CMSC424: Database Design

Instructor: Amol Deshpande

amol@cs.umd.edu

Spring 2020 – Online Instruction Plan

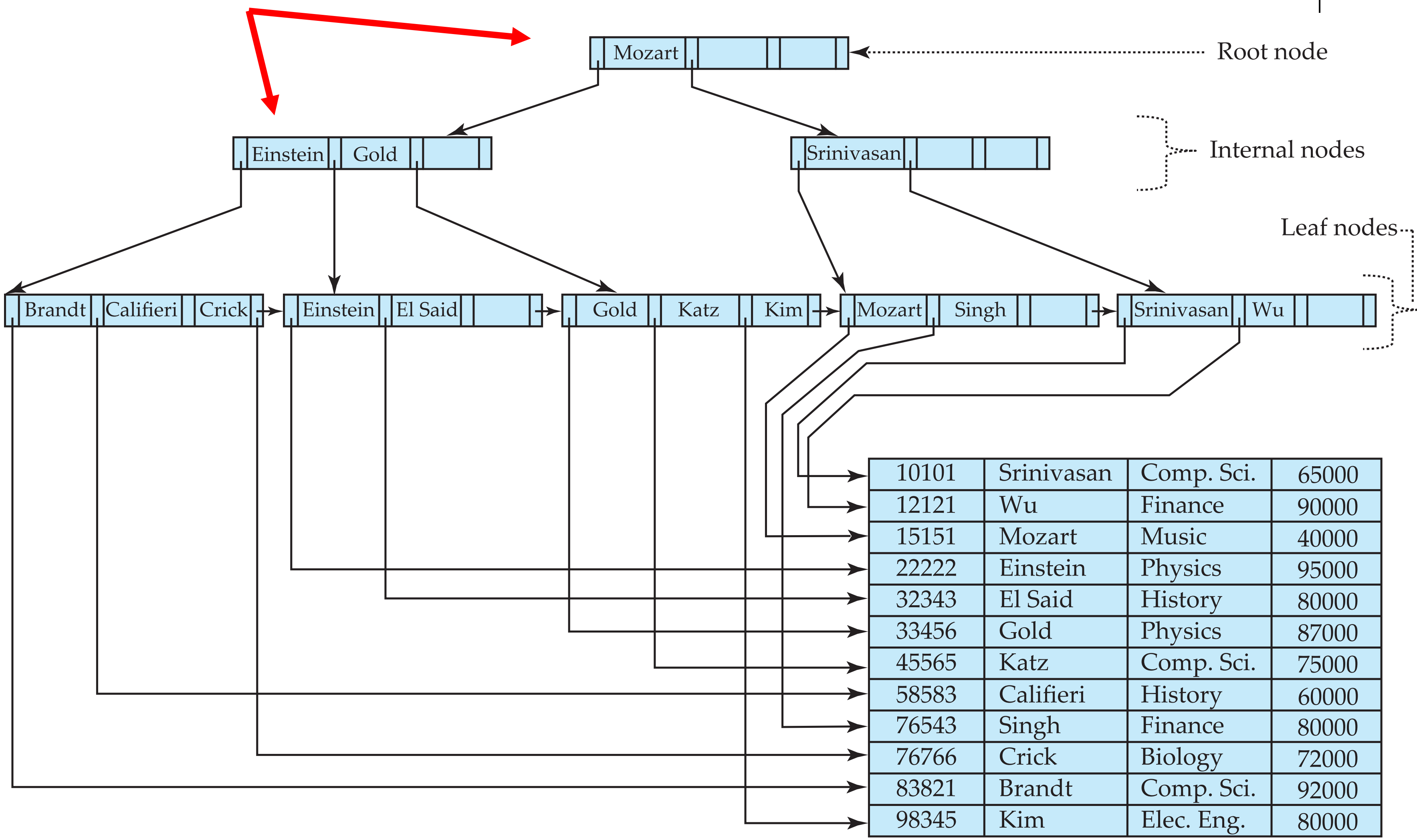
- Week 1 (March 30 – April 2):
  - File Organization and Overview of Indexes
  - B+-Trees
  - Hashing
  - Miscellaneous topics in Indexes
- Week 2: Query Processing
- Week 3: Transactions 1
- Week 4: Transactions 2
- Week 5: Parallel Database and MapReduce

B+-Trees

- Book Chapters
  - 11.3
- Key topics:
  - B+-Trees as a multi-level index, and basic properties
  - How to search in a B+-Tree?
  - How to update B+-Tree when a new tuple in inserted in the relation?
    - Key challenge: keeping the index “balanced” and all the pages “sufficiently full”
  - How to handle a delete from the underlying relation?
    - Same key challenge

Example B+-Tree Index

Index Disk Blocks



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 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)
  - *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	$\ldots$	P_{n-1}	K_{n-1}	P_n

Properties

- Interior nodes

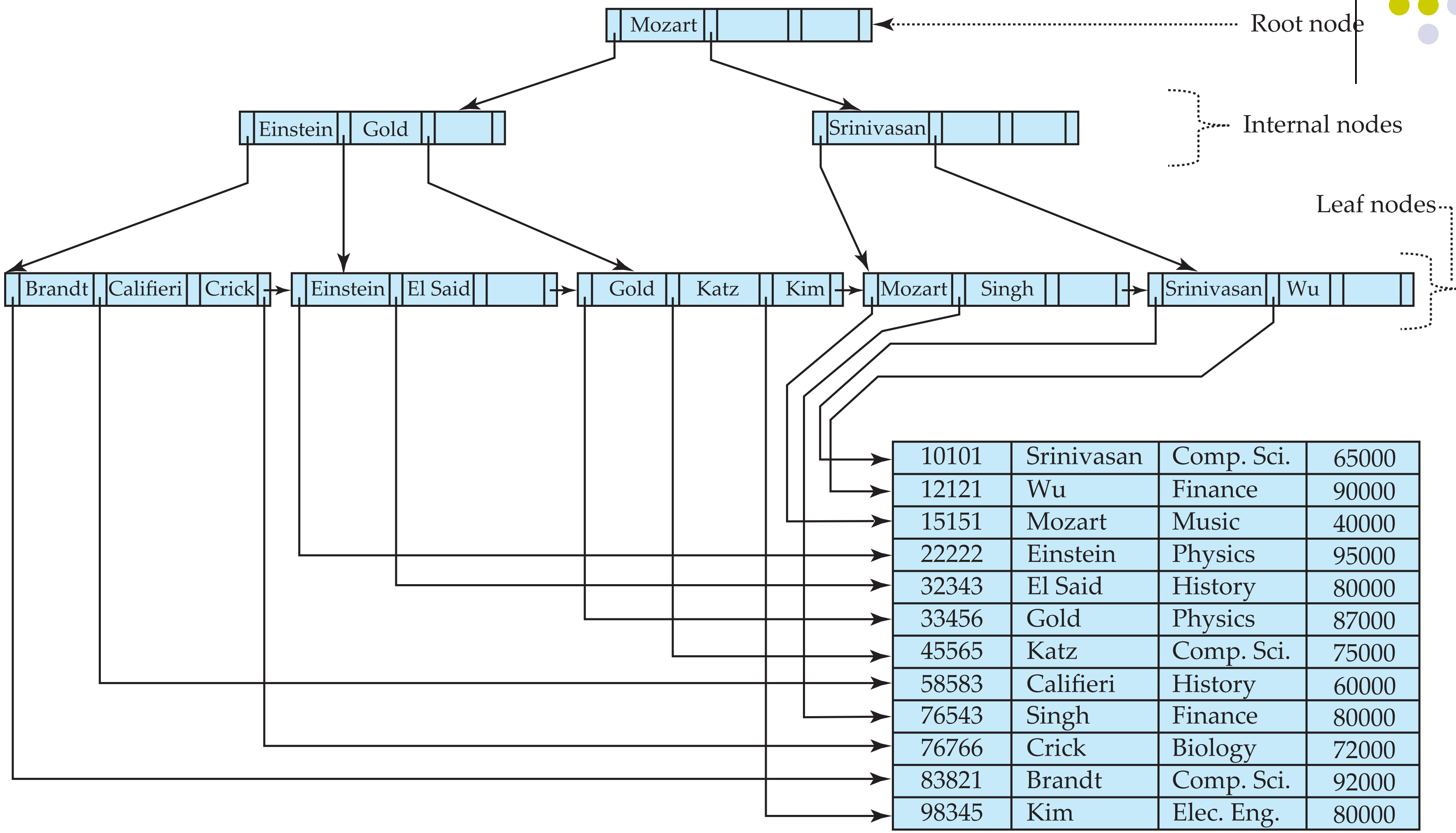
P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

- 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 *Pn,* are all larger than *Kn-1*
- Must contain at least n/2 entries (unless root)

B+-Trees - Searching

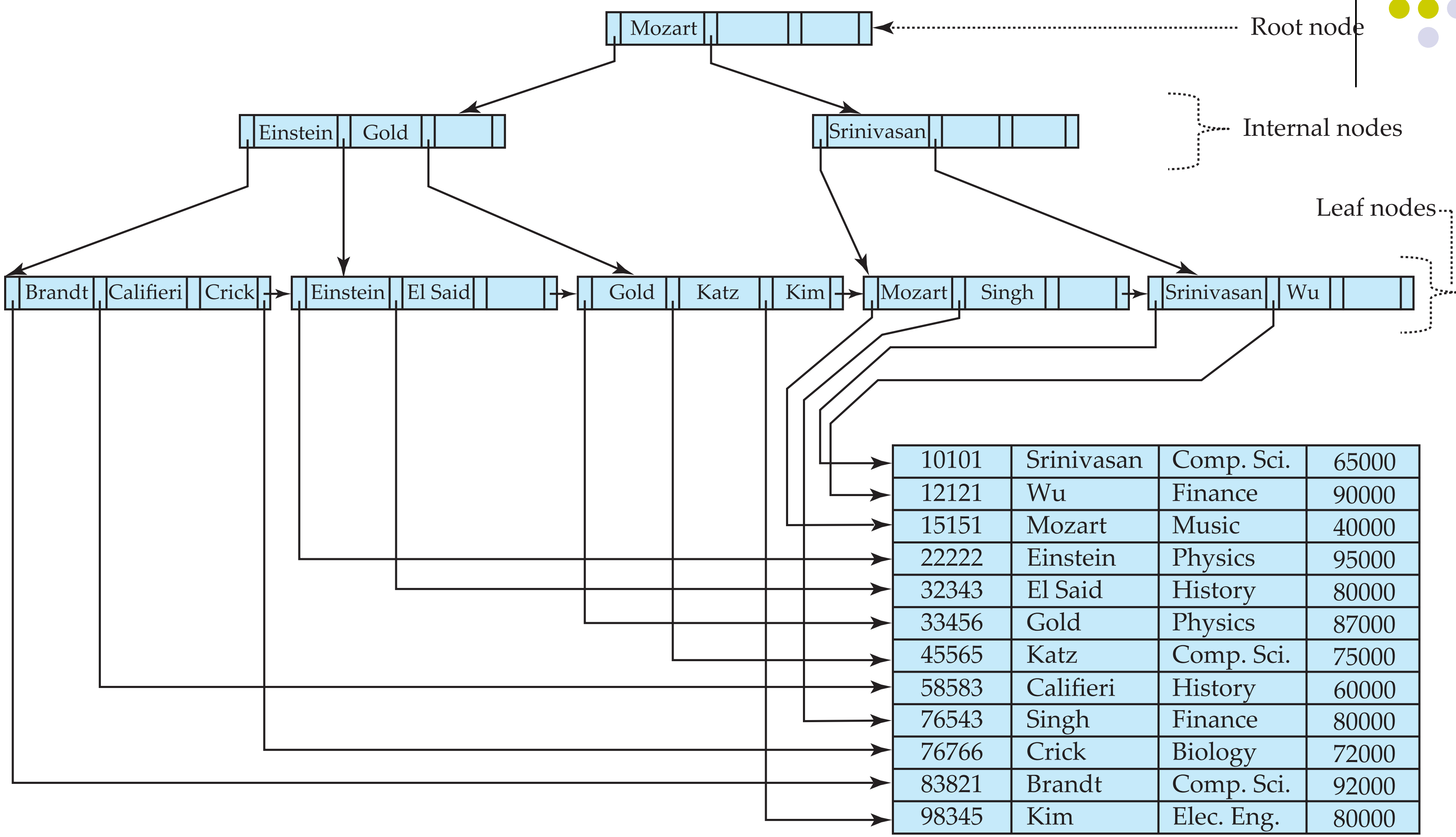
- How to search ?
  - Follow the pointers
- Logarithmic
  - $log_{B/2}(N)$, where *B = Number of entries per block*
  - *B* is also called the order of the B+-Tree Index
    - Typically 100 or so
- If a relation contains1,000,000,000 entries, takes only 4 random accesses
- The top levels are typically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

Example B+-Tree Index



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

Example B+-Tree Index



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

If this were a “primary” index, then not all ”keys” are present in the index

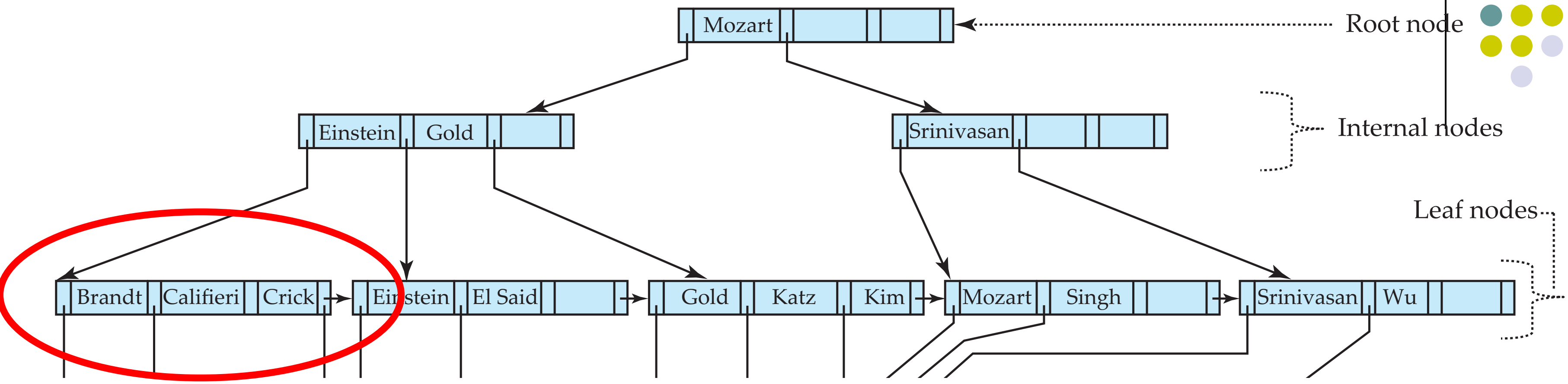
Tuple Insertion

- 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



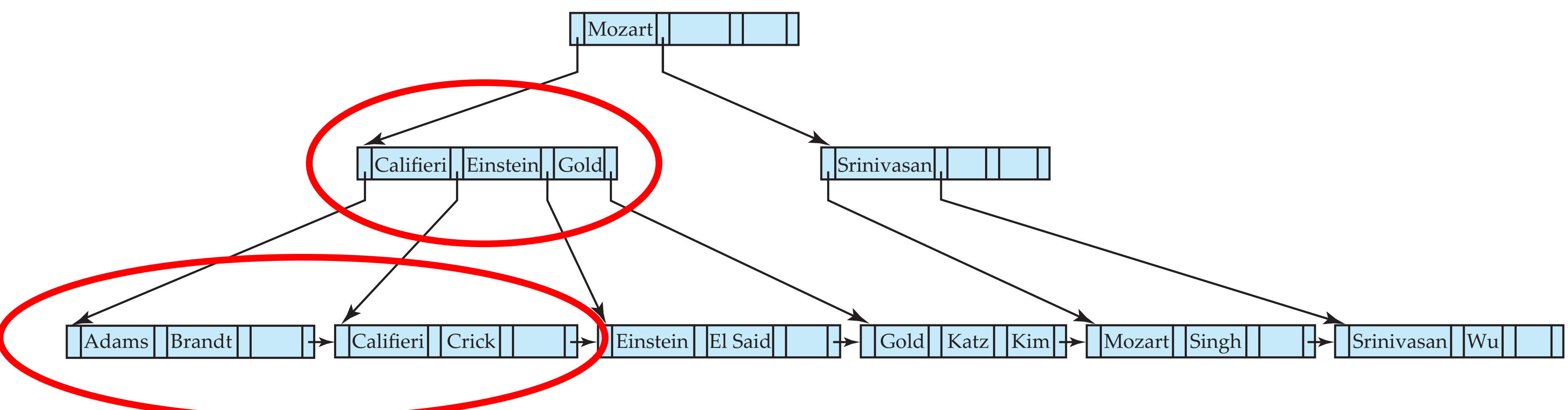


Figure 11.13 Insertion of "Adams" into the B+-tree of Figure 11.9.

B⁺-Trees: Insertion

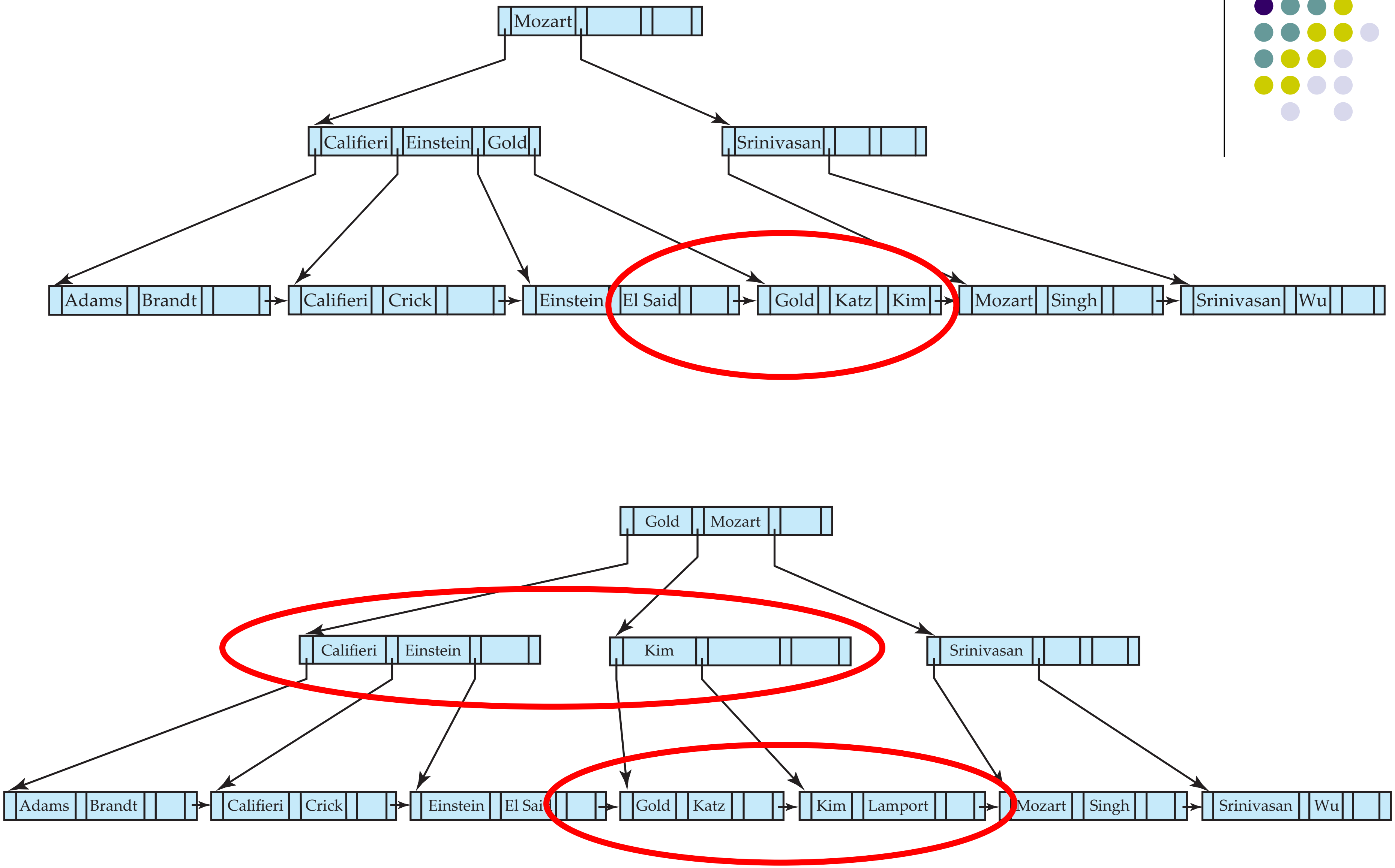


Figure 11.14 Insertion of "Lamport" into the B^+-tree of Figure 11.13.

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

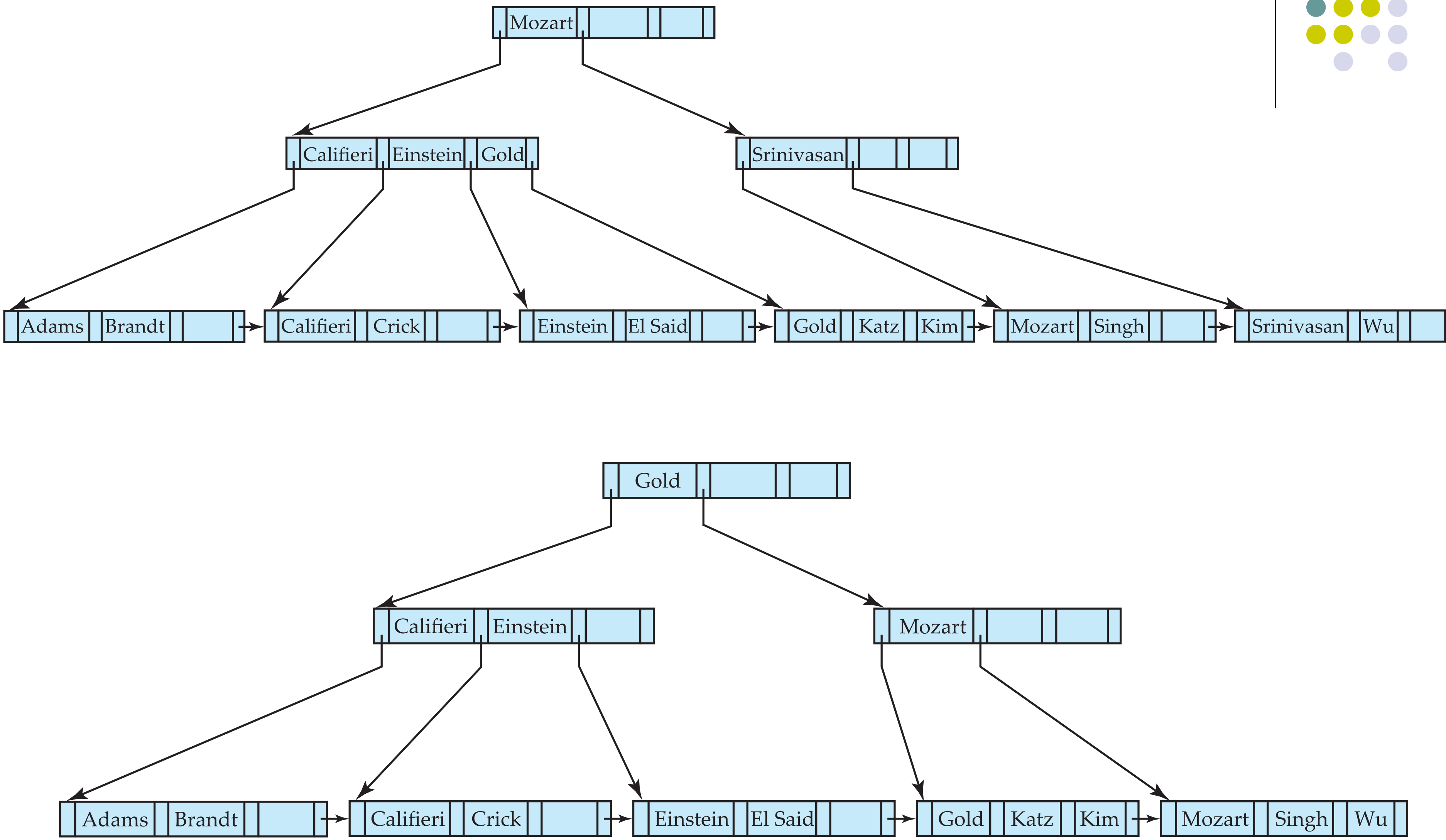
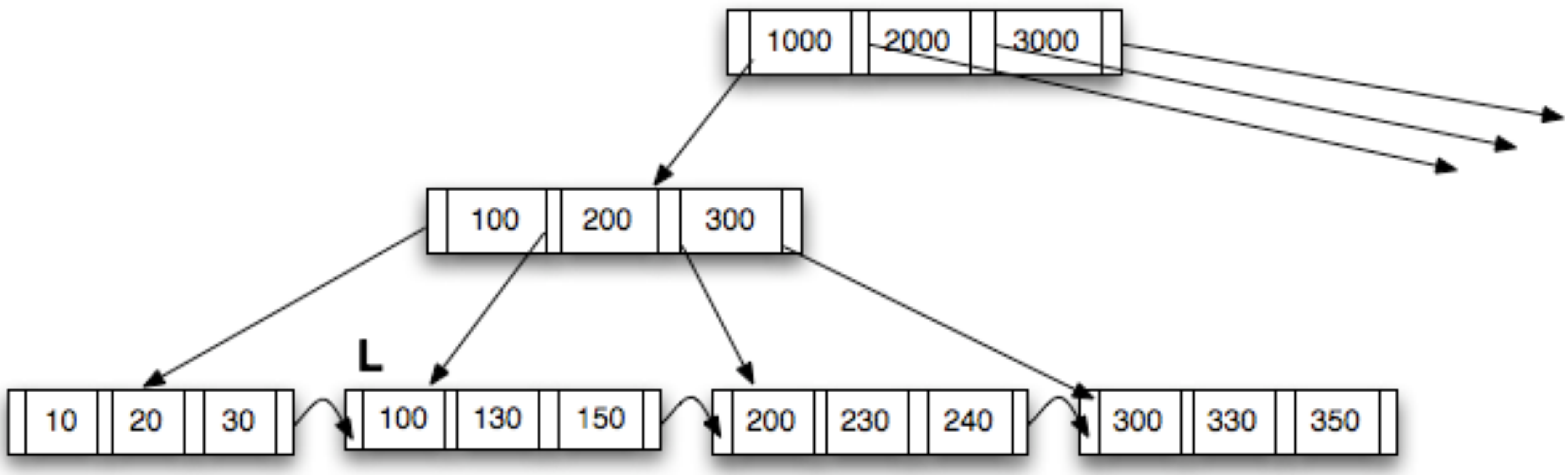


Figure 11.16 Deletion of "Srinivasan" from the B$^+$-tree of Figure 11.13.

Another B+Tree Insertion Example

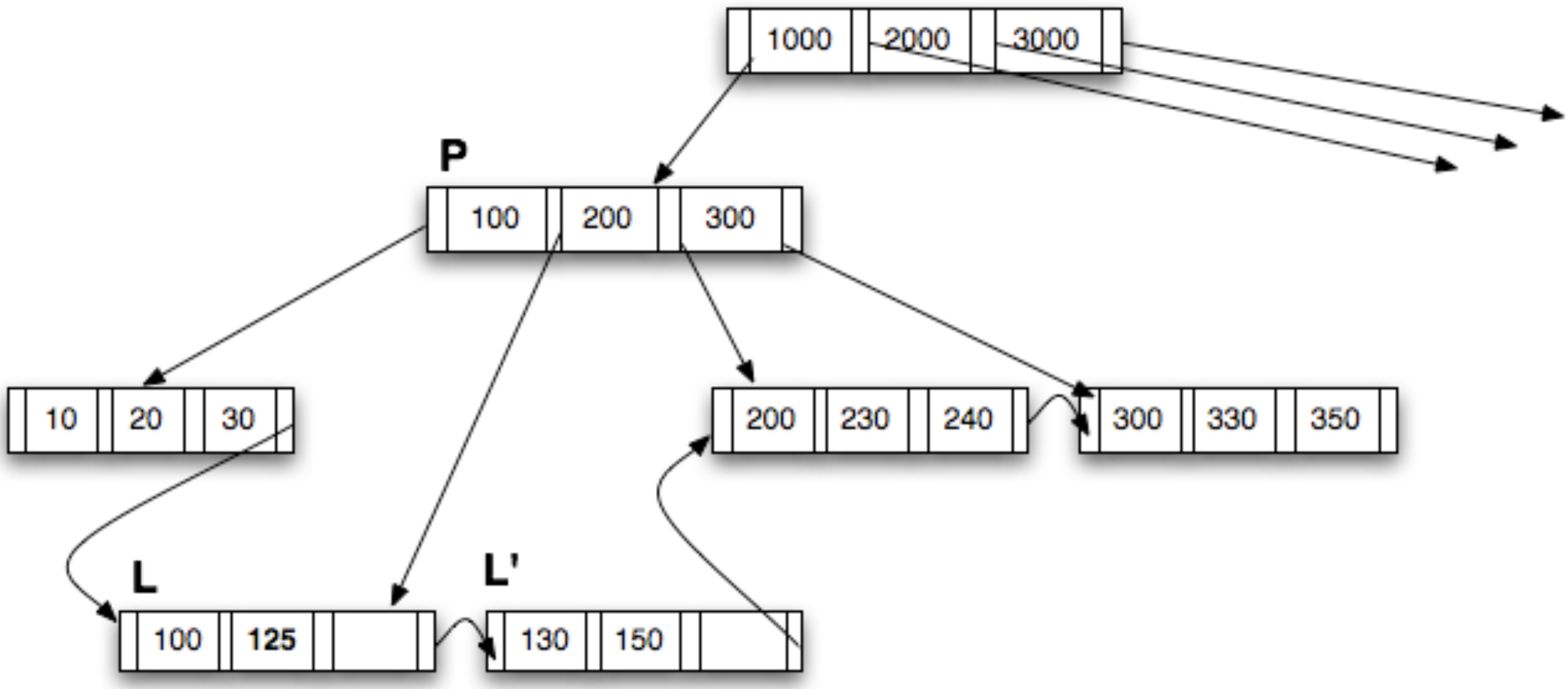
INITIAL TREE



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

Another Example: INSERT (125)

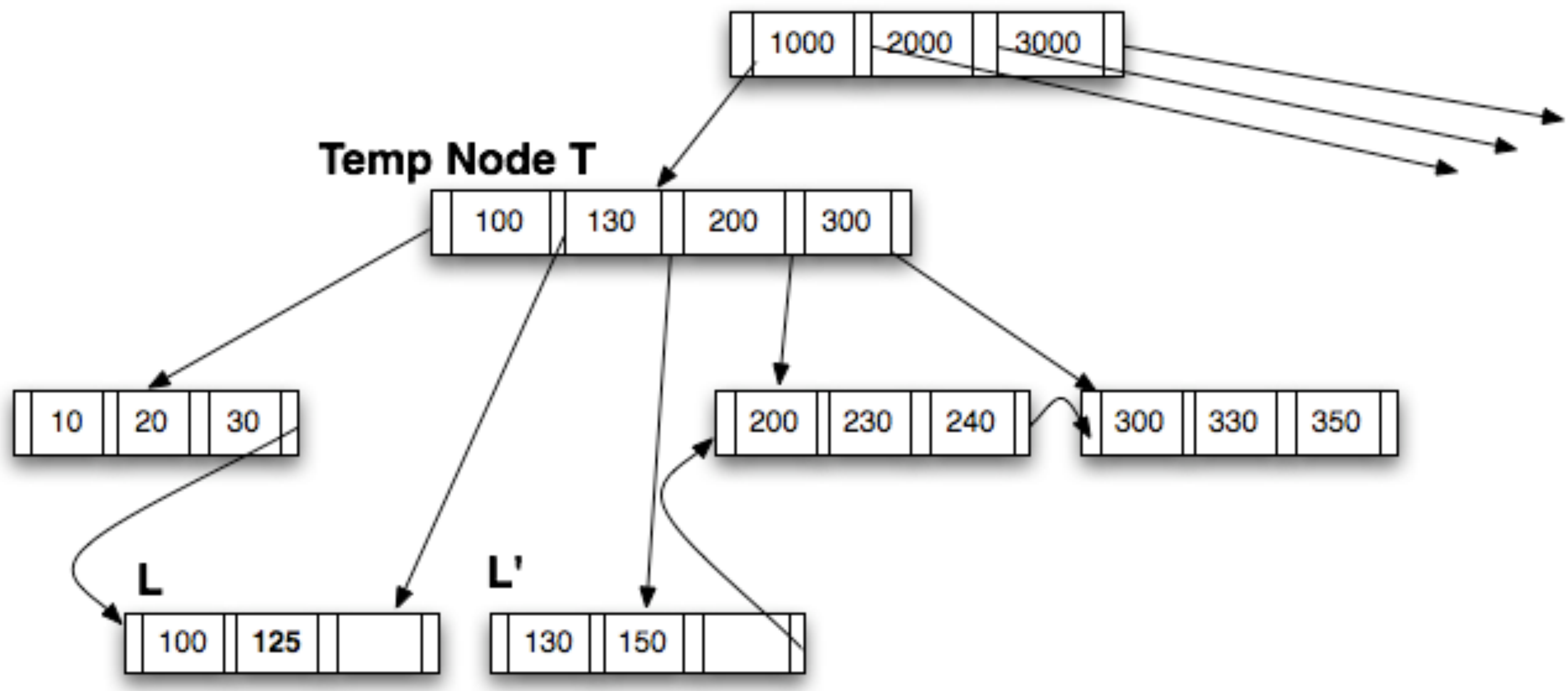
Step 1: Split L to create L’



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

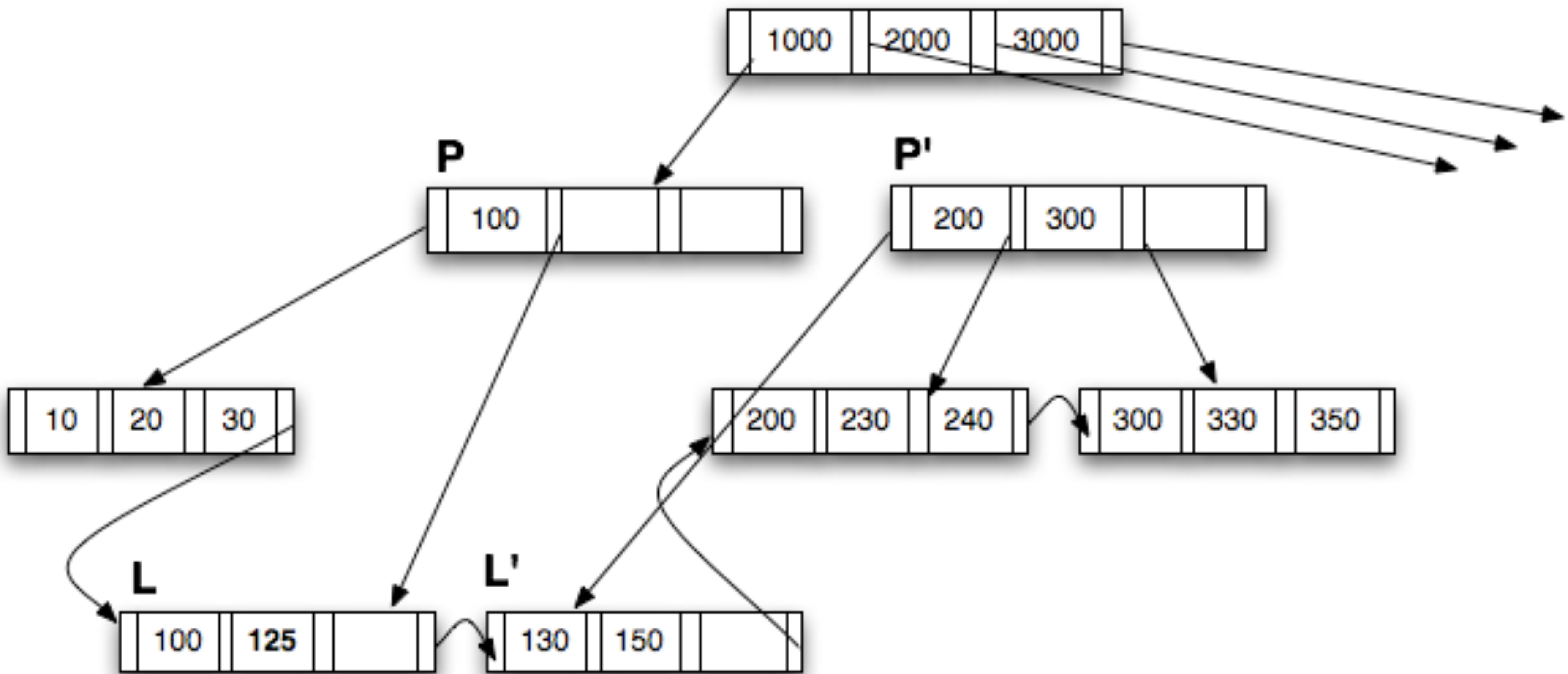
Another Example: INSERT (125)

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



Another Example: INSERT (125)

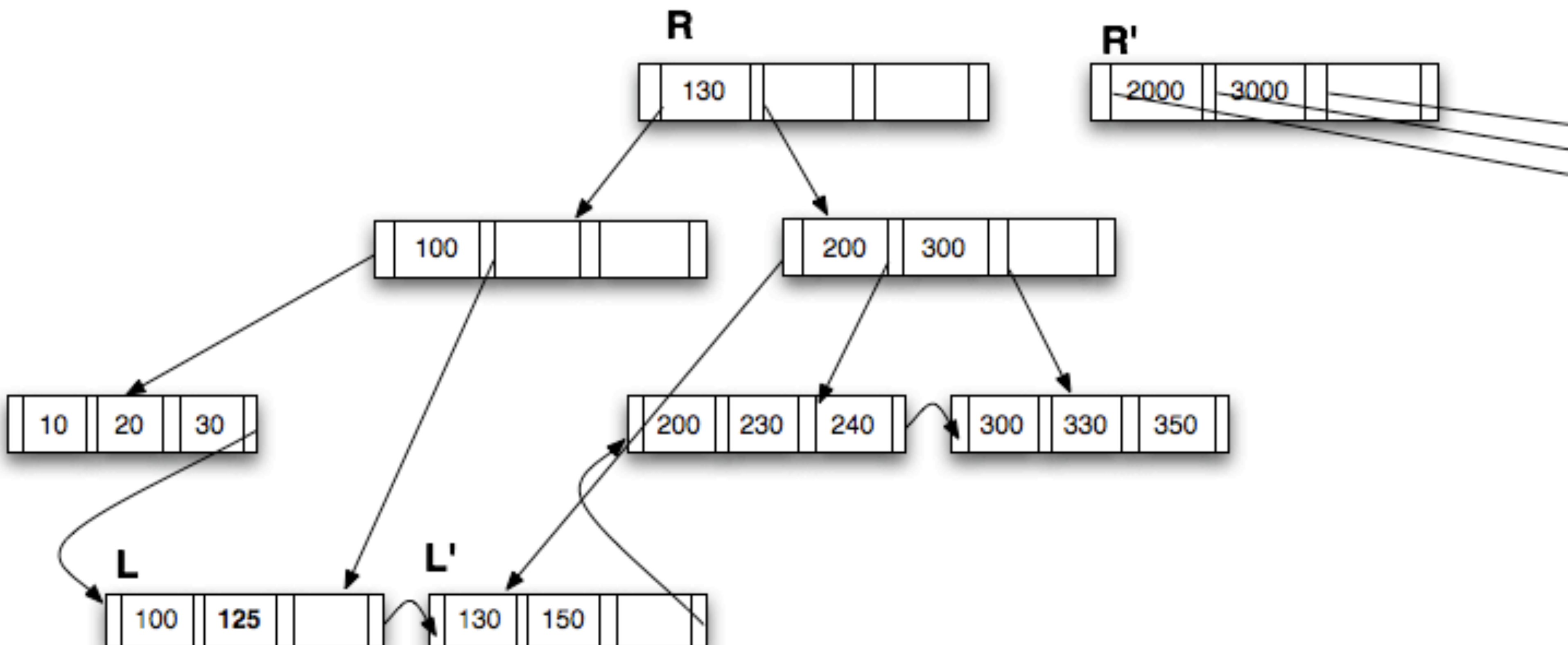
Step 3: Create P'; distribute from T into P and P'



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)

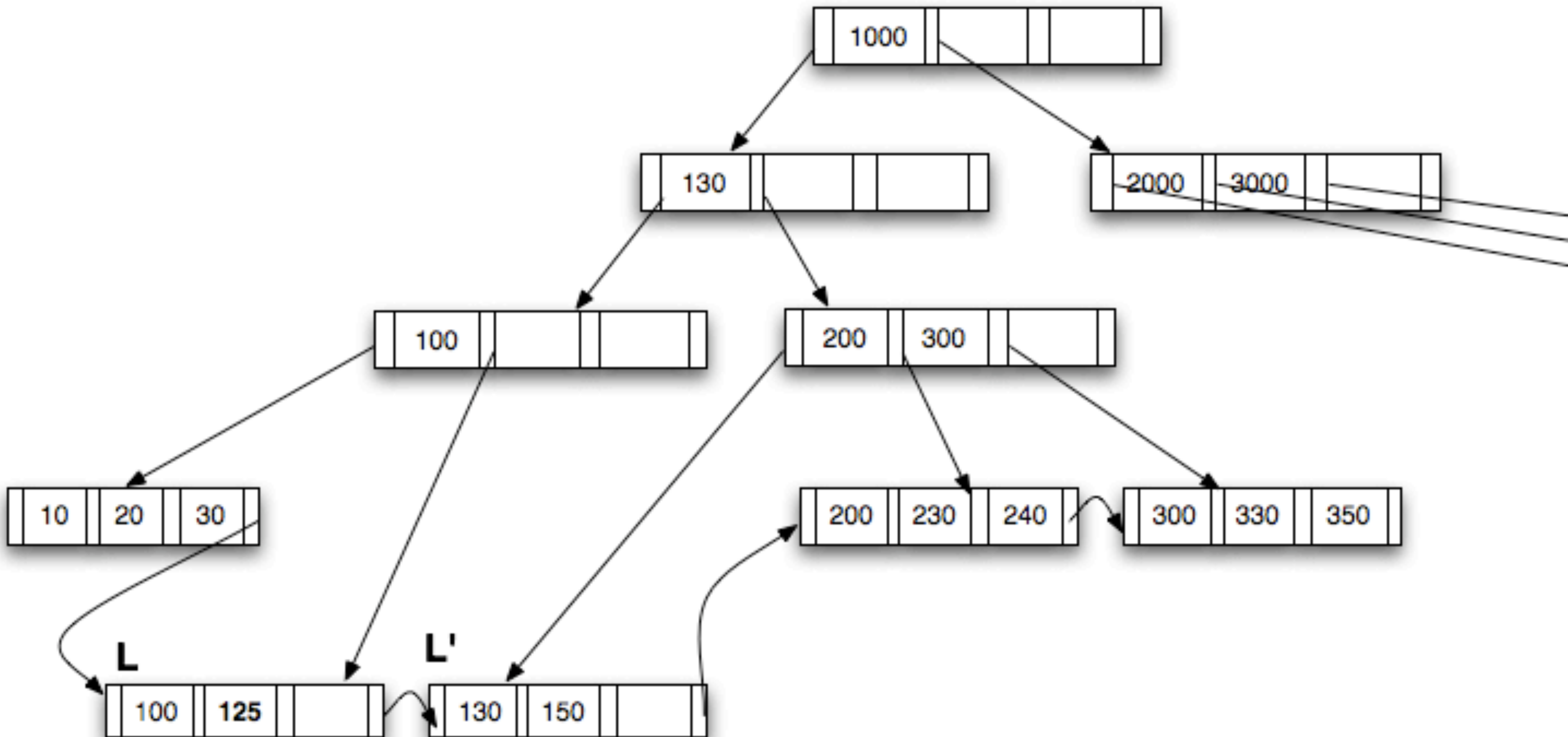
Step 4: Insert (130) into the parent (R); create R'



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