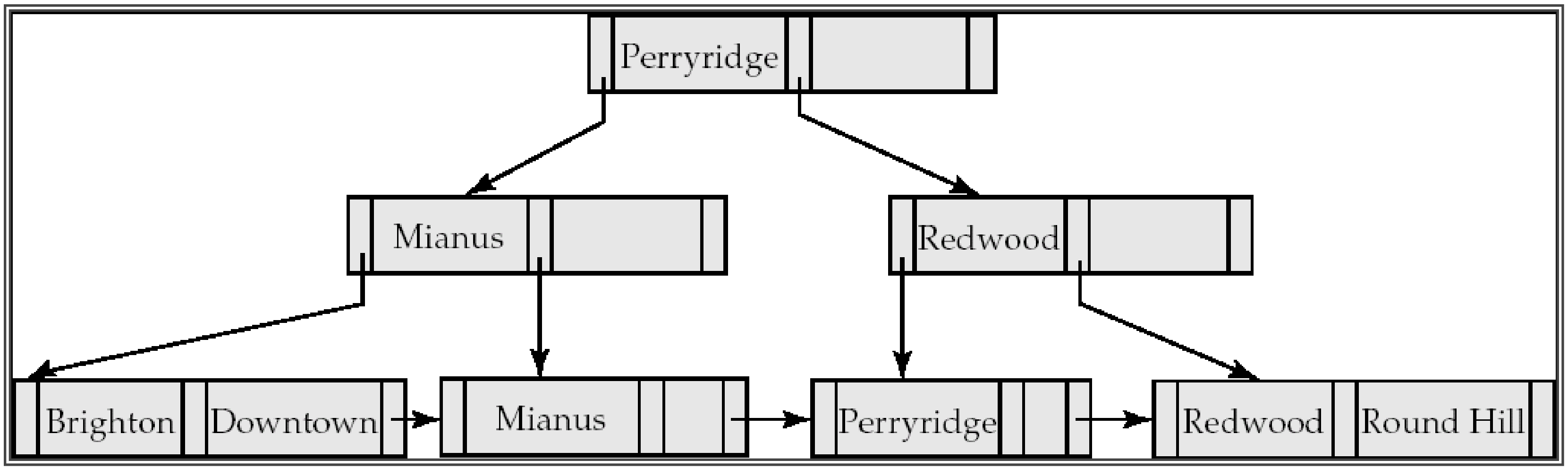
B+ trees

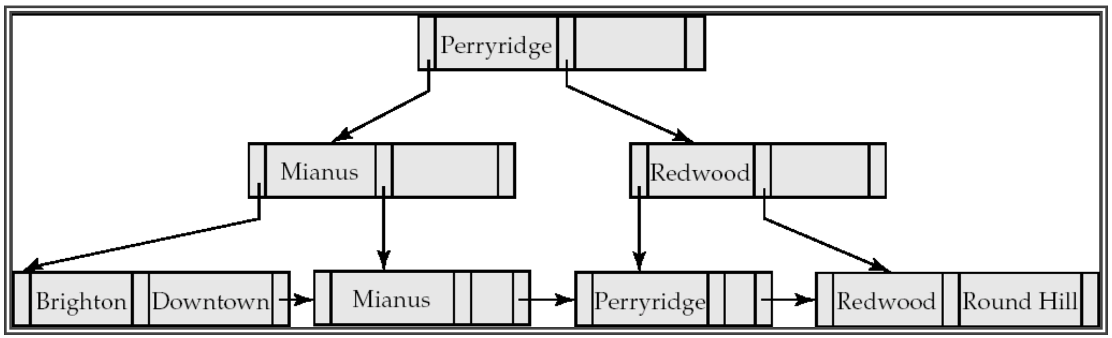
Example B+-Tree Index



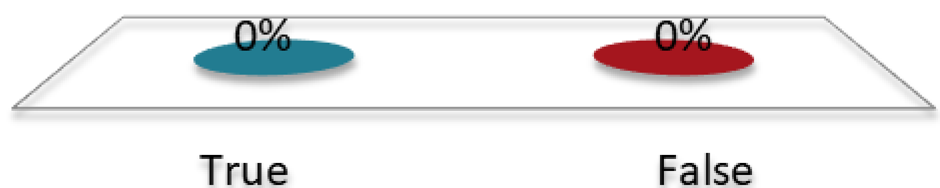
This is a sparse index

A. True

B. False



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	



B^+-Tree Node Structure

- Typical node

P_1	K_1	P_2	$\ldots$	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

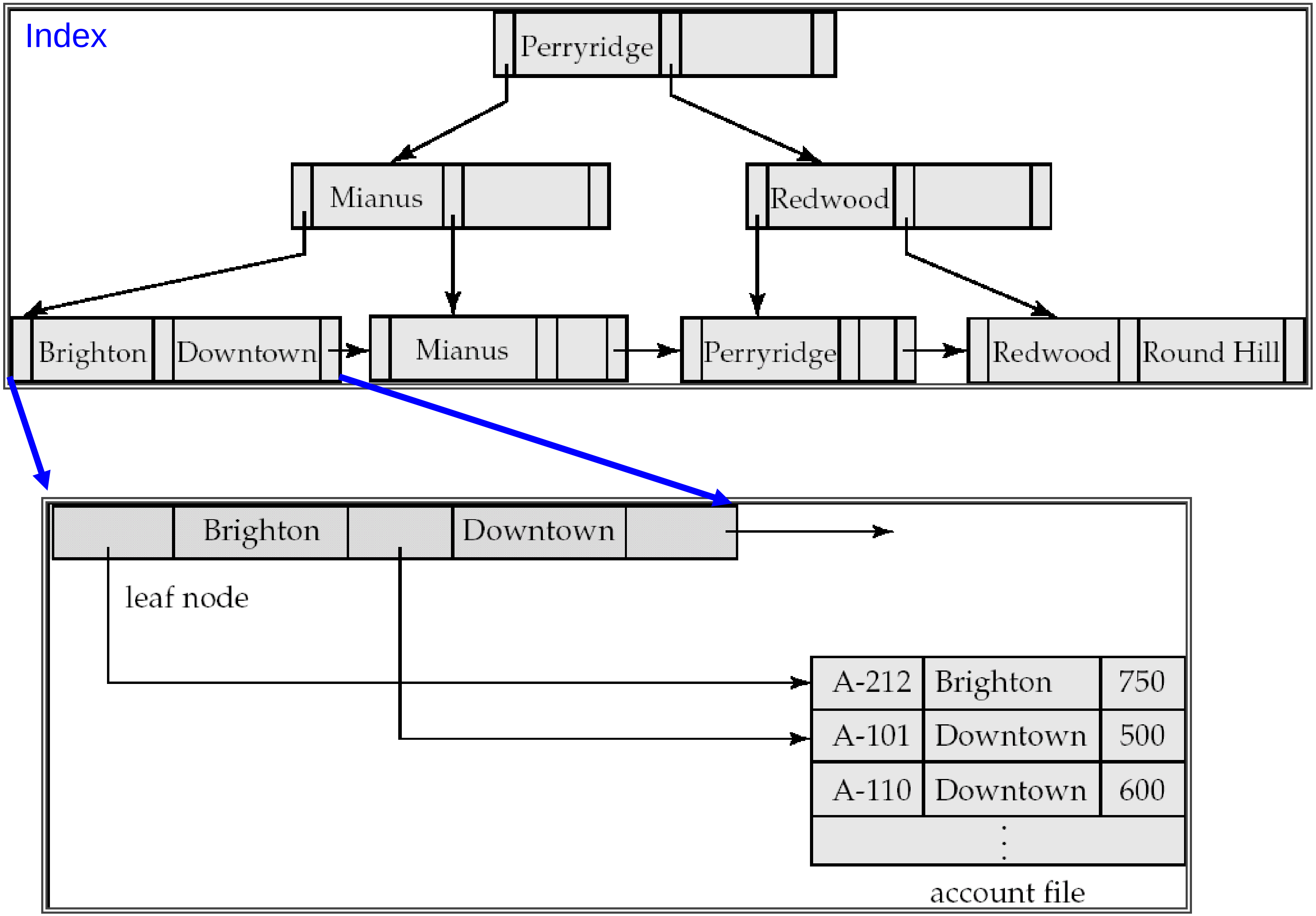
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Properties of B+-Trees

- It is balanced
  - Every path from the root to a leaf is same length
- Leaf nodes (at the bottom)
  - P_i contains the pointers to tuple(s) with search value K_i
  - …
  - Last pointer, P_n, is a pointer to the *next* leaf node
  - Must contain at least ceil((n-1)/2) keys

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

Example B+-Tree Index



Properties

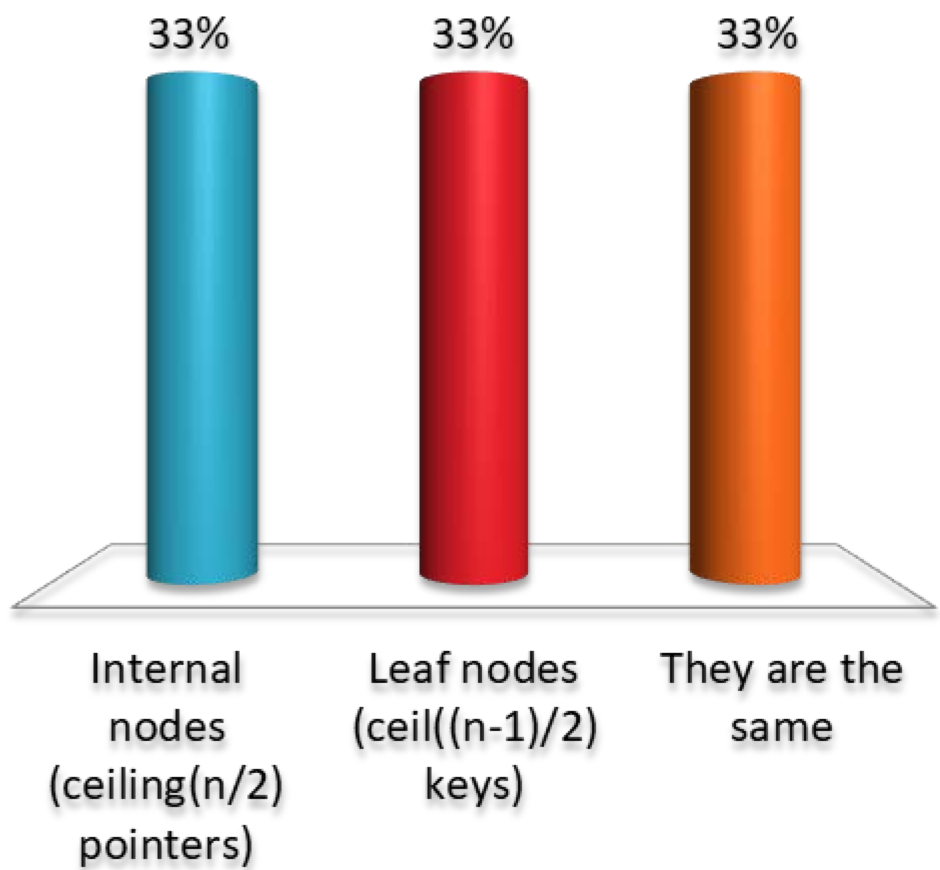
- Interior nodes

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

  - All tuples in the subtree pointed to by P_i, have search key < K_i
  - To find a tuple with key $K_j < K_i$, follow P_i
  - …
  - Finally, search keys in the tuples contained in the subtree pointed to by P_n, are all larger than or equal to K_{n-1}
  - Must contain at least ceiling(n/2) entries (pointers), unless root

Which has larger minimum size?

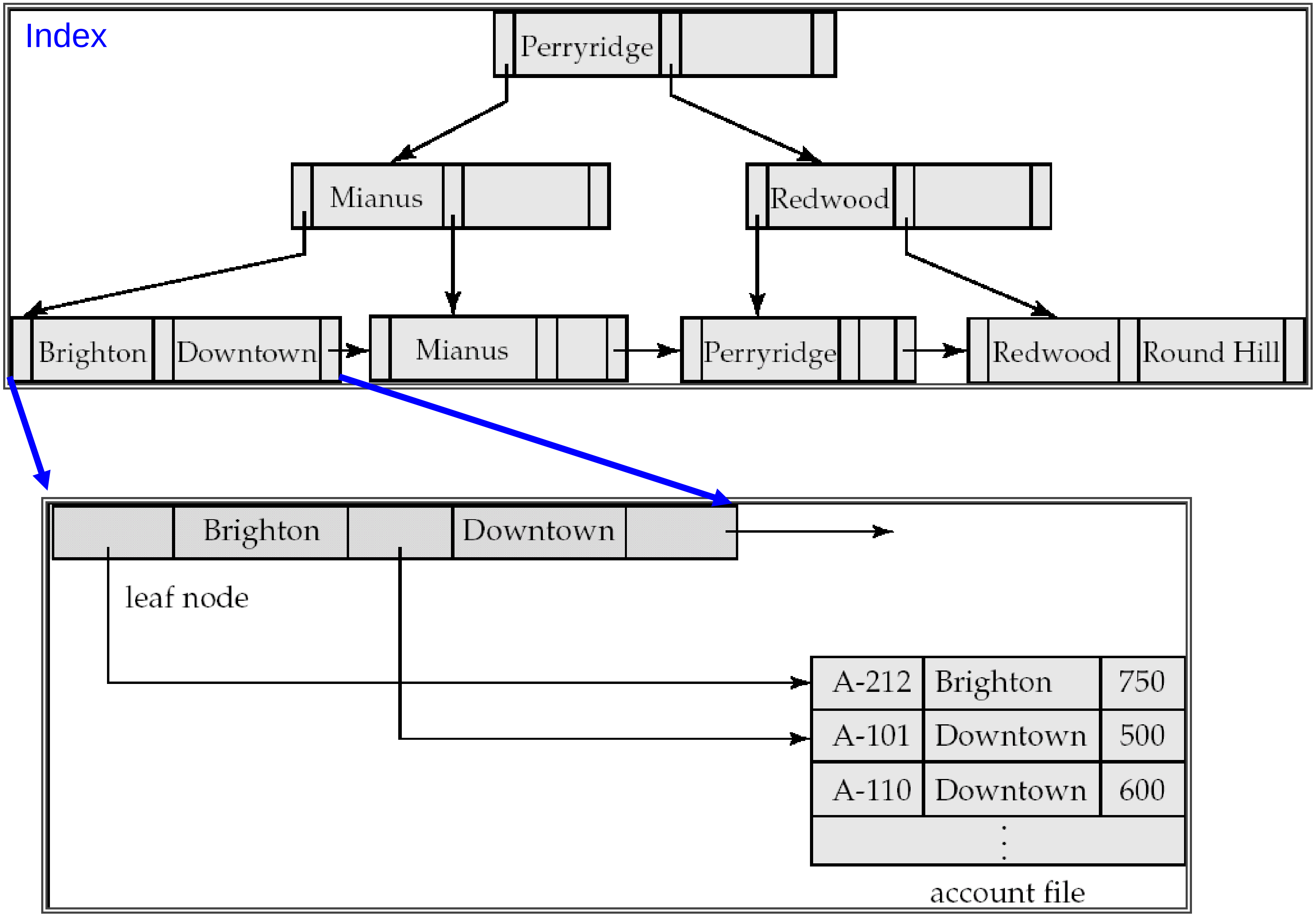
A. Internal nodes (ceiling(n/2) pointers)
B. Leaf nodes (ceil((n-1)/2) keys)
C. They are the same



The discussion in class focused on why A is the wrong answer:
When n is even, ceiling(n/2) = ceiling((n-1)/2)
When n is odd, ceiling(n/2) is one more than ceiling((n-1/2))

BUT, each node has one more pointer than key
SO, when n is odd, they have the exact same minimum size
and when n is even, leaf nodes have a larger minimum size

Example B+-Tree Index



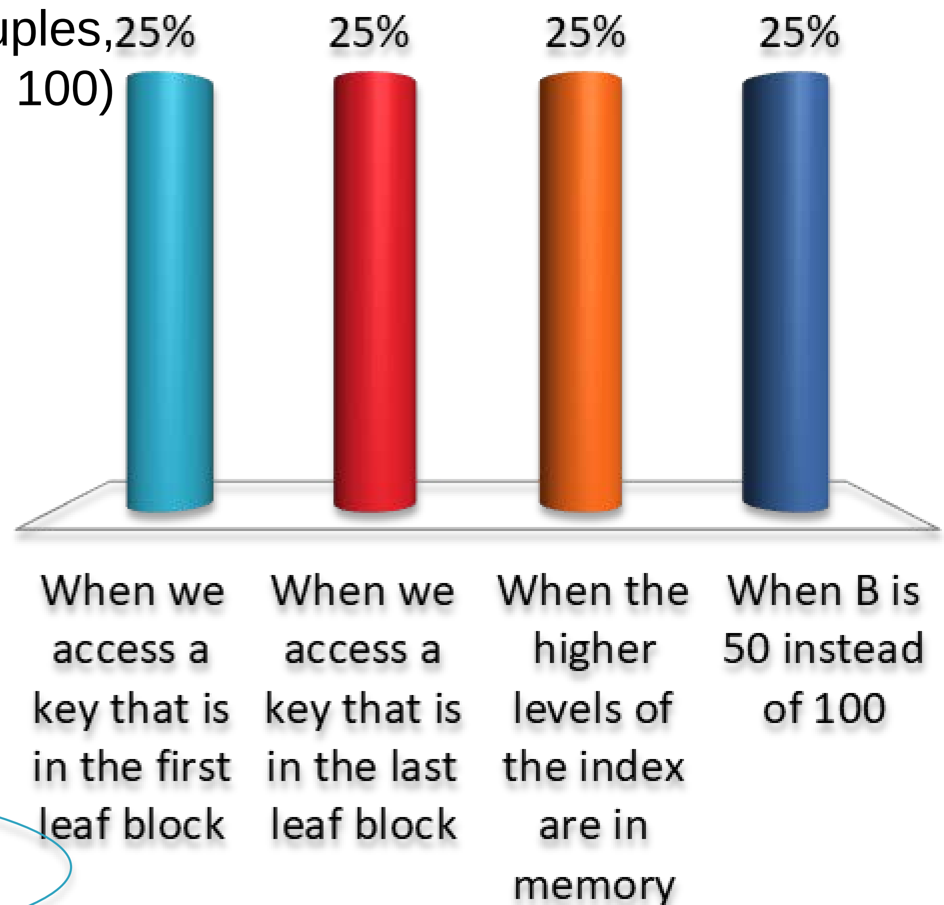
B+-Trees - Searching

- How to search ?
  - Follow the pointers
- Logarithmic
  - *$log_{B/2}(N)$,* where *B = Number of pointers per block*
  - *B* is also called the order of the B+-Tree Index
    - Typically 100 or so
- If a relation contains1,000,000 entries, takes only 4 random accesses (ceiling of $\log_{50}(1000000)$), *plus file access costs*

When will example from previous slide take fewer than 4 random disk IOs?

(Example was table with 1,000,000 tuples, number of pointers per block (B) = 100)

A. When we access a key that is in the first leaf block
B. When we access a key that is in the last leaf block
C. When the higher levels of the index are in memory
D. When B is 50 instead of 100



B+-Trees - Searching

- How to search ?
  - Follow the pointers
- Logarithmic
  - *$log_{B/2}(N)$,* where *B = Number of pointers per block*
  - *B* is also called the order of the B+-Tree Index
    - Typically 100 or so
- If a relation contains1,000,000 entries, takes only 4 random accesses (ceiling of $\log_{50}(1000000)$), *plus file access costs*
- 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 (K_i, P_i) pair to the leaf node
    - Recall the keys in the nodes are sorted
  - What if there is no space ?

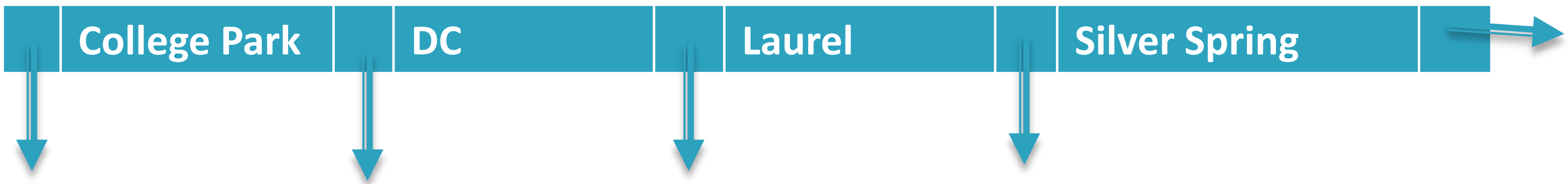
Tuple Insertion

- 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
    - Insert new K and P into oversized L (called T in textbook).
    - Create new L'
    - P_1 through $K_{ceil(n/2)}$ in (original) L, rest in L'
    - Let K' be smallest key in L'
      - Insert (K', L') in L-parent
  - Recursively go up the tree
    - In parent, ceil((N+1)/2) **pointers** go in left node, rest of **pointers** go right
    - (Key between the pointer split point goes next level up)
      - We will illustrate the difference in this splitting process on the next two slides
    - 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 !! *Read Fig 11.15…*

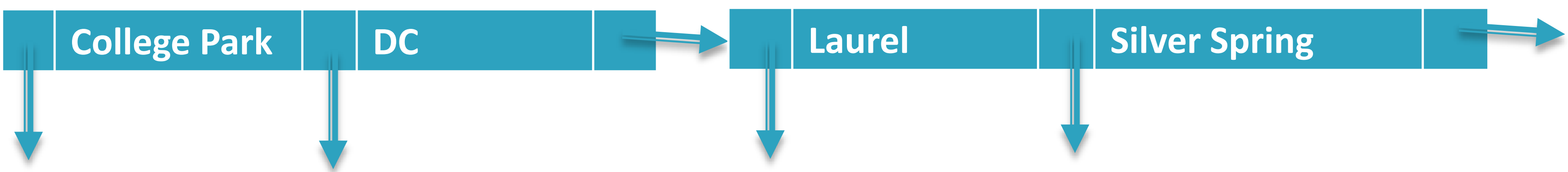
Leaf splitting

Add DC to:

Add DC into temporary oversized block, T



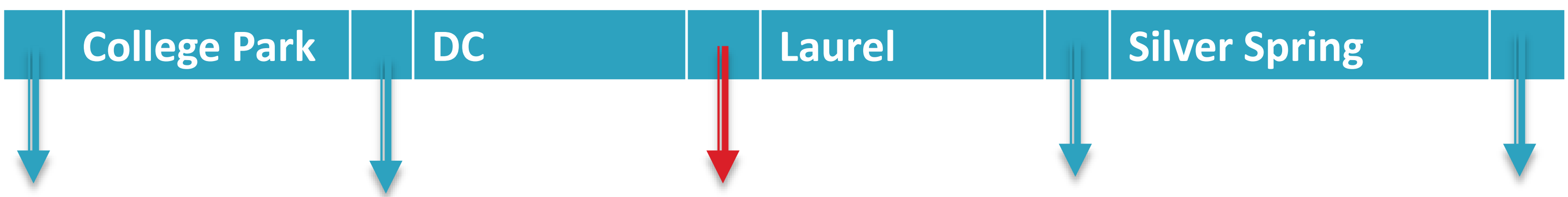
Split oversized block into two blocks



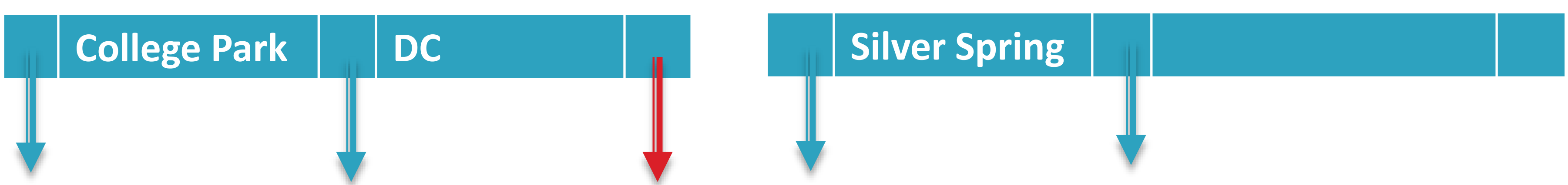
Non-leaf splitting

Add DC, pointer to DC block to:

Add DC into temporary oversized block, T

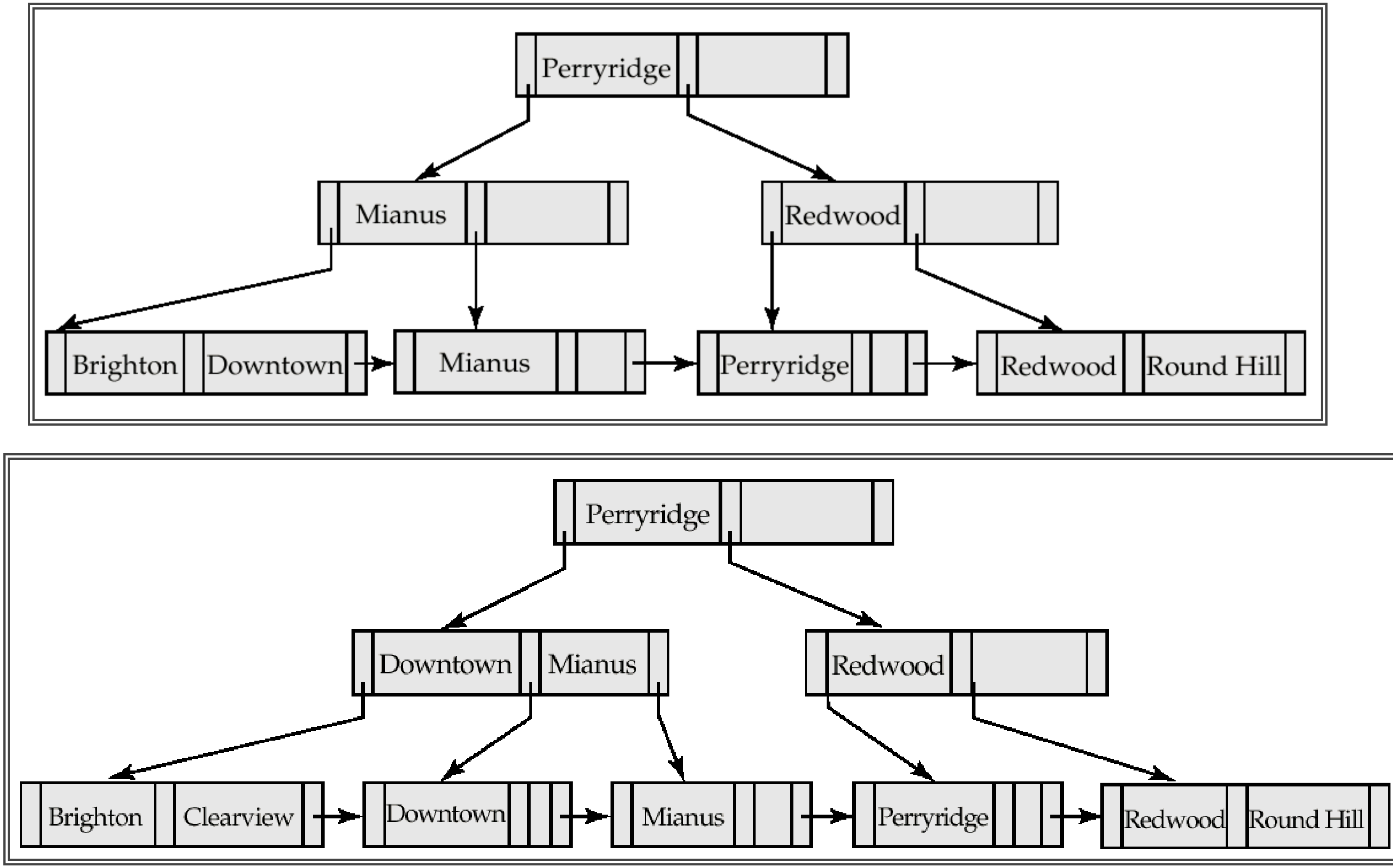


Split oversized block into two blocks



Laurel and pointer to the Silver Spring node gets inserted into next level up

B+-Trees: Simple Insertion



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

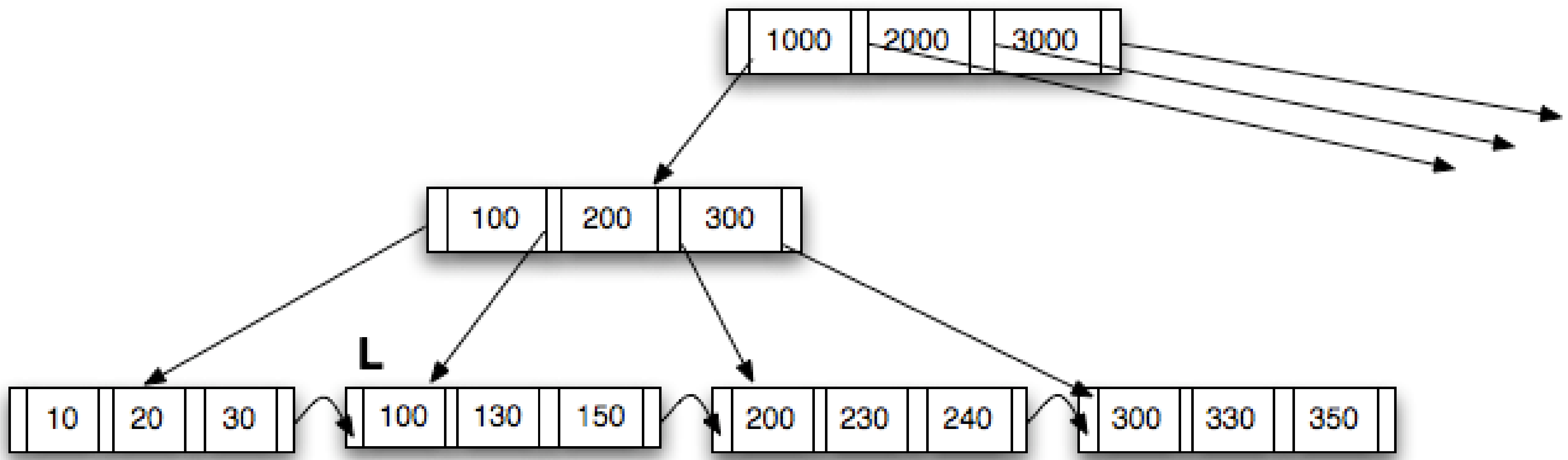
Another B+Tree Insertion Example

n=4

2 ptrs interior nodes

2 ptrs leaves

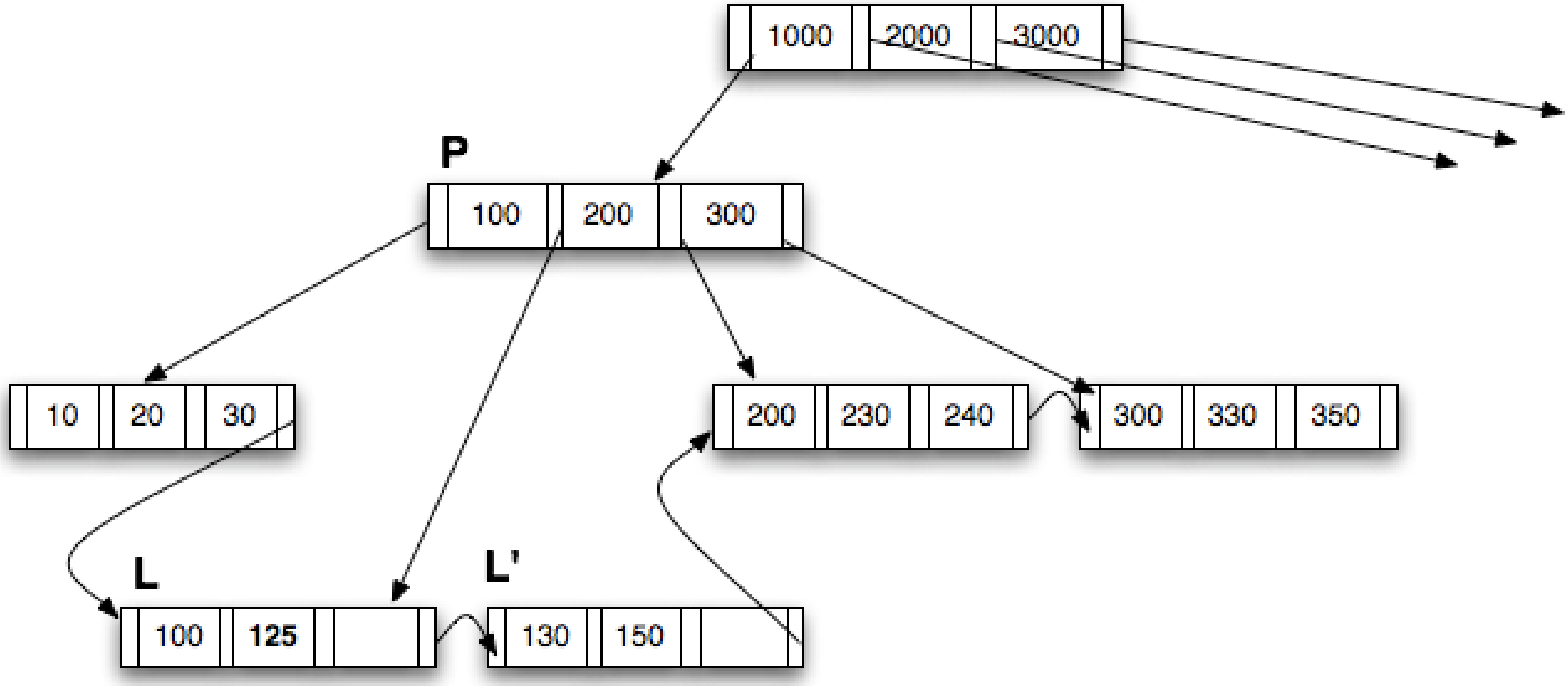
INITIAL TREE



Next slides show the insertion of (125) into this tree

Another Example: INSERT (125)

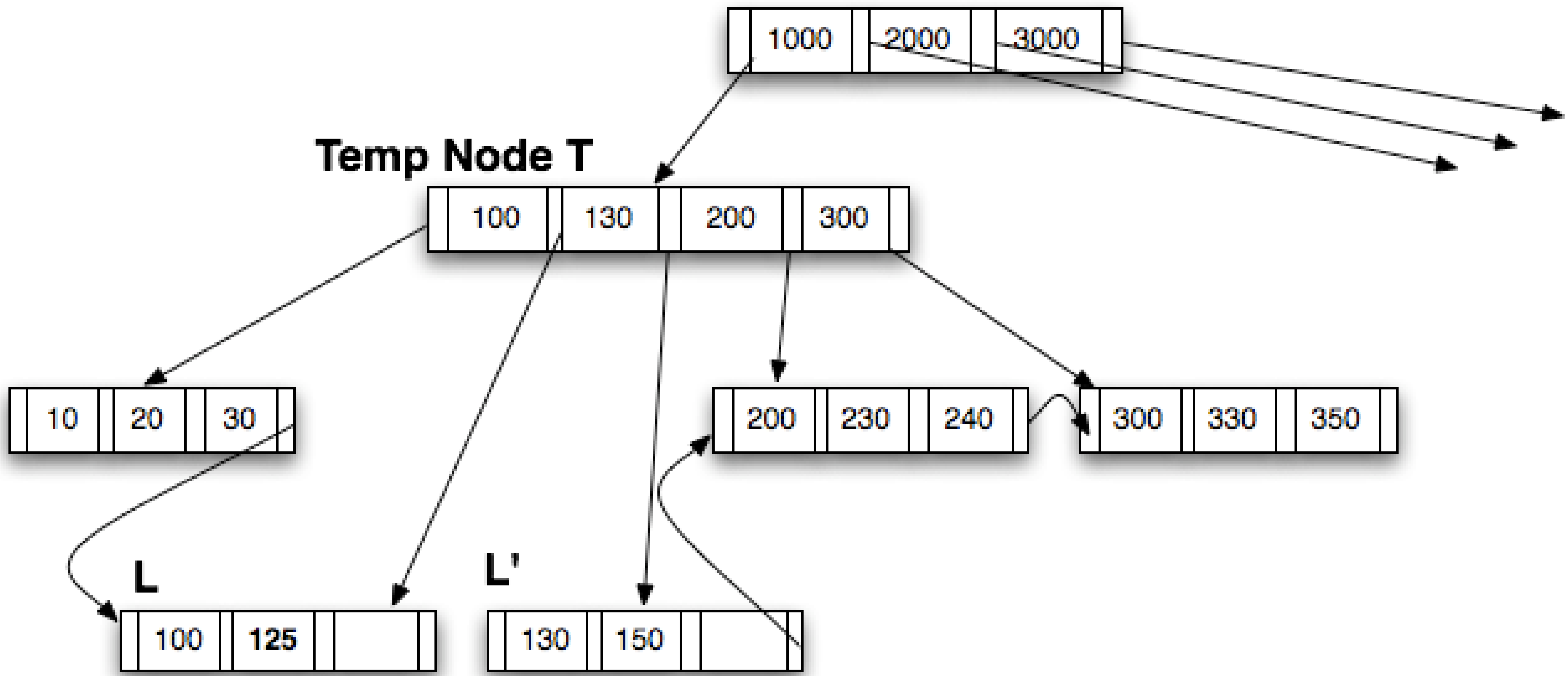
Step 1: Split L to create L'



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

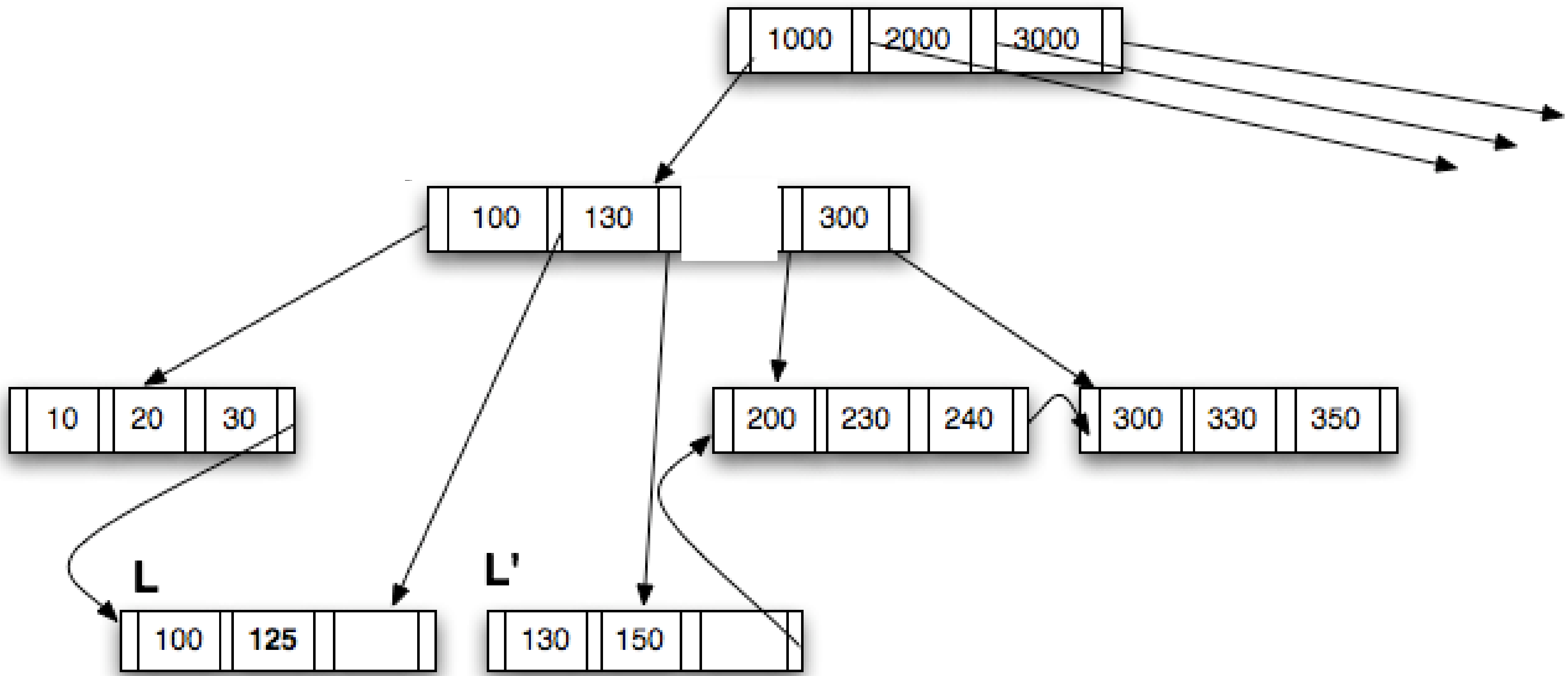
Another Example: INSERT (125)

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



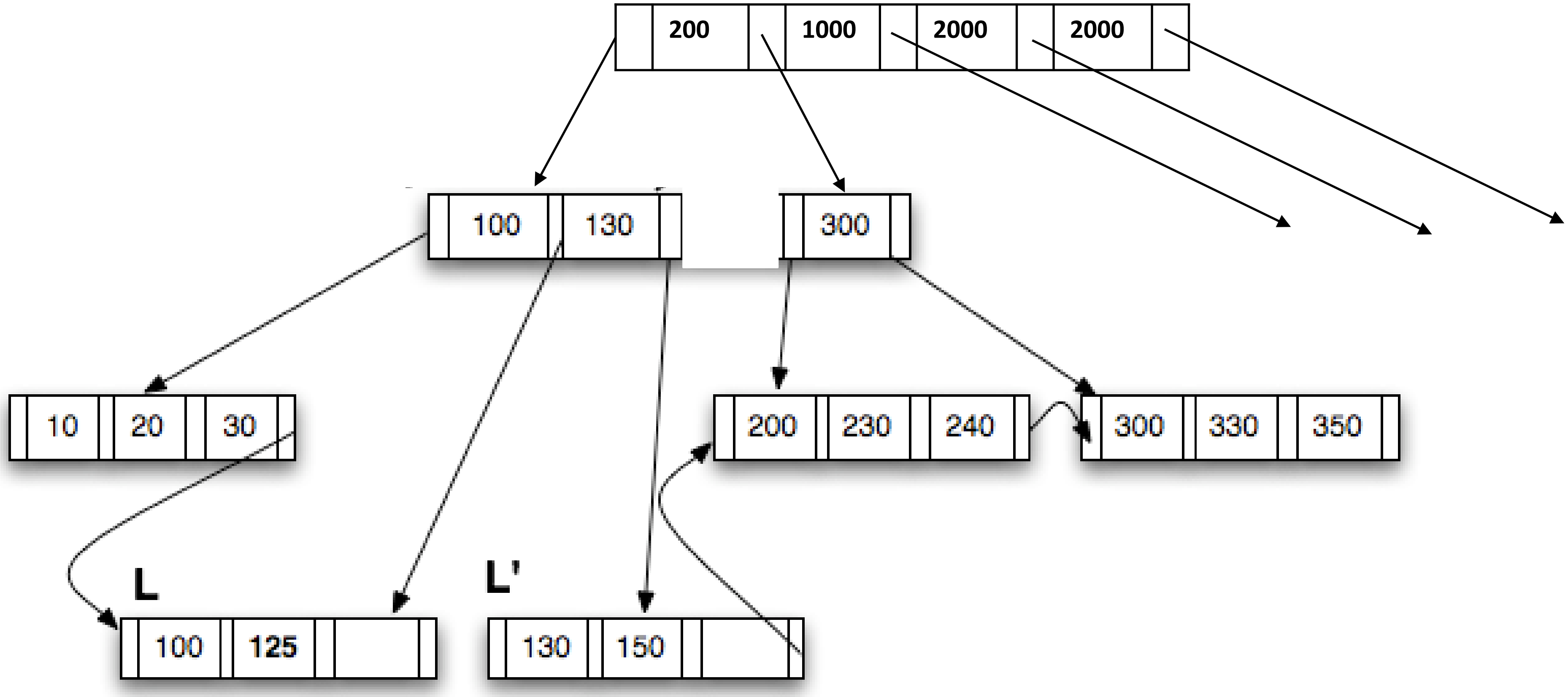
Another Example: INSERT (125)

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



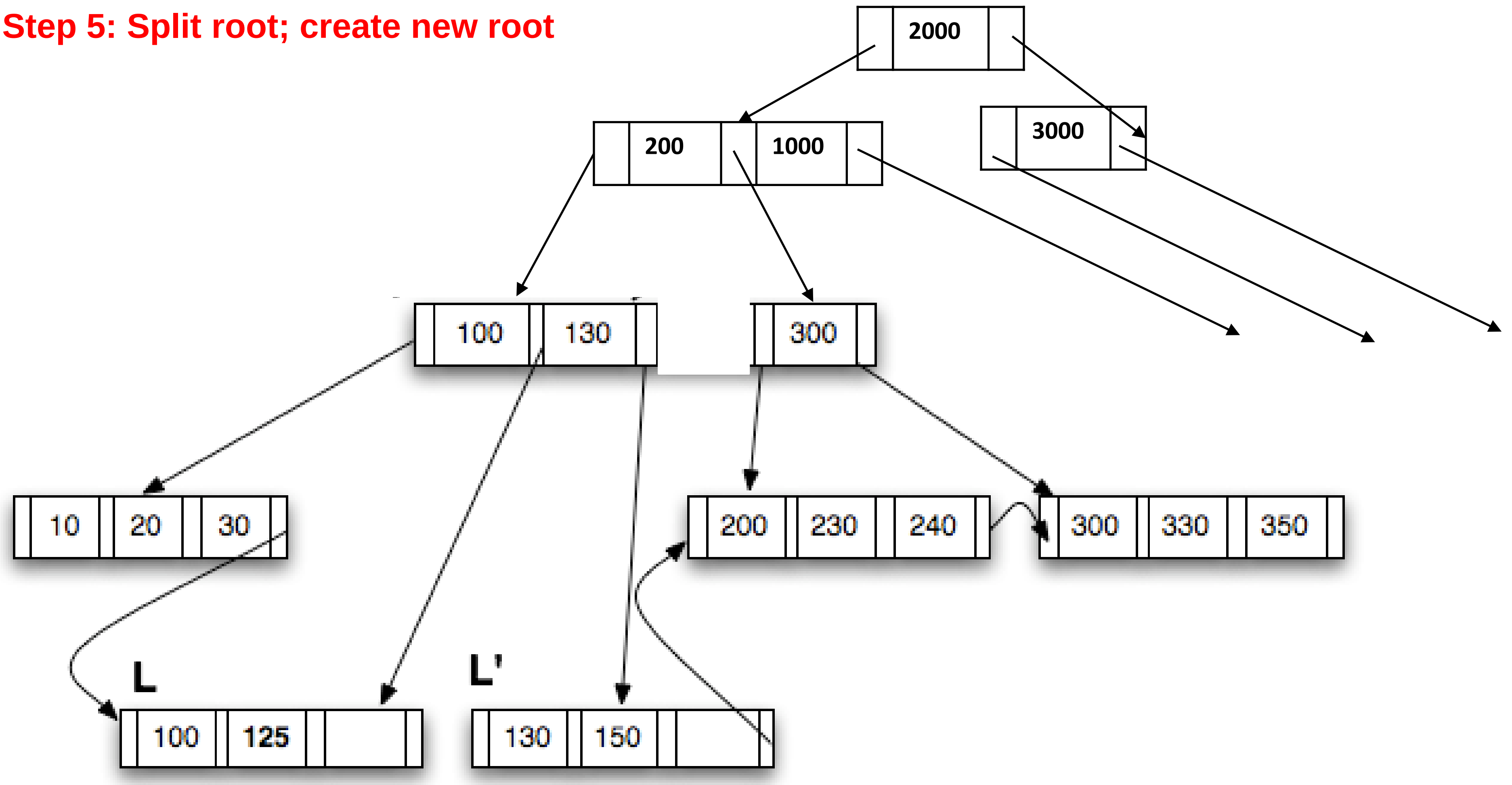
Another Example: INSERT (125)

Step 4: Insert 200 into parent

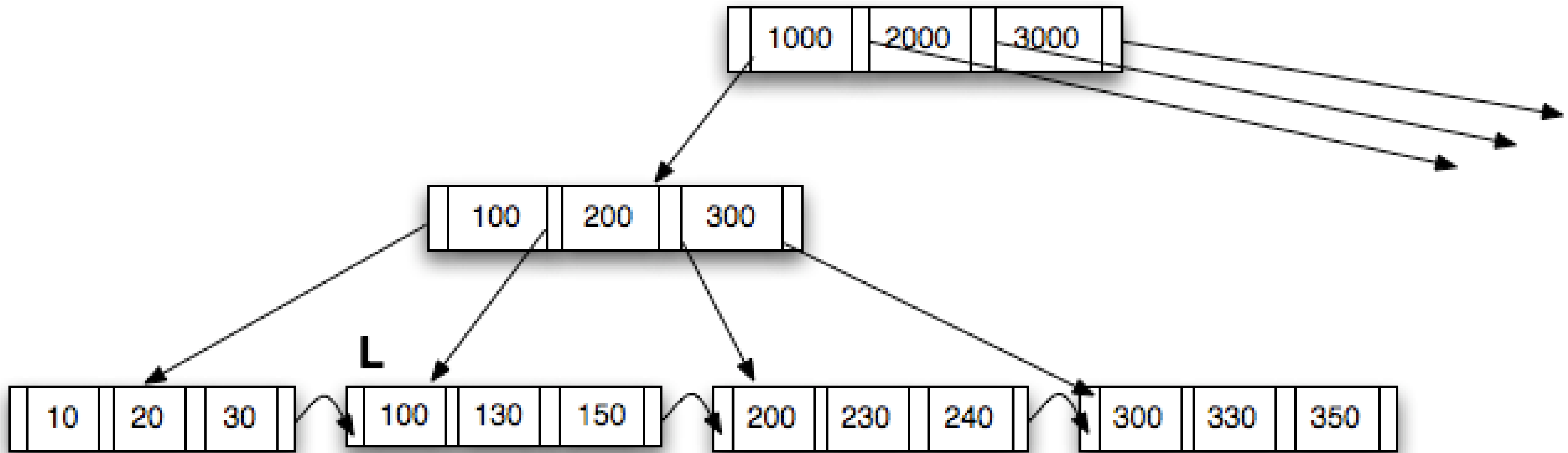


Another Example: INSERT (125)

Step 5: Split root; create new root



What keys does parent of the leaf node with 350 contain after insert of 340 into:



A. 100,200,300

B. 300,350

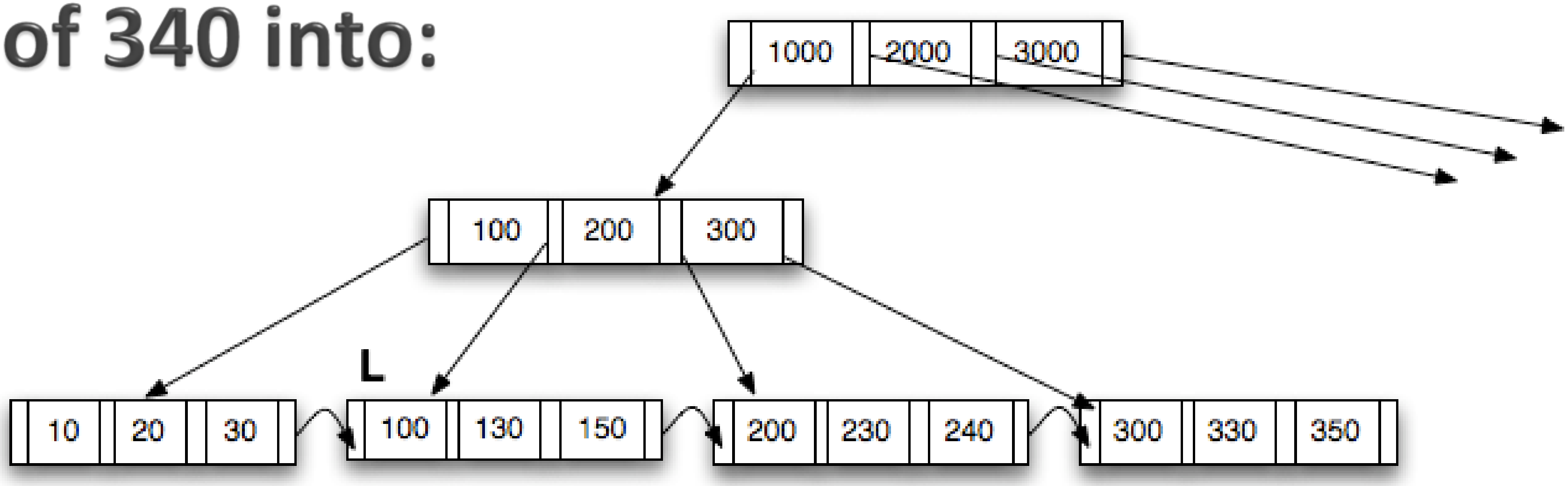
C. 300,340

D. 300

E. 340

F. 350

What keys does root contain after insert of 340 into:



A. 1000,2000,3000

B. 1000,2000

C. 340

D. 350

E. 1000

F. 2000

G. 3000

B^+-Trees: Deletion

- Find the record, delete it.
- *Maybe* remove the (search-key, pointer) pair from leaf node
  - Note that there might be another tuple with the same search-key
  - In that case, this is not needed (if primary index)
- Issue:
  - The leaf node now may contain too few entries
    - Why do we care ?
  - Solution:
    1. Merge, if possible

When is it impossible to merge with a sibling?

A. When the sibling to the left is full
B. When the sibling to the right is full
C. When there is no sibling to the left
D. When there is no sibling to the right
E. When both immediate siblings are full
F. When both immediate siblings have fewer than (ceiling(n/2) -1) free pointers

B^+-Trees: Deletion

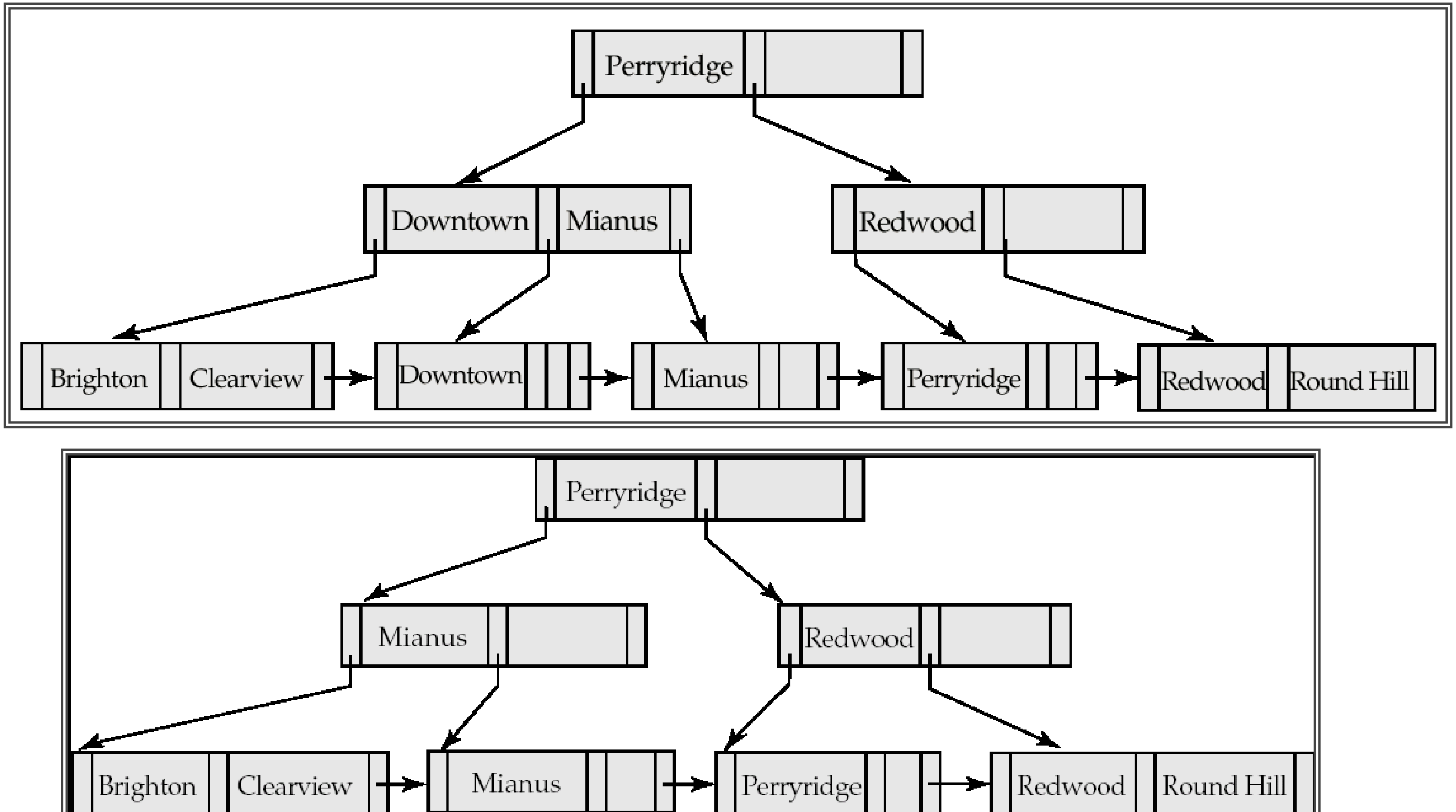
- Find the record, delete it.
- *Maybe* remove the (search-key, pointer) pair from leaf node
  - Note that there might be another tuple with the same search-key
  - In that case, this is not needed (if primary index)
- Issue:
  - The leaf node now may contain too few entries
    - Why do we care ?
  - Solution:
    1. Merge, if possible
    2. Borrow from adjacent sibling, otherwise
  - May end up merging all the way to the root
  - In fact, may reduce the height of the tree by one

Approach to B$^+$-tree Deletion

```
procedure delete_entry(node N, value K, pointer P)
    delete (K, P) from N
    if (N is the root and N has only one remaining child)
    then make the child of N the new root of the tree and delete N
    else if (N has too few values/pointers) then begin
        Let N′ be the previous or next child of parent(N)
        Let K′ be the value between pointers N and N′ in parent(N)
        if (entries in N and N′ can fit in a single node)
            then begin /* Coalesce nodes */
                if (N is a predecessor of N′) then swap_variables(N, N′)
                if (N is not a leaf)
                    then append K′ and all pointers and values in N to N′
                    else  append all (K_i, P_i) pairs in N to N′; set N′.P_n = N.P_n
                delete_entry(parent(N), K′, N); delete node N
            end
        else begin /* Redistribution: borrow an entry from N′ */
            if (N′ is a predecessor of N) then begin
                if (N is a nonleaf node) then begin
                    let m be such that N′.P_m is the last pointer in N′
                    remove (N′.K_{m−1}, N′.P_m) from N′
                    insert (N′.P_m, K′) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K′ in parent(N) by N′.K_{m−1}
                end
                else begin
                    let m be such that (N′.P_m, N′.K_m) is the last pointer/value
                        pair in N′
                    remove (N′.P_m, N′.K_m) from N′
                    insert (N′.P_m, N′.K_m) as the first pointer and value in N,
                        by shifting other pointers and values right
                    replace K′ in parent(N) by N′.K_m
                end
            end
            else  ... symmetric to the then case ...
        end
    end
```

11.19 in book

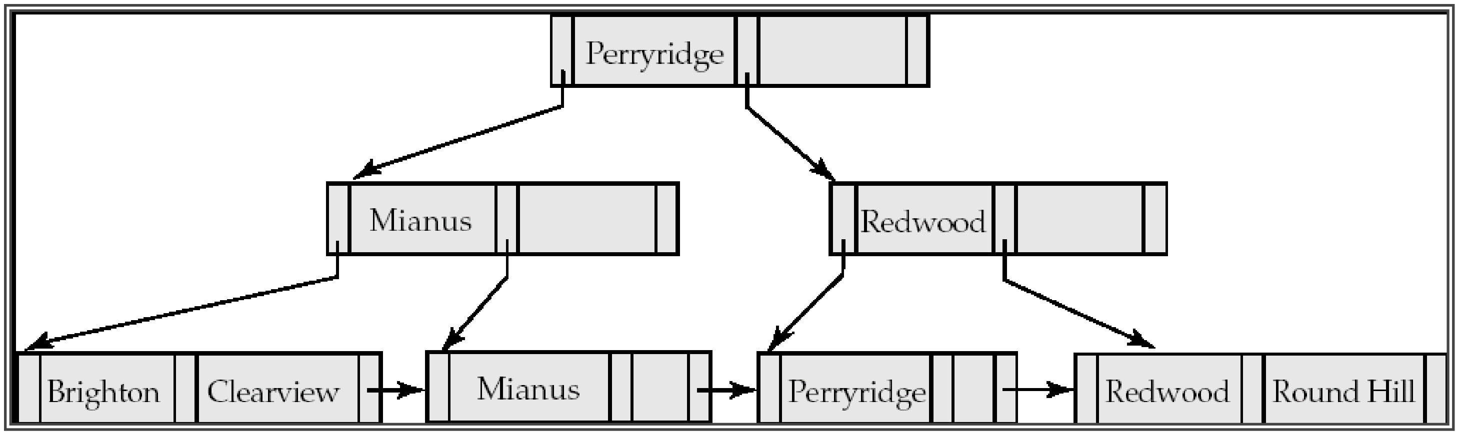
Examples of B+-Tree Deletion



Before and after deleting “Downtown”

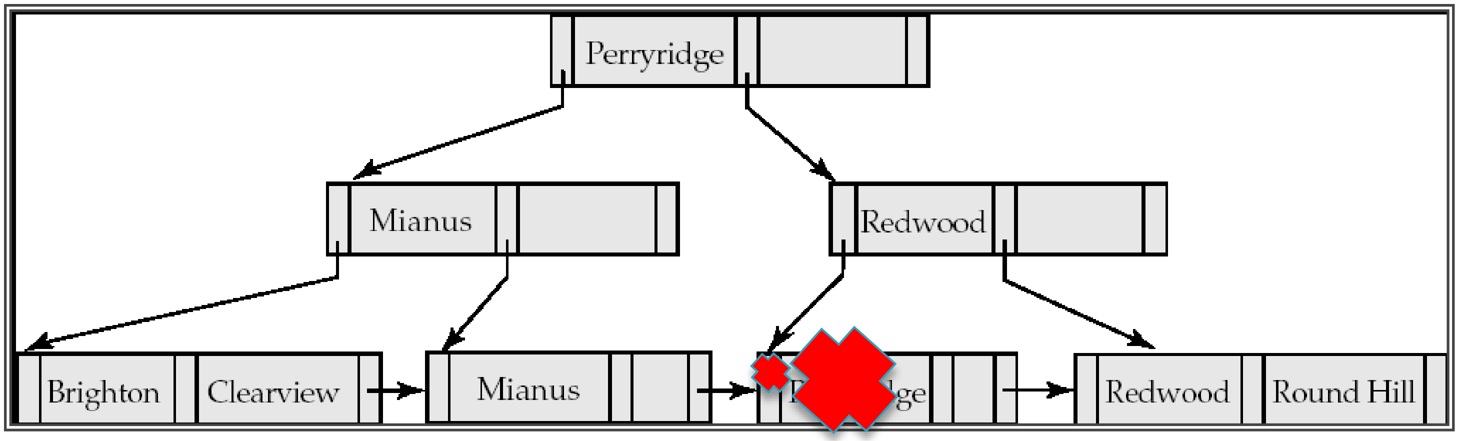
- Deleting “Downtown” causes merging of under-full leaves

Examples of B+-Tree Deletion



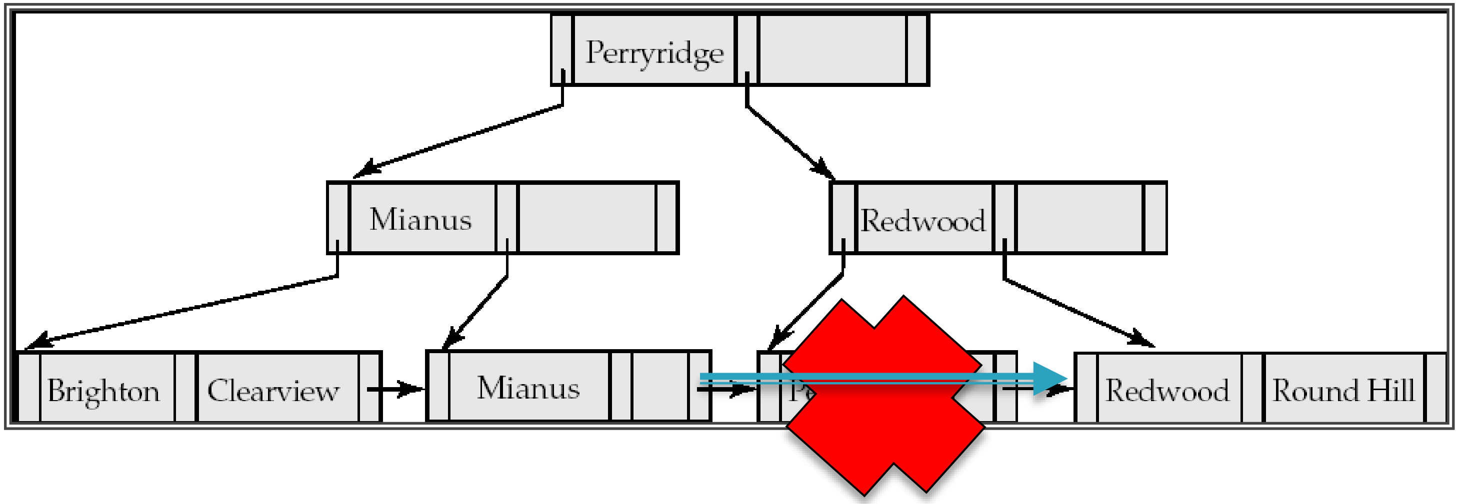
Deletion of “Perryridge”

Examples of B+-Tree Deletion



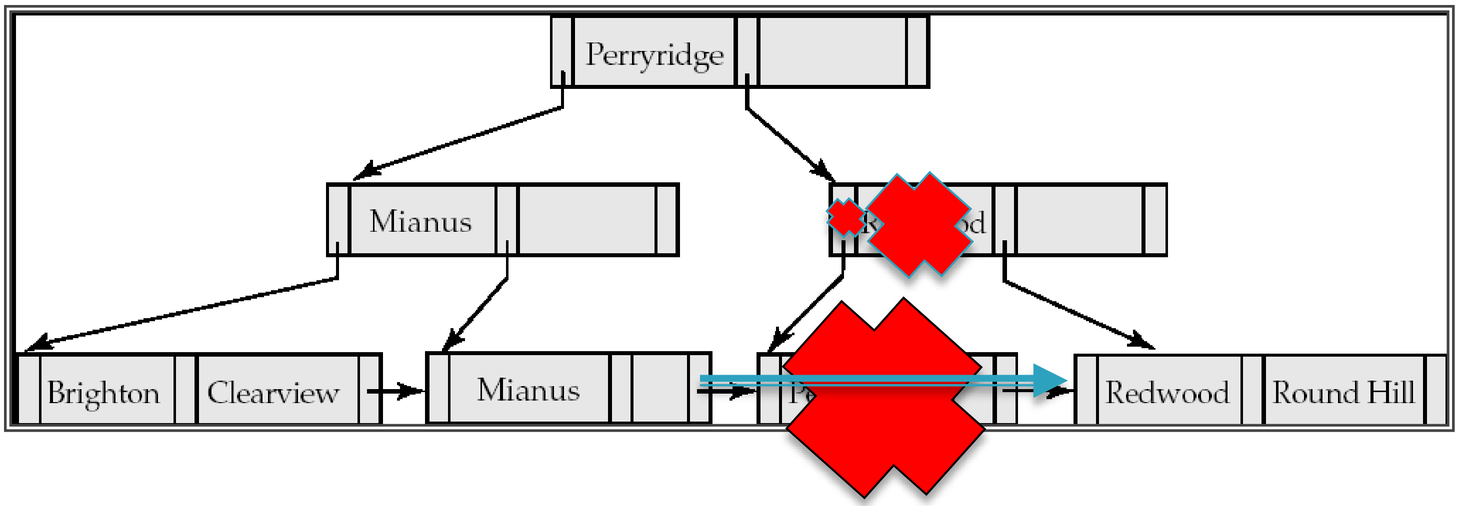
Deletion of “Perryridge”

Examples of B+-Tree Deletion



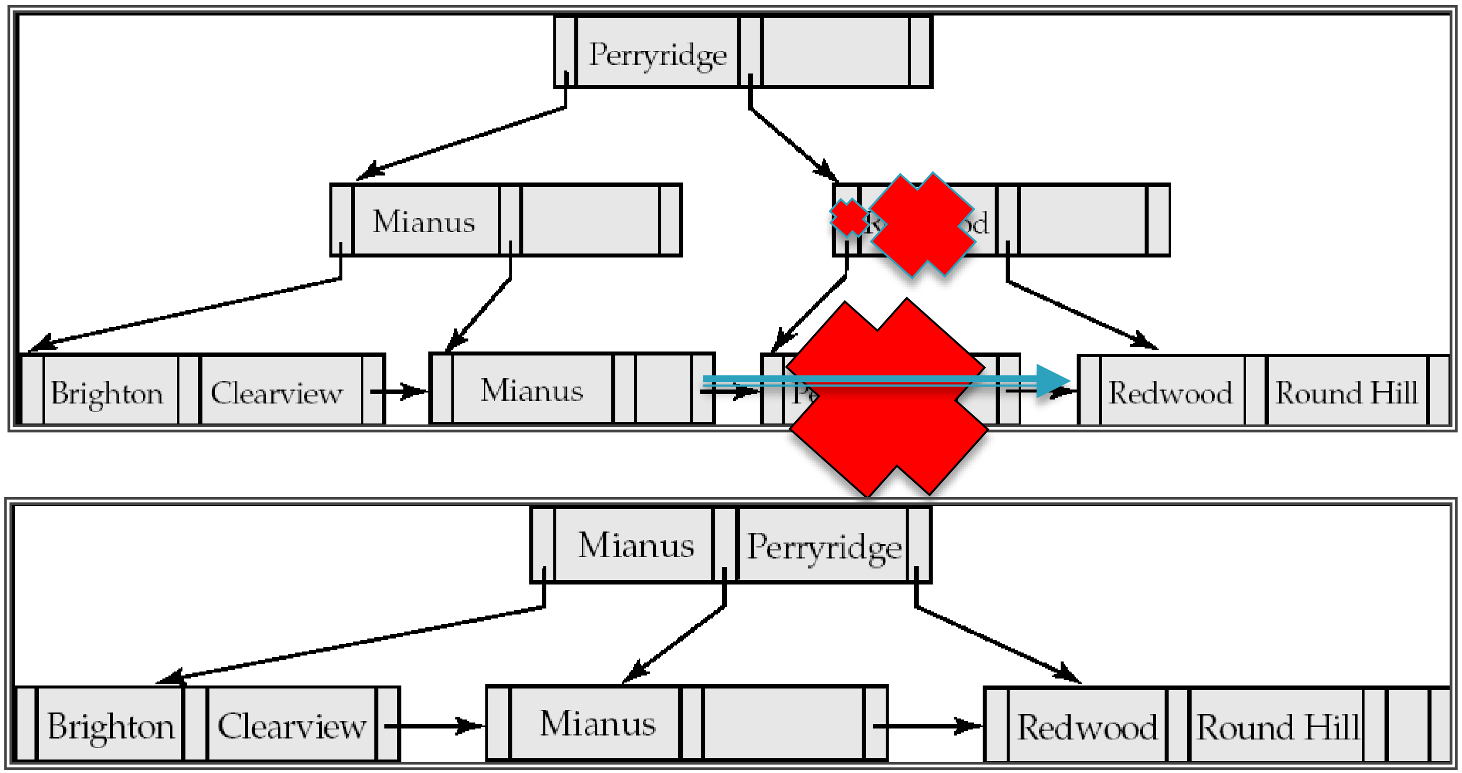
Deletion of “Perryridge”

Examples of B+-Tree Deletion



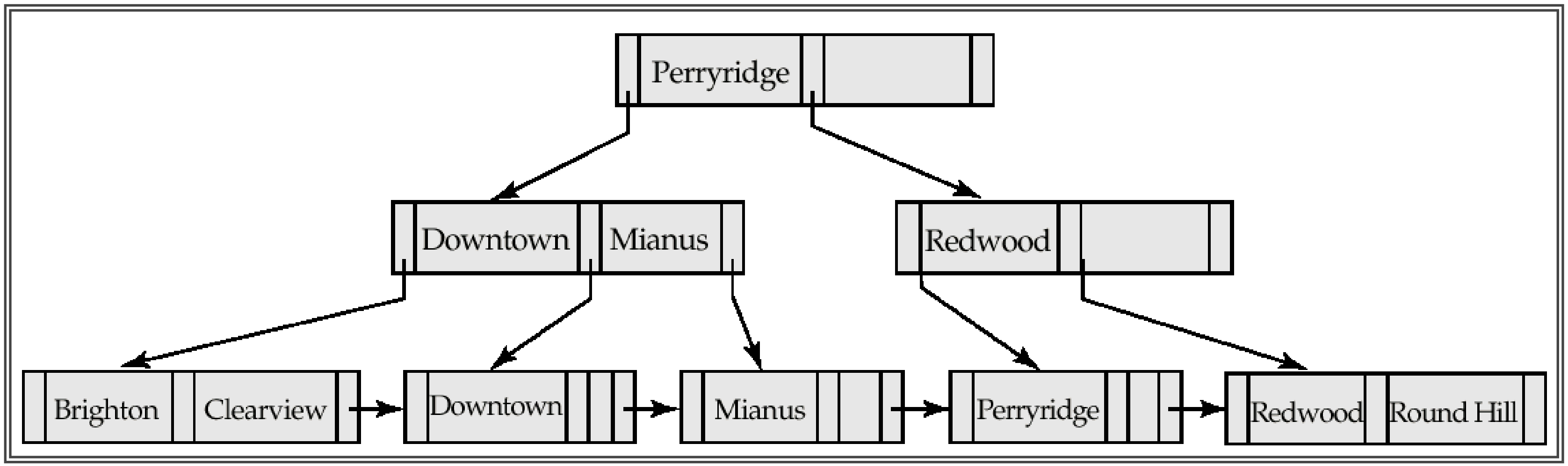
Deletion of “Perryridge”

Examples of B$^+$-Tree Deletion



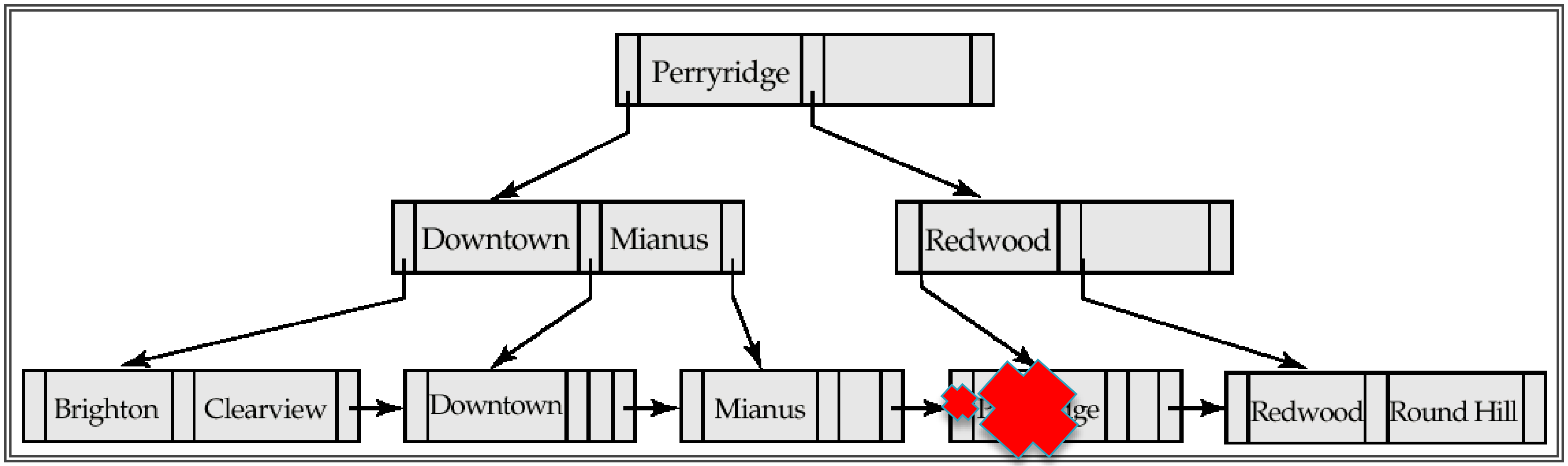
Deletion of “Perryridge”

Examples of B^+-Tree Deletion



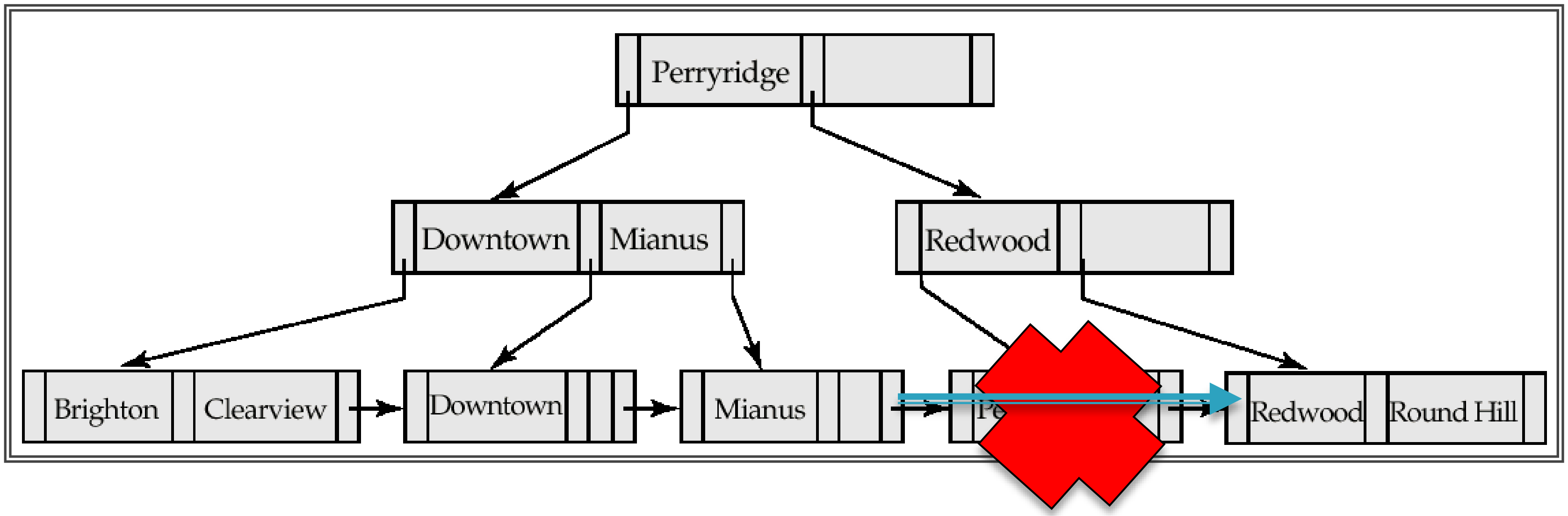
- What if we delete Perryridge from this original index instead?

Examples of B$^+$-Tree Deletion



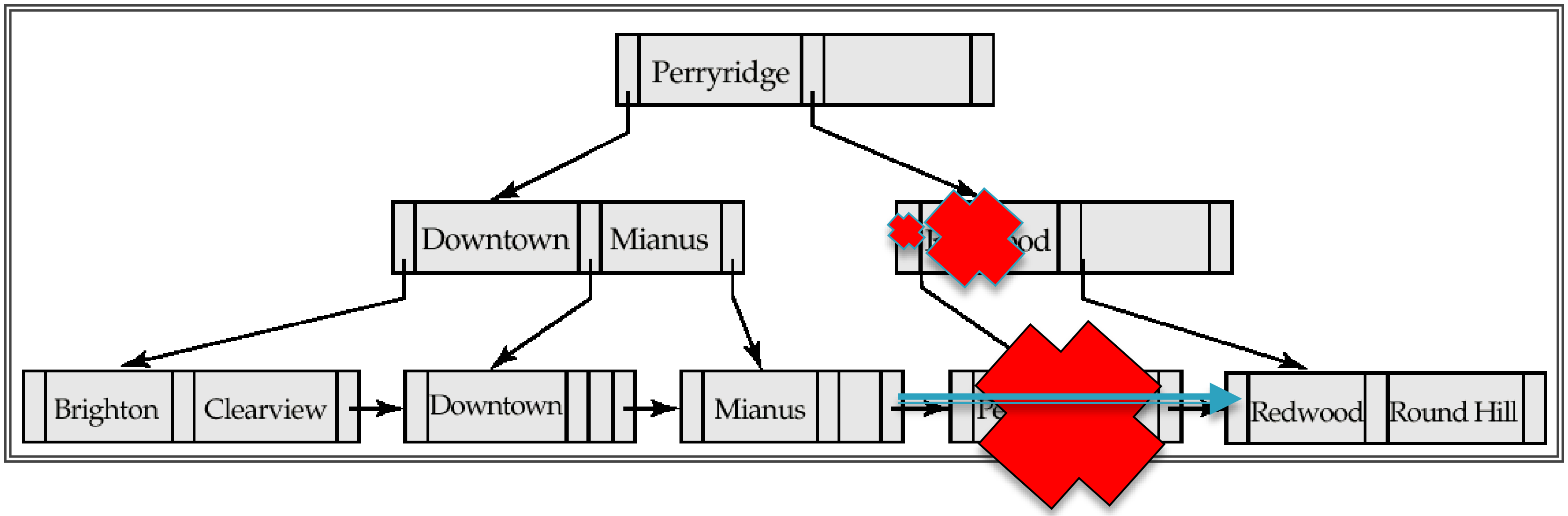
- What if we delete Perryridge from this original index instead?

Examples of B$^+$-Tree Deletion



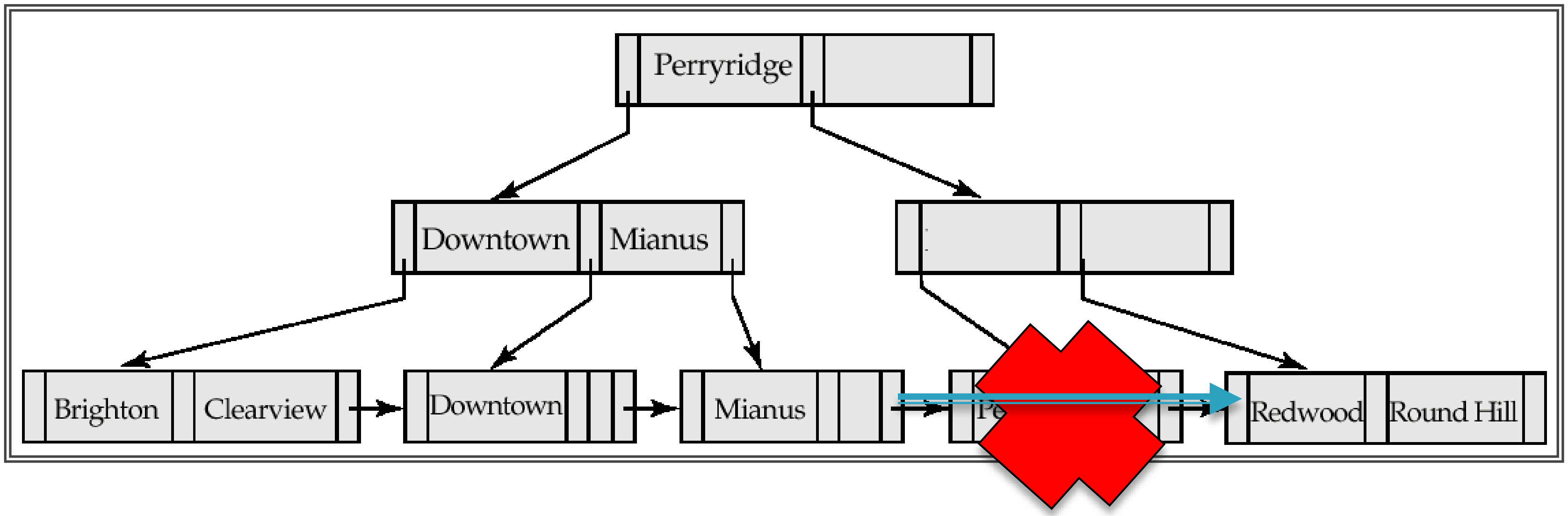
- What if we delete Perryridge from this original index instead?

Examples of B+-Tree Deletion



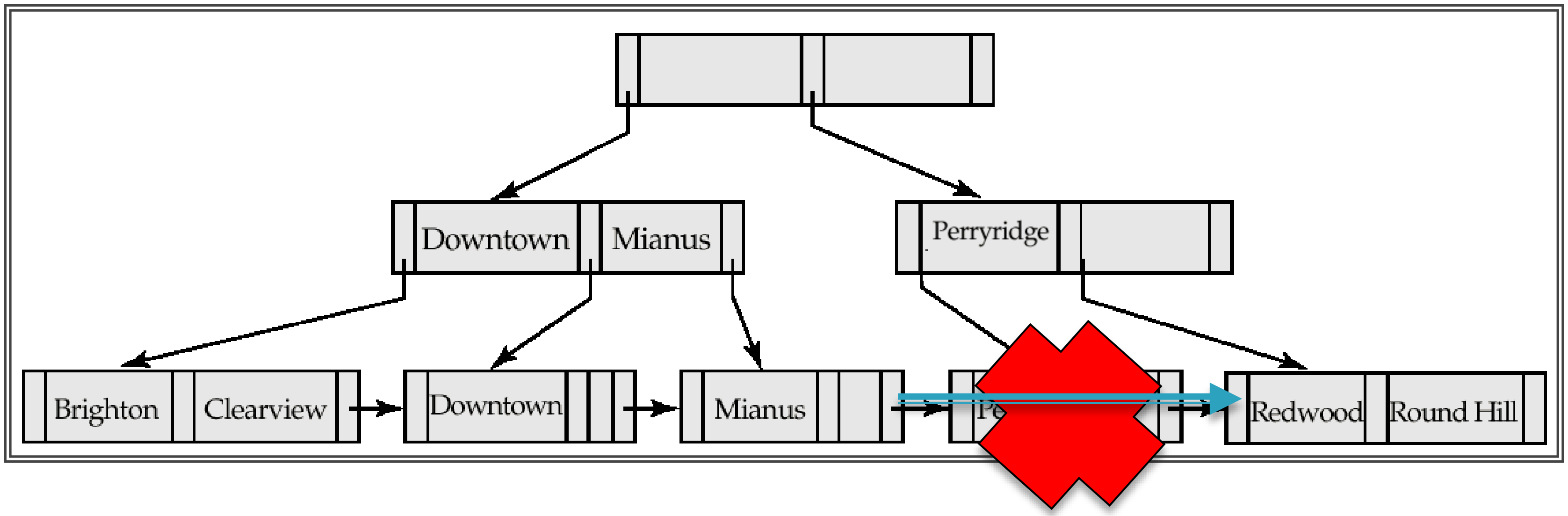
- What if we delete Perryridge from this original index instead?

Examples of B$^+$-Tree Deletion



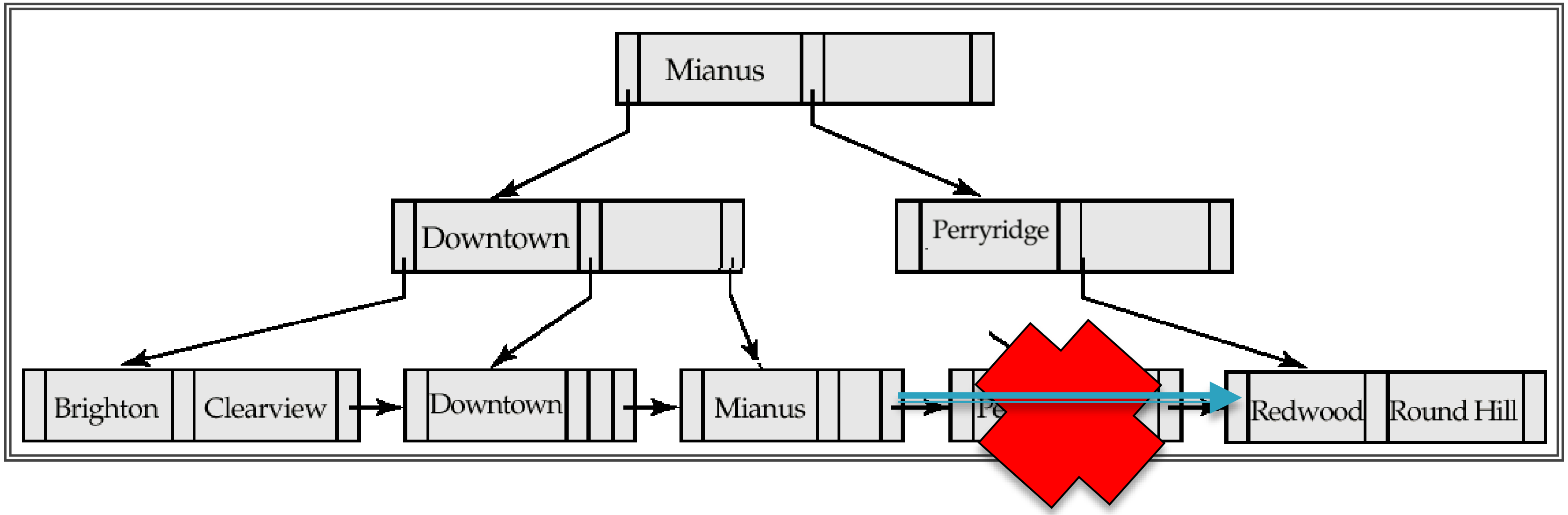
- What if we delete Perryridge from this original index instead?

Examples of B+-Tree Deletion



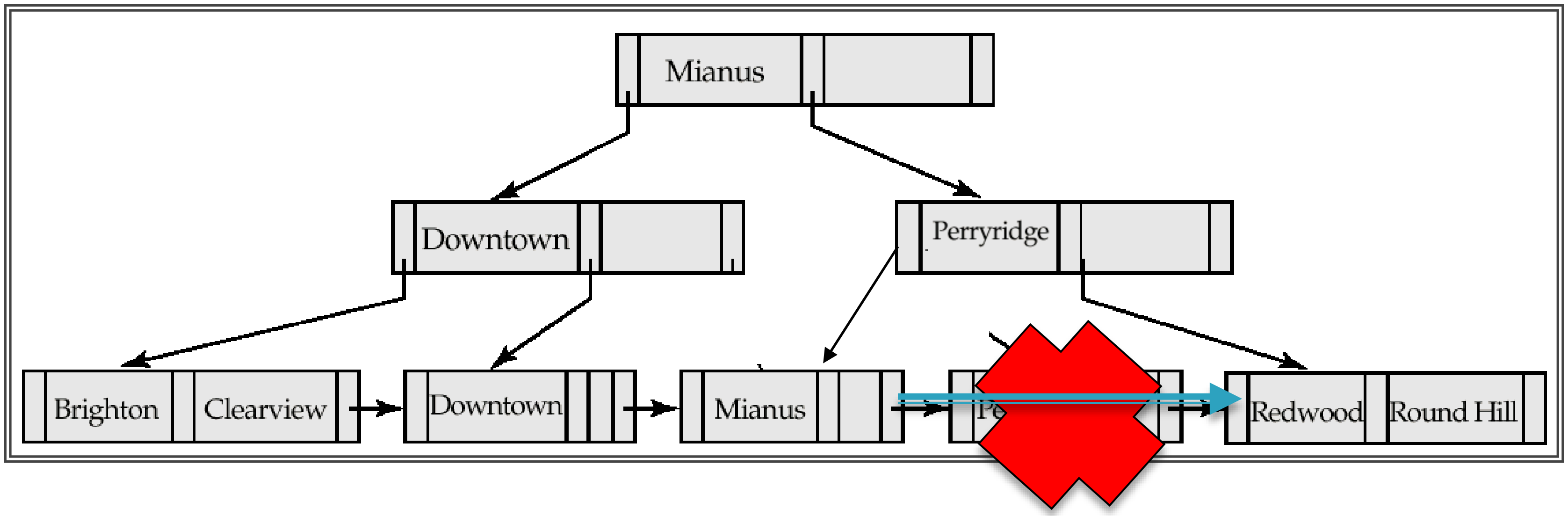
- What if we delete Perryridge from this original index instead?

Examples of B$^+$-Tree Deletion



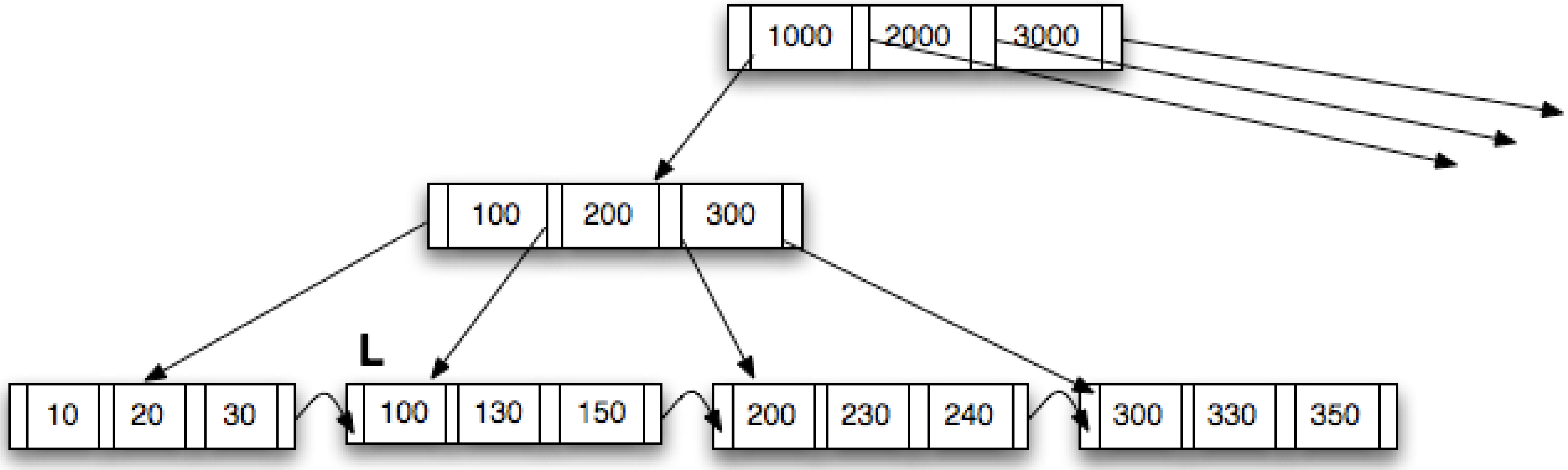
- What if we delete Perryridge from this original index instead?

Examples of B⁺-Tree Deletion



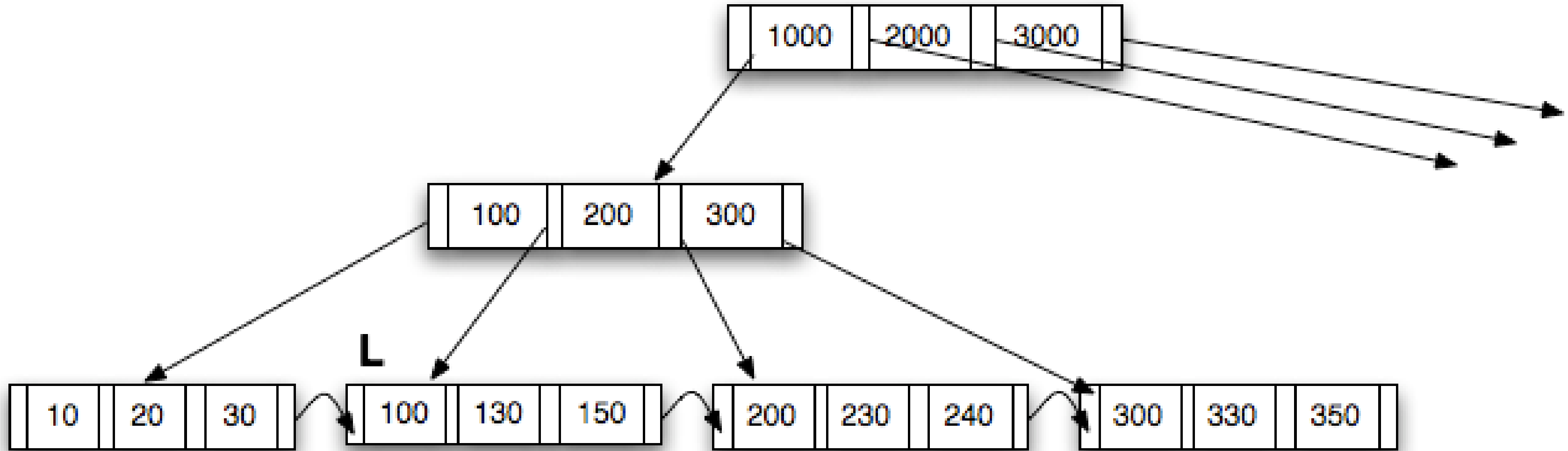
- What if we delete Perryridge from this original index instead?

What keys does parent of the leaf node with 350 contain after delete of 300 from:



A. 100,200,300
B. 100,200,330
C. 100,200
D. 300
E. 330
F. 200,330

What keys does parent of the leaf node with 350 contain after delete of 300 and 330 from:



A. 100,200,300
B. 100,200,330
C. 100,200,240
D. 240
E. 300
F. 350
G. 240,300

B+ Trees in Practice

- Typical order: 200. 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