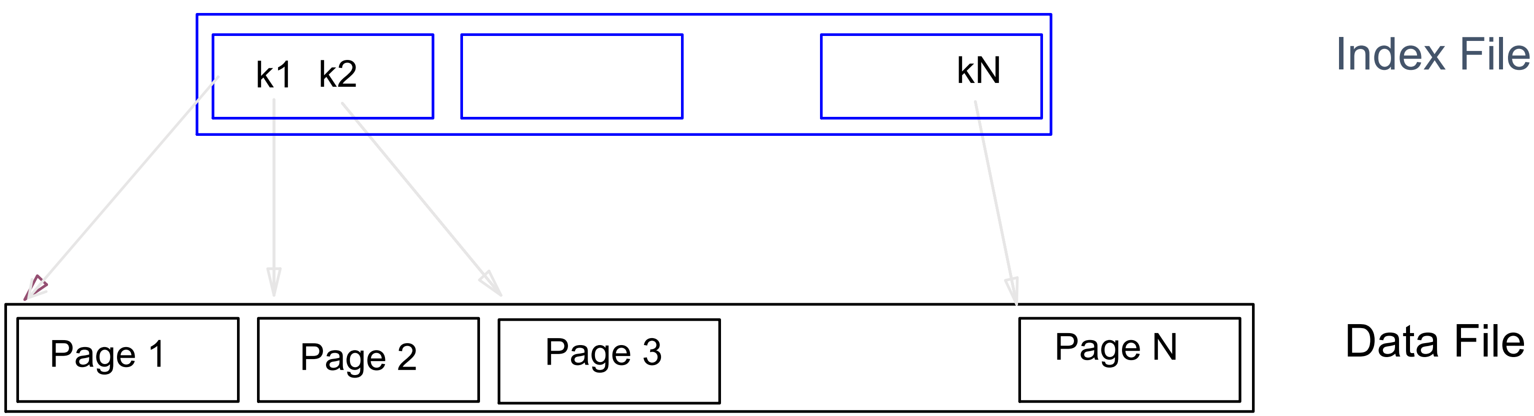
Indexing Structures
B+-tree, Hashing

Range Searches

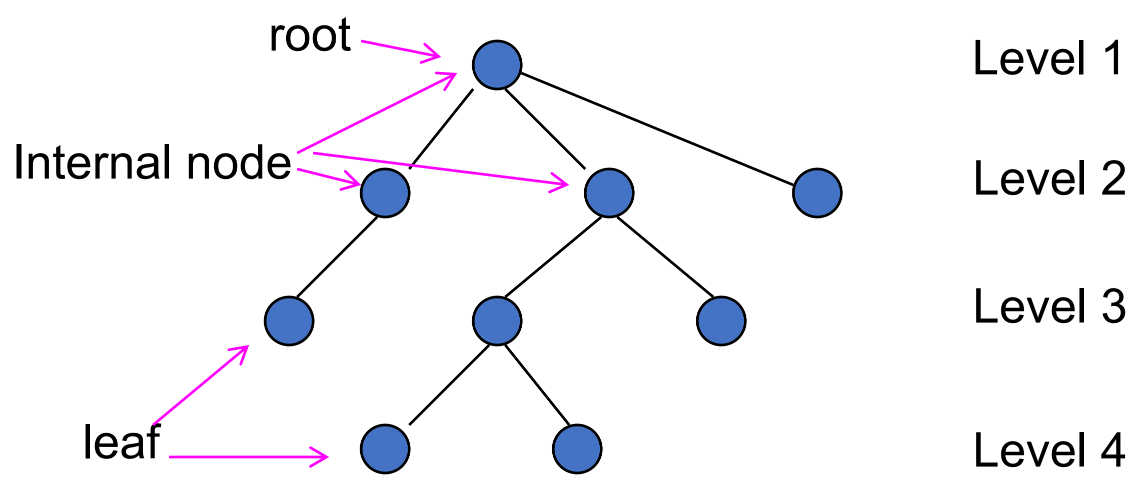
- *``Find all students with gpa > 3.0''*
  - If records are sorted on gpa, do binary search to find first such student, then scan to find others.
  - Cost of binary search can be quite high.
- Simple idea: Create an `index' file.



☛ *Can do binary search on (smaller) index file!*

B+ Tree: Most Widely Used Index

- General concept of a tree:

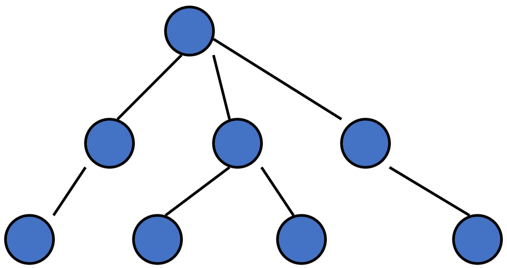


I. Height of a node: its distance to the root

II. If a higher level node is connected to a lower level node, then the higher level node is called a parent (grandparent, ancestor, etc.) of the lower level node

B+ Tree (cont.)

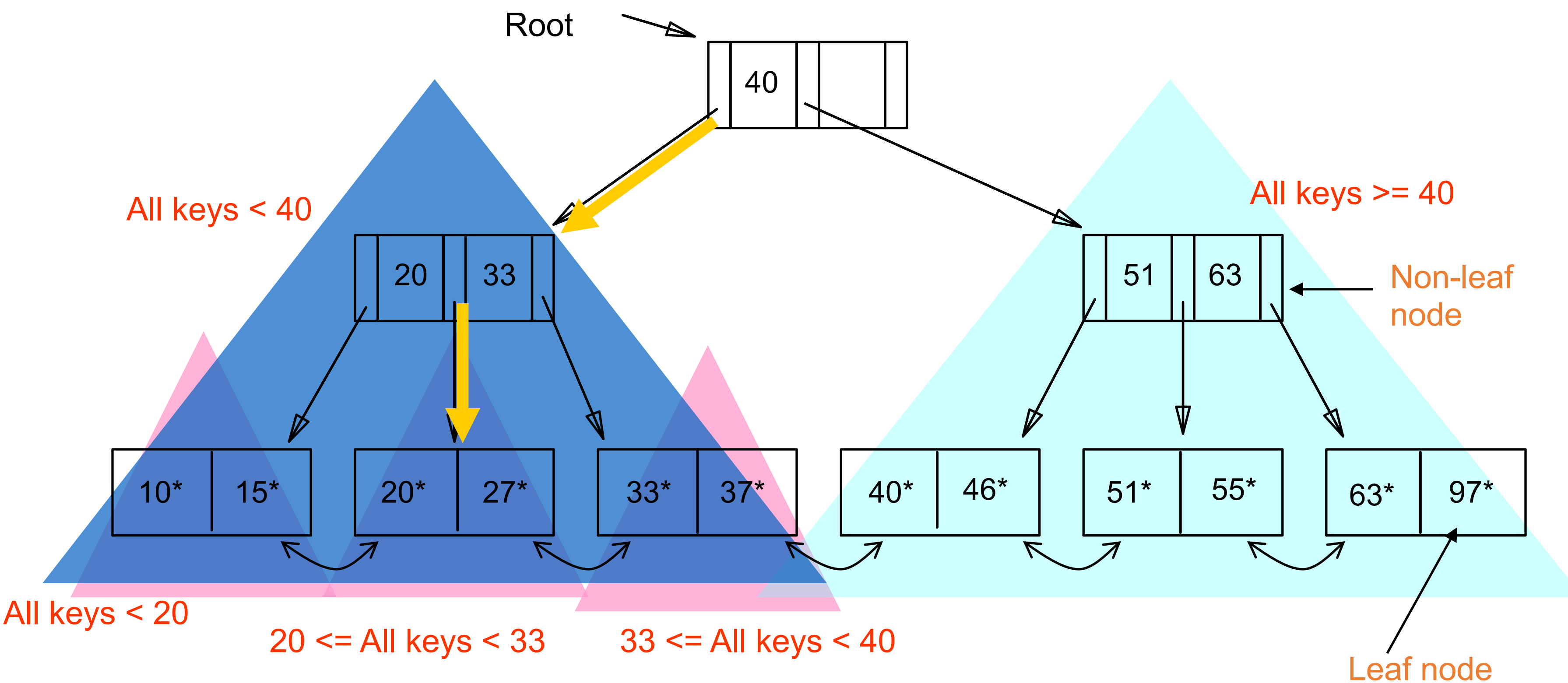
- Balanced tree: all leaves at the same level
  - Example:



- Structure of a B+ tree:
  - It is ***balanced:*** all leaf nodes are at the same level
  - Each node has a special structure

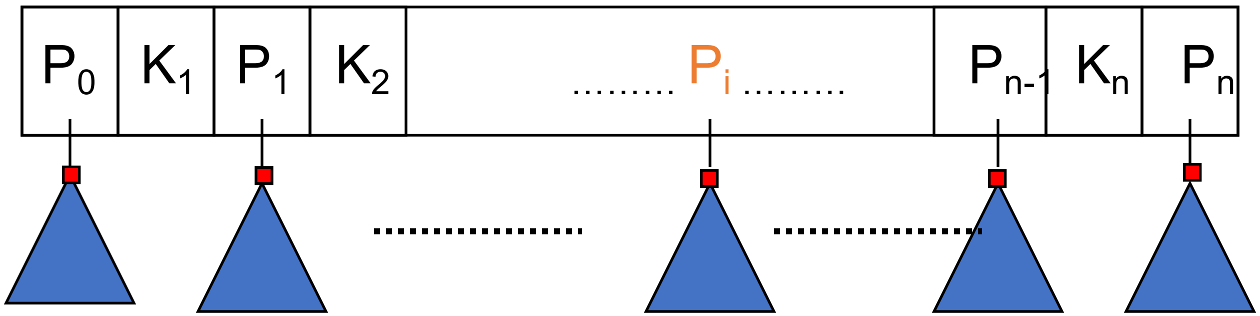
Example: A B+ tree with order of 1

- Each node must hold at least 1 entry, and at most 2 entries



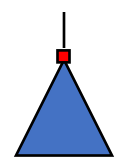
- Given search key values 27 how to find the rids?
  - Search begins at the root, and key comparisons direct it to a leaf
  - Note that key values are sorted at each level

B+ tree: Internal node structure

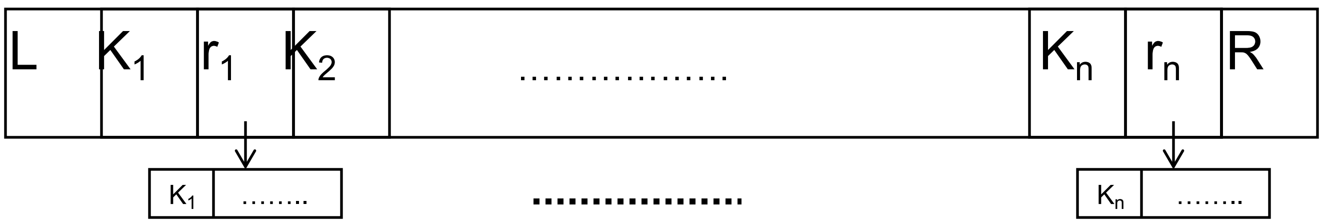


Each P_i is a pointer to a child node, each K_i is a search key value
Pointers out number search key values by *exactly one.*

- $K_1 < K_2 < \ldots < K_n$
- If the node is not the root, we require $d \leq n \leq 2d$ where d is a pre-determined value for this B+ tree, called its *order*
- If the node is the root, $1 \leq n \leq 2d$

- For any search key value K in the subtree  pointed by P_i,
  - If $P_i = P_0$, we require that $K < K_1$
  - If $P_i = P_1, \ldots, P_{n-1}$, we require that $K_i \leq K < K_{i+1}$
  - If $P_i = P_n$, we require $K_n \leq K$

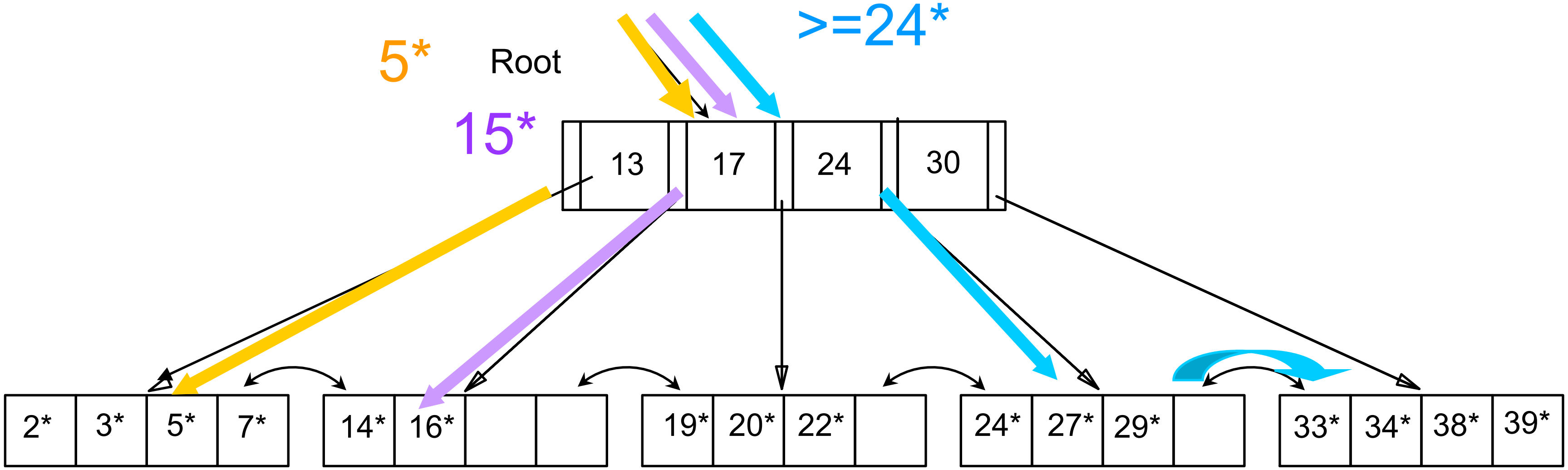
B+ tree: leaf node structure



- Each r_i is a pointer to a record that contains search key value K_i
- L points to the left neighbor, and R points to the right neighbor
- $K_1 < K_2 < \ldots < K_n$
- $d \leq n \leq 2d$ where d is the order of this B+ tree
- We will use K_i* for the pair (K_i, r_i) and omit L and R for simplicity in the remaining slides

Example: a B+ tree with order 2

- Search for 5*, 15*, all data entries >= 24* ...
- The last one is a range search, we need to do the sequential scan, starting from the first leaf containing a value >= 24.

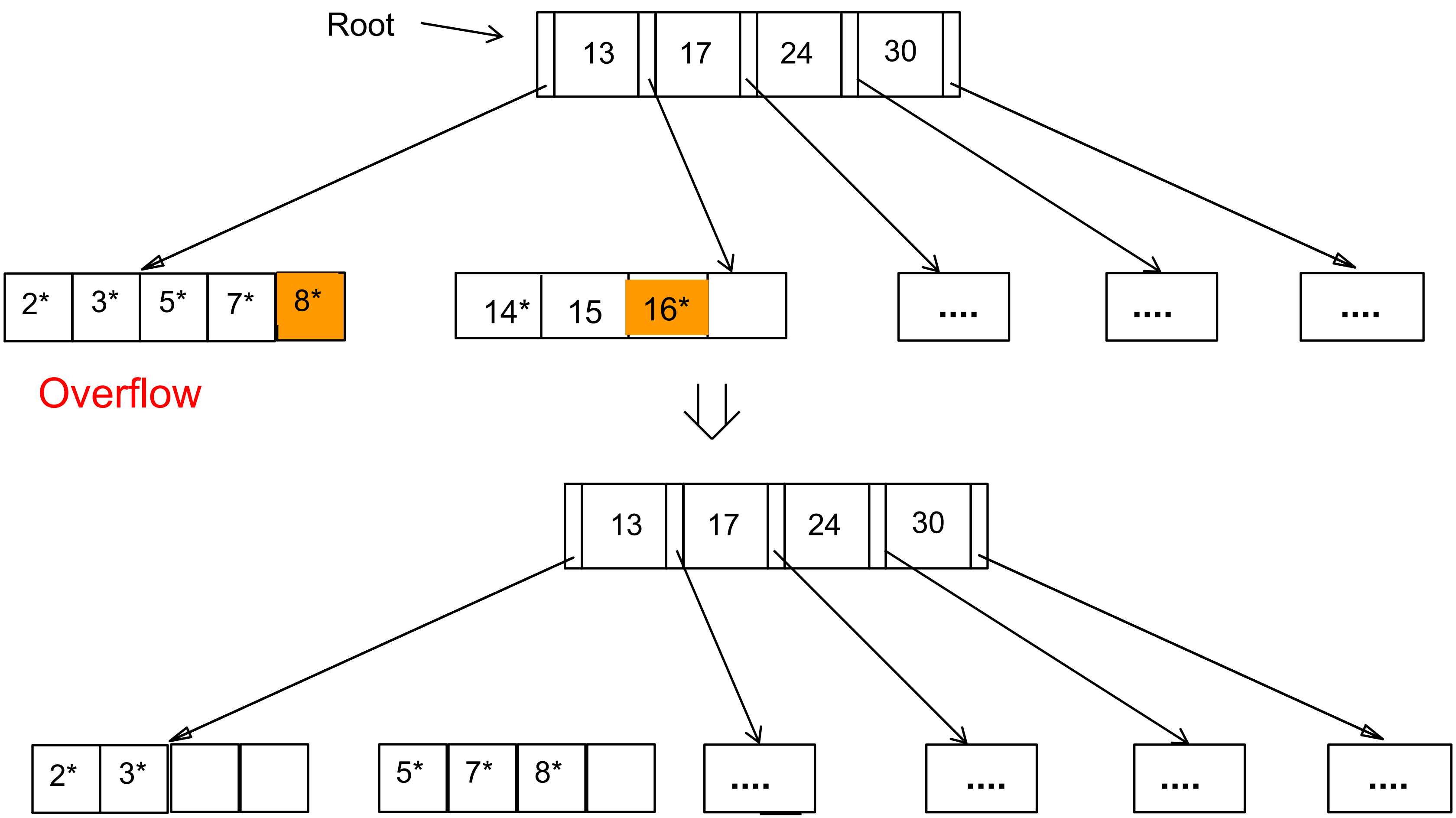


Inserting a Data Entry into a B+ Tree

- Find correct leaf *L.*
- Put data entry onto L.
  - If L has enough space, done!
  - Else, must split L (into L and a new node L2)
    - Redistribute entries evenly, put middle key in L2
    - **copy up** middle key.
    - Insert index entry pointing to L2 into parent of L.
- This can happen recursively
  - To split index node, redistribute entries evenly,
  - but **push up** middle key. (Contrast with leaf splits.)

- Splits "grow" tree; root split increases height.
  - Tree growth: gets wider or one level taller at top.

Inserting 16*, 8* into Example B+ tree



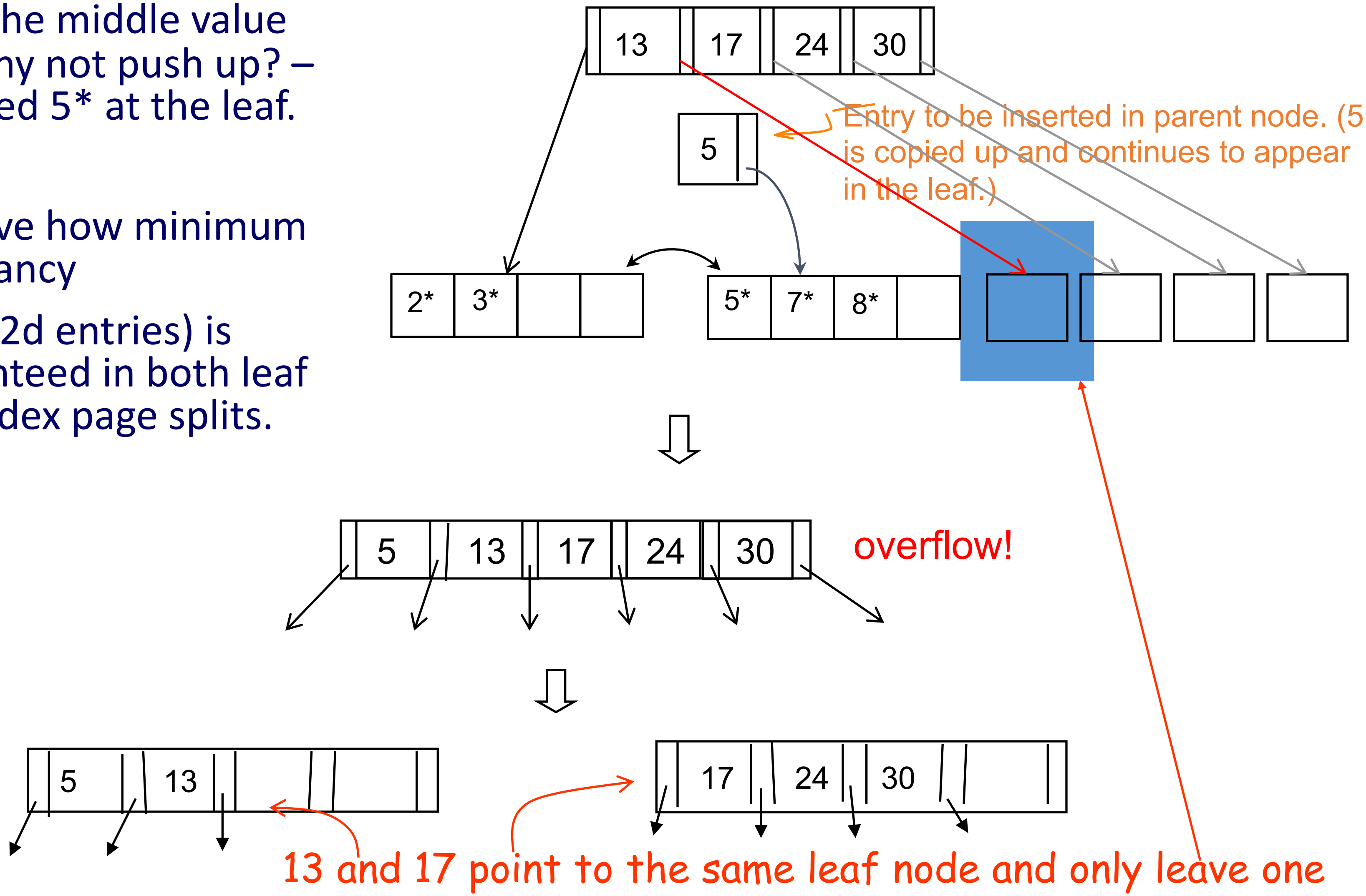
One more child generated, must add one more pointer to its parent, thus one more key value as well.

Inserting 8* into Example B+ Tree (order 2)

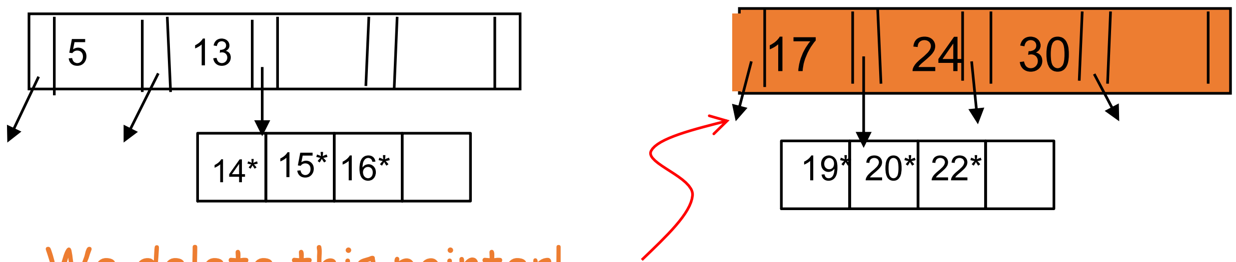
- Copy the middle value up. Why not push up? – we need 5* at the leaf.

- Observe how minimum occupancy

  (2 to 2d entries) is guaranteed in both leaf and index page splits.



Insertion into B+ tree (cont.)



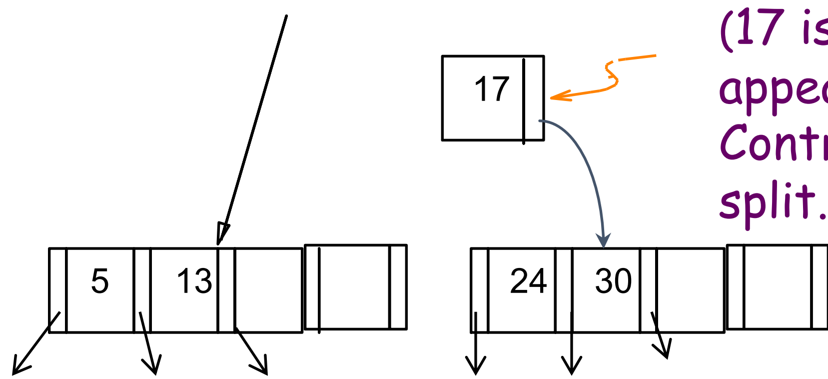
- We delete this pointer!

17 <= k

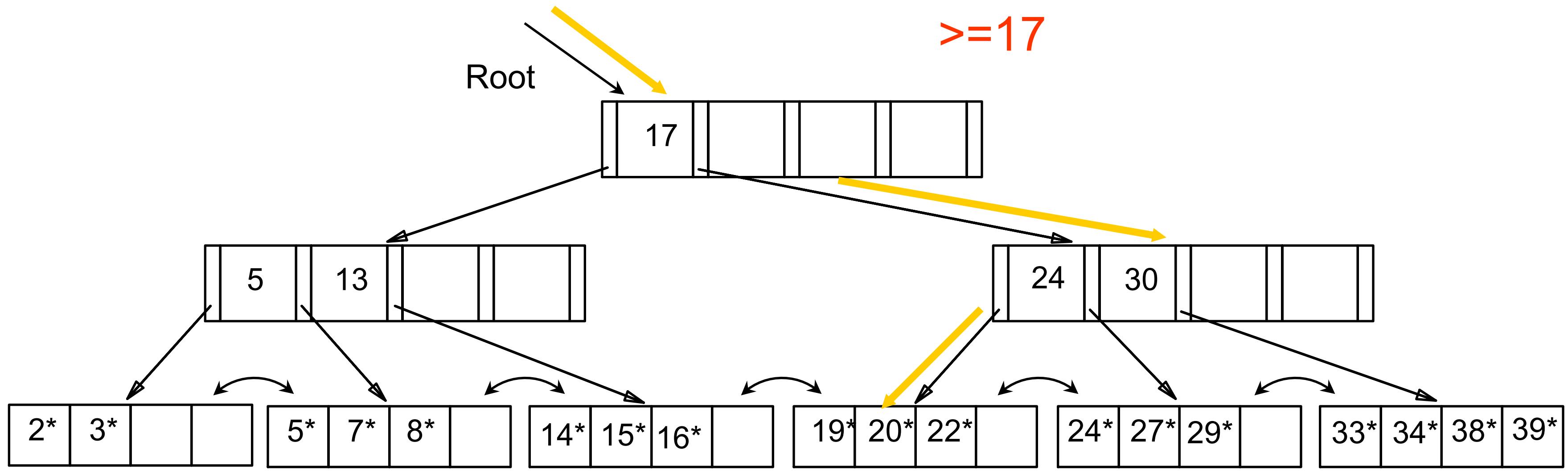
- For internal nodes, we only have one copy of key values. Thus, 17 is deleted and push up to the parent node.

Entry to be inserted in parent node.

(17 is pushed up and only appears once in the index. Contrast this with a leaf split.)



Example B+ Tree After Inserting 8*



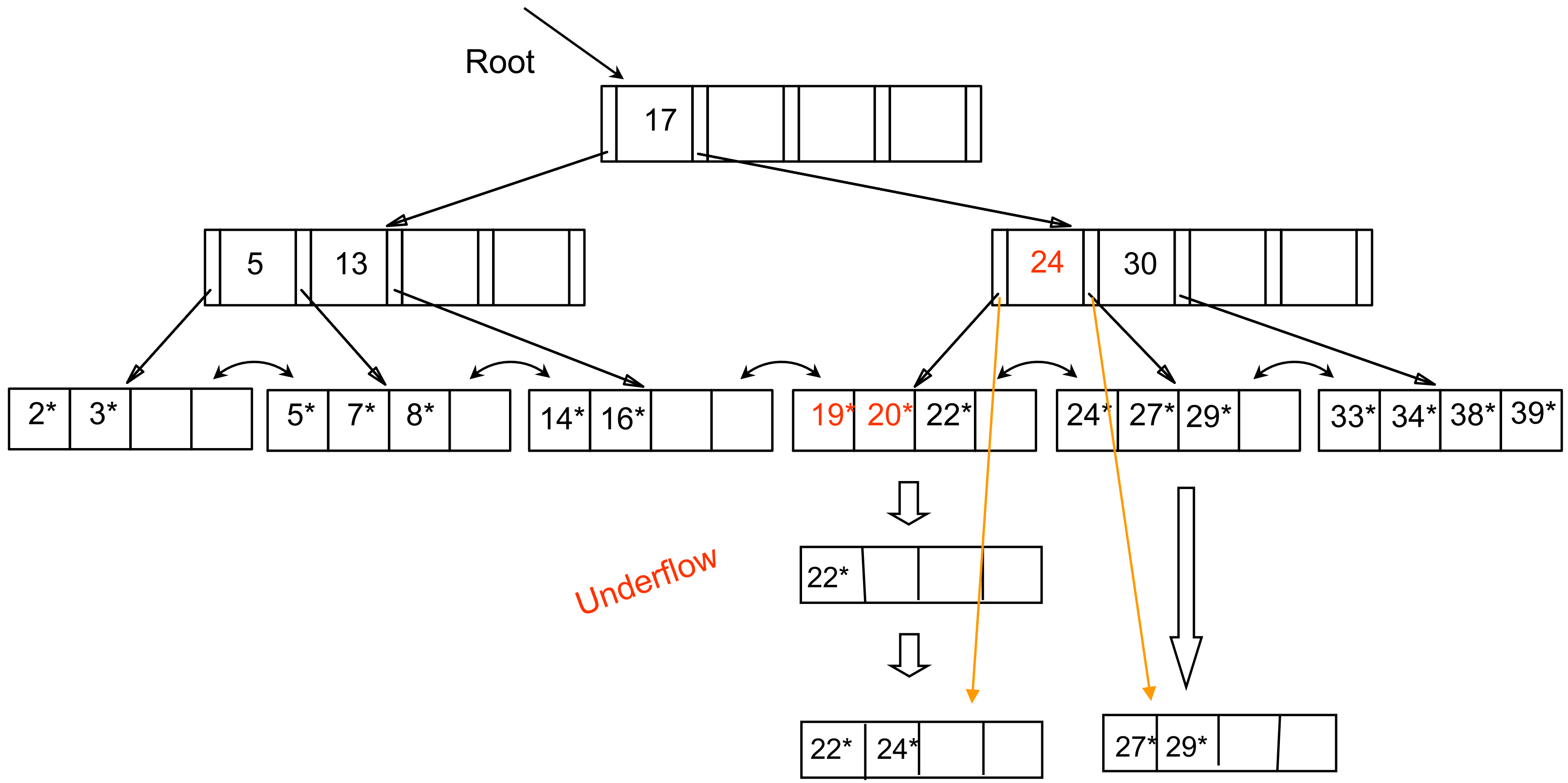
- the root was split, leading to increase in height.

- In this example, we can avoid splitting by re-distributing entries.

Deleting a Data Entry from a B+ Tree

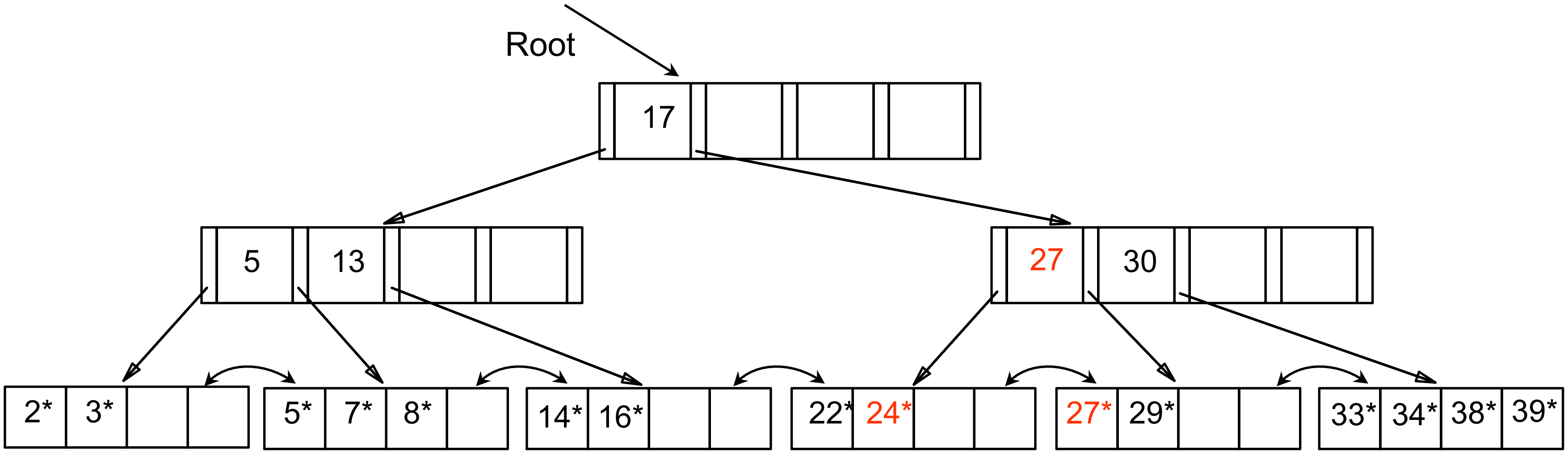
- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has only d-1 entries,
    - Try to re-distribute, borrowing from sibling (adjacent node with the same parent as L).
    - If re-distribution fails, merge L and sibling.
- If merge occurred,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

Delete 19* and 20*



Have we still forgotten something?
The left child key values should be smaller than 24.

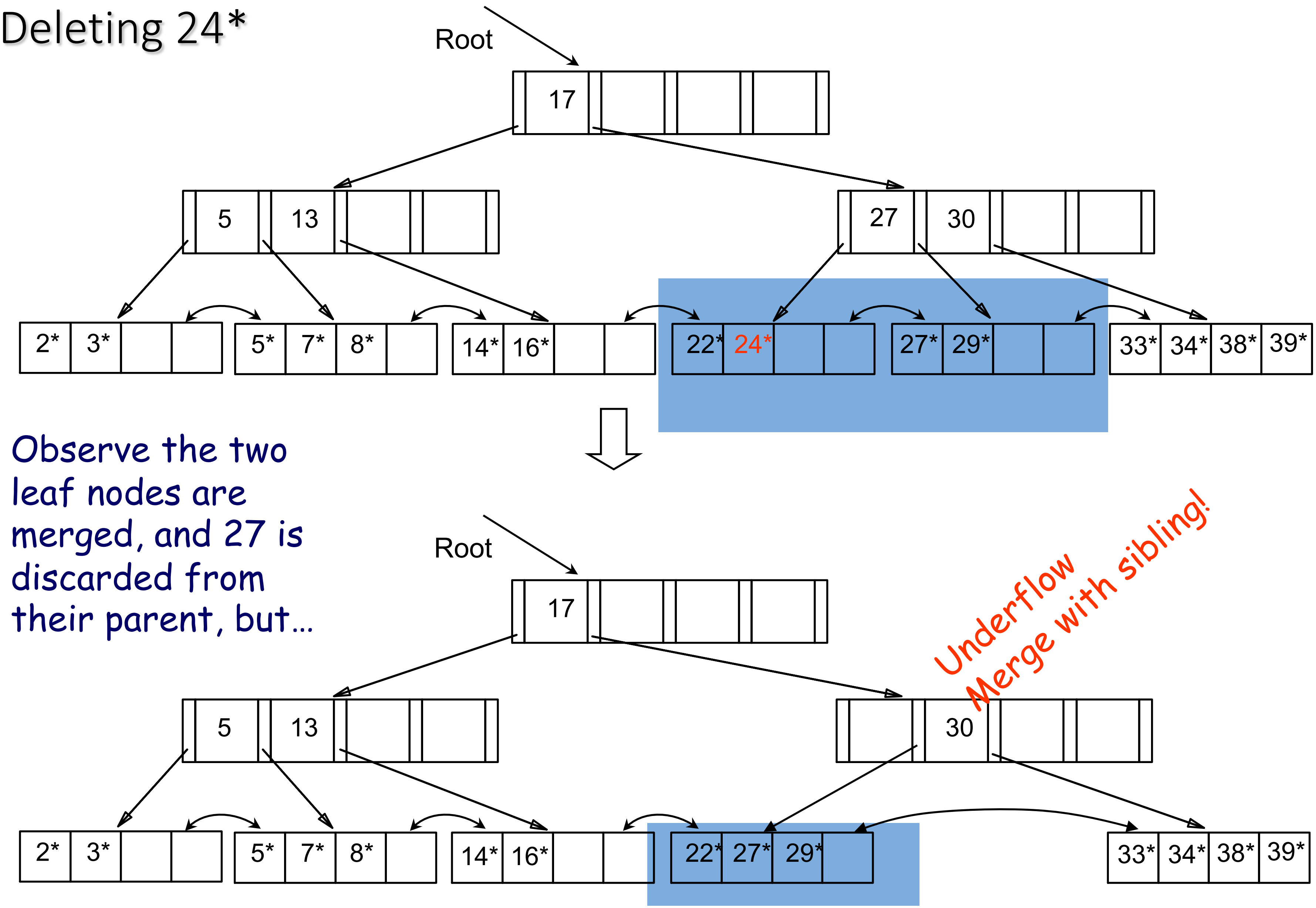
Deleting 19* and 20* (cont.)



- Notice how 27 is copied up.

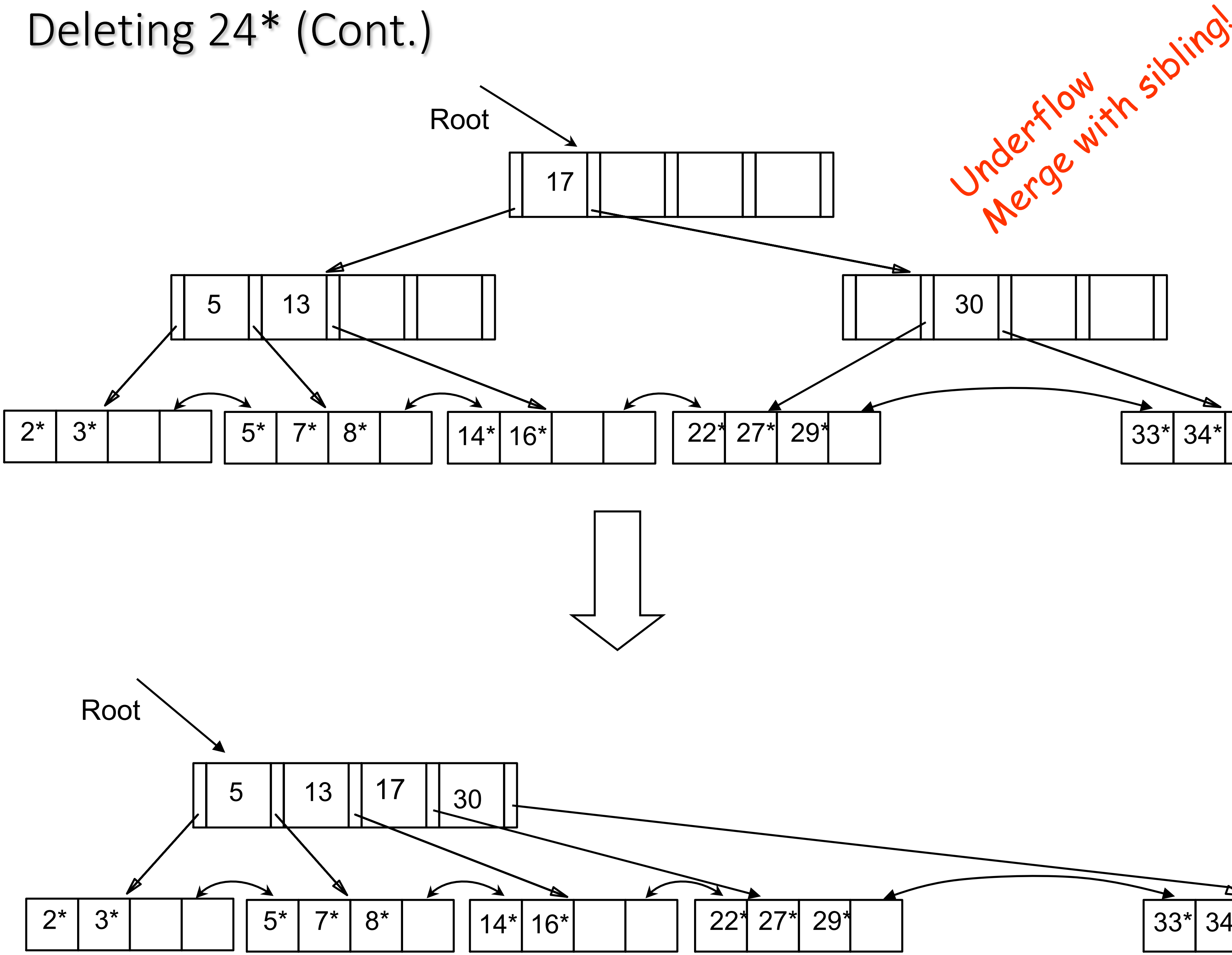
- Now we want to delete 24
- Underflow again! But can we redistribute this time?(Leave it to you)

Deleting 24*



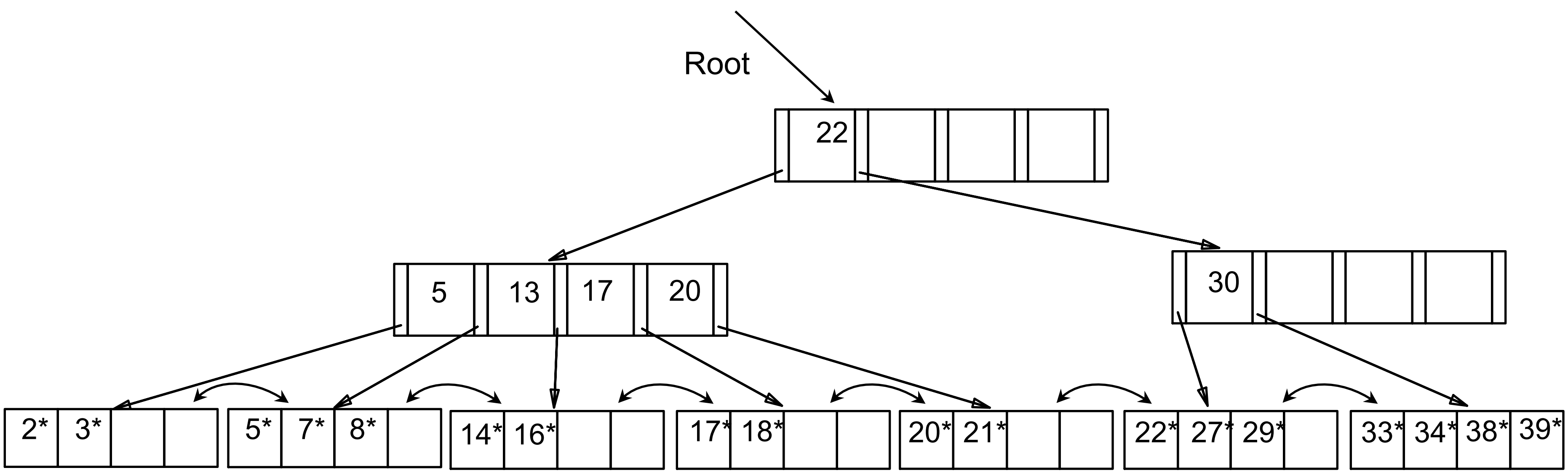
Observe the two leaf nodes are merged, and 27 is discarded from their parent, but…

Deleting 24* (Cont.)

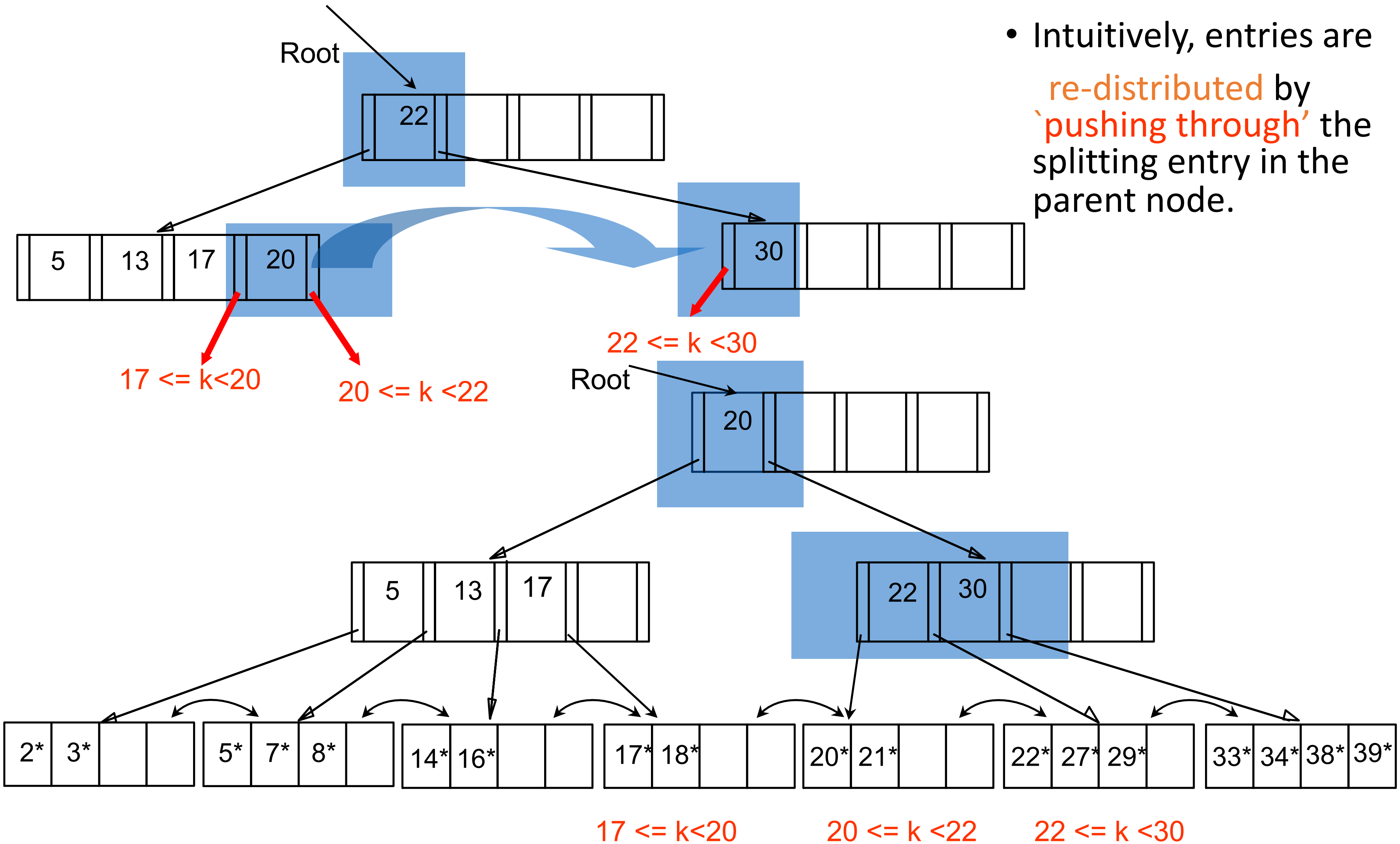


Example of Non-leaf Re-distribution

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

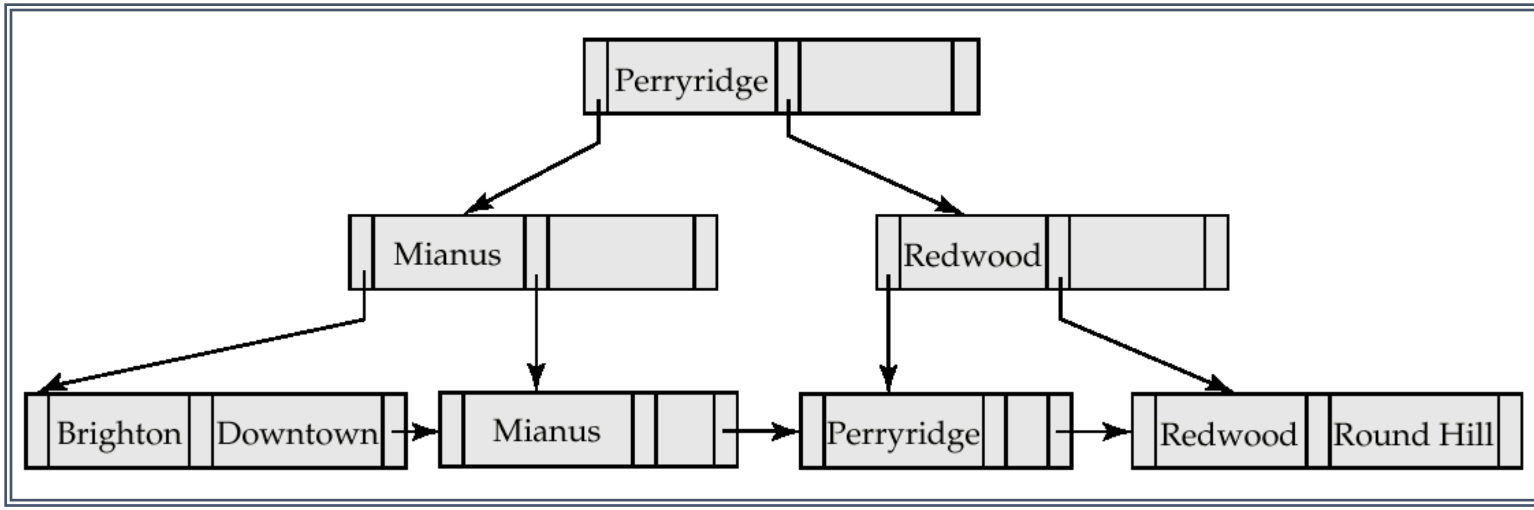


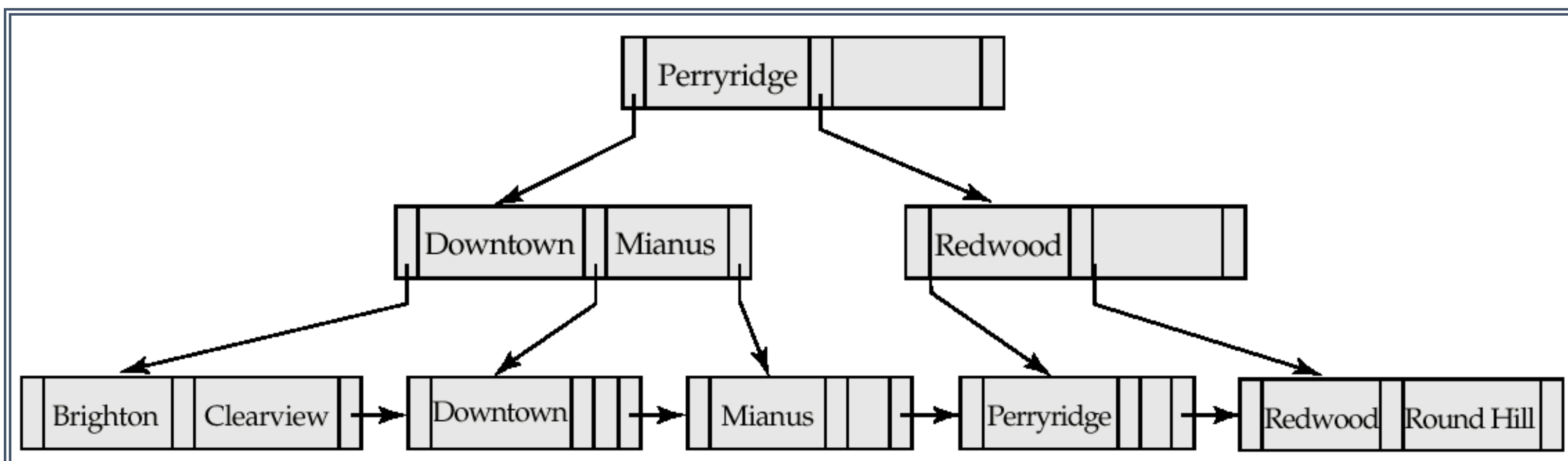
Non-leaf Re-distribution (Cont.)



- Intuitively, entries are re-distributed by `pushing through' the splitting entry in the parent node.

B+-Trees: Insertion



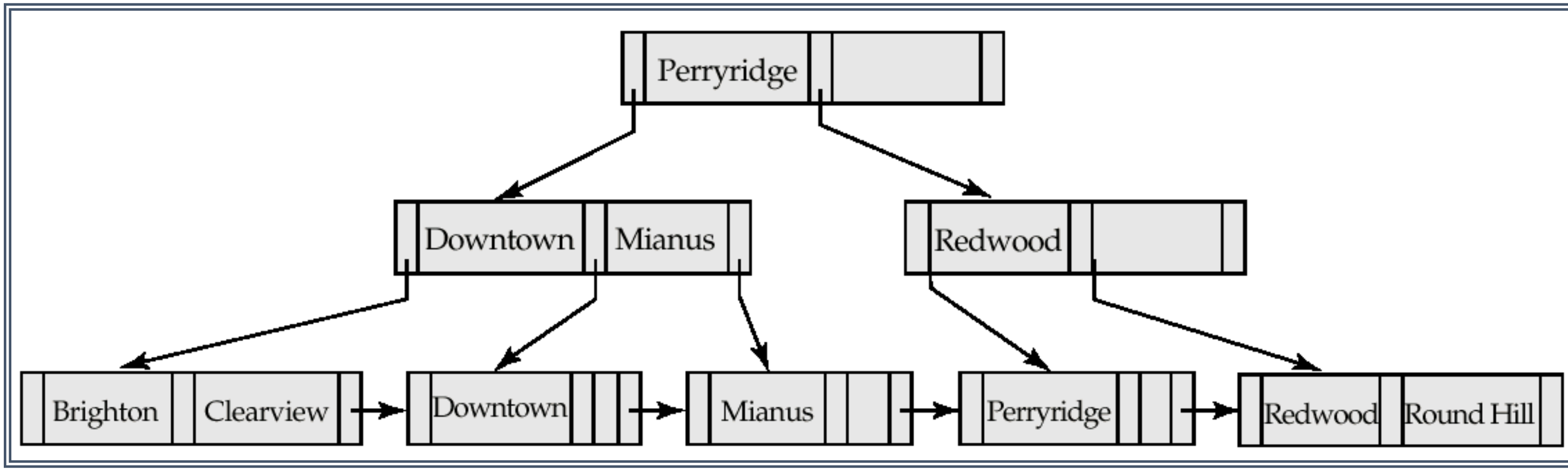


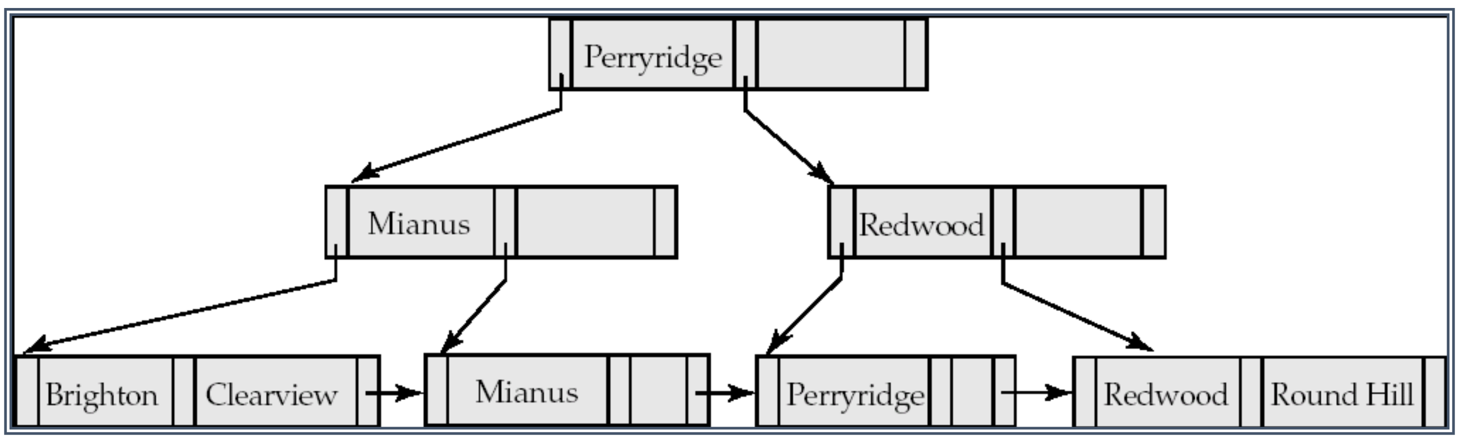
B+-Tree before and after insertion of “Clearview”

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

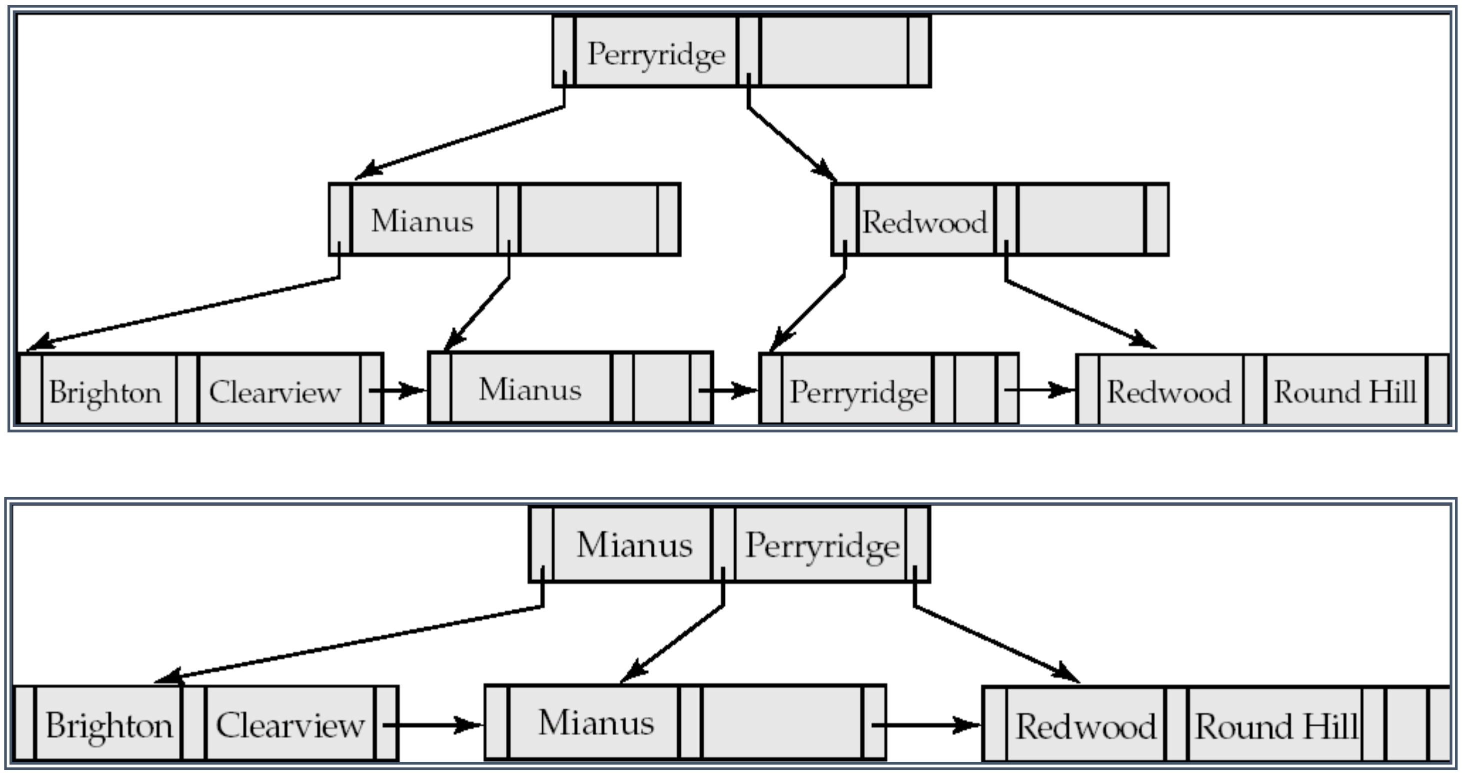




Before and after deleting "Downtown"

- 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

B+ Trees in Practice

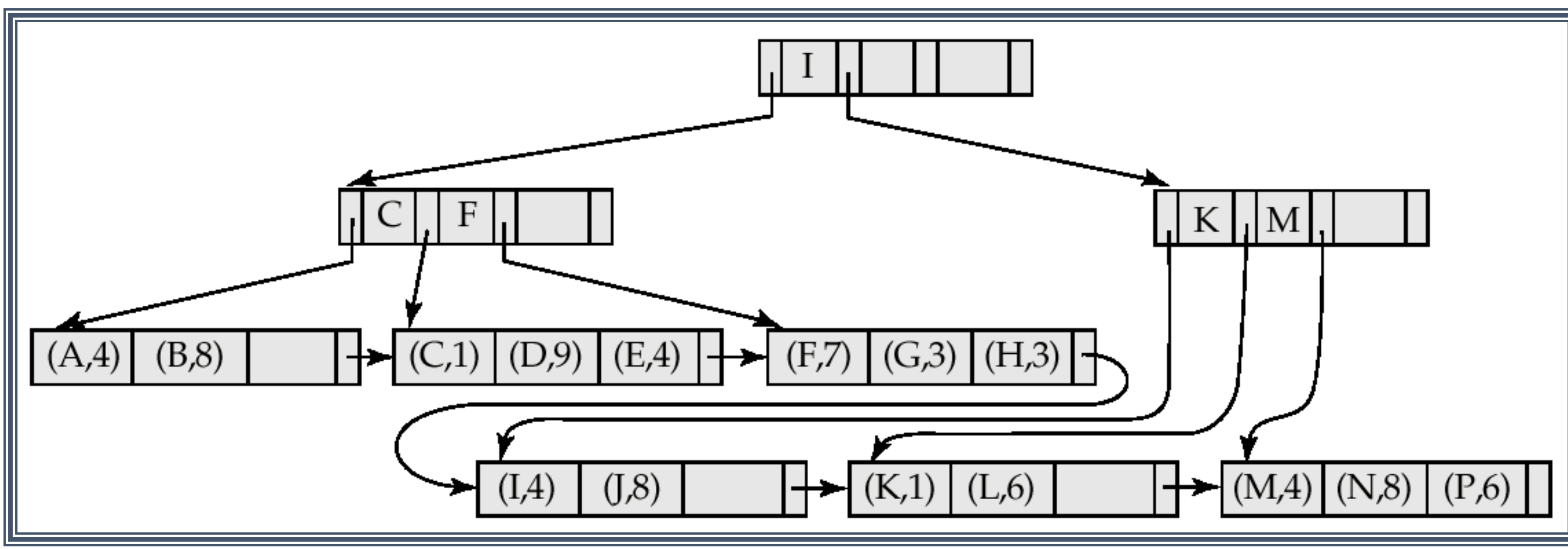
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 3: 133^3 = 2,352,637 entries
  - Height 4: 133^4 = 312,900,700 entries
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 =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

B+-Tree File Organization

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



Hash-based File Organization

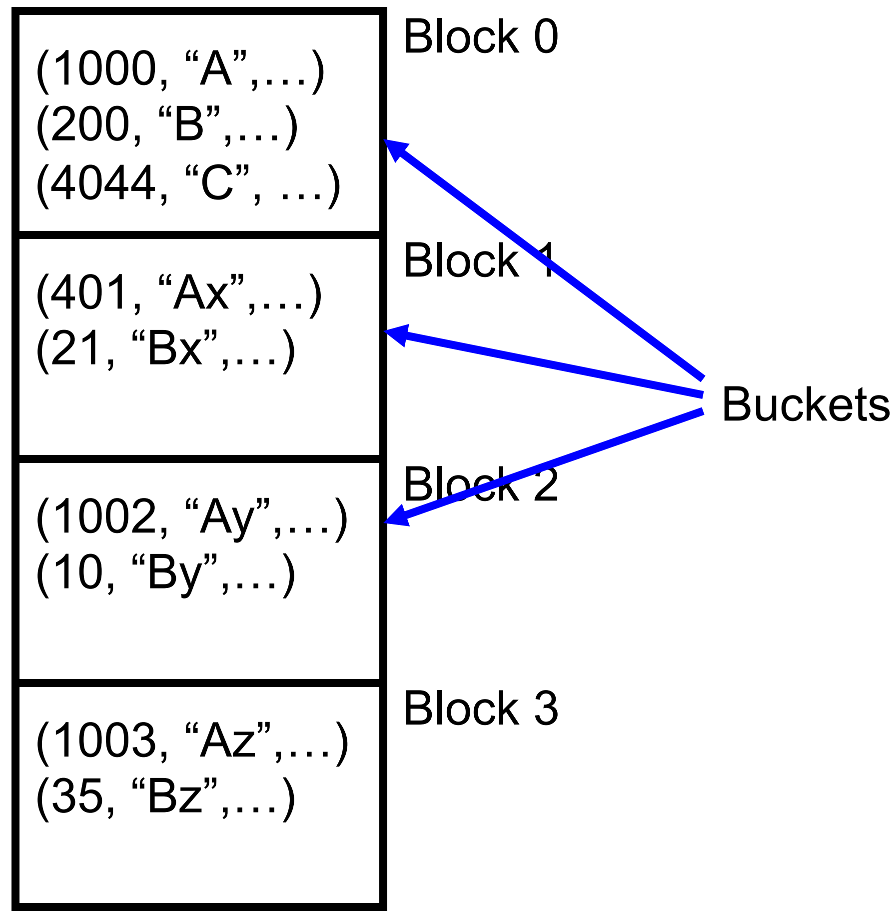
Store record with search key *k* in block number *h(k)*

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

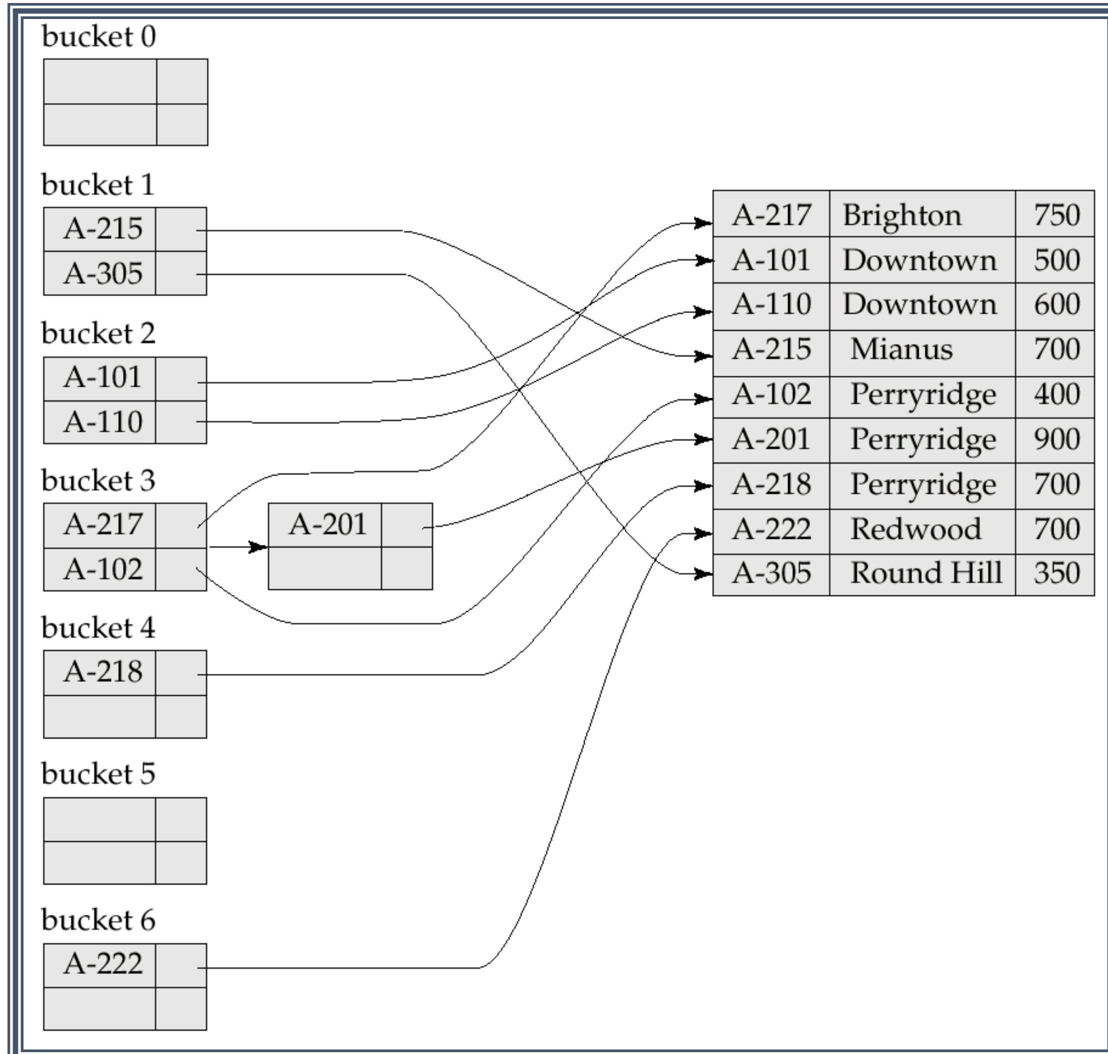
Search:

Find the block with search key

Follow the pointer

Range search ?

a < X < b ?

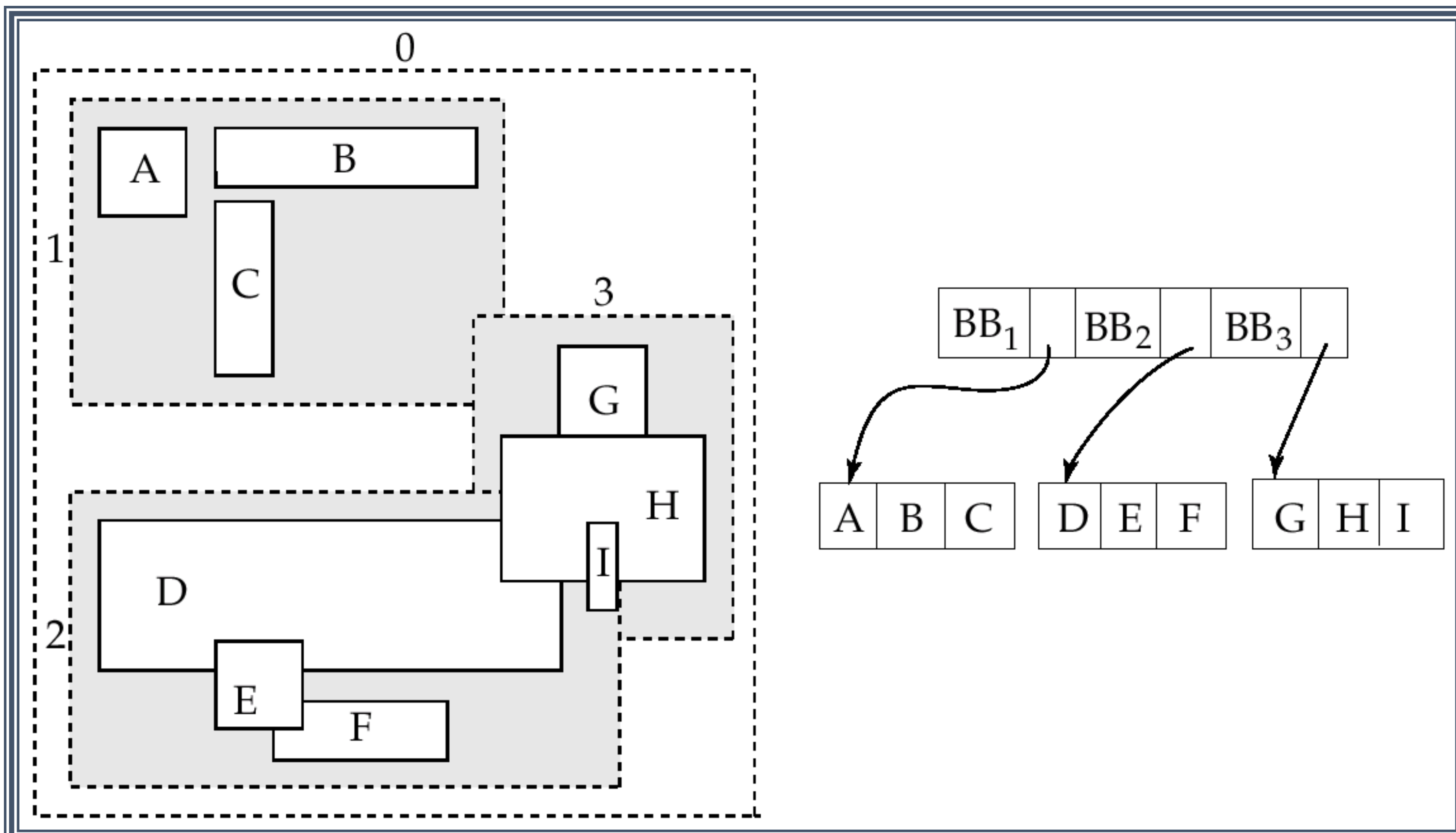


Hash Indexes

- 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

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 exists
  - For different types of data
  - For different types of queries
    - E.g. "nearest-neighbor" queries