B^+-tree

Younghoon Kim
(nongaussian@hanyang.ac.kr)

B^+-tree

- Basic Concepts
- Ordered Indices
- Building a B^+-Tree
  - Insertion
  - Deletion

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to 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

- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

Index Evaluation Metrics

- Access types supported efficiently.
  - E.g.,
    - records with a specified value in the attribute
    - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value.
  - E.g., author catalog in library
- **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index

Dense Index Files

- **Dense index** — Index record 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	

Sparse Index Files

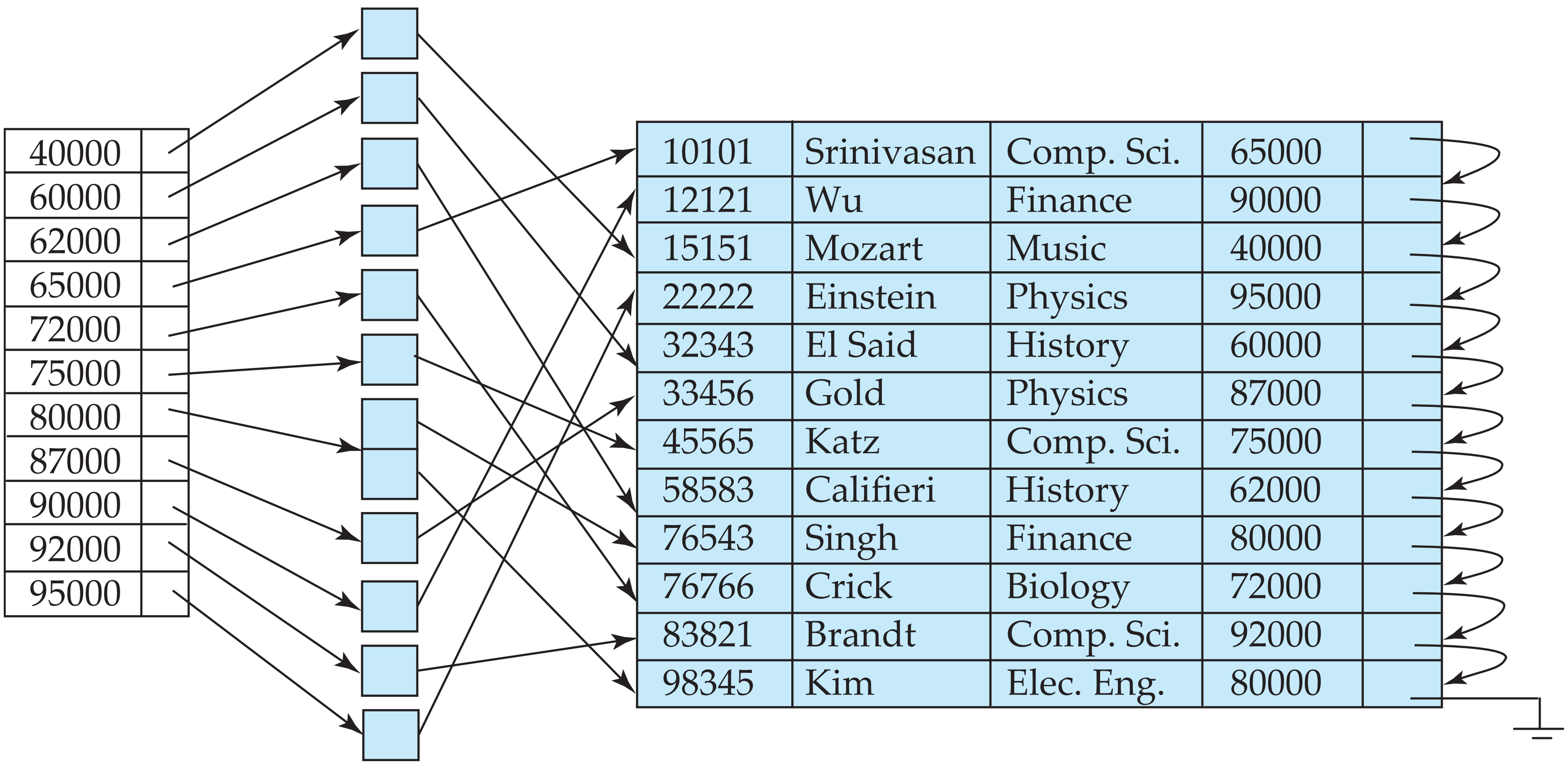
- **Sparse Index**: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value < *K*
  - Search file sequentially starting at the record to which the index record 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	

Secondary Indices Example

- 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



Secondary index on *salary* field of *instructor*

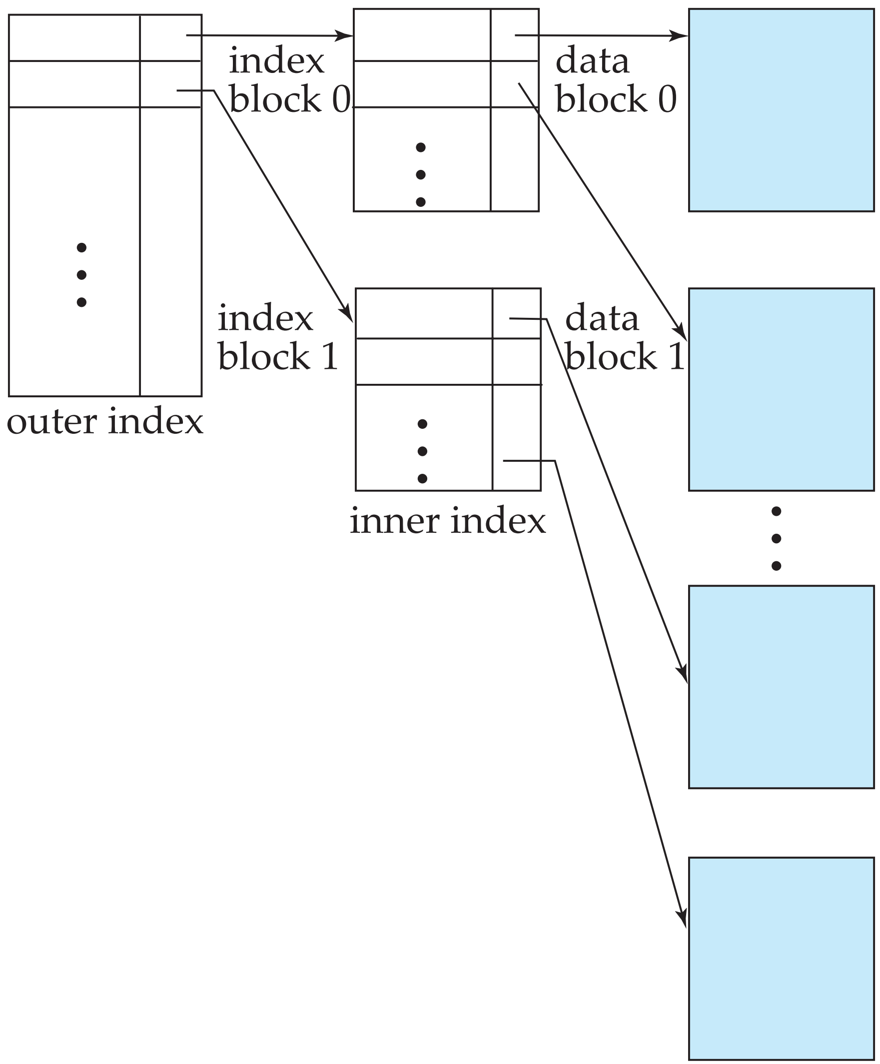
Primary and Secondary Indices

- 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

Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - 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.
- Indices at all levels must be updated on insertion or deletion from the file.

Multilevel Index (Cont.)

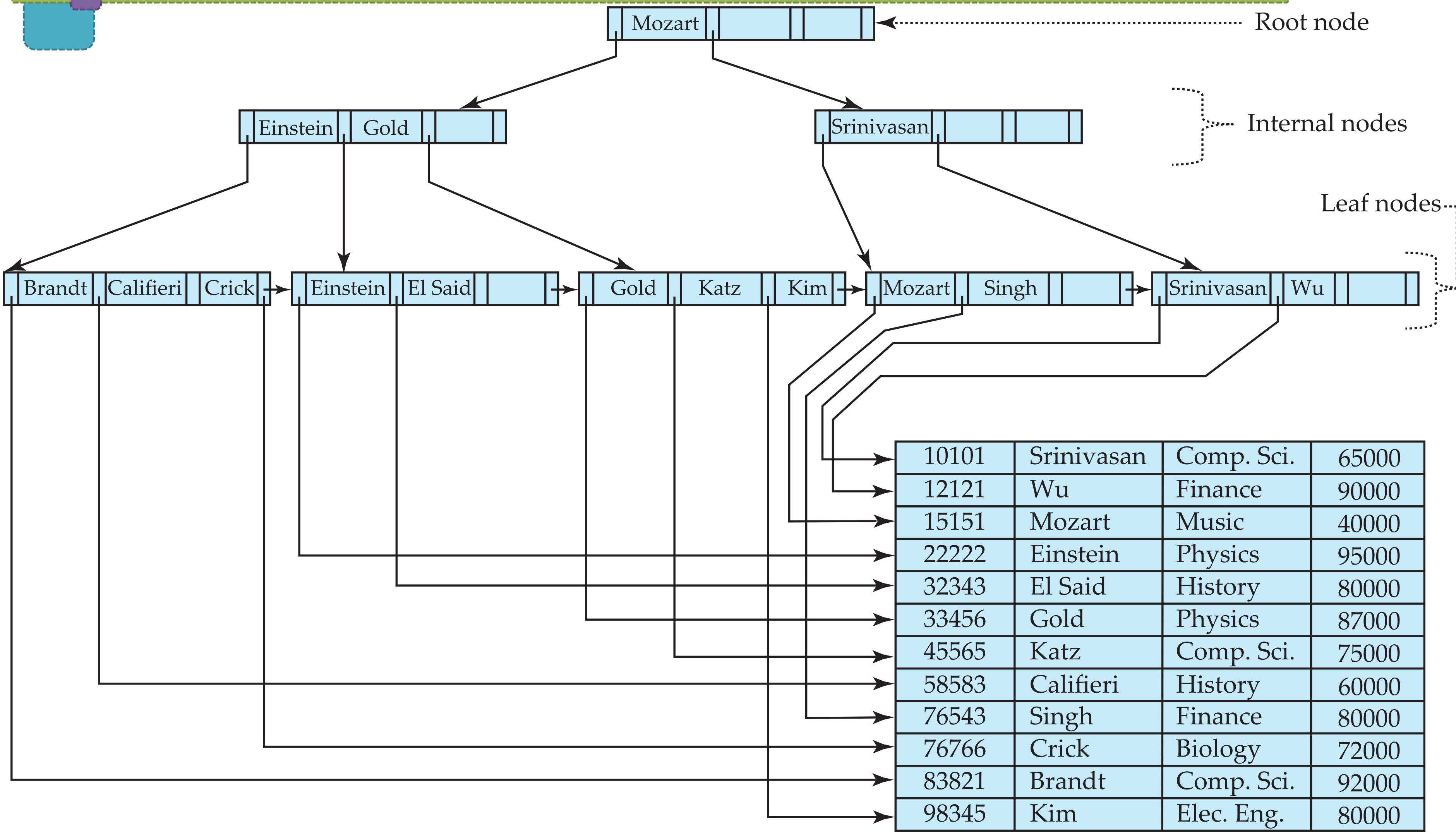


B^+-Tree Index Files

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)
- **Each node that is not a root or a leaf** has between ⌈n/2⌉ and n children.
- **A leaf node** has between ⌈(n–1)/2⌉ 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.
- Disk-base data structure (not main memory)
  - Page
- *Fanout **n** of a node*: the number of pointers out of the node

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

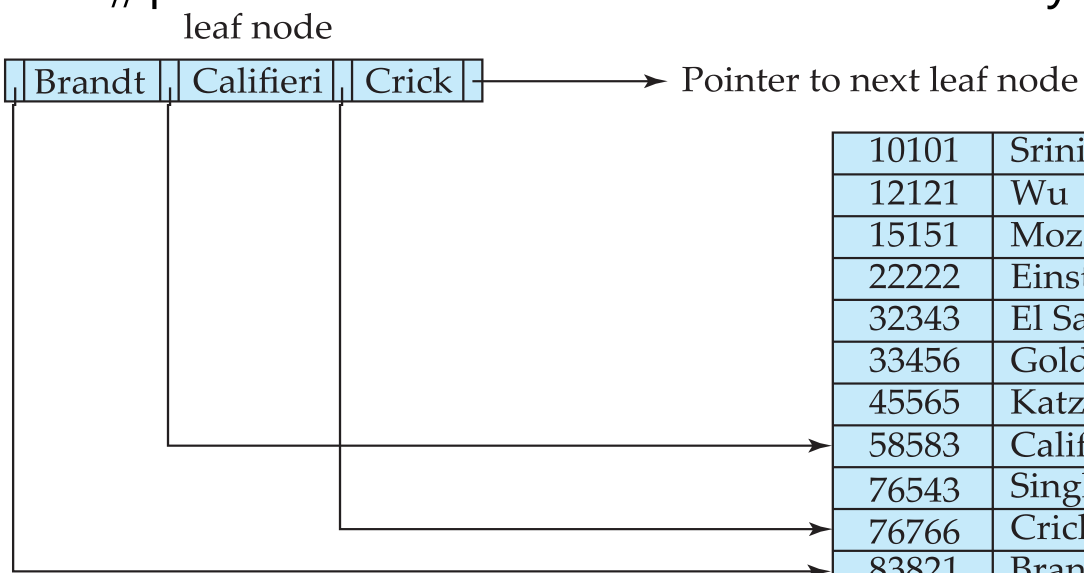
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially <u>assume no duplicate keys</u>, address duplicates later)

Leaf Nodes in B$^+$-Trees

Properties of a leaf node:

- For i = 1, 2, . . ., n–1, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values (i.e., increasing or decreasing order)
- P_n points to next leaf node in search-key order



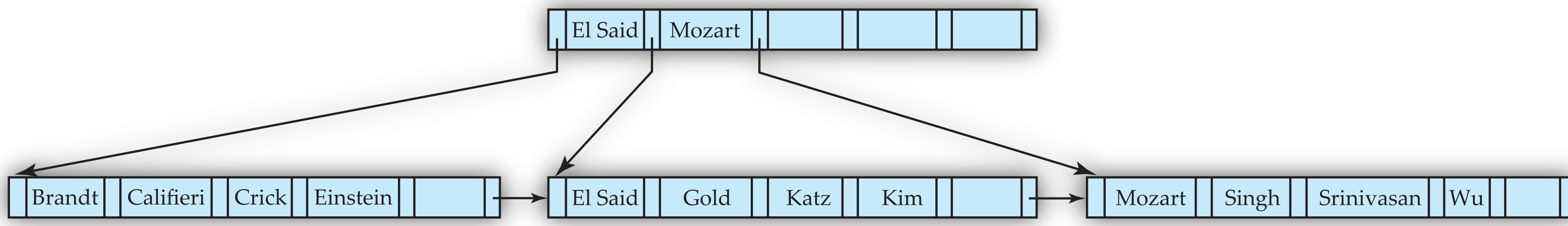
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

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 $2 \le i \le n - 1$, 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}

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

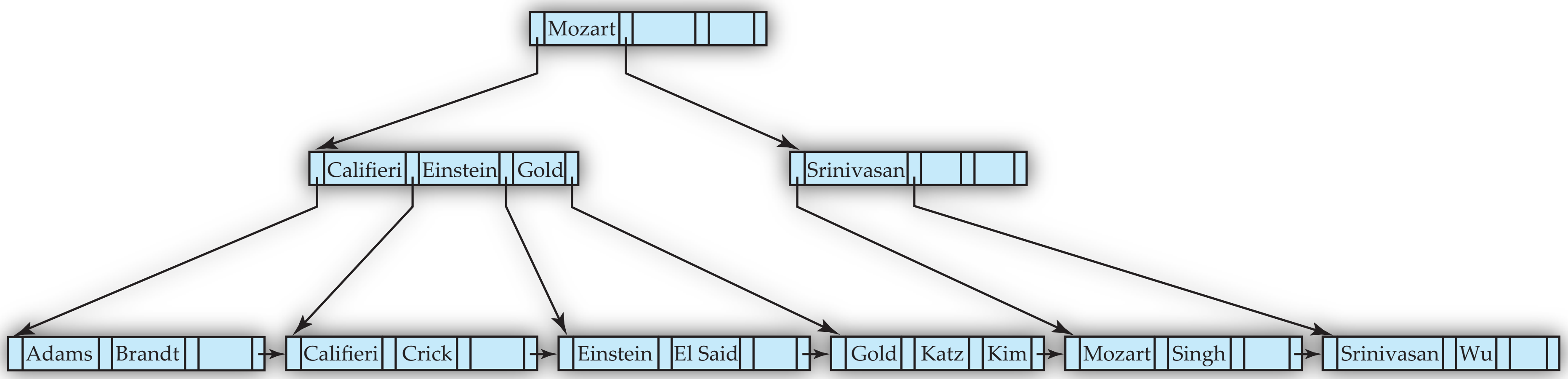
- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
    - Level below root has at least 2* ⌈n/2⌉ values
    - Next level has at least 2* ⌈n/2⌉ * ⌈n/2⌉ 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 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.)

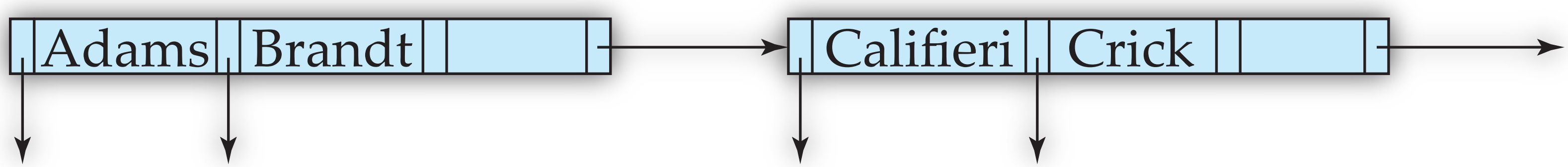
- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil} K \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  – and n is typically around 100 (40 bytes per index entry).
- With 1 million search key values and n = 100
  – at most $\lceil \log_{50} 1M \rceil$ = 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
   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.)

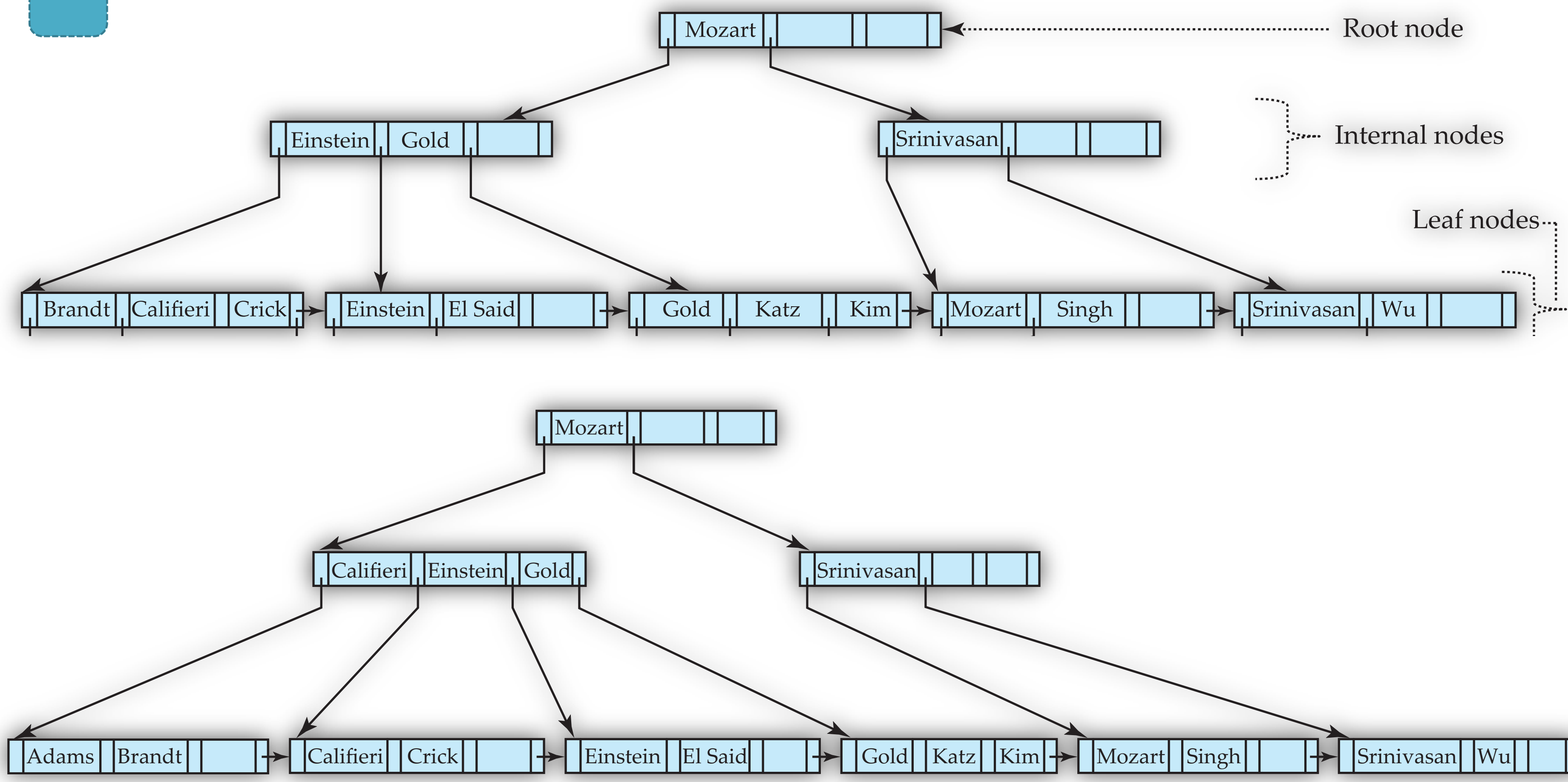
- **Splitting a leaf node**:
  - Take the *n* (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - Let the new node be *p,* and let *k* be the least(i.e., first) key value in *p*. 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

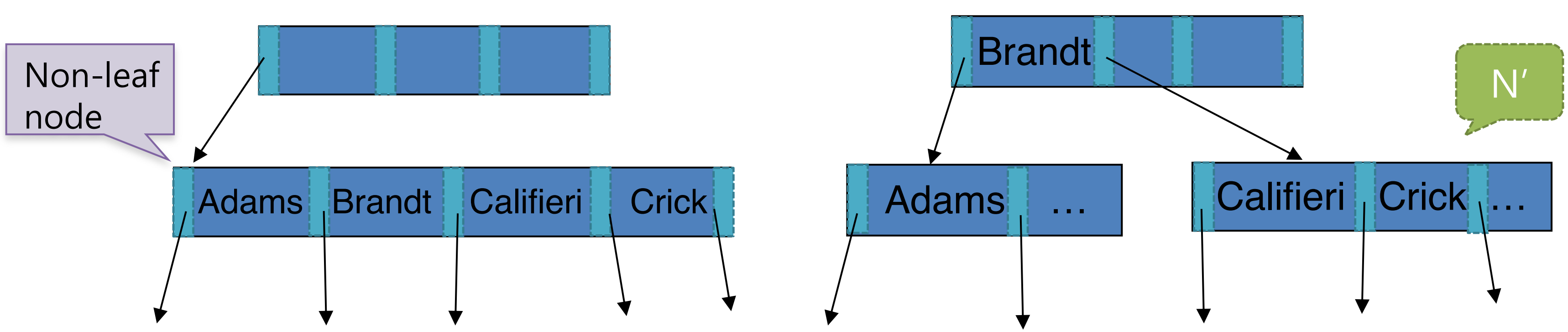
B+-Tree Insertion



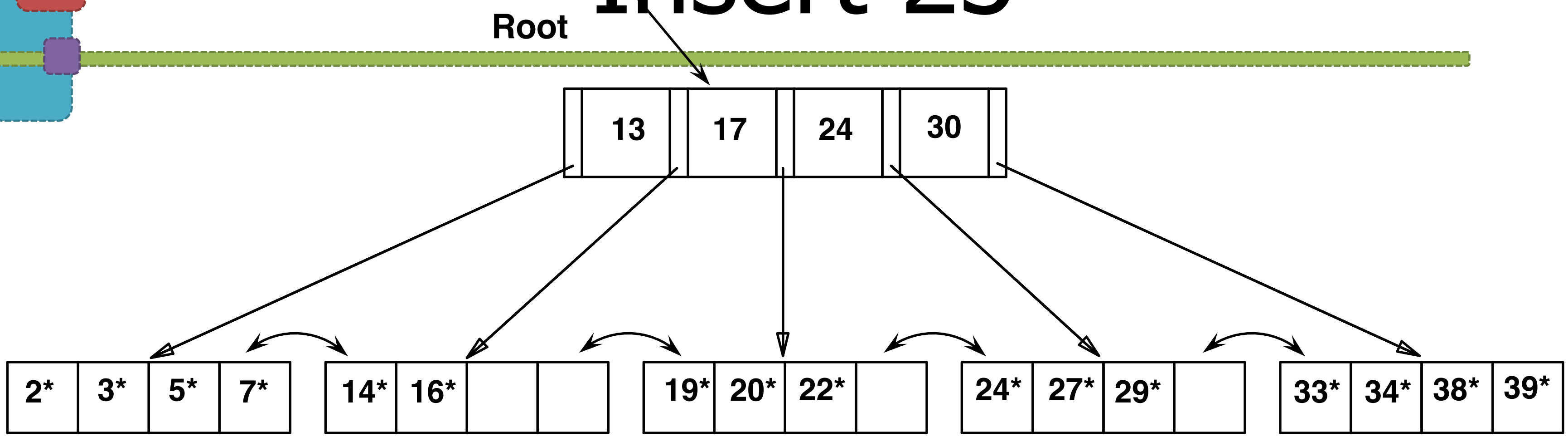
B+-Tree before and after insertion of “Adams”

Insertion in B+-Trees (Cont.)

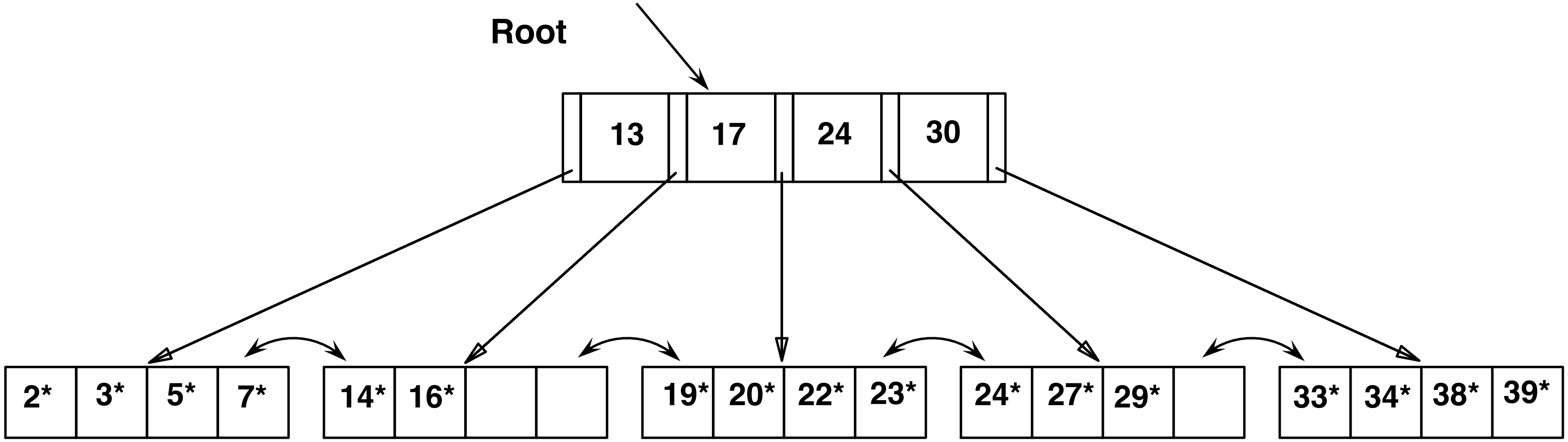
- **Splitting a non-leaf node**: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil \frac{n}{2} \rceil - 1}, P_{\lceil \frac{n}{2} \rceil}$ from M back into node N
  - Copy, $P_{\lceil \frac{n}{2} \rceil + 1}, \mathrm{K}_{\lceil \frac{n}{2} \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into **newly allocated node N′**
  - Insert $(K_{\lceil \frac{n}{2} \rceil}, N')$ into the parent of N



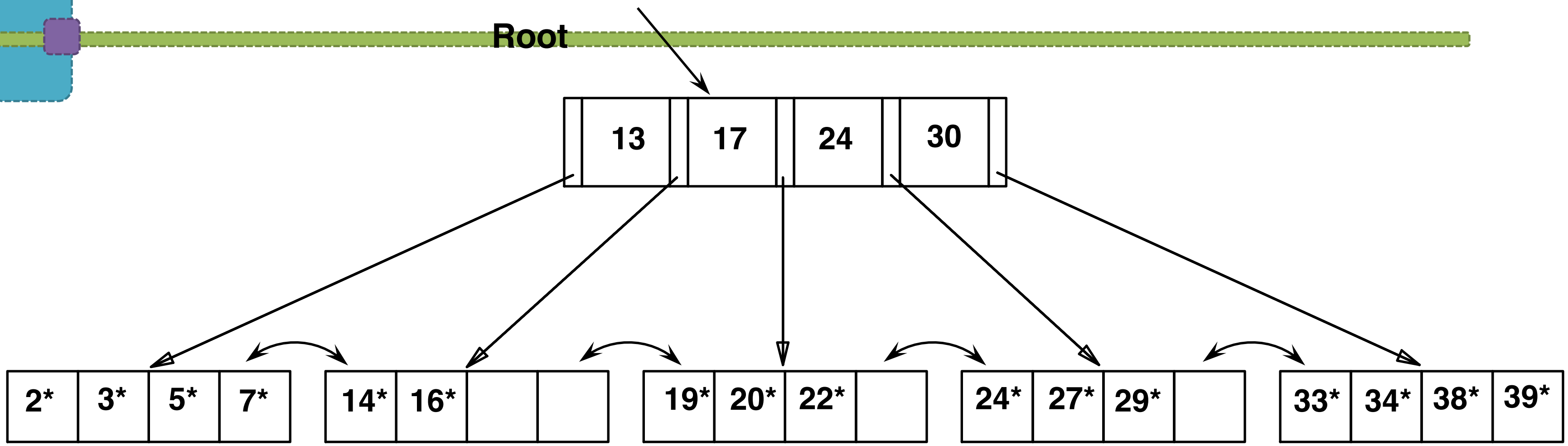
Insert 23*



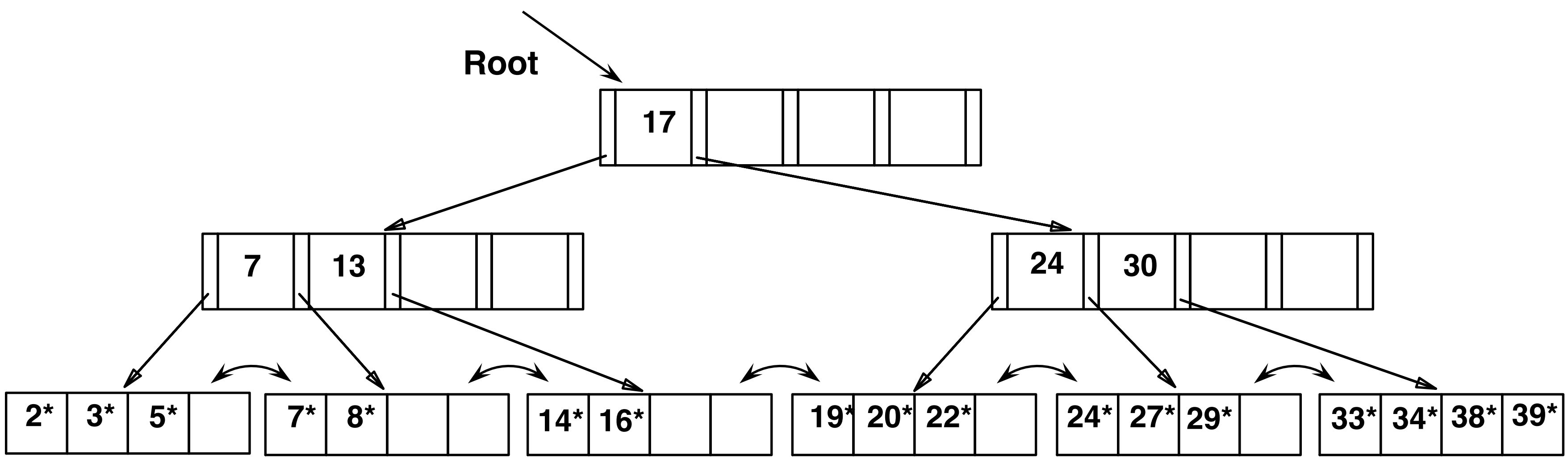
No splitting required.



Insert 8*

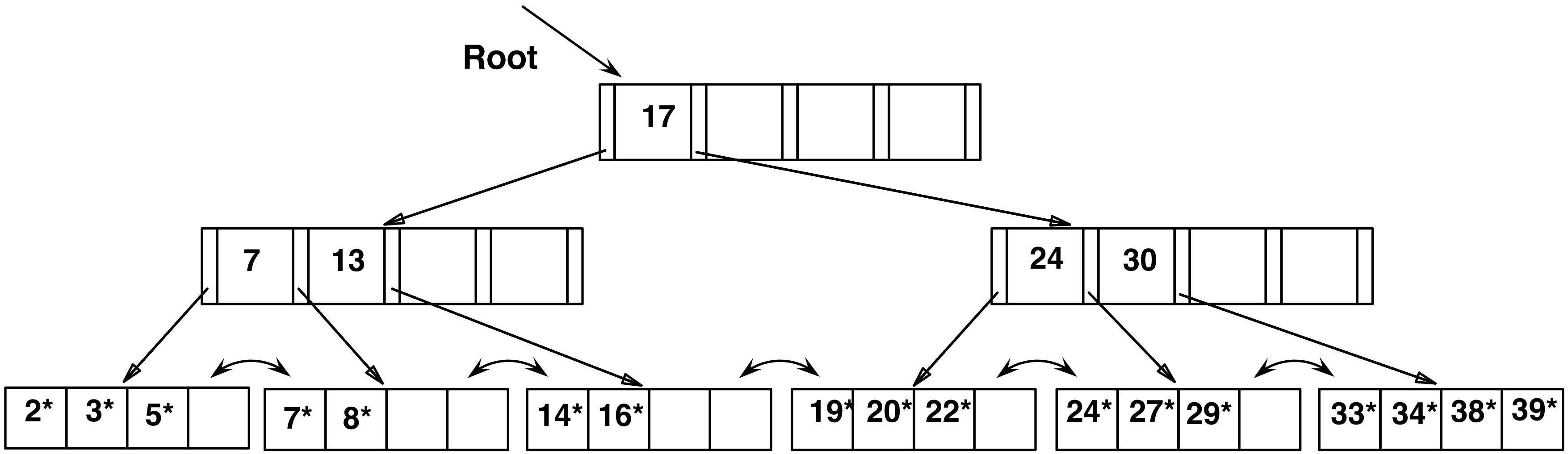


Root
13
17
24
30
2*
3*
5*
7*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*



Root
17
7
13
24
30
2*
3*
5*
7*
8*
14*
16*
19*
20*
22*
24*
27*
29*
33*
34*
38*
39*

Example B+ Tree - Inserting 8*



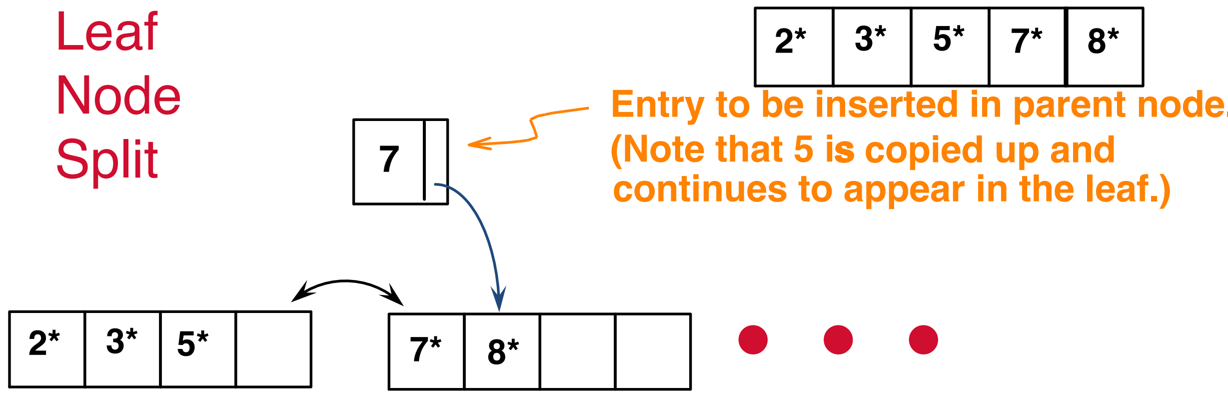
❖ Notice that root was split, leading to increase in height.

❖ In this example, we can avoid split by re-distributing entries; however, this is usually not done in practice.

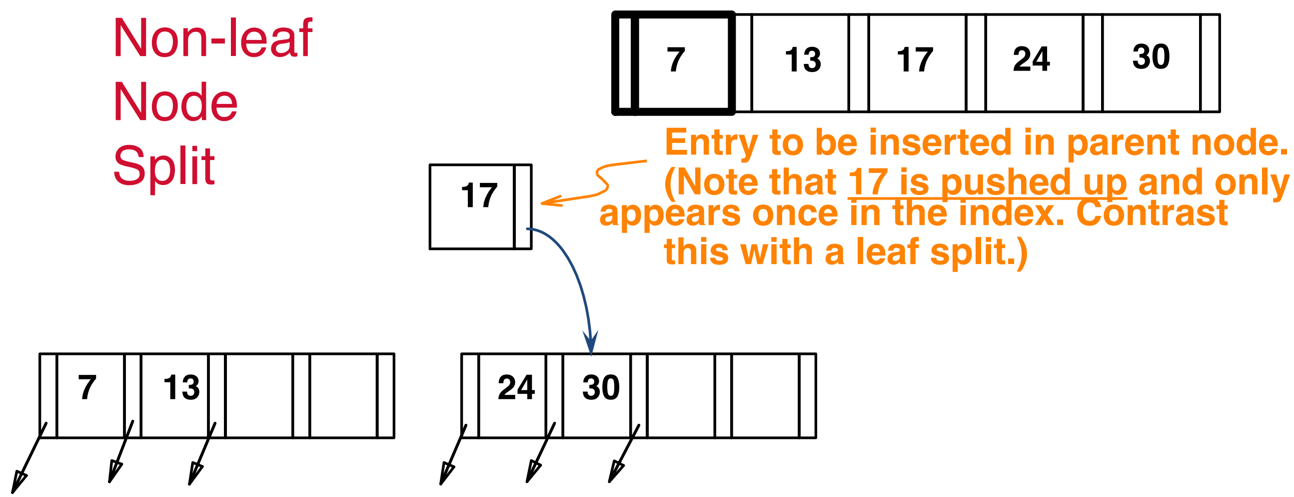
Leaf vs. Non-leaf Node Split
(from previous example of inserting "8")

- Observe how minimum occupancy is guaranteed in both leaf and non-leaf splits.
- Note difference between *copy-up* and *push-up*; be sure you understand the reasons for this.

Leaf Node Split

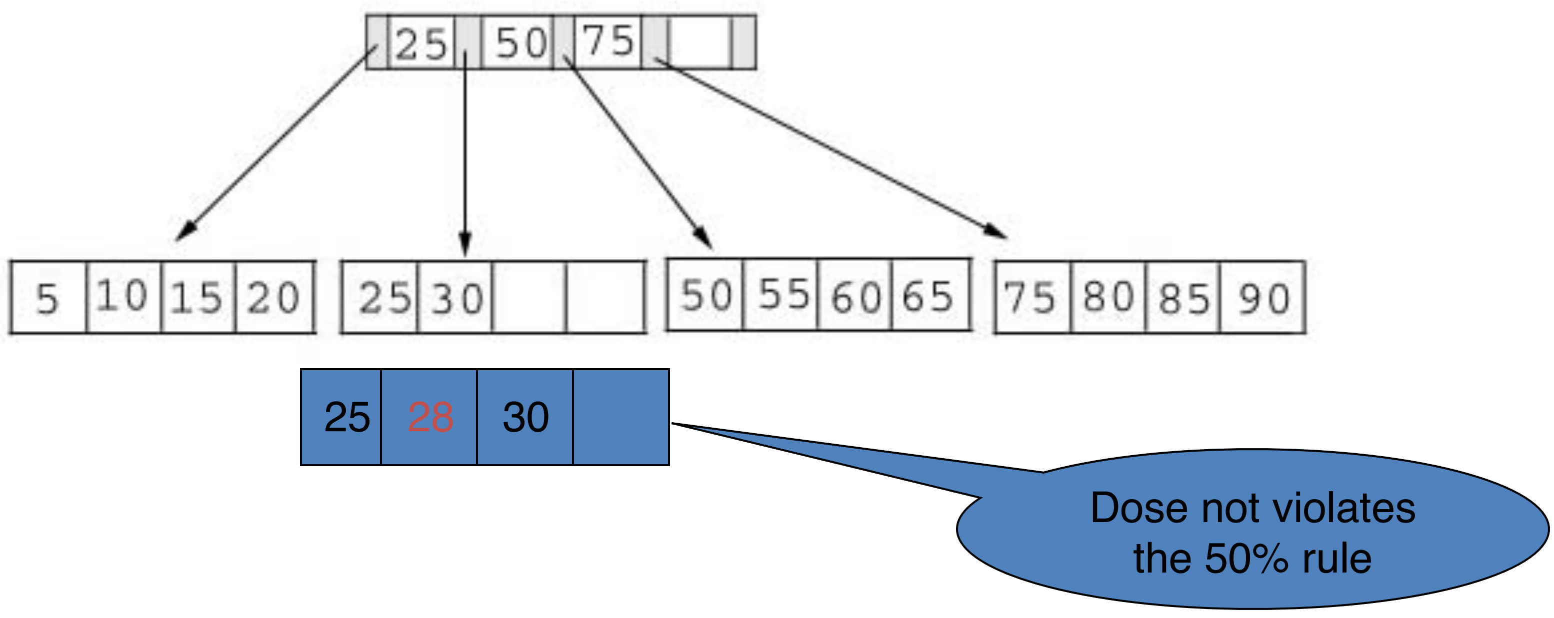


Non-leaf Node Split



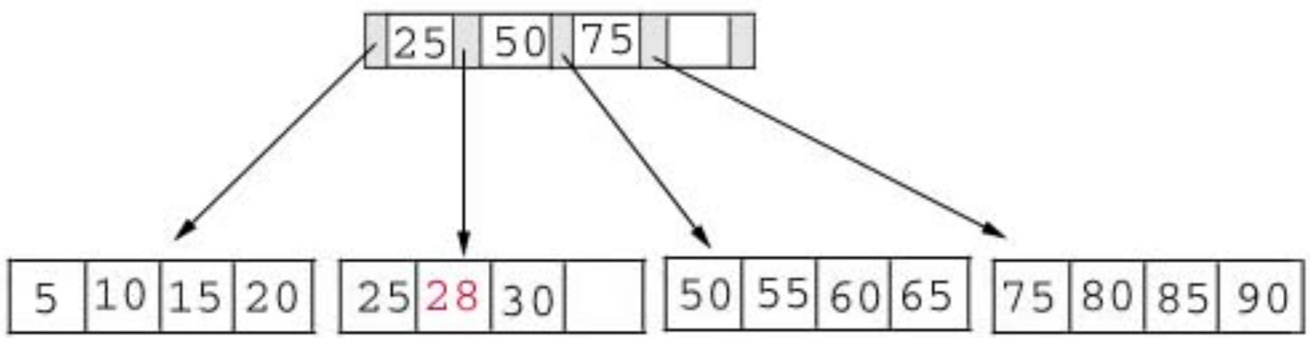
Exercise: Insertion

- Since insert a value into a B+ tree may cause the tree unbalance, so rearrange the tree if needed.
- Example #1: insert 28 into the below tree.



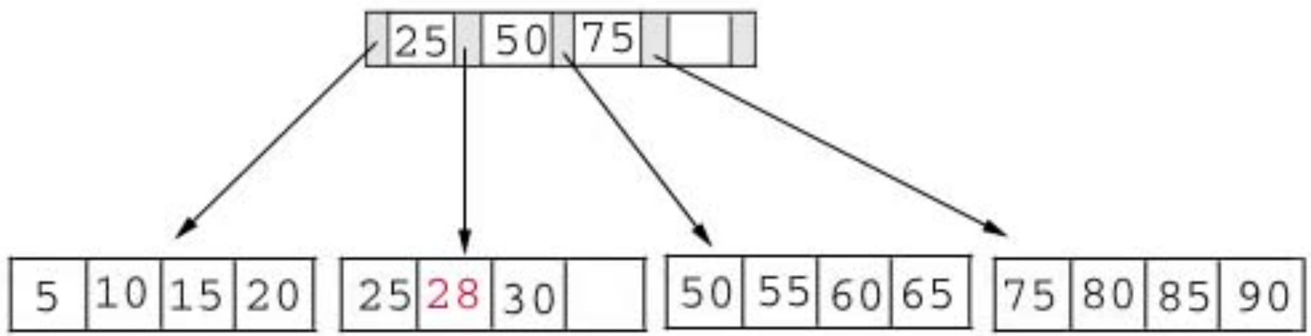
Insertion

- Result:



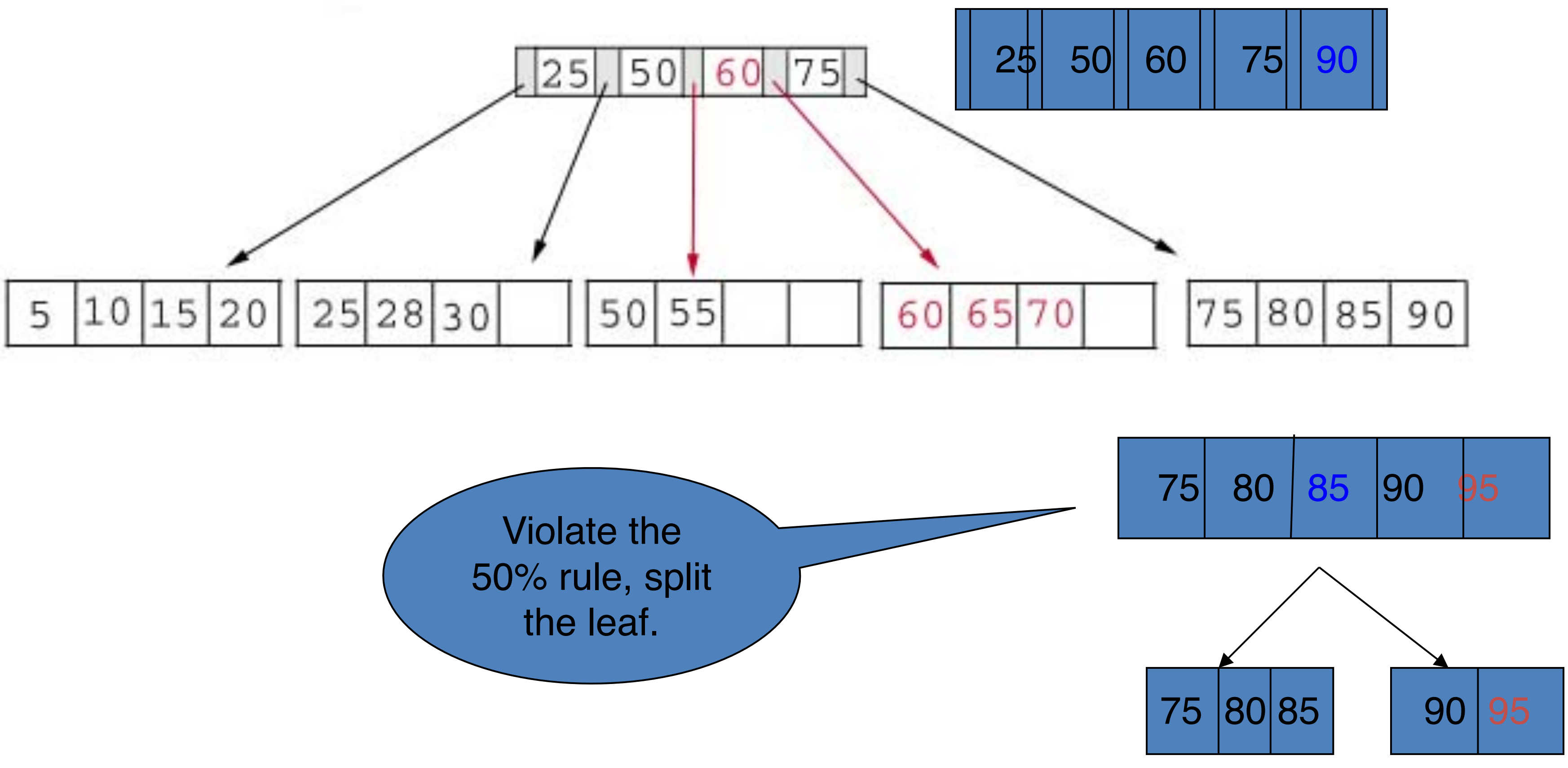
Insertion

- Example #2: insert 70 into below tree



Exercise: Insertion

- Add a key value 95 to the below tree

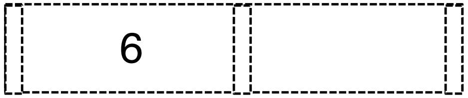


Inserting into a B+ Tree (cont.)

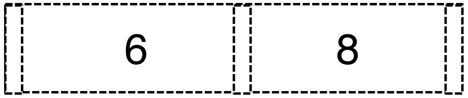
- Example:
  - Suppose we had a B+ tree with n = 3
    - 2 keys max. at each internal node
    - 3 pointers max. at each internal node

Inserting Into B+ Trees (cont.)

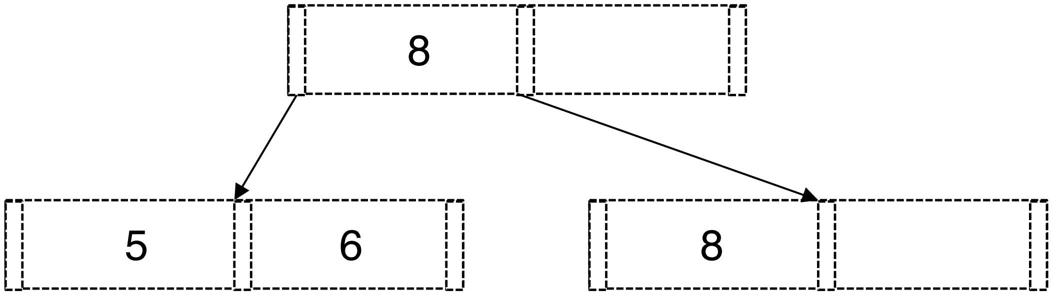
- Case 1: empty root
  - Insert 6



  - Insert 8

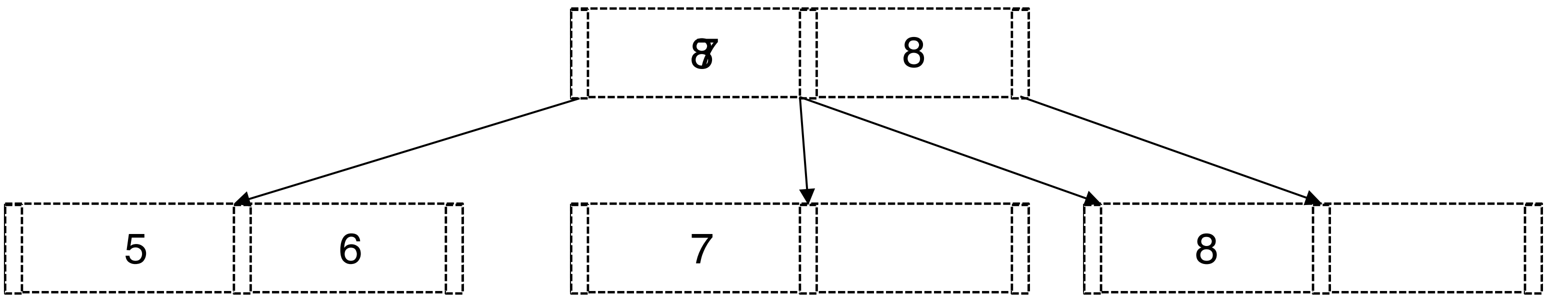


- Case 2: full root
  - Insert 5



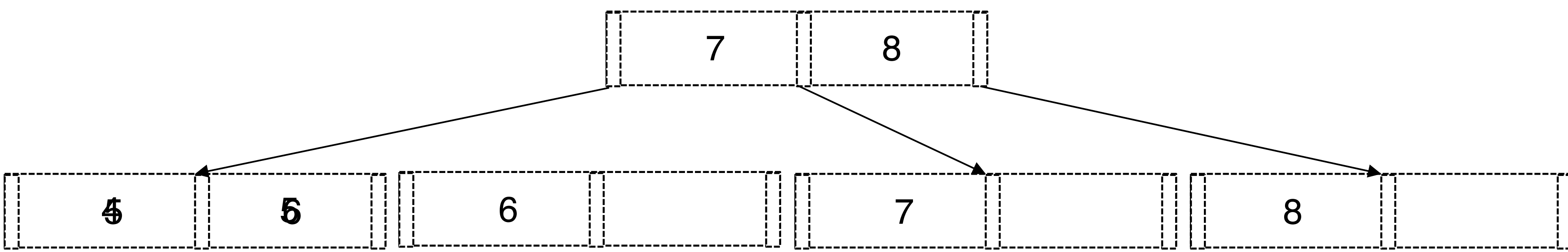
Inserting into B+ Trees (cont.)

- Case 3: Adding to a full node
  - Insert 7 into our tree:



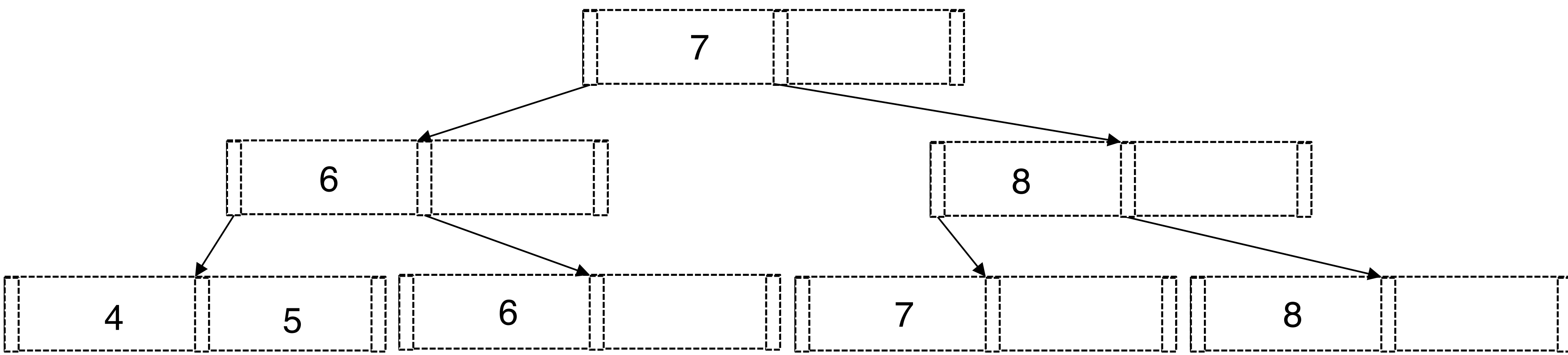
Inserting into B+ Trees (cont.)

- Case 4: Inserting on a full leaf, requiring a split at least 1 level up
  - Insert 4.

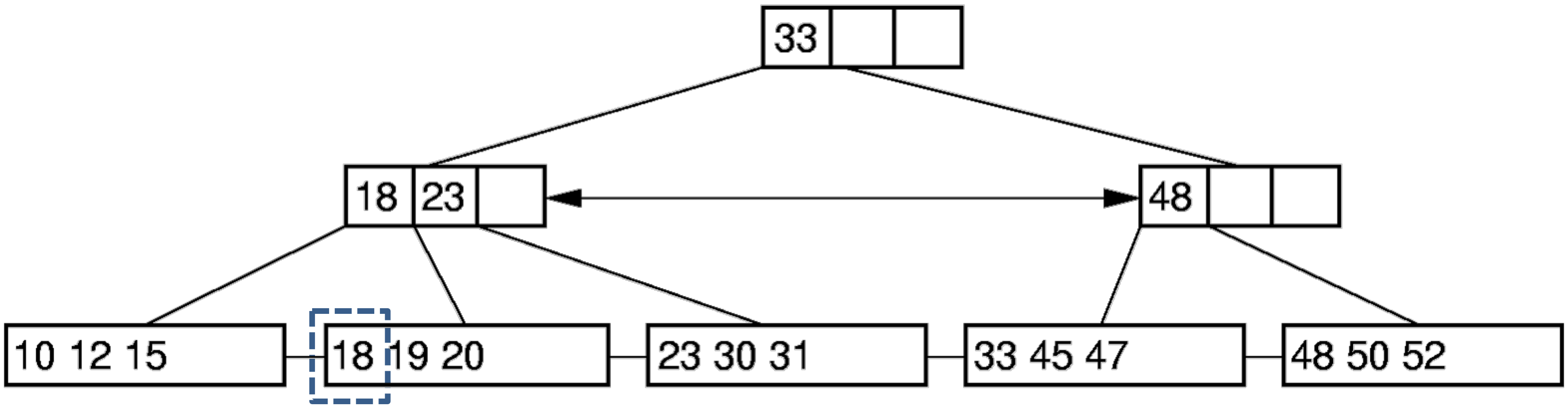


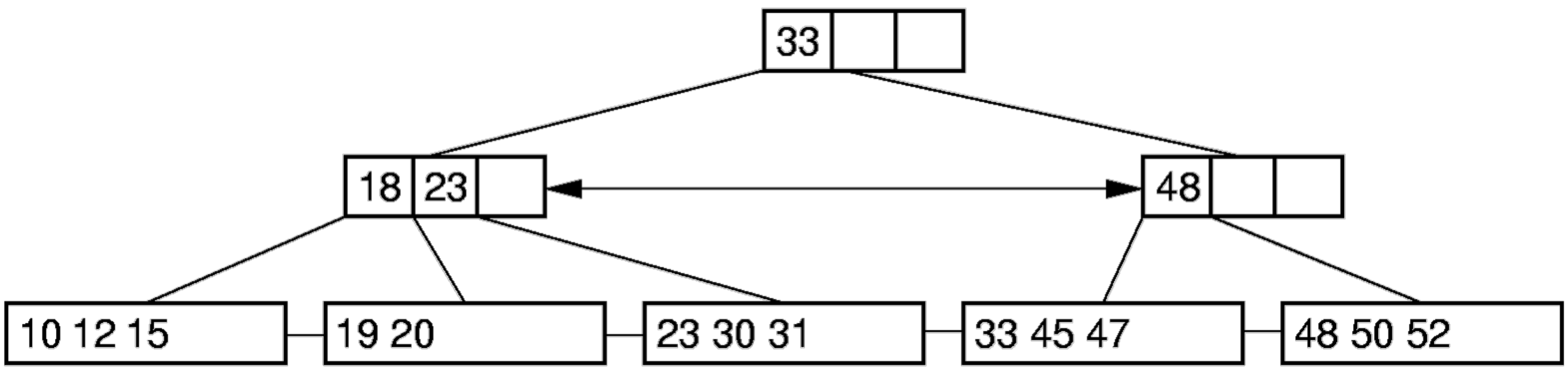
Inserting into B+ Trees (cont.)

- Case 4: Inserting on a full leaf, requiring a split at least 1 level up
  - Insert 4.

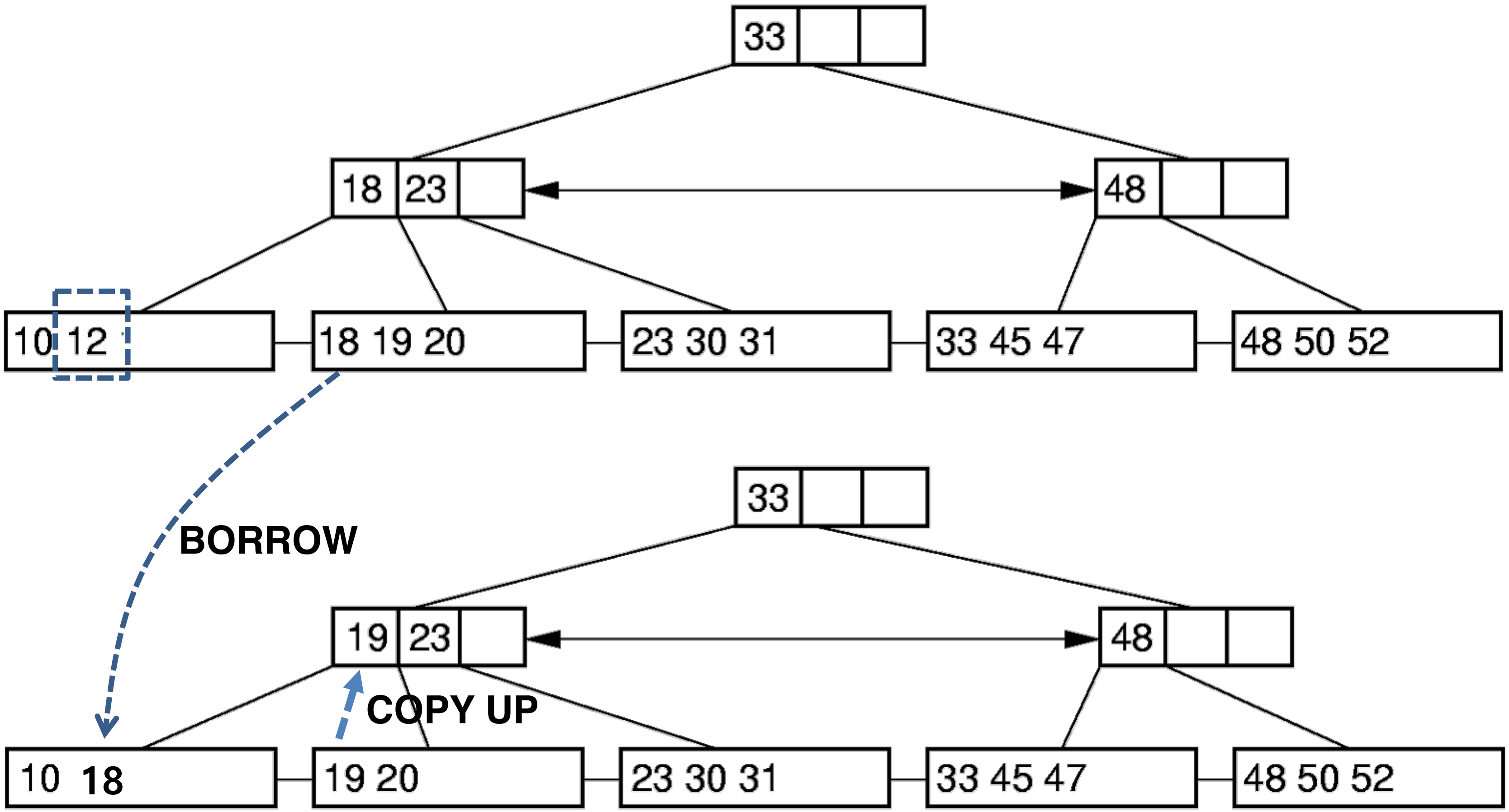


B+-Tree Deletion (1) [Delete 18] : Leaf node has enough keys



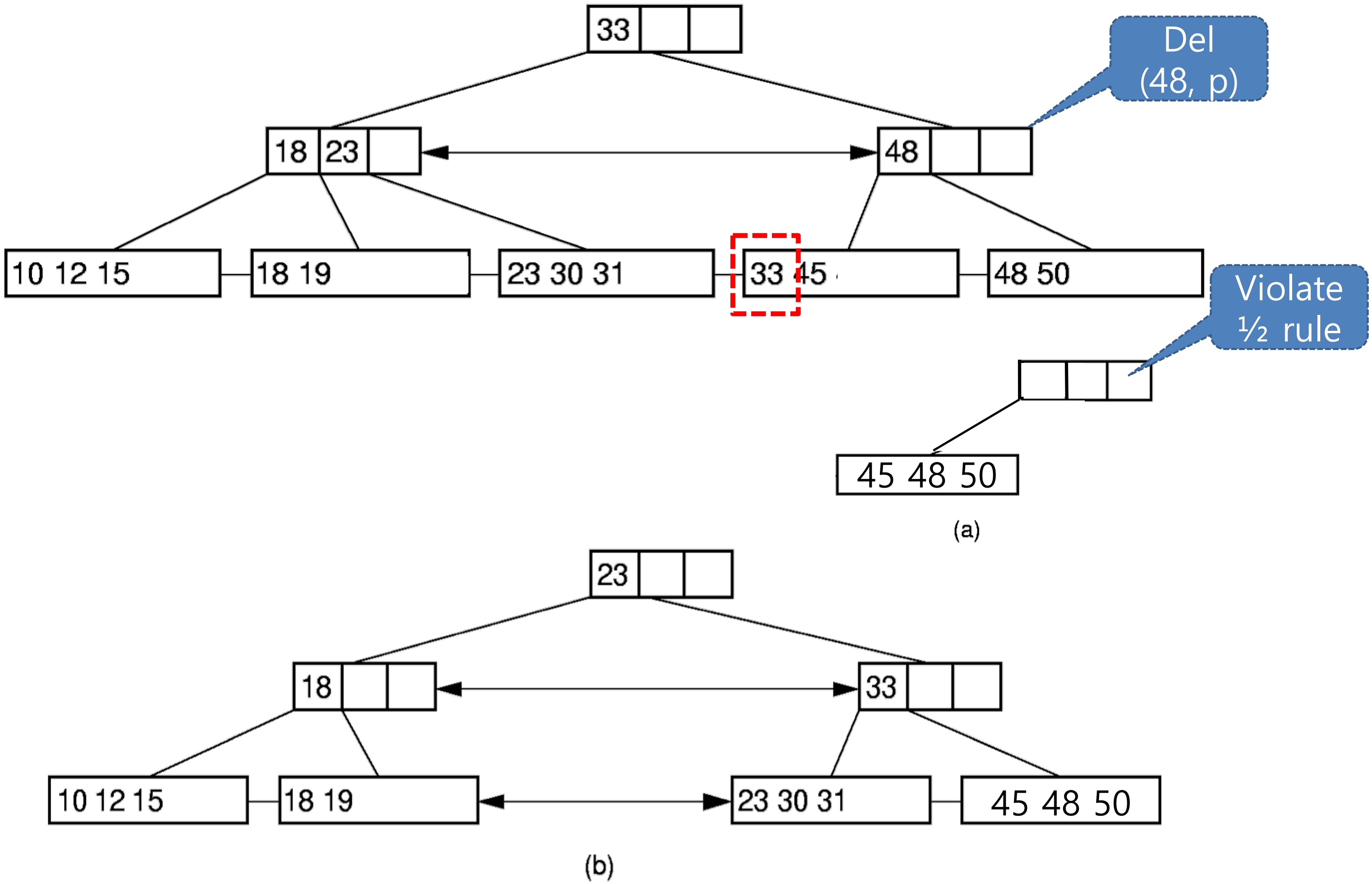


B+-Tree Deletion (2) [Delete 12]: Re-distribution in Leaf Nodes

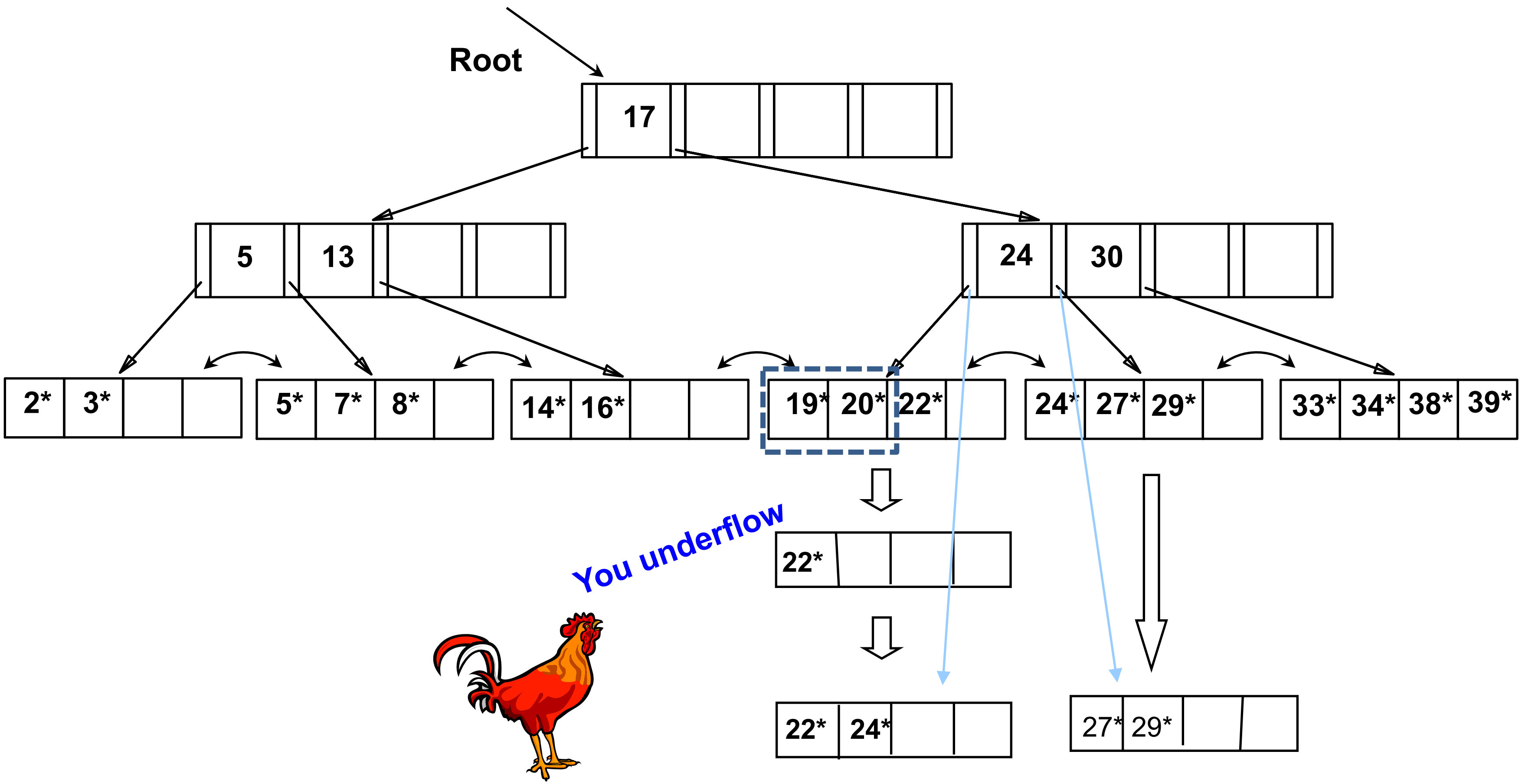


B+-Tree Deletion (3) [Delete 33]:

Merge in Leaf Nodes and Re-distribution in Non-leaf Nodes



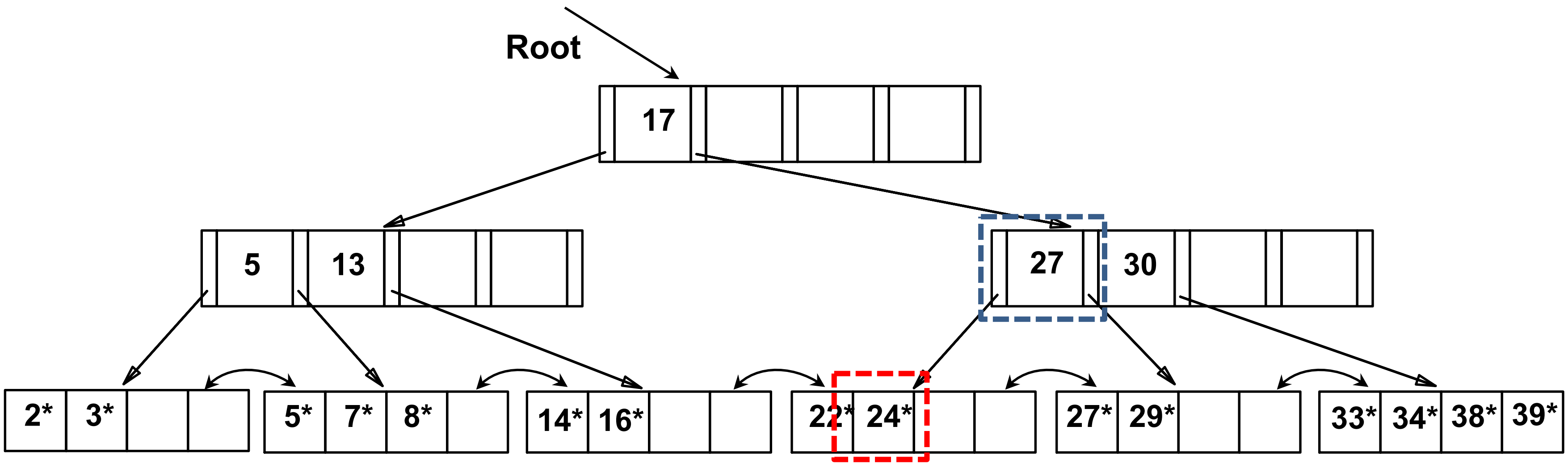
More Examples: Delete 19* and 20*



You underflow

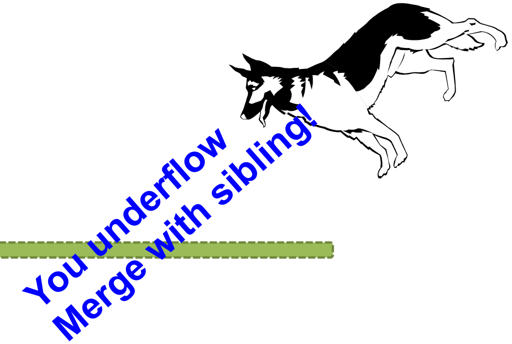
Have we still forgot something?

Deleting 19* and 20* (cont.)

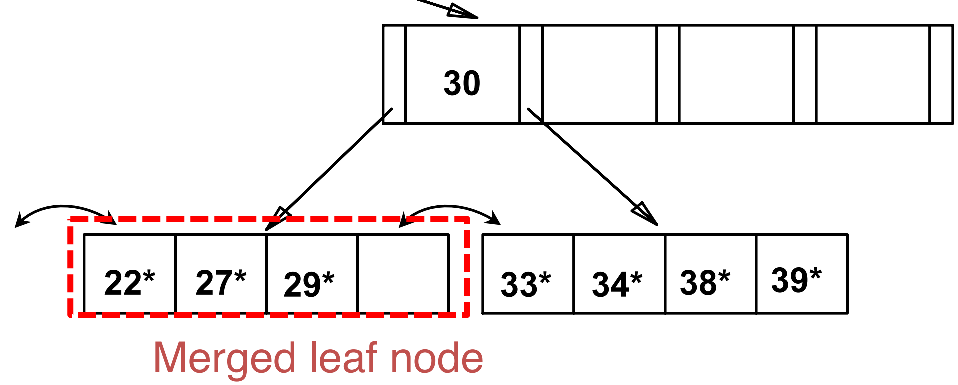


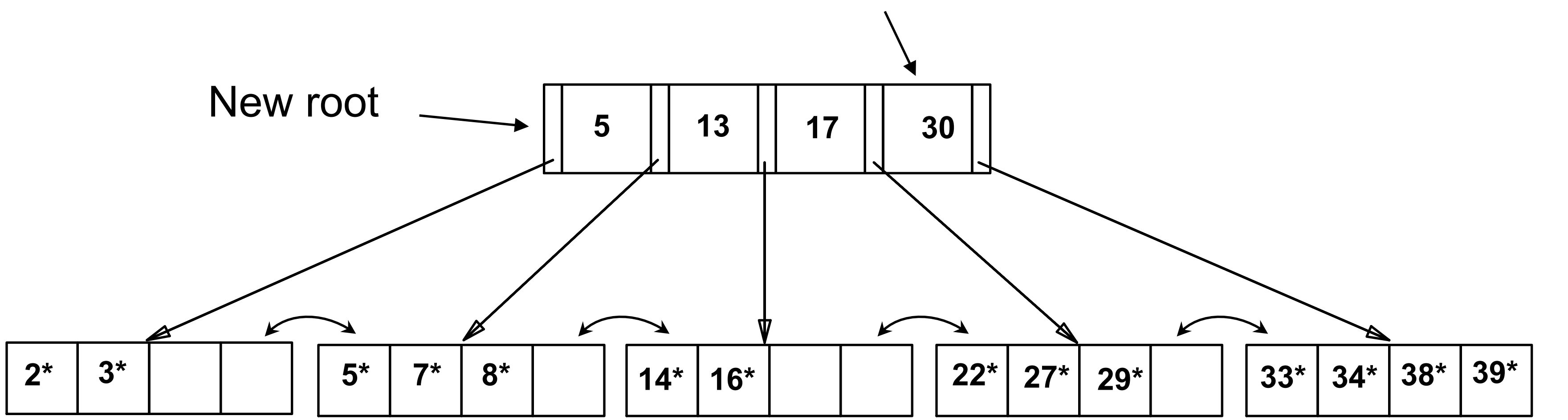
- Notice how 27 is *copied up*.
- But can we move it up?
- Now we want to delete 24
- Underflow again! But can we redistribute this time?

Deleting 24*



- Observe the two leaf nodes are merged, and 27 is discarded from their parent, but …
- Observe *`pull down'* of index entry (below).



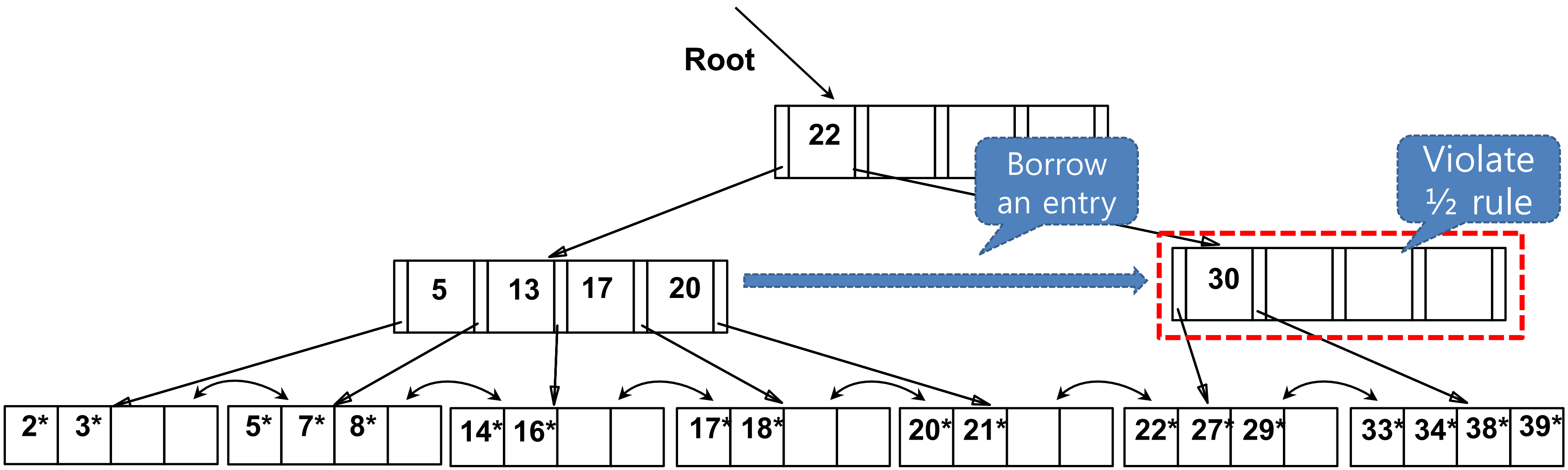


Deleting a Data Entry from a B+ Tree: Summary

- Start at root, find leaf *L* where entry belongs
- Remove the entry from *L*
- If L is at least half-full, *done!*
- If L has only ⌈(n−1)/2⌉-1 entries,
  - Try to re-distribute, borrow from *sibling* (***adjacent node with same parent as L***).
  - If re-distribution fails, *merge* *L* and sibling.
    - If merge occurred, must delete ***entry pointing to L or sibling*** from parent of *L*.
    - Merge could propagate to root, decreasing height.

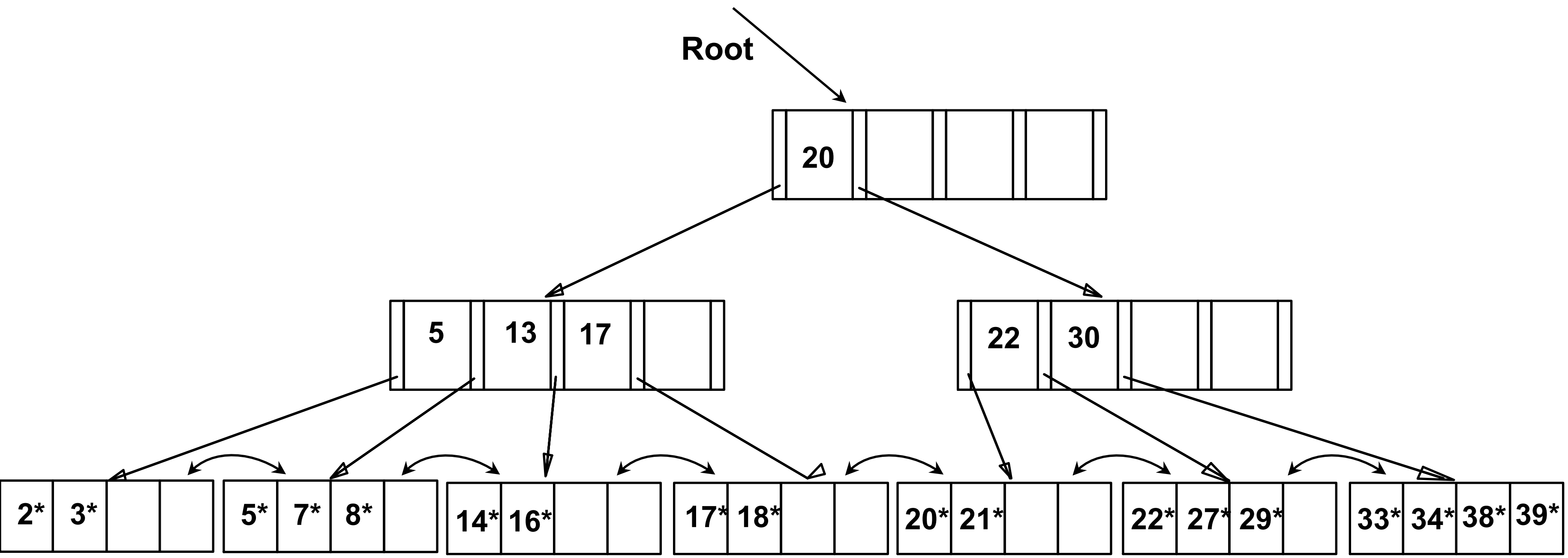
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*
- In contrast to previous example, <u>can re-distribute entry from left child of root to right child</u>.



After Re-distribution

- Intuitively, entries are re-distributed by *`pushing through'* the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



Deletion Pseudocode

- procedure delete (value K, pointer P) {
  - Find the leaf node with (K, P)
  - delete_entry (L, K, P);
- }

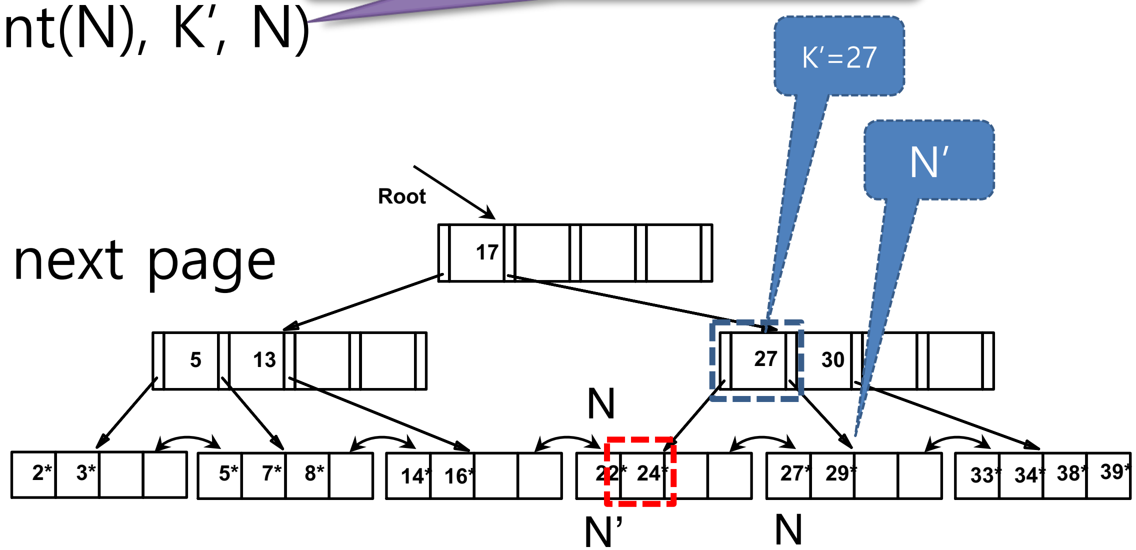
Deletion Pseudocode

- procedure delete_entry (node N, value K, pointer P) {
  - delete (K, P) from N
  - if (N is root and N has only one child) {
    - Change the child to a new root node & delete the old root
  - } else if (N is against ½ rule) {
    - → next page
  - }
- }

Deletion Pseudocode

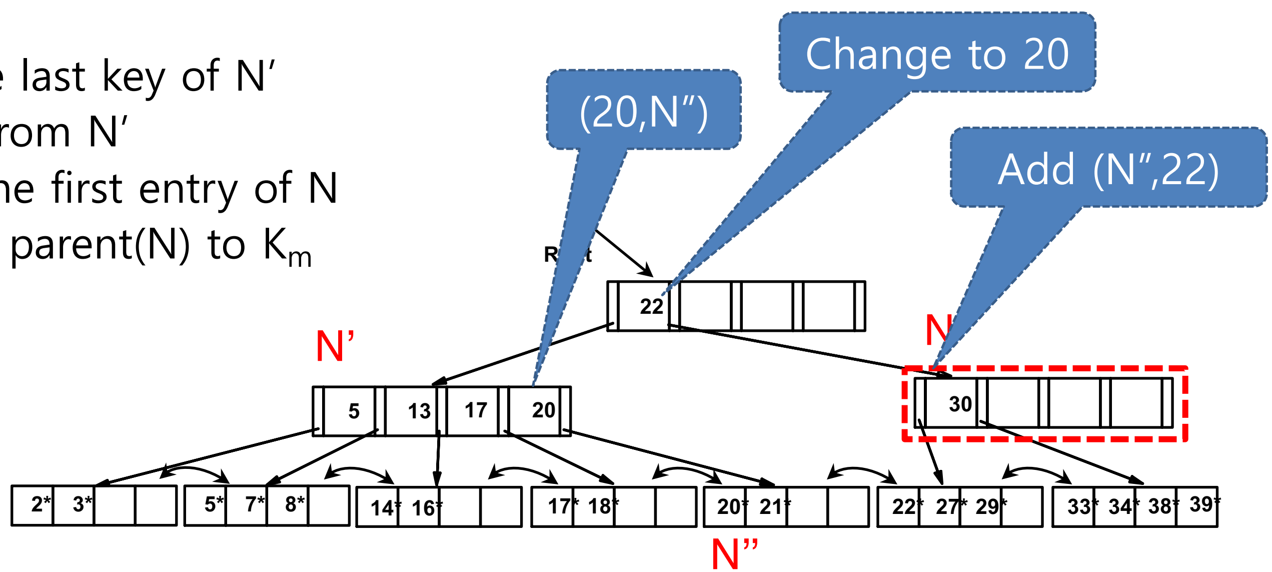
- N′ ← {Sibling of N (i.e., a node sharing the same parent as N)}
- K′ ← {The key between N and N′ in parent(N)}
- if (N and N′ can be merged into a node) {
  - if (N precedes N′) swap (N, N′)
  - if (N is not a leaf)
    - Attach K′ to N′ & move all N′s (K, P) pairs to N′
  - else
    - Move all N′s (K, P) pairs to N′
  - delete_entry (parent(N), K′, N)
- }
- else {
  - Re-distribution → next page
- }

delete_entry(parent(N), 27, N)



Deletion Pseudocode

- // N′ ← a sibling of N
- // K′ ← key between N and N′ in parent(N)
- if (N′ precedes N) {
  - if (N is a non-leaf) {
    - Let P_m be the last pointer of N′
    - Remove the last (K_{m-1}, P_m) from N′
    - Push (P_m, K′) as P_1 and K_1 to N
    - Change K′ in parent(N) to K_{m-1}
  - }
  - else {
    - Let K_m be the last key of N′
    - Remove K_m from N′
    - Push K_m to the first entry of N
    - Change K′ in parent(N) to K_m
  - }
- }

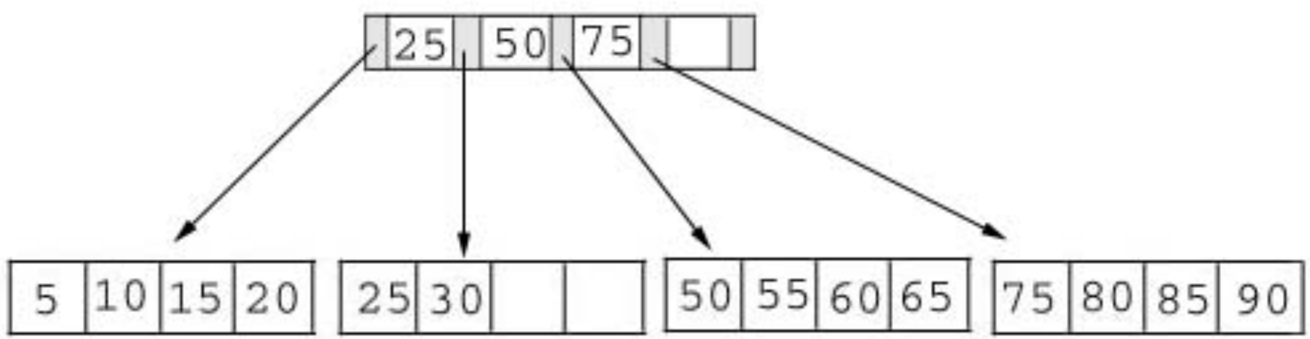


Procedure delete_entry(node L, value V, pointer P)

```
delete (V,P) from L
if (L is root and L has only one remaining child)
then make the child of L the new root and delete L
else if (L has too few values)
    Let L' be the prev or next child of parent(L)
    Let V' be the value between pointers L and L' in parent(L)
    if (entries in L and L' can fit in a single node)
            if (L is a predecessor of L') then swap(L, L')
            if (L is not a leaf)
                then append V' and all pointers and values in L to L'
                else append all (K,P) pairs in L to L' and set L'.P_n = L.P_n
            delete_entry(parent(L), V', L) and delete node L
    else
            if (L' is a predecessor of L)
                if (L is a non-leaf node)
                    let m be s.t. L'.P_m is the last pointer in L'
                    remove (L'.K_m-1, L'.P_m) from L'
                    insert (L'.P_m, V') as the first pointer and value in L
                    replace V' in paranet(L) by L'.K_m-1
                else
                    let m be s.t. (L'.P_m, L'.K_m) is ther last pointer and value in L'
                    remove (L'.P_m, L'.K_m) from L'
                    insert (L'.P_m, L'.K_m) as the first pointer and value in L
                    replace V' in paranet(L) by L'.K_m-1
            else
                (symmetric case)
```

Exercise: Searching

- Since no structure change in a B+ tree during a searching process, so just compare the key value with the data in the tree, then give the result back.
- For example: find the value 45, and 15 in below tree.



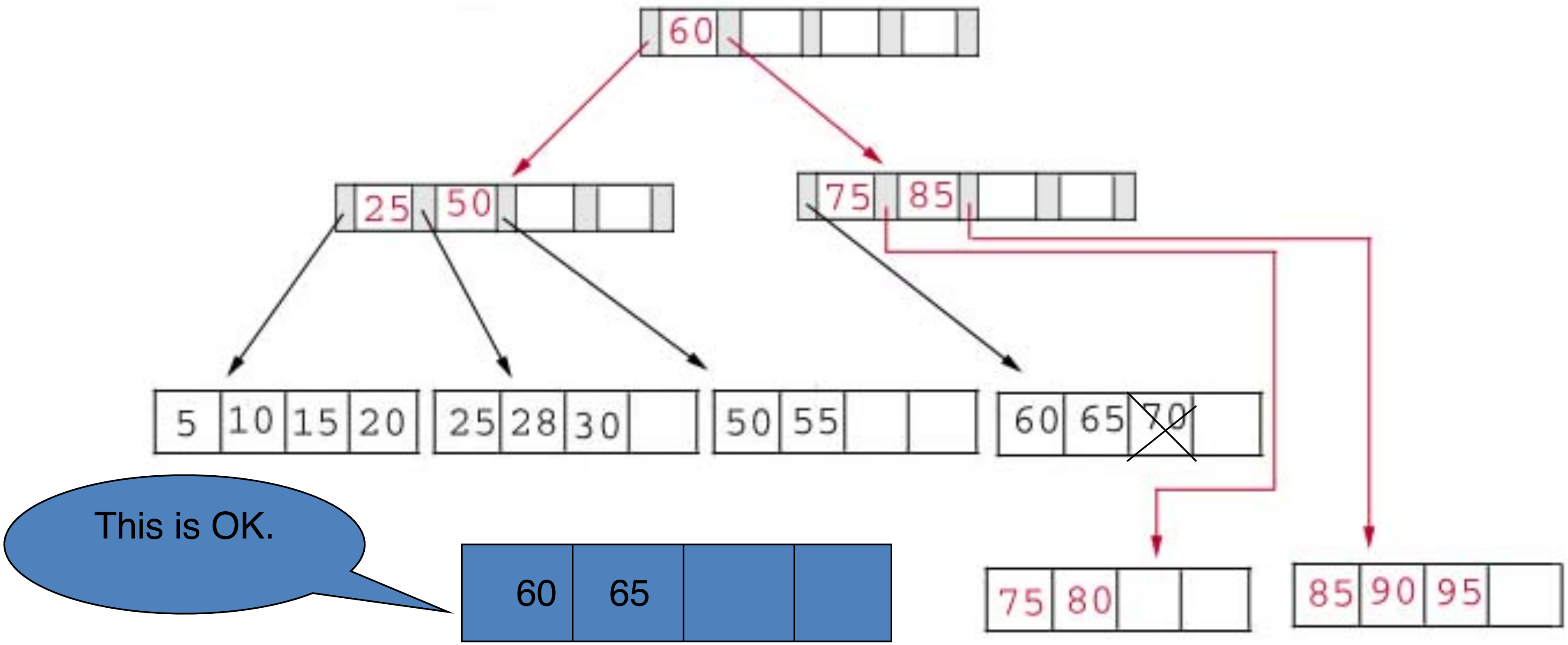
Searching

- Result:

1. For the value of 45, not found.
2. For the value of 15, return the position where the pointer located.

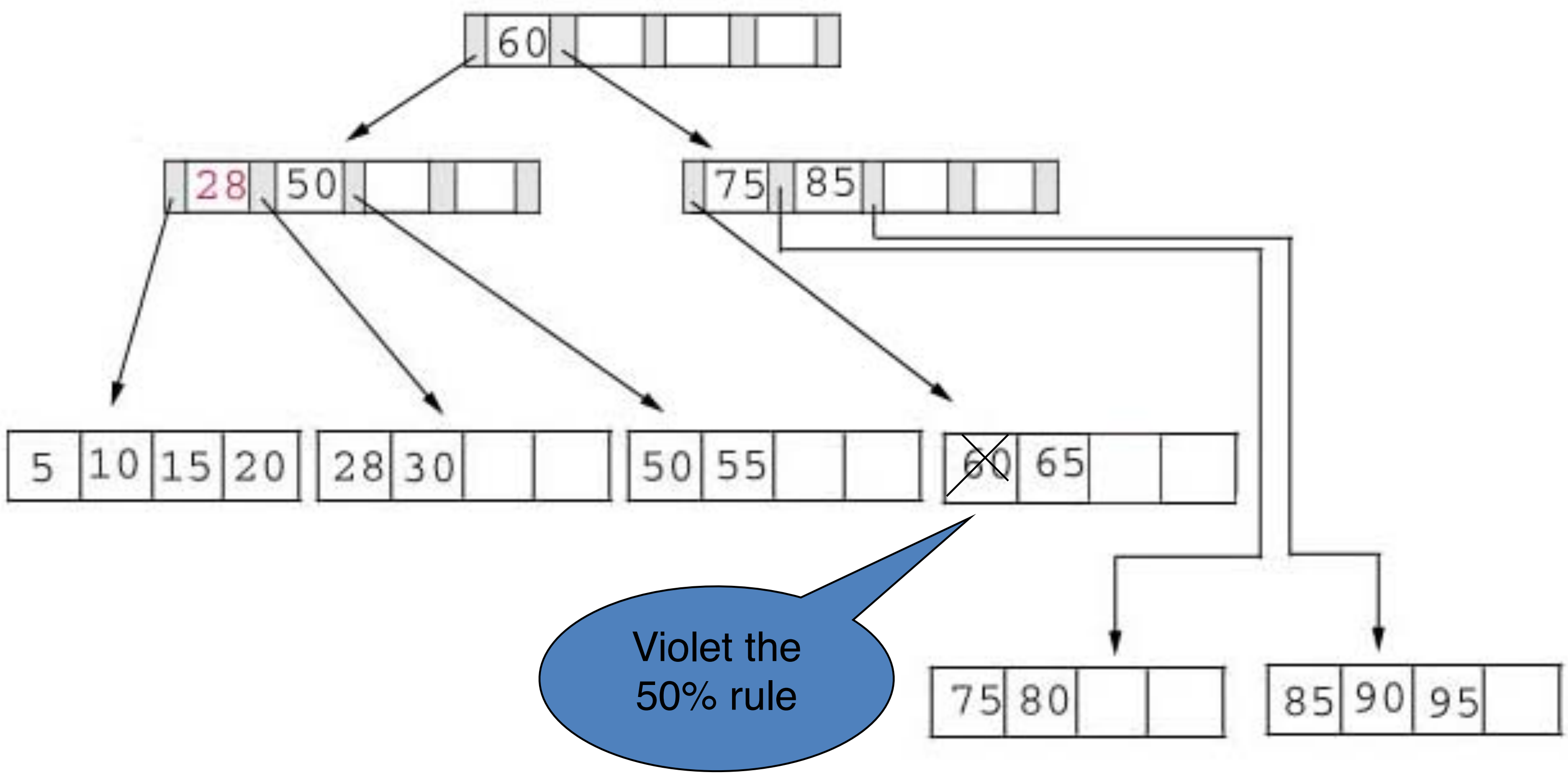
Exercise: Deletion

- Same as insertion, the tree has to be rebuild if the deletion result violate the rule of B+ tree.
- Example #1: delete 70 from the tree



Deletion

- Example #3: delete 60 from the below tree



Exercise

- Build a B^+-tree of fan-out 5 created by these data:
  - 3, 7, 9, 23, 45, 1, 5, 14, 25, 24, 13, 11, 8, 19, 4, 31, 35, 56
- Add these further keys: 2, 6, 12
- Delete these keys: 4, 5, 7, 3, 14

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 nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

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

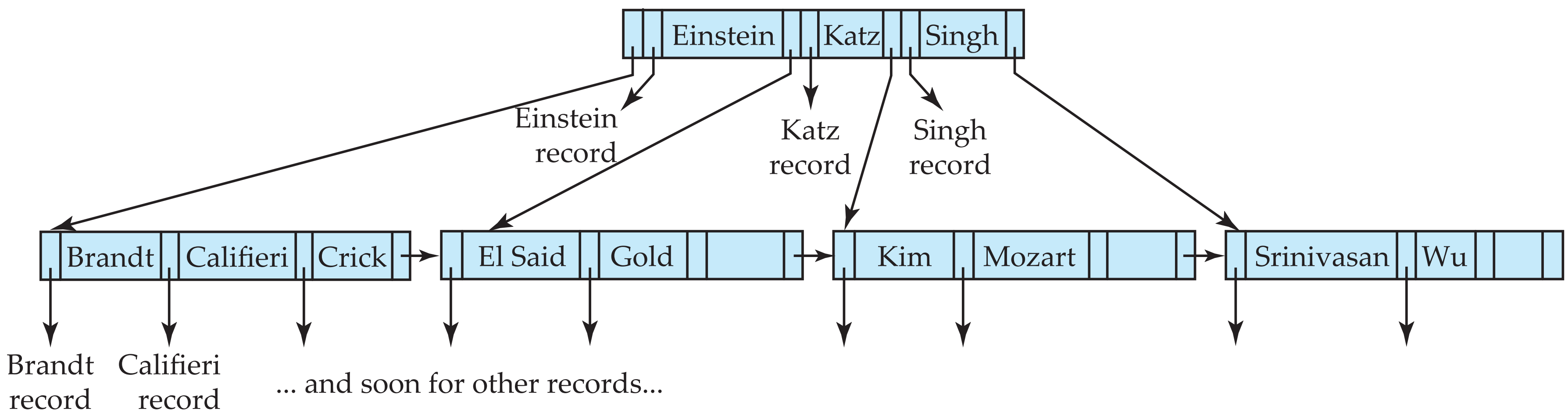
(a)

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

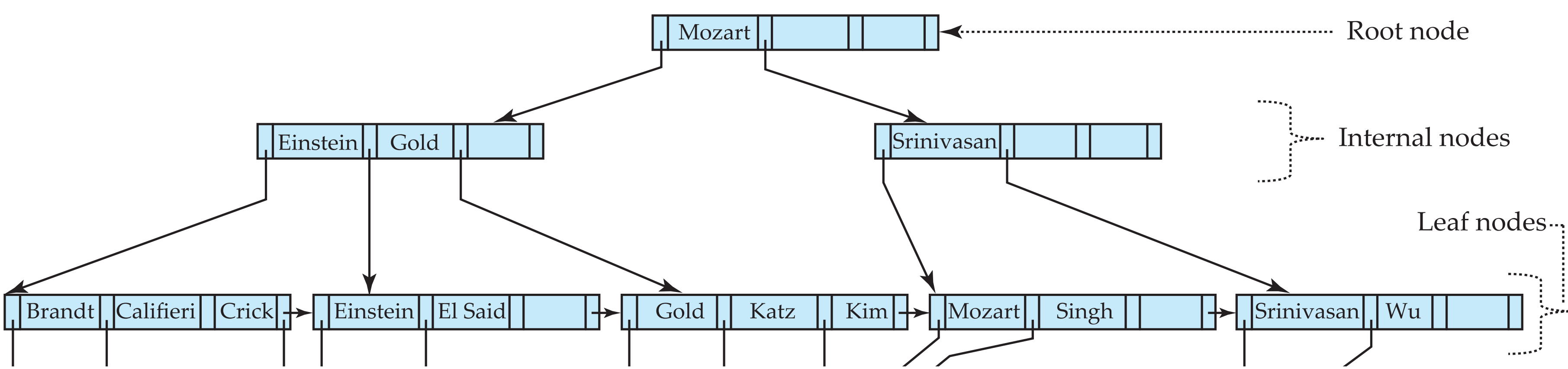
(b)

- Nonleaf node – pointers Bi are the bucket or file record pointers.

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.
  - 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 weigh disadvantages.