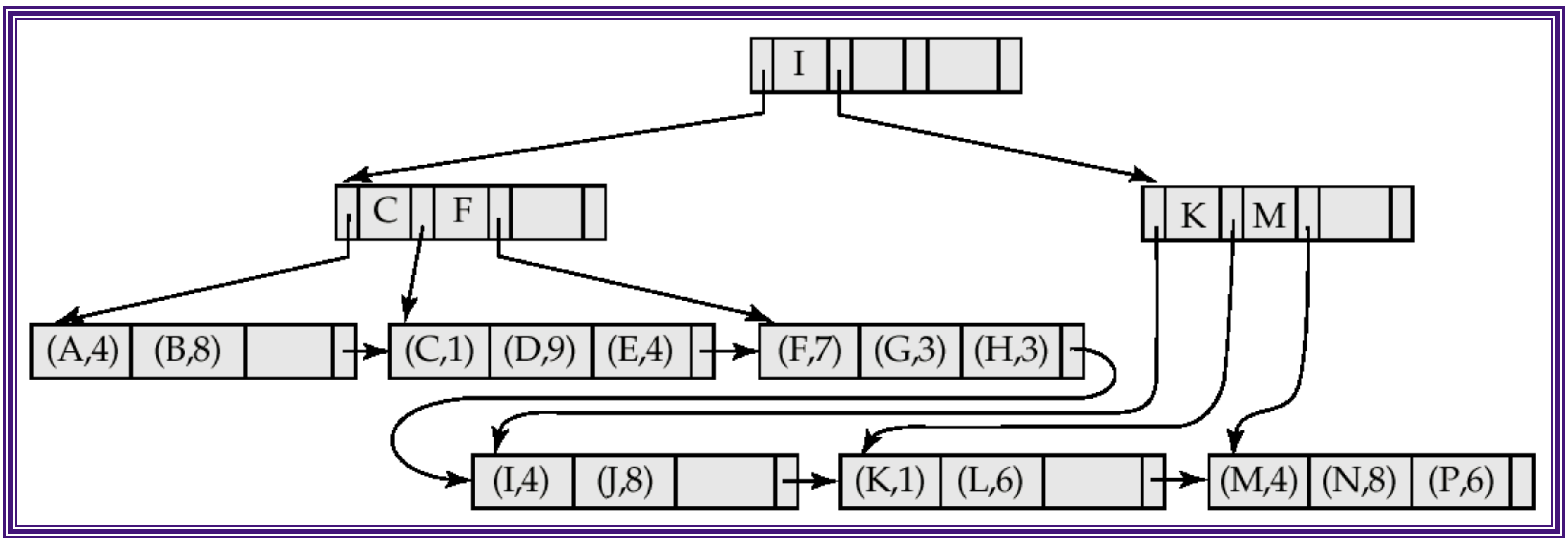
Secondary Index

- If relation not sorted by search key, called a *secondary index*
  - Not all tuples with the same search key will be together
  - Searching is more expensive



B+-Tree File Organization

- Store the records at the leaves
- Sorted order etc..



B-Tree

- Predates
- Different treatment of search keys
- Less storage
- Significantly harder to implement
- Not used.

Hash-based File Organization

Store record with search key *k*
in block number *h(k)*

e.g. for a person file,

h(SSN) = SSN % 4

Blocks called "*buckets*"

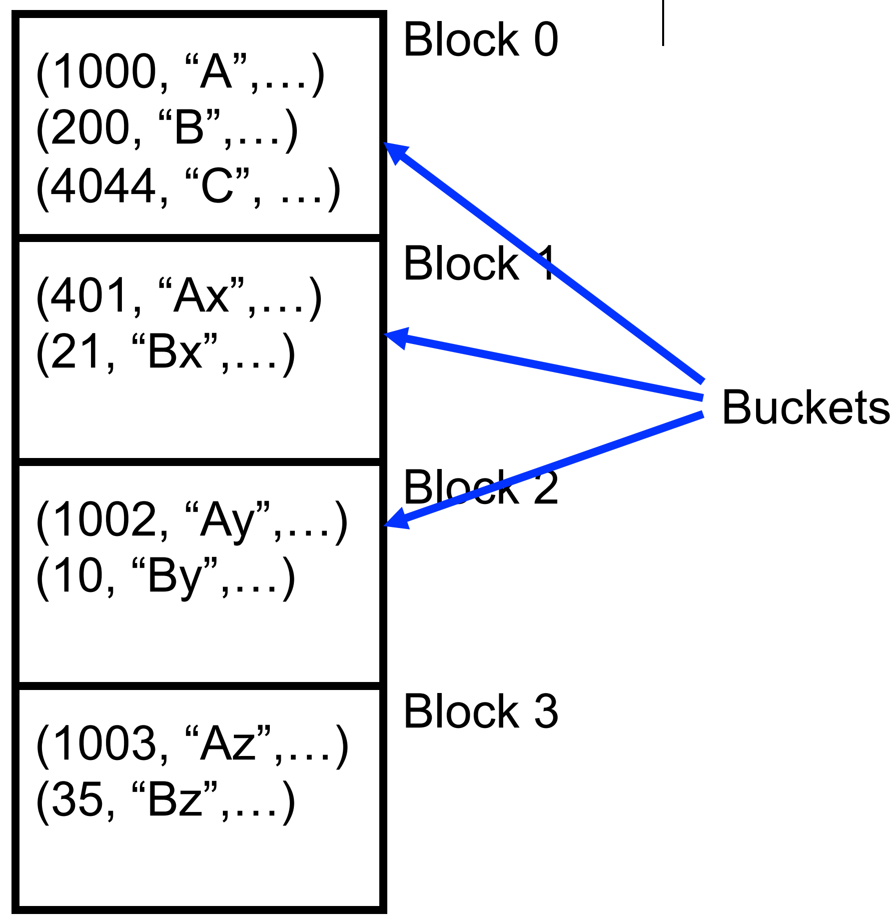
What if the block becomes full ?

Overflow pages

Uniformity property:

Don't want all tuples to map to the same bucket

h(SSN) = SSN % 2 would be bad



Hash-based File Organization

Hashed on “branch-name”

Hash function:

$a = 1, b = 2, .., z = 26$

$h(abz)$

$= (1 + 2 + 26) \% 10$

$= 9$

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 Indexes

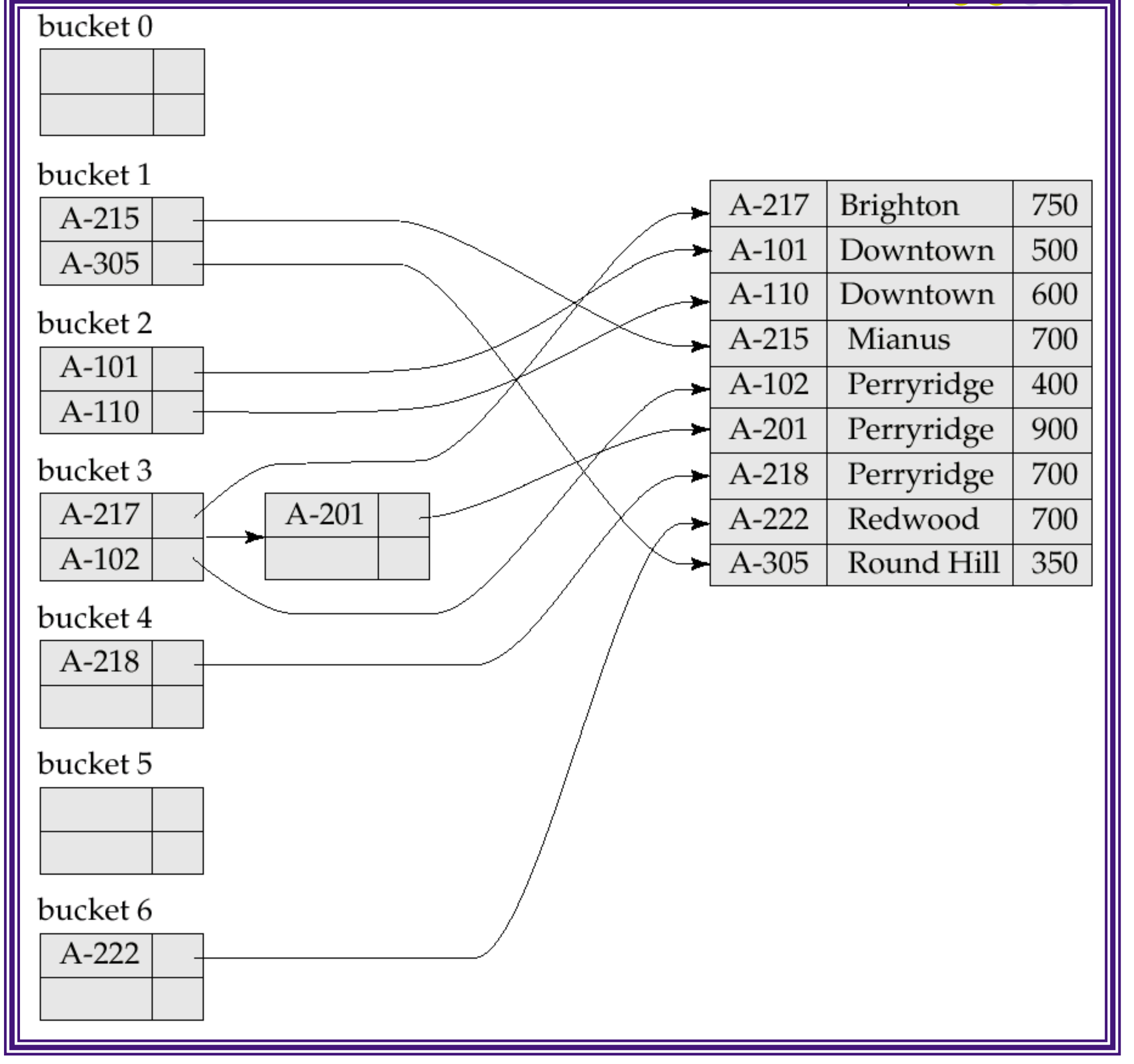
Extends the basic idea

Search:

- Find the block with search key
- Follow the pointer

Range search ?

$a < X < b$?



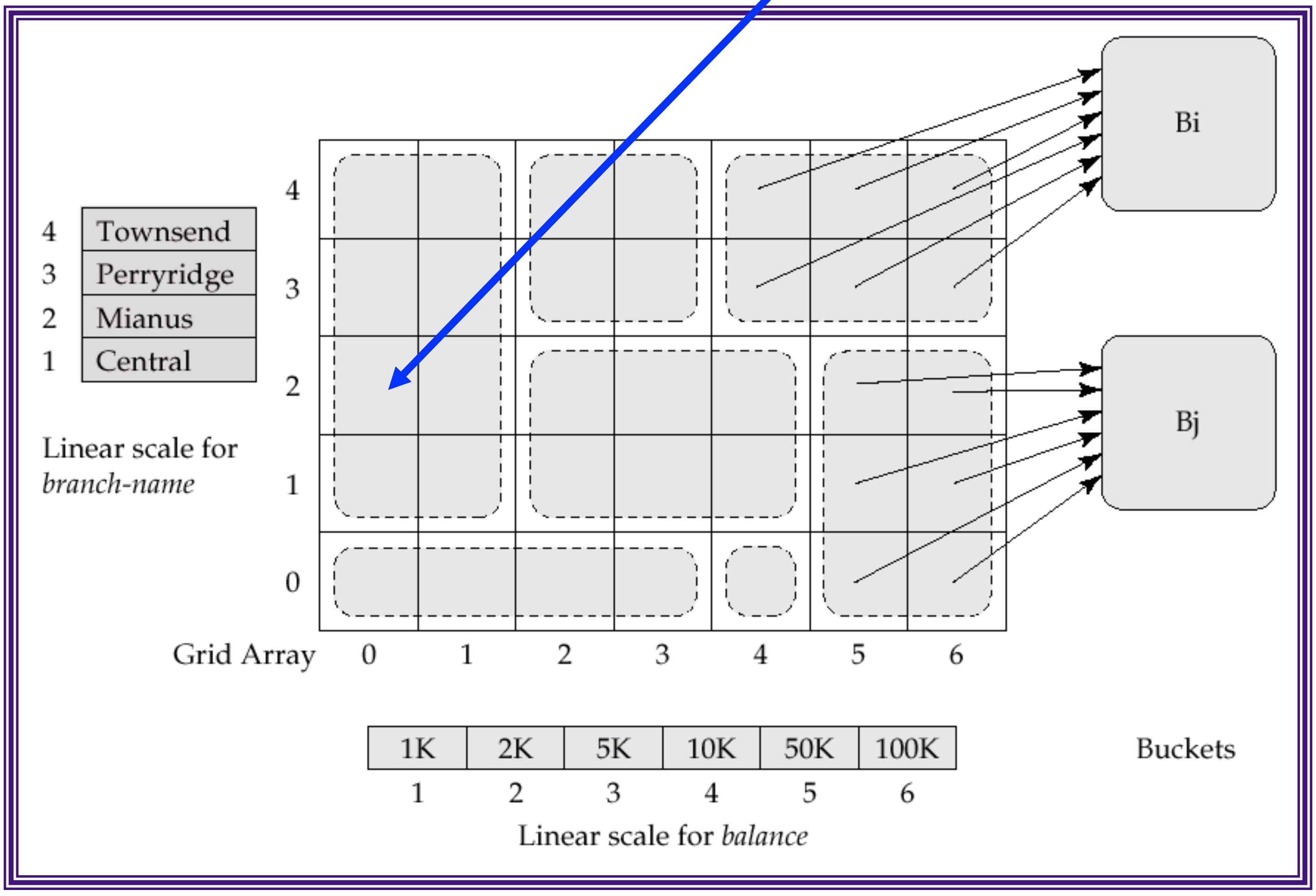
Hash Indexes

- Very fast search on equality
- Can’t search for “ranges” at all
  - Must scan the file
- Inserts/Deletes
  - Overflow pages can degrade the performance
- Two approaches
  - Dynamic hashing
  - Extendible hashing

Grid Files

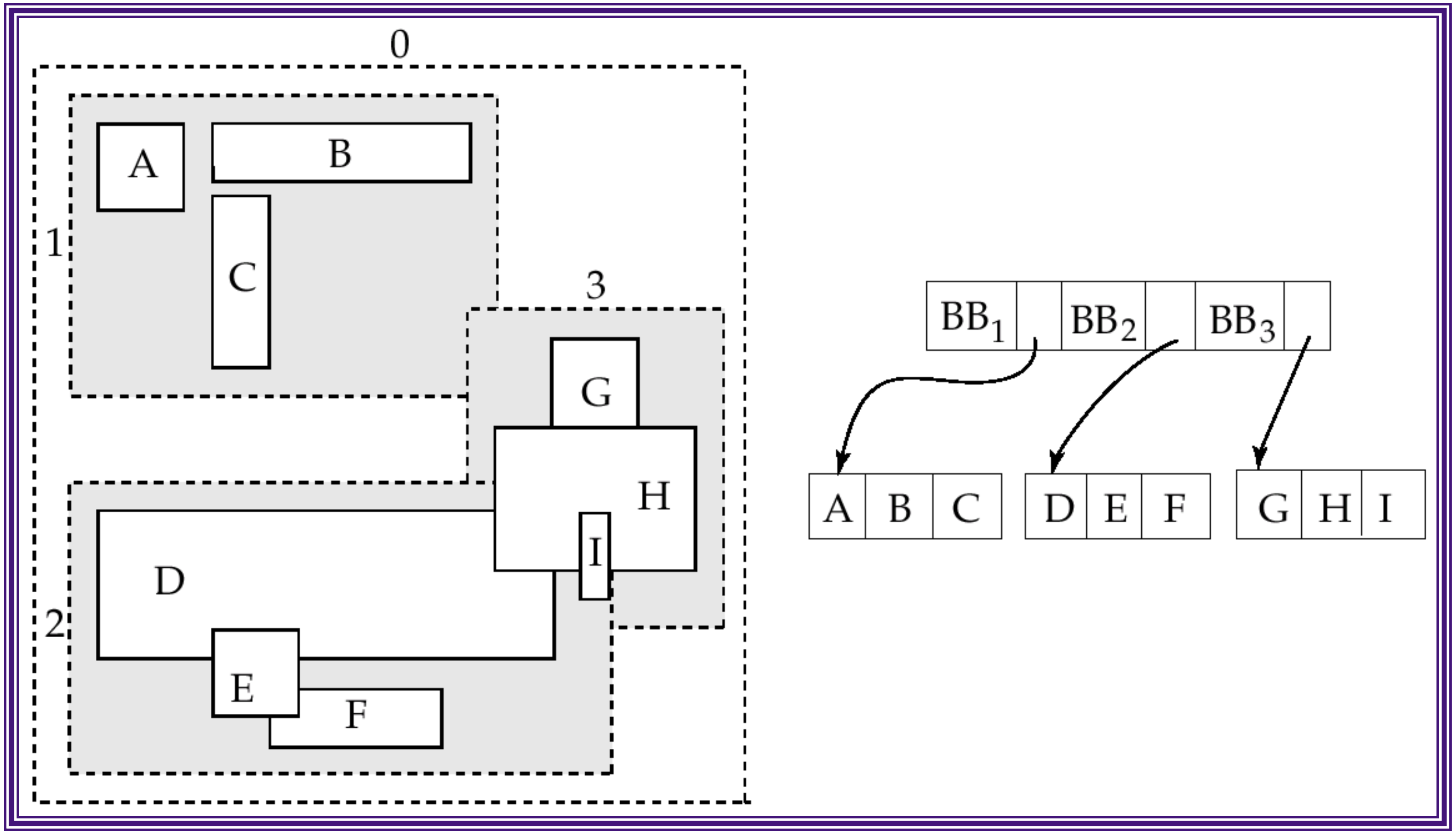
Multidimensional index structure
Can handle: *X = x1 and Y = y1*
a < X < b and c < Y < d

Stores pointers to tuples with :
branch-name between Mianus and Perryridge
and balance < 1k



R-Trees

For spatial data (e.g. maps, rectangles, GPS data etc)



Conclusions

- Indexing Goal: “Quickly find the tuples that match certain conditions”
- Equality and range queries most common
  - Hence B+-Trees the predominant structure for on-disk representation
  - Hashing is used more commonly for in-memory operations
- Many many more types of indexing structures exist
  - For different types of data
  - For different types of queries
    - E.g. “nearest-neighbor” queries