Access Methods I: Tree-based indexes

"If you don't find it in the index, look very carefully through the entire catalogue."

-- Sears, Roebuck, and Co., Consumer's Guide, 1897

Example

Relation R

name	age	weight
Moe	10	55
Curly	10	65
Larry	12	70
Bob	15	60
Alice	17	48
Lucy	17	45
John	17	59
Charlie	20	69
Marcie	22	50
Linus	23	60
Sally	23	48
Tom	25	56

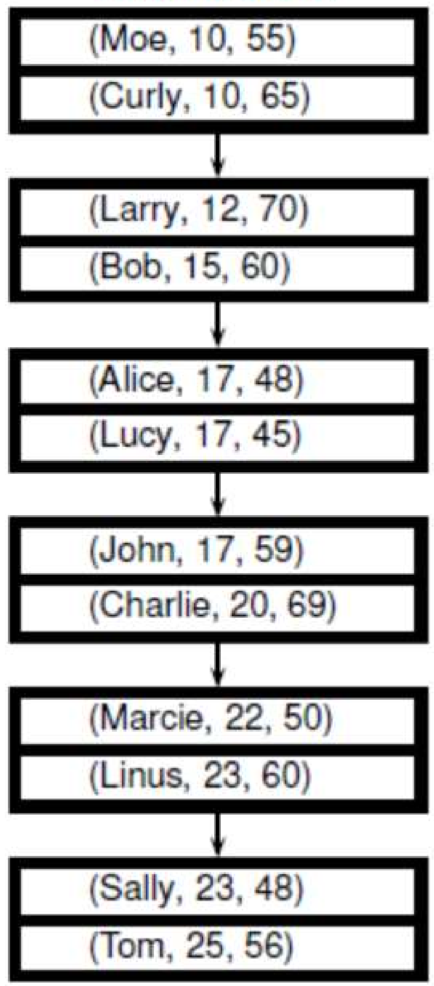
SELECT * FROM R WHERE weight BETWEEN 50 AND 55

Example

Relation R

name	age	weight
Moe	10	55
Curly	10	65
Larry	12	70
Bob	15	60
Alice	17	48
Lucy	17	45
John	17	59
Charlie	20	69
Marcie	22	50
Linus	23	60
Sally	23	48
Tom	25	56

Data File for R



Records organized into pages on disk (e.g., 2 records per page; pages are chained up)

SELECT * FROM R WHERE weight BETWEEN 50 AND 55

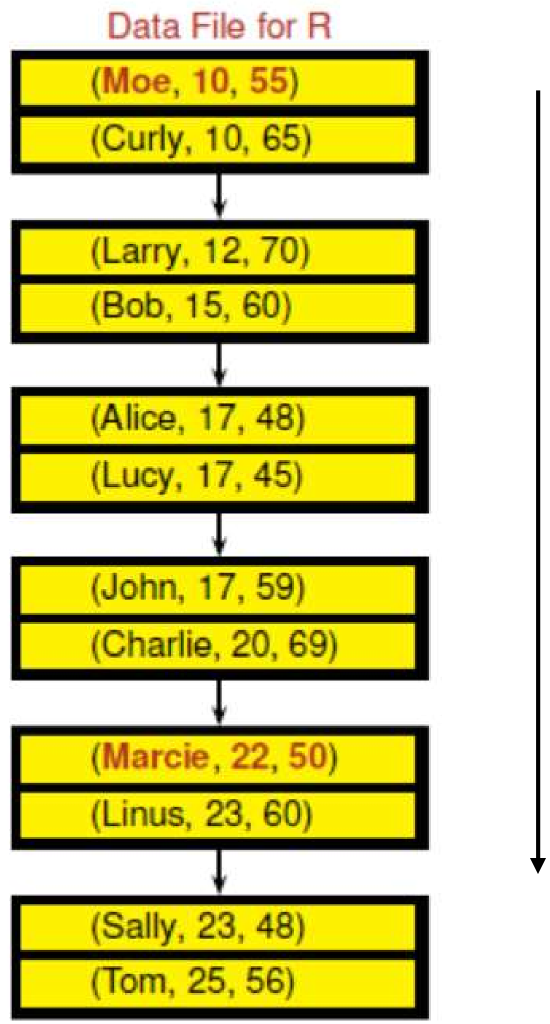
If each page is 2 records, then how many pages need to be scanned to find the correct answer for the query

- Since not sure if done searching
- This will take 6 I/O since there are 6 pages

Example

Relation R

name	age	weight
Moe	10	55
Curly	10	65
Larry	12	70
Bob	15	60
Alice	17	48
Lucy	17	45
John	17	59
Charlie	20	69
Marcie	22	50
Linus	23	60
Sally	23	48
Tom	25	56



Scan the entire relation

SELECT * FROM R WHERE weight BETWEEN 50 AND 55

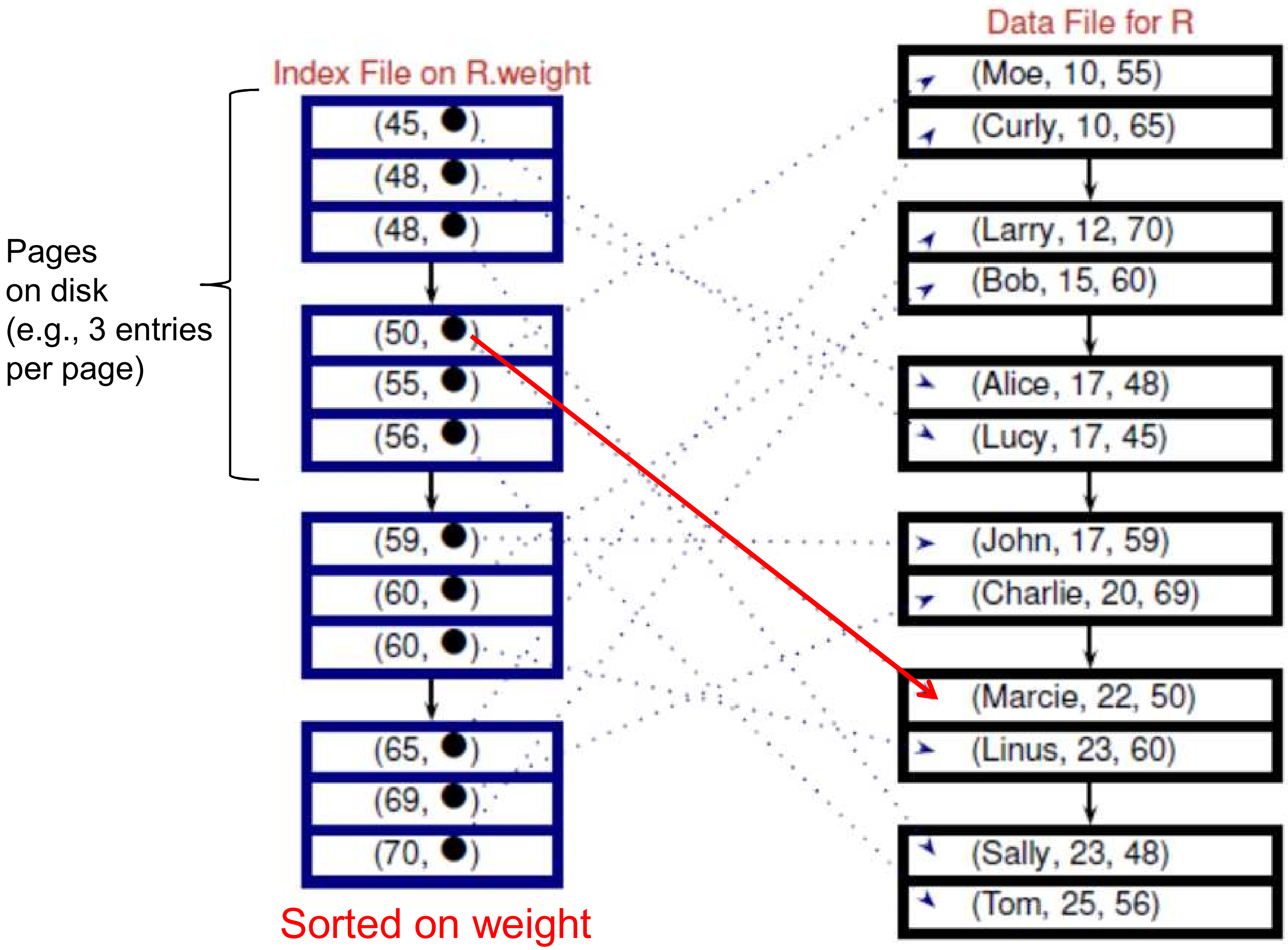
Single Record and Range Searches

- Single record retrievals
  - *``Find student name whose matric# = 921000Y13″*
- Range queries
  - *``Find all students with 2.0 < cap < 2.5″*
- Sequentially scanning the file is costly
- If data is sorted in the search condition
  - Can stop once you find the desired record(s)

Indexes

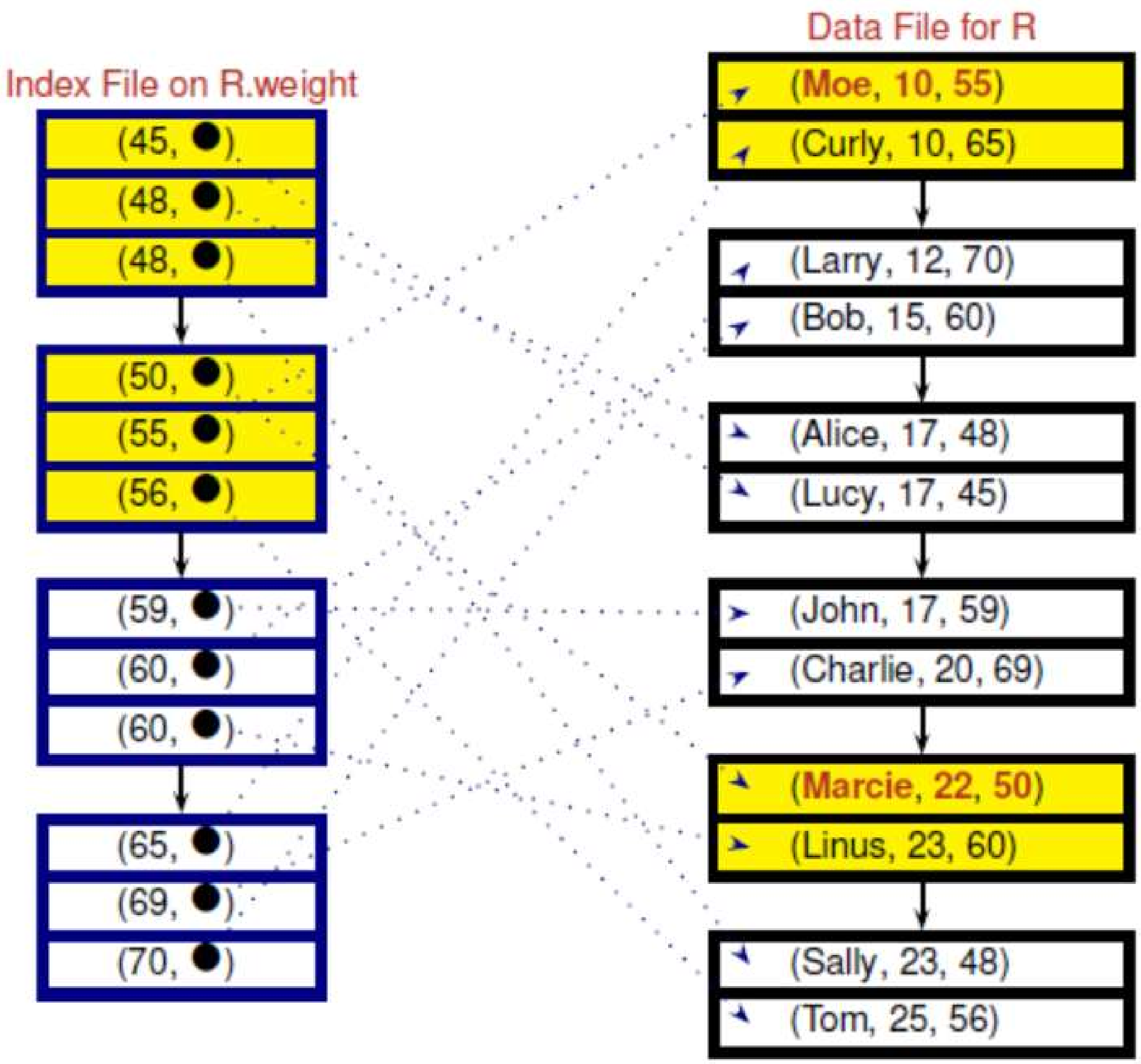
- An index is a data structure on a file to speed up retrieval/selections based on some ***search*** key
  - Any subset of the fields of a relation can be the search key for an index on the relation
  - *Search key* is **NOT** the same as *key* (minimal set of fields that uniquely identify a record in a relation)
    - e.g., consider Student(<u>matric#,</u> name, addr, cap), the key is matric#, but the search key can be matric#, name, addr, cap or any combination of them (e.g., (name, address))
  - For each search key, you can build an index, i.e., there can be multiple indexes on a single relation that provides different access paths
- An index is a unique index if its search key is a candidate key; otherwise, it is a non-unique index
- An index is stored as a file
  - Records in an index file are referred to as data entries

Simple Index File: Unsorted Data File



(50, •) is the data entry for the data record (Marcie, 22, 50) thius this means that only need to sort the index file instead of the entire data file

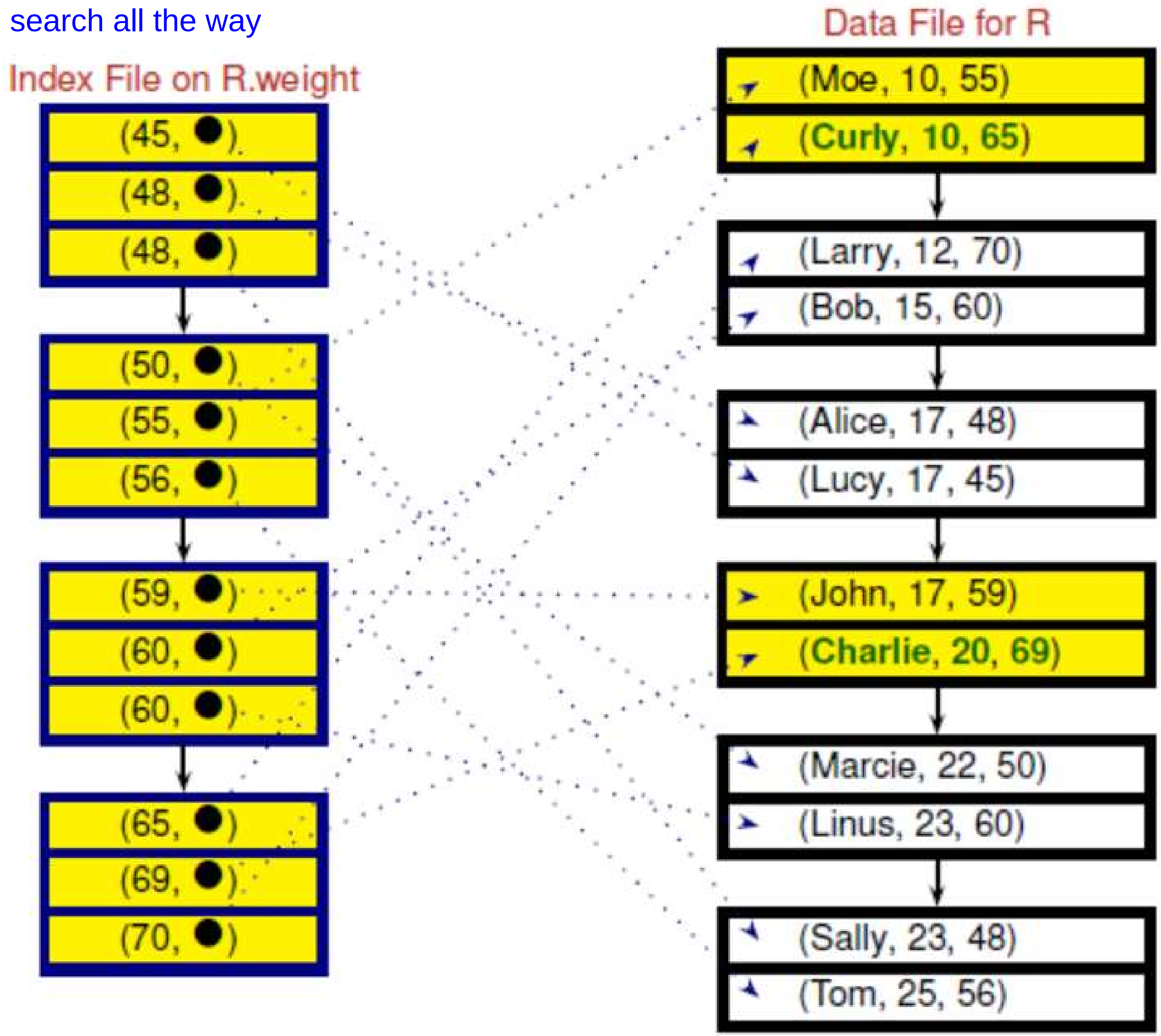
Simple Index File



SELECT * FROM R WHERE weight BETWEEN 50 AND 55

but this might not be the best method
(sorting by weight)
since if the weight required is all the way at
the bottom then still need to search all the way

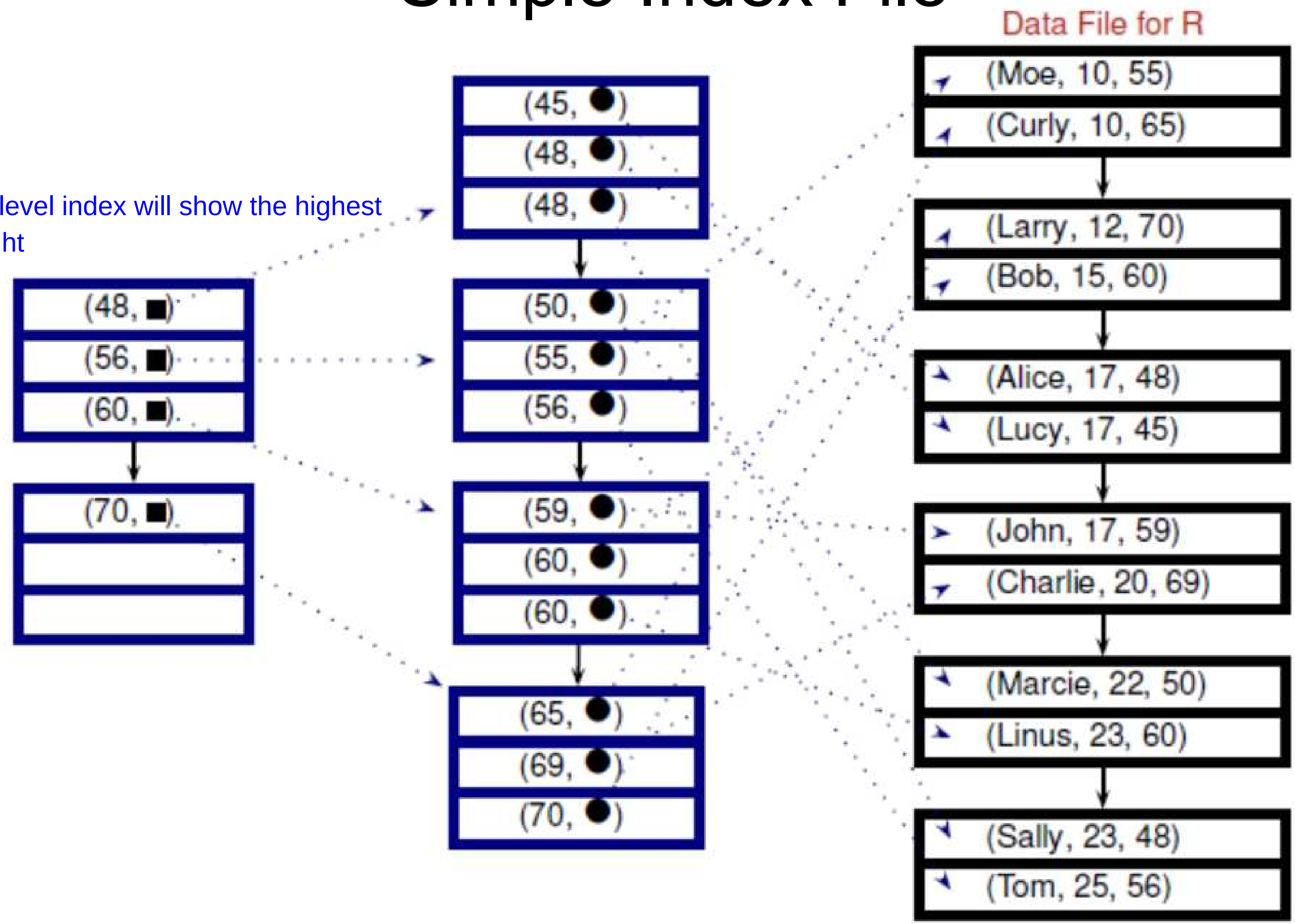
Simple Index File



SELECT * FROM R WHERE weight BETWEEN 65 AND 69

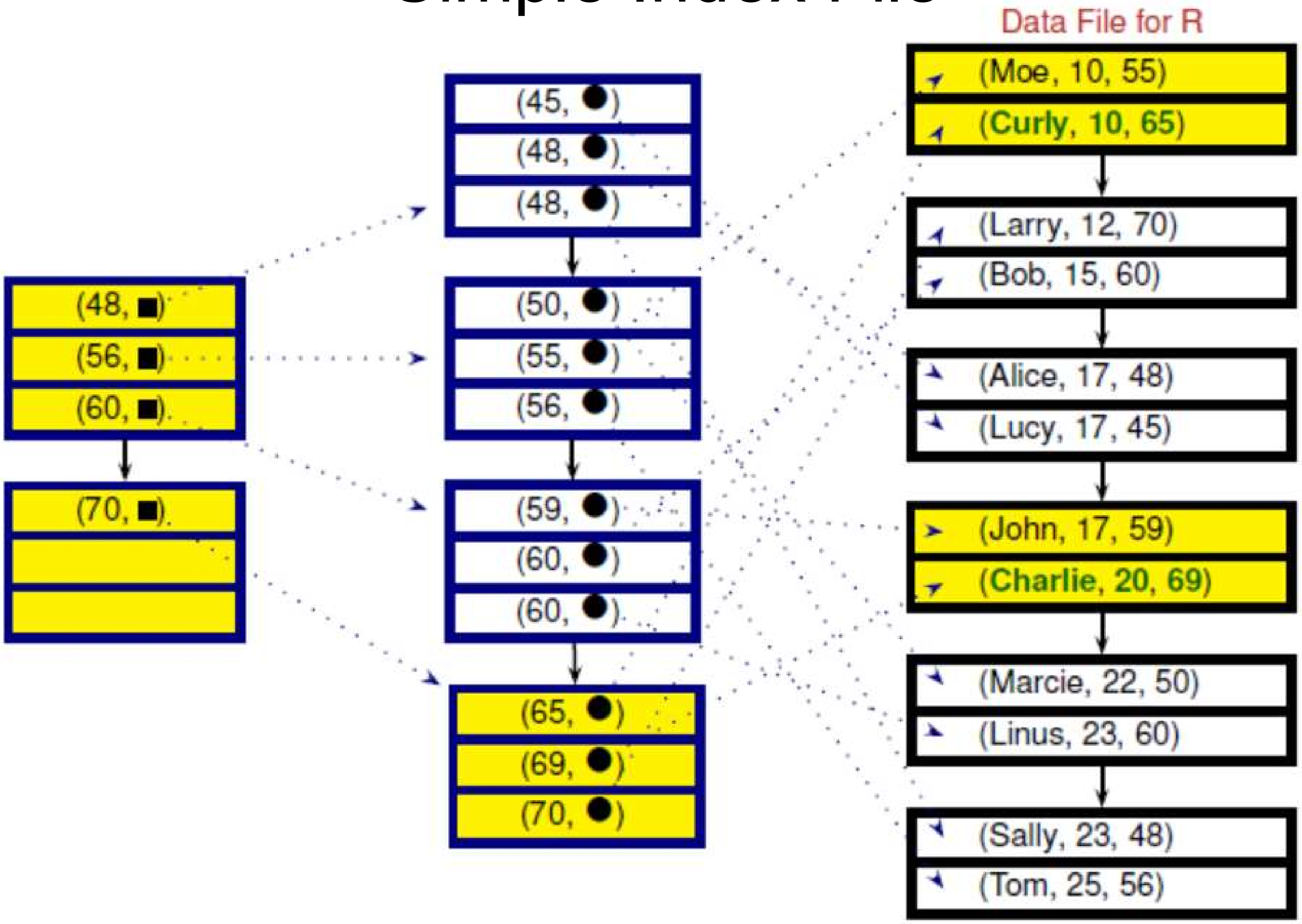
Simple Index File

2nd level index will show the highest weight



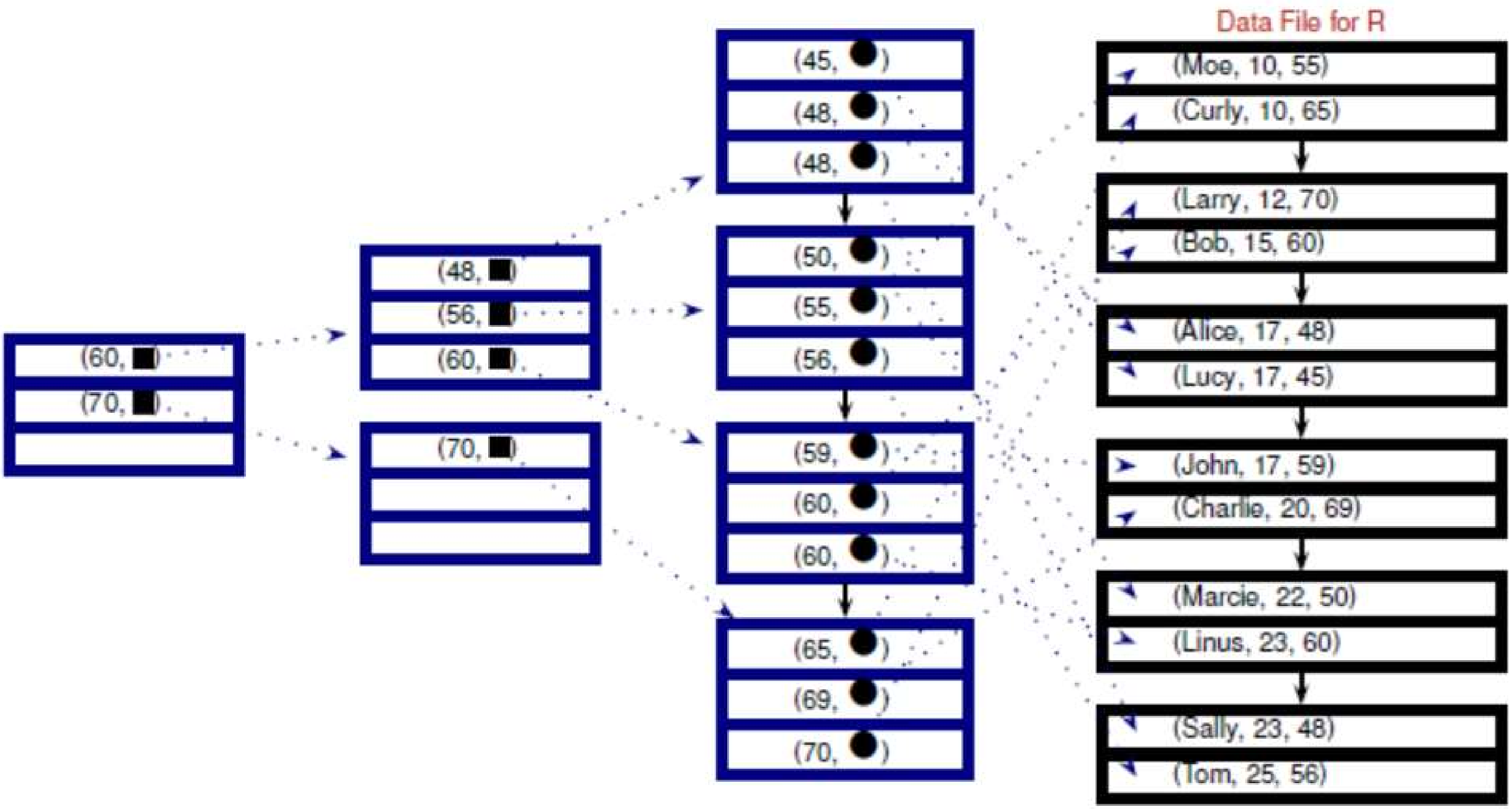
(48, ■) is the index entry for the first page of data entries

Simple Index File



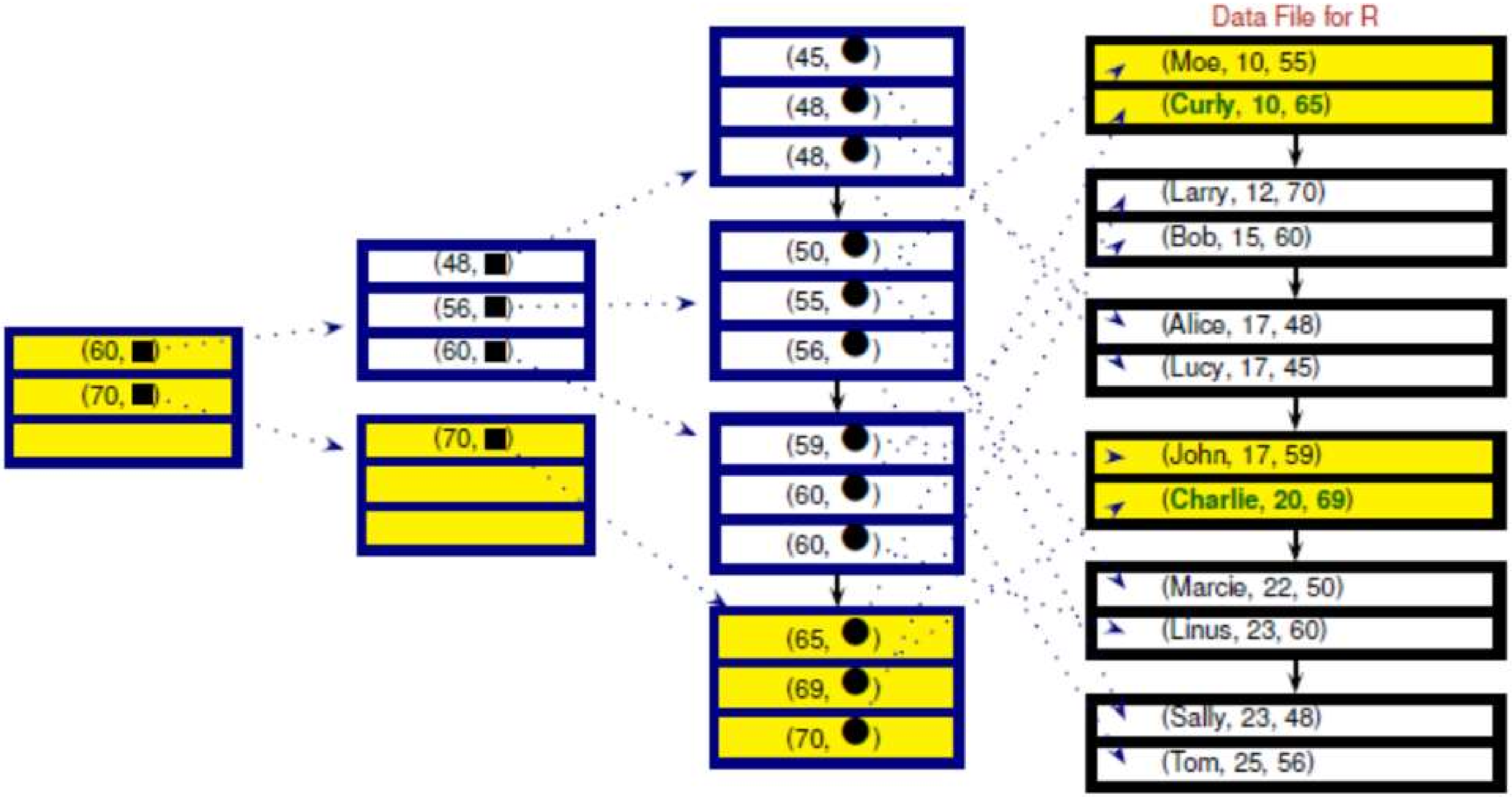
SELECT * FROM R WHERE weight BETWEEN 65 AND 69

Simple Index File



Simple Index File

in this case a 3 level index will not be more useful
since it will cap at the maximum I/O already



SELECT * FROM R WHERE weight BETWEEN 65 AND 69

What if data file is sorted (on search key)?

order of the index leaf will be exactly the same as the data file (since they will all be sorted)

- thus this would mean that the very 1st index level can be removed for unsorted data file it is easy as for the data file just need to add to the bottom (might create a new page) but this would mean that there is more work to update the index nodes - pushing down all the index nodes into the next page

for sorted data file

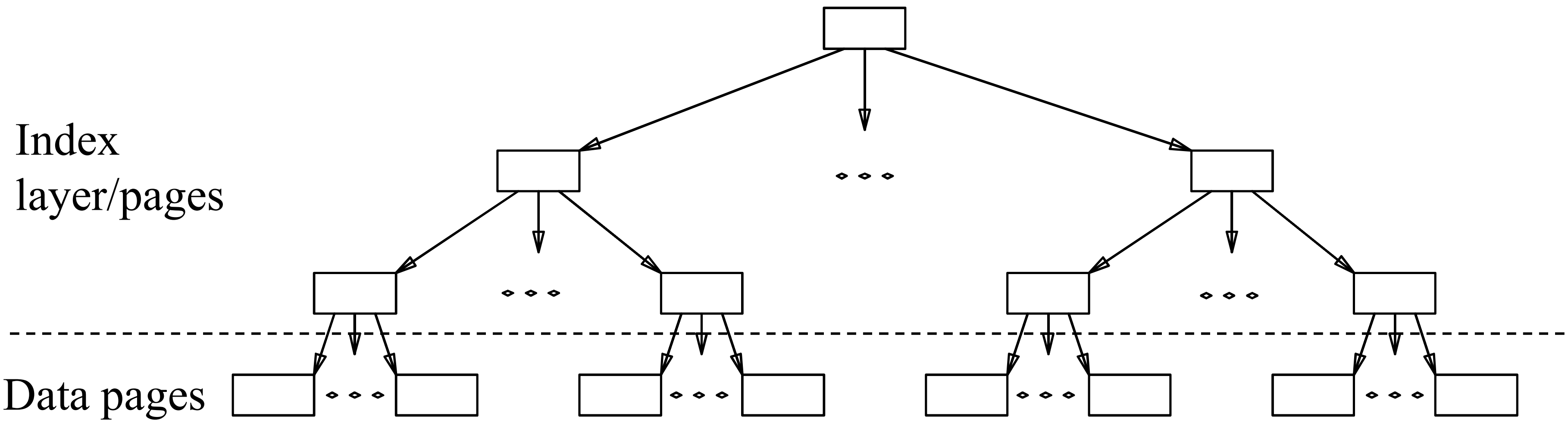
What if you want to insert a new record, say (Judy, 24, 47) for (un)sorted data file?

Index Types

- Two main types of indexes
  - Tree-based index
    - Based on sorting of search key values
    - Examples: ISAM, B^+-tree
  - Hash-based index
    - Data entries are accessed using hashing function
    - Examples: static hashing, extensible hashing, linear hashing
- Things to consider when choosing an index
  - Search performance
    - Equality search: $k = v$
    - Range search: $v_1 \leq k \leq v_2$
  - Storage overhead
  - Update performance

Tree-Structured Indexing

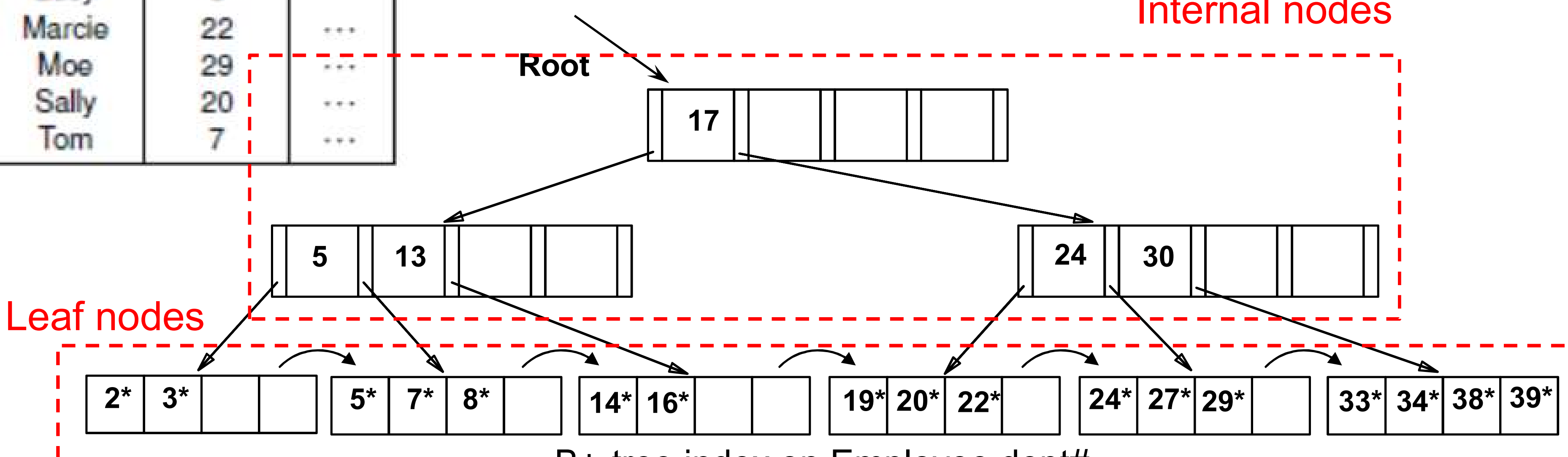
- Tree-structured indexing techniques support both *range searches* and *equality searches*



B+ Tree: The Most Widely Used Index

Employee

name	dept#	...
Alice	5	...
Bob	16	...
Charlie	19	...
Curly	39	...
Dave	38	...
Eve	14	...
Fred	33	...
Harry	2	...
John	34	...
Kate	8	...
Larry	27	...
Linus	24	...
Lucy	3	...
Marcie	22	...
Moe	29	...
Sally	20	...
Tom	7	...



B+-tree index on Employee.dept#

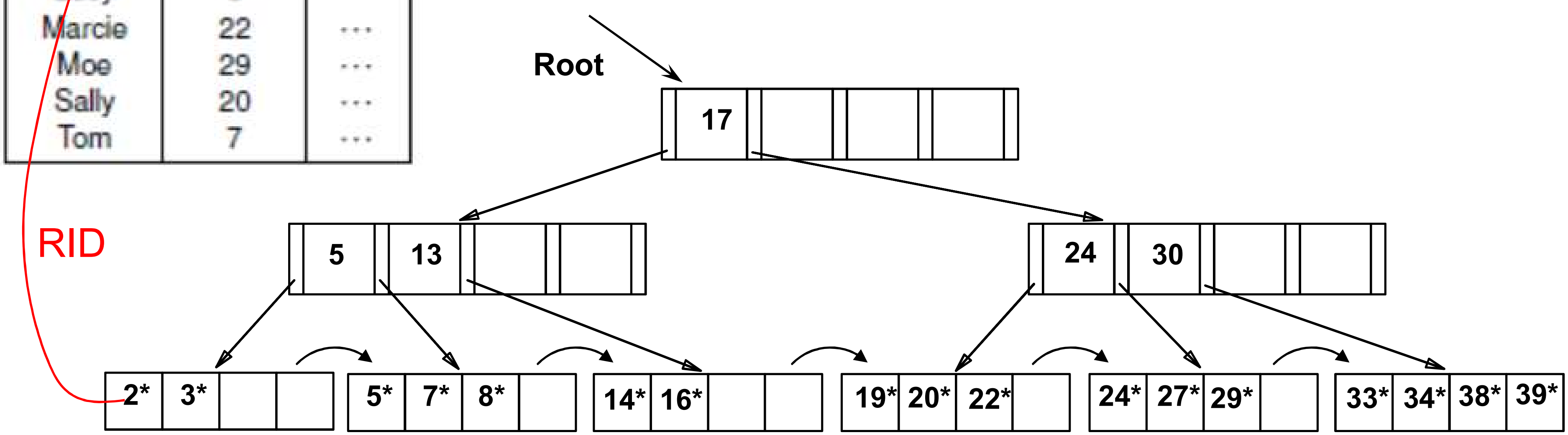
(each node can hold 4 entries (order = 2))

Employee

name	dept#	...
Alice	5	...
Bob	16	...
Charlie	19	...
Curly	39	...
Dave	38	...
Eve	14	...
Fred	33	...
Harry	2	...
John	34	...
Kate	8	...
Larry	27	...
Linus	24	...
Lucy	3	...
Marcie	22	...
Moe	29	...
Sally	20	...
Tom	7	...

B+ Tree: The Most Widely Used Index

- Leaf nodes stored sorted data entries
  - k* denote a data entry of the form (k, RID)
    - k = search key value of corresponding data record
    - RID = RID of corresponding record
  - Lead nodes are either singly or doubly linked
  - *Each data record has an entry in the leaf node (dense index)*
  - Efficient for range search



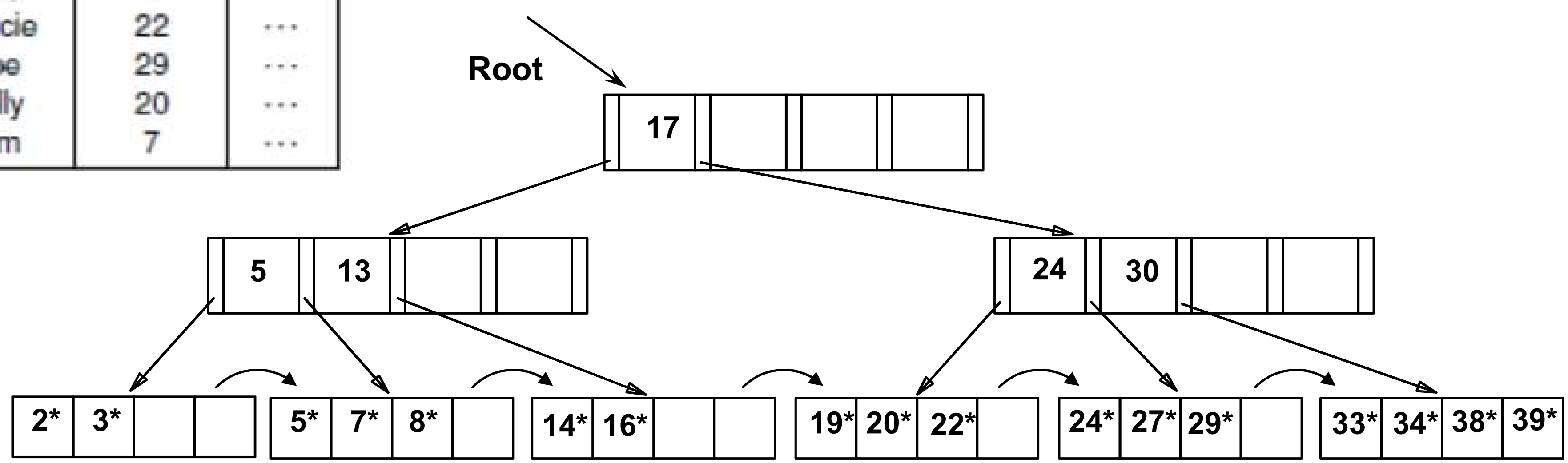
B+-tree index on Employee.dept#

(each node can hold 4 entries (order = 2))

Employee

name	dept#	···
Alice	5	···
Bob	16	···
Charlie	19	···
Curly	39	···
Dave	38	···
Eve	14	···
Fred	33	···
Harry	2	···
John	34	···
Kate	8	···
Larry	27	···
Linus	24	···
Lucy	3	···
Marcie	22	···
Moe	29	···
Sally	20	···
Tom	7	···

- Internal nodes store index entries of the form (p_0, k_1, p_1, k_2, p_2, · · · , p_n)
  - $k_1 < k_2 < \ldots < k_n$
  - p_i = disk page address (root node of an index subtree T_i)
  - For each data entry k* in T_0, $k < k_1$
  - For each data entry k* in T_i ($i \in [1,n)$), $k \in [k_i, k_{i+1})$
  - For each data entry k* in T_n, $k \geq k_n$
- Key values for index entries (internal nodes) are separators (not necessarily correspond to any key values)
  - e.g., 13, 30. How did this happen???

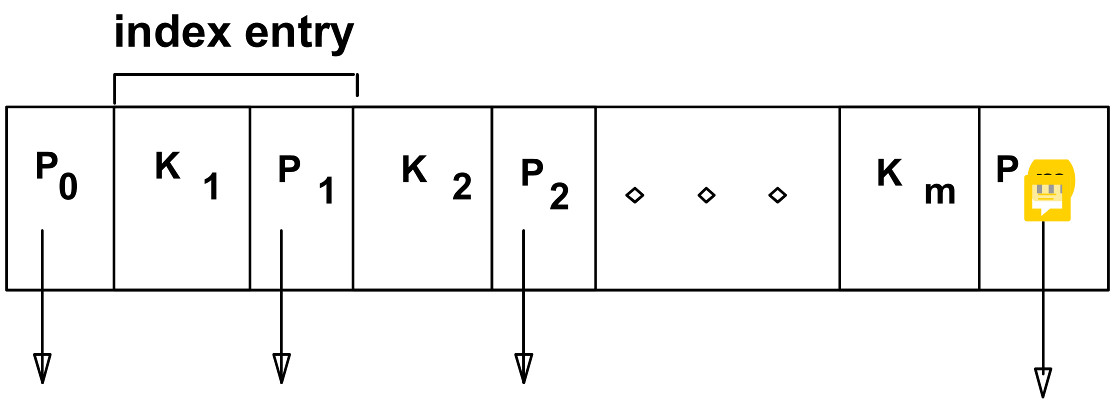


B+-tree index on Employee.dept#

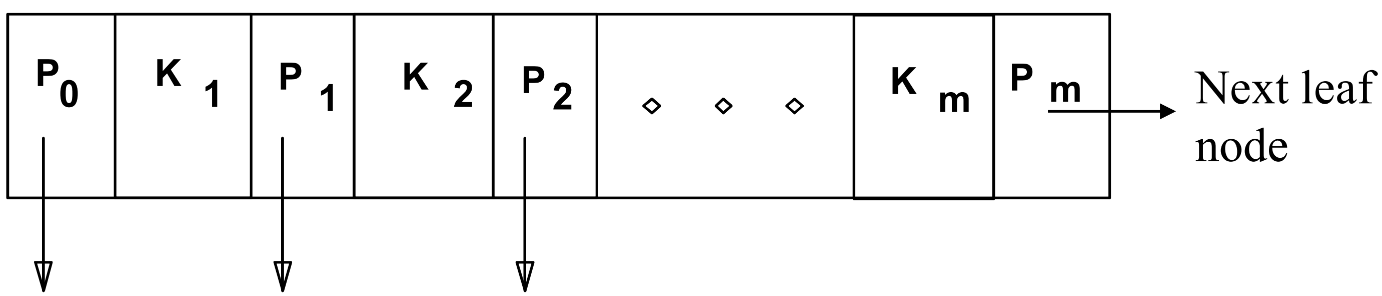
order means the minimum is 2 max is 4

Node structure

- A node essentially corresponds to a page (each node access costs an I/O)
- Non-leaf node



- Leaf node



How to determine the Order?

- A node essentially corresponds to a page
- Assume 8 KB page, 8-byte key, 4-byte pointer, we have
  - m = 2*d
  - (2d+1)*4 + 2d*8 ≤ 8096
  - d ~ 336
- m = 672 (though we can actually store 673 entries ~ 8080 bytes!)

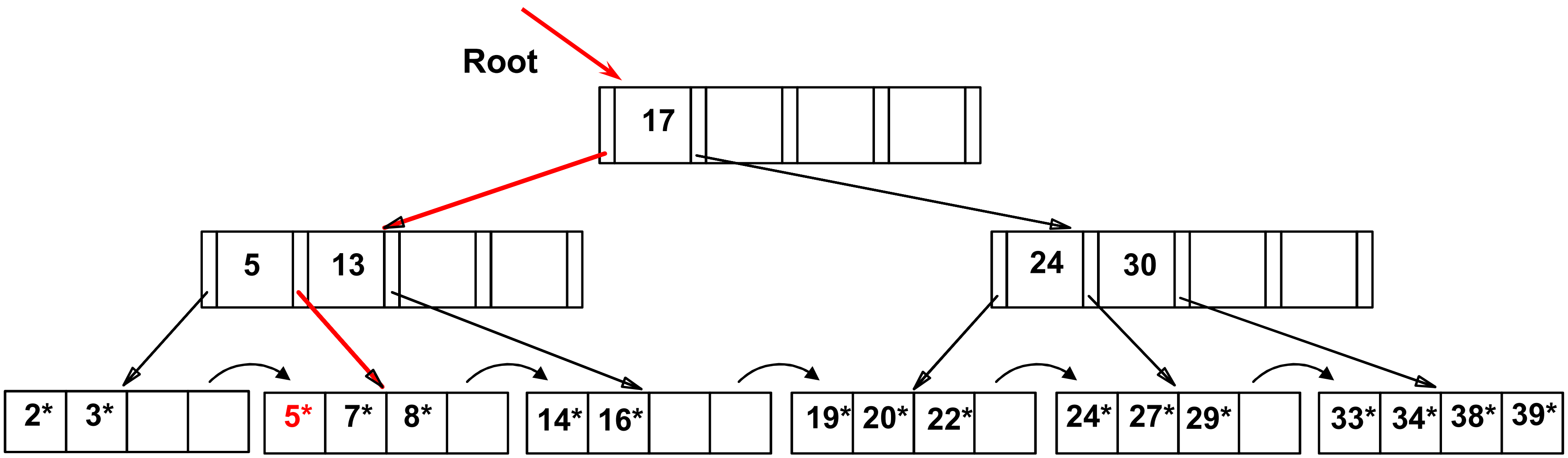
Properties of B^+ Tree

- *Height-balanced (search and update efficient)*
  - Insert/delete at $\log_F N$ cost (F = fanout, N = # leaf pages)
- Grow and shrink dynamically (update efficient)
- Minimum 50% occupancy (except for root) (storage efficient)
  - Each non-leaf node contains **d** ≤ *m* ≤ 2**d** entries. The parameter **d** is called the *order* of the tree
  - *Order* (**d**) concept replaced by physical space criterion in practice (‘*at least half-full*’)
- next-leaf-pointer to chain up the leaf nodes (*efficient range search*)
- Data entries at leaf are sorted

Searching in B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
  – At each internal node N, find the largest key k_j in N s.t. $k \geq k_j$
    - If k_j exists, then search subtree at p_j
    - Otherwise, search subtree at p_0
- Search for 5

3 I/O for the tree + 1 I/O for actually accessing the data

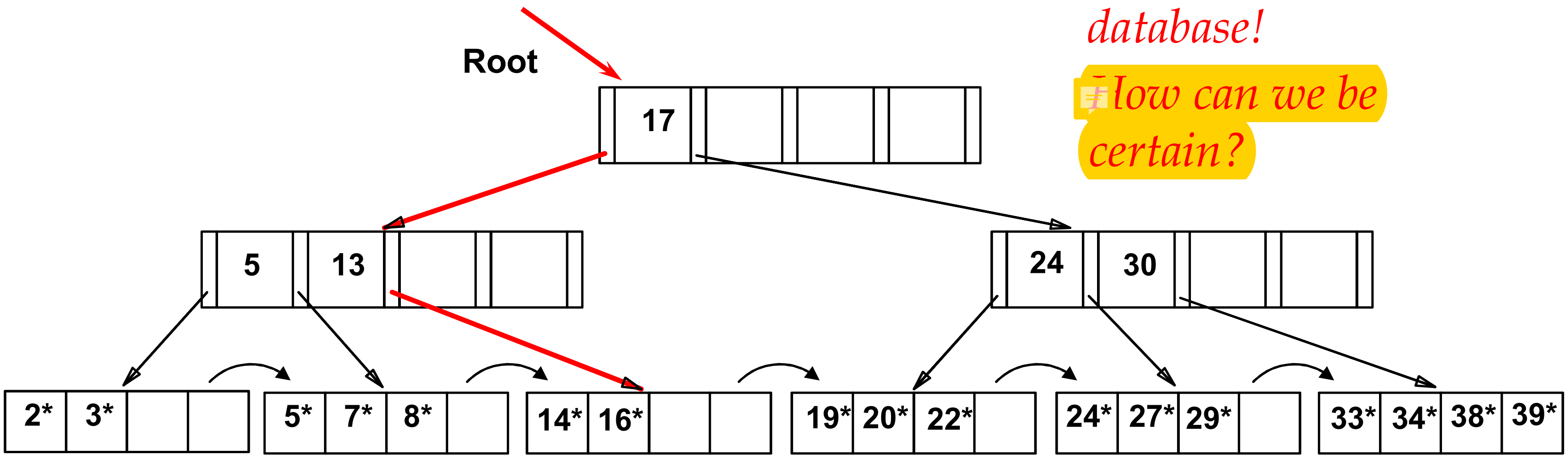


Searching in B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
  - At each internal node N, find the largest key k_j in N s.t. $k \geq k_j$
    - If k_j exists, then search subtree at p_j
    - Otherwise, search subtree at p_0
- Search for 15

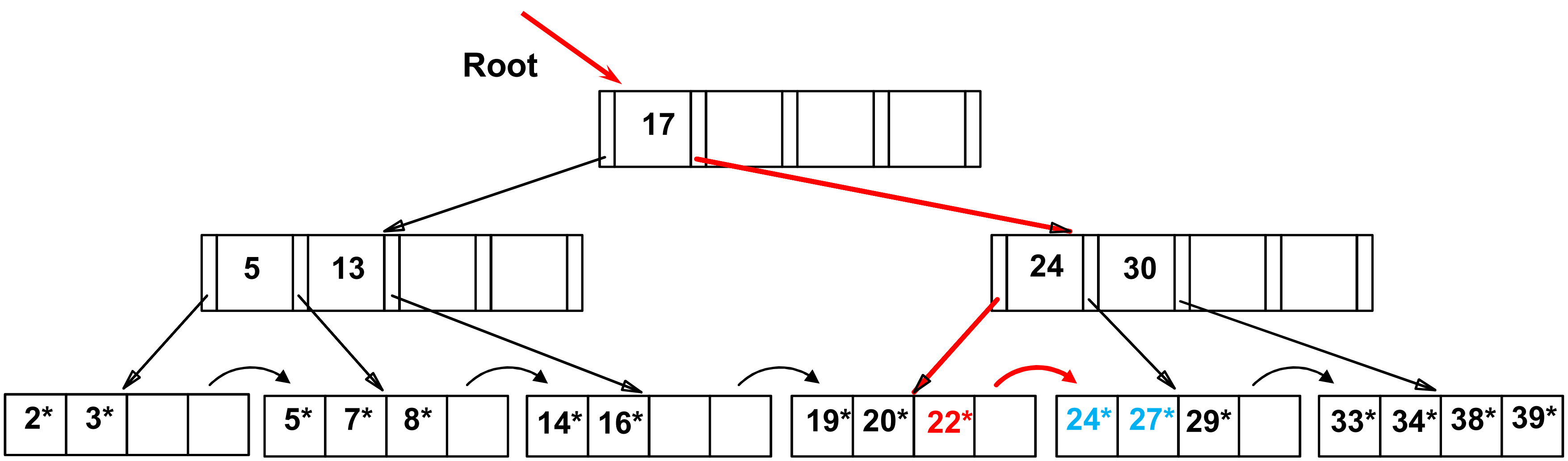
Based on the search for 15, we know no such records in the database!*

How can we be certain?



Searching in B+ Tree

- Search begins at root, and key comparisons direct it to a leaf
  - At each internal node N, find the largest key k_j in N s.t. $k \geq k_j$
    - If k_j exists, then search subtree at p_j
    - Otherwise, search subtree at p_0
- Search for all data entries between 22 and 27



B+ Trees in Practice

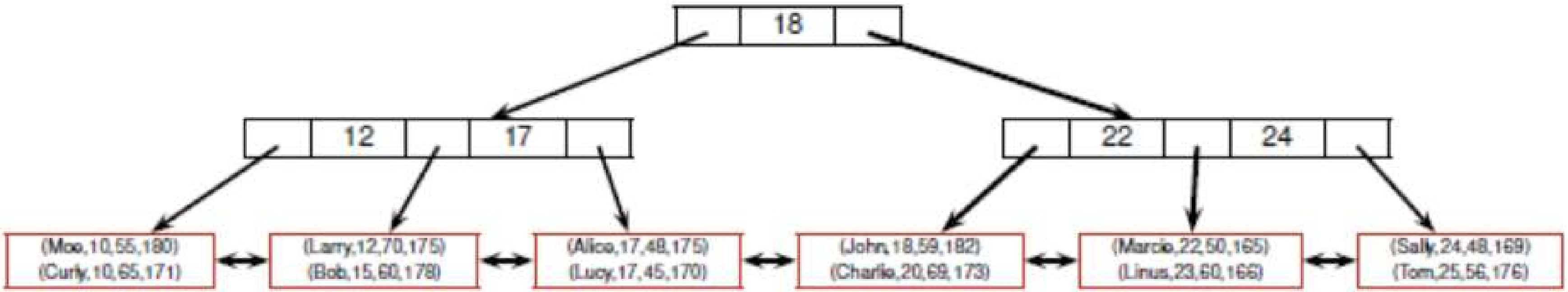
- Typical order: 100. Typical fill-factor: 67%
  - average fanout = 133
- Typical capacities:
  - Level 4: $133 \times 133^3 = 133 \times 2{,}352{,}637$ pointers
  - Level 5: $133 \times 133^4 = 41{,}615{,}795{,}893$ pointers!
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

since all the 1st 3 levels added together is still quite small, there is a chance that they can still be stored in memory - so no I/O cost in going to the 3 levels

Formats of Data Entries

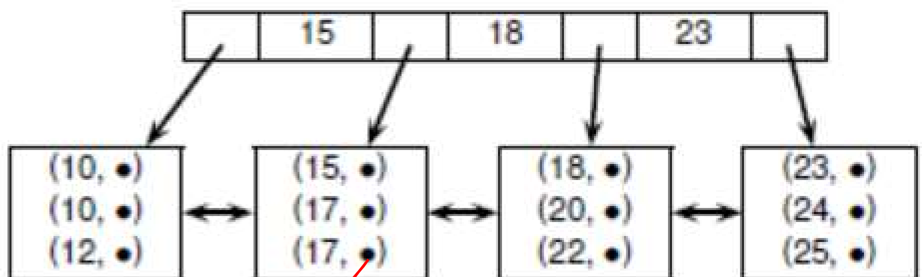
- So far, our examples assume k* is of the form (k, rid), where rid is the record identifier of a data record with search key value k
  - This is referred to as Format 2 (Default unless otherwise stated)
- Two other different formats for data entries:
  - Format 1: k* is the (actual) data record (with search key value k)
  - Format 3: k* is of the form (k, rid-list), where rid-list is a list of record identifiers of data records with search key value k

Formats of Data Entries: Example



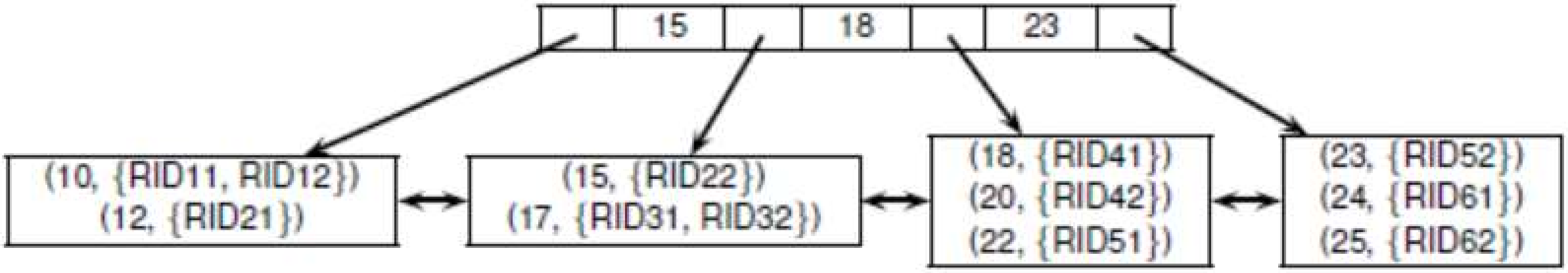
B^+-tree index on R.age (Format 1)

name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	60	166
Sally	24	48	169
Tom	25	56	176



B^+-tree index on R.age (Format 2)

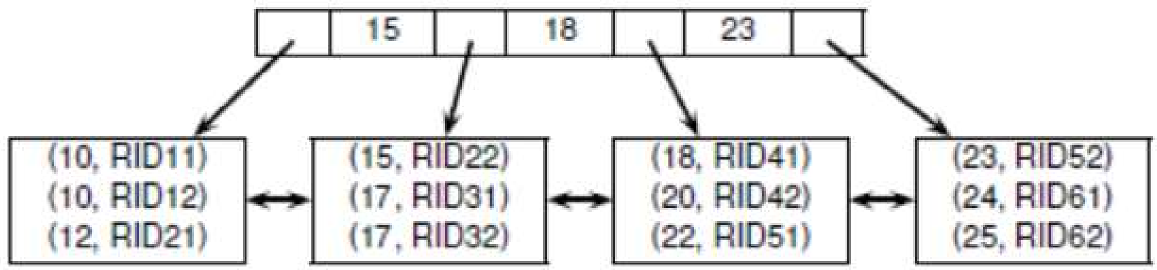
Formats of Data Entries: Example



Index on R.age (format 3)

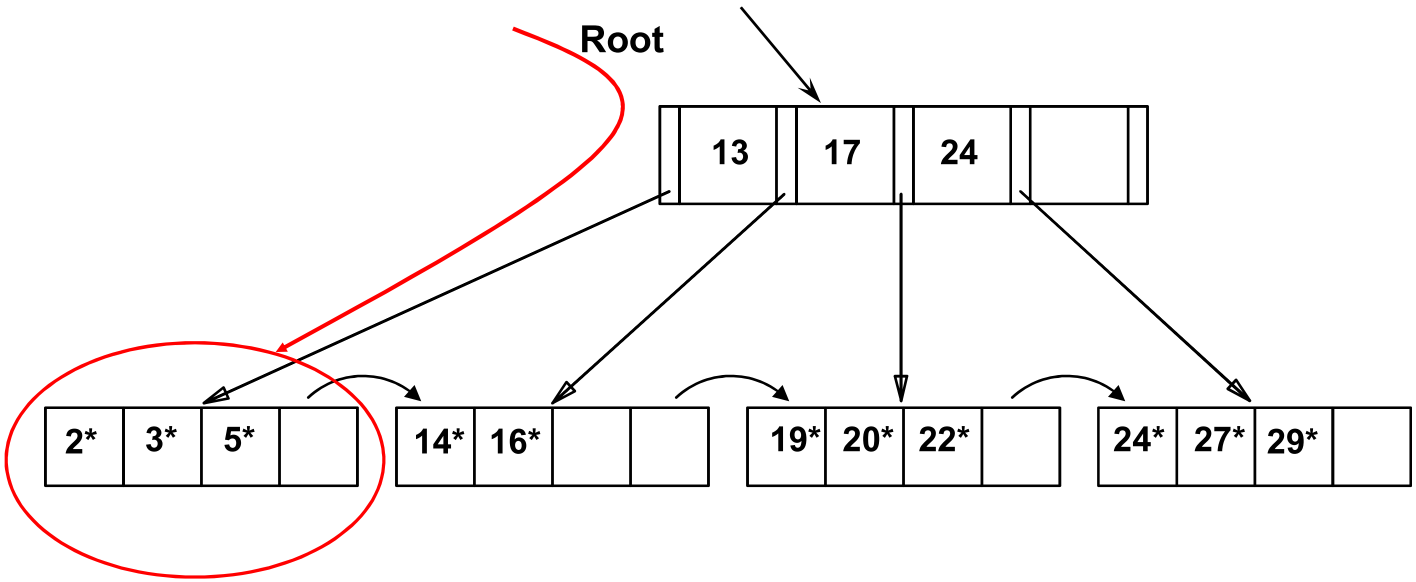
Relation R

name	age	weight	height
Moe	10	55	180
Curly	10	65	171
Larry	12	70	175
Bob	15	60	178
Alice	17	48	175
Lucy	17	45	170
John	18	59	182
Charlie	20	69	173
Marcie	22	50	165
Linus	23	60	166
Sally	24	48	169
Tom	25	56	176



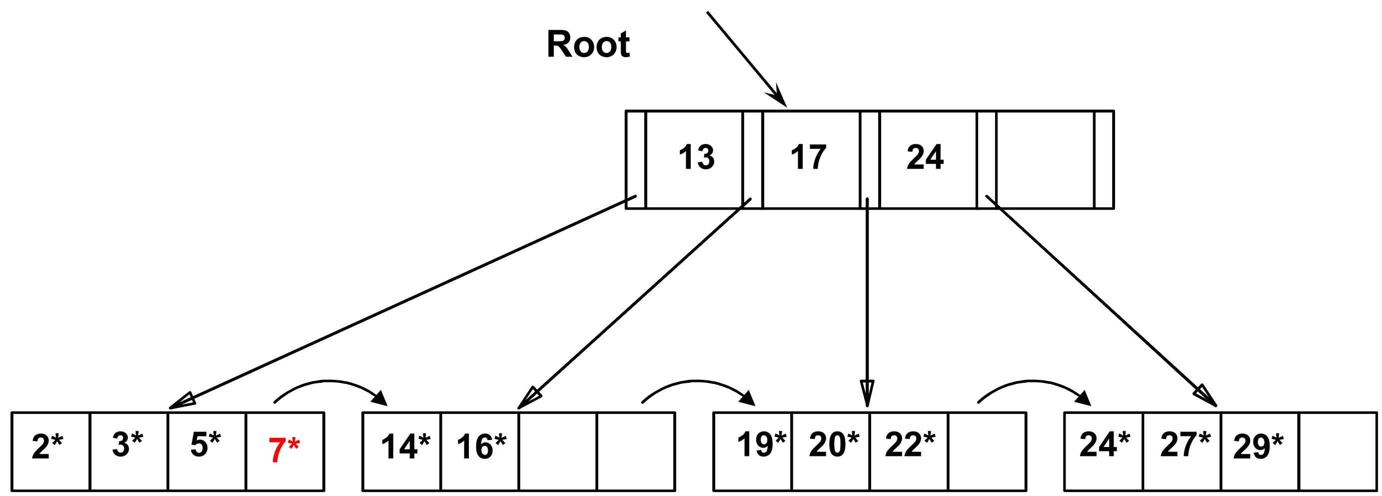
Index on R.age (format 2)

Inserting 7 into Example B+ Tree



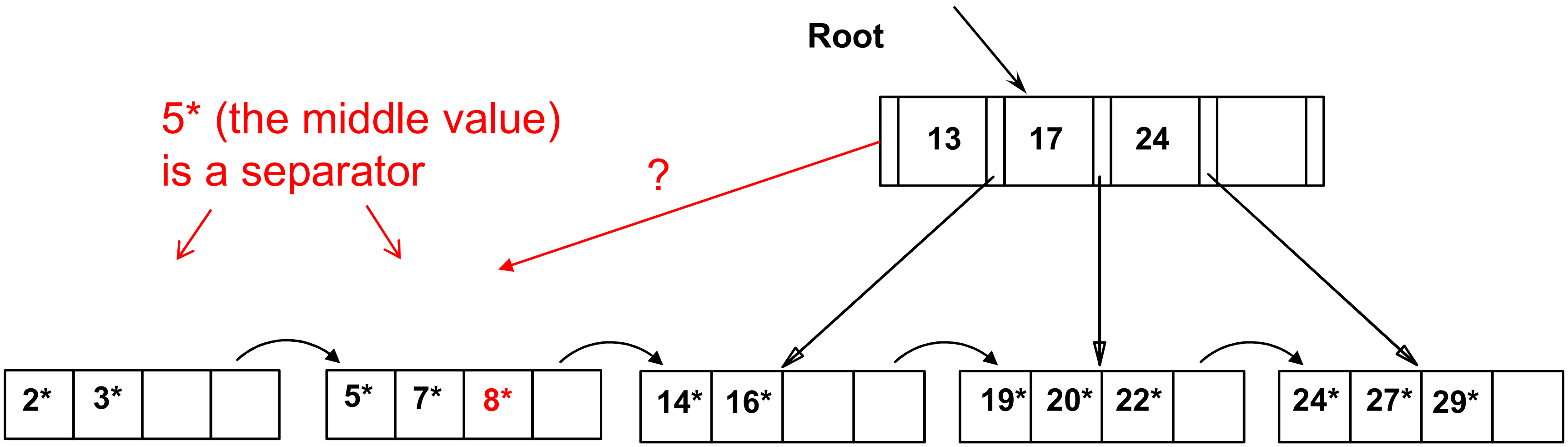
Find the leaf node to insert

Inserting 7 into Example B+ Tree

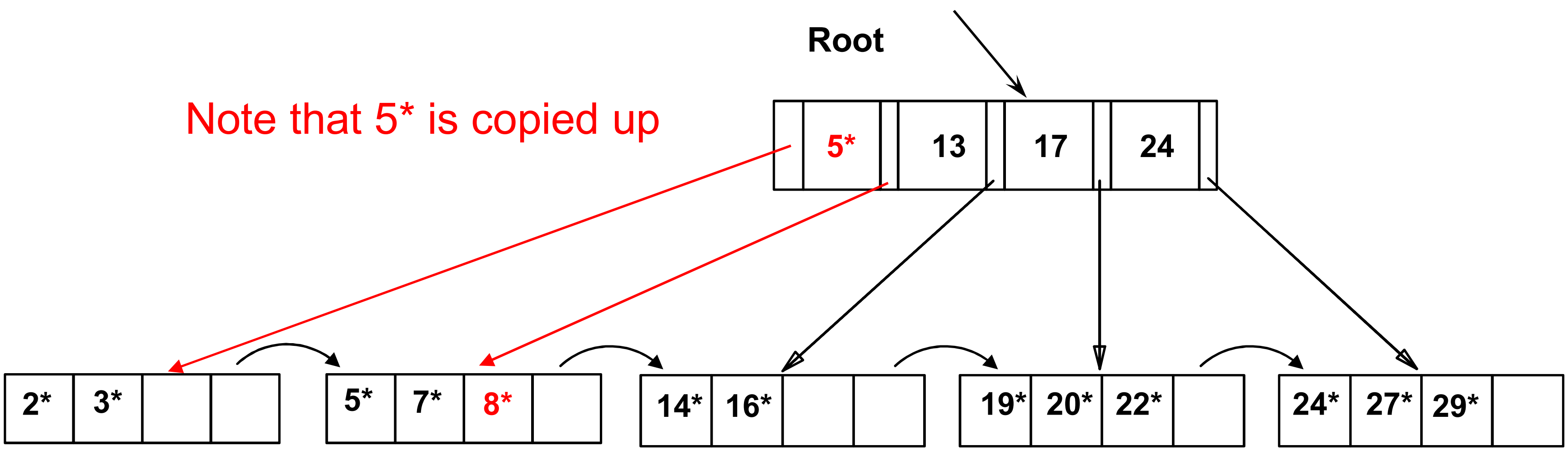


What if we insert 8 now?

Inserting 8 into Example B+ Tree



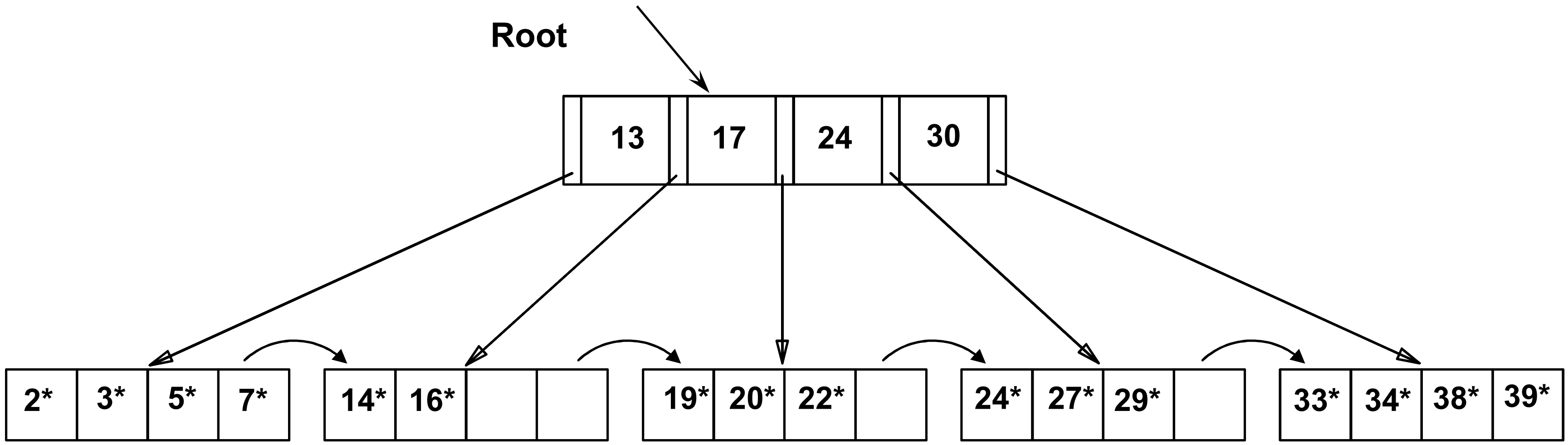
Inserting 8 into Example B+ Tree



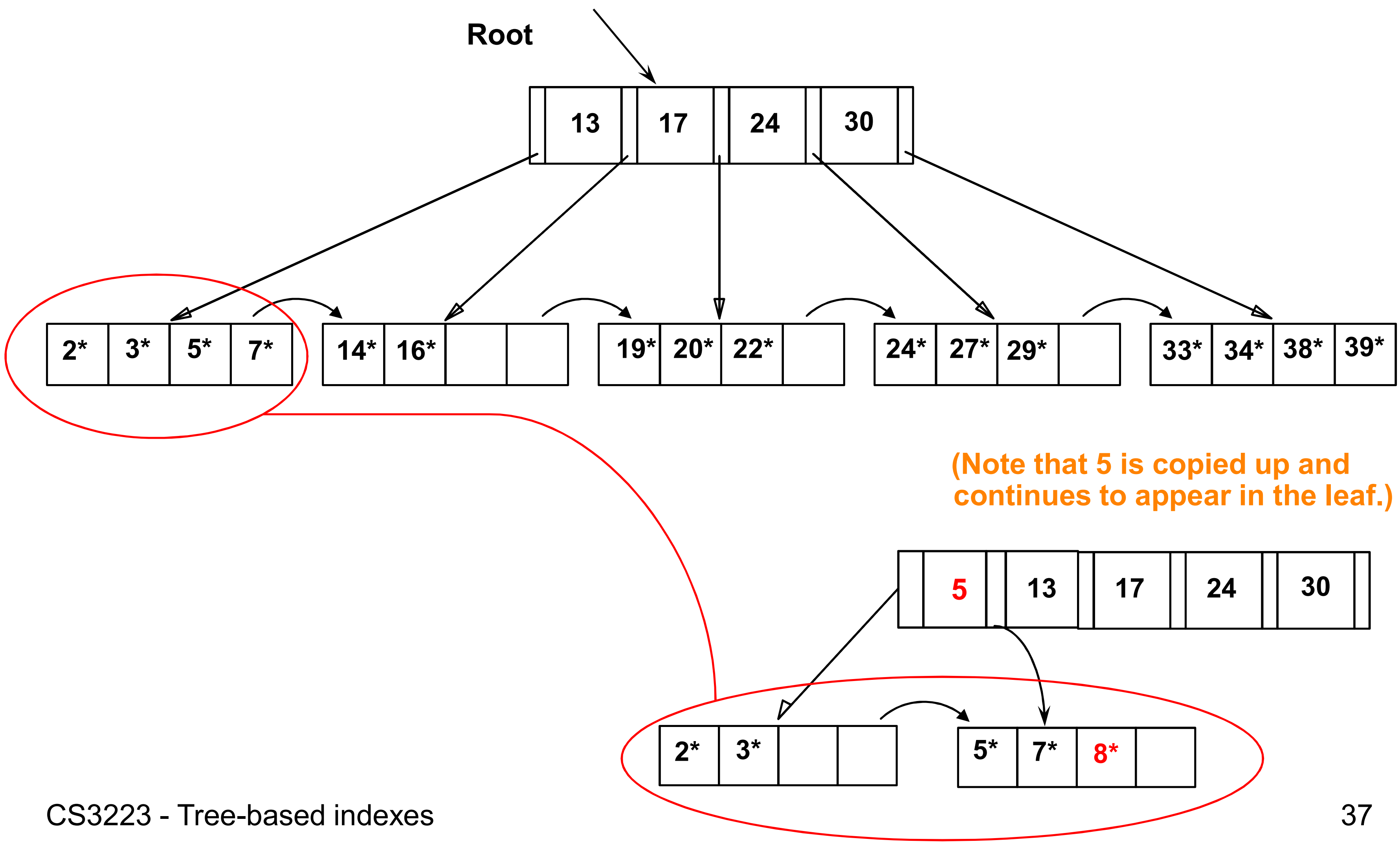
Insertion Algorithm

- Find correct leaf *L*
- Put data entry into *L*
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key
    - Insert index entry pointing to *L2* into parent of *L*
- This can happen recursively (index node can be full!)
  - 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 8 into Example B+ Tree (Internal node is full)

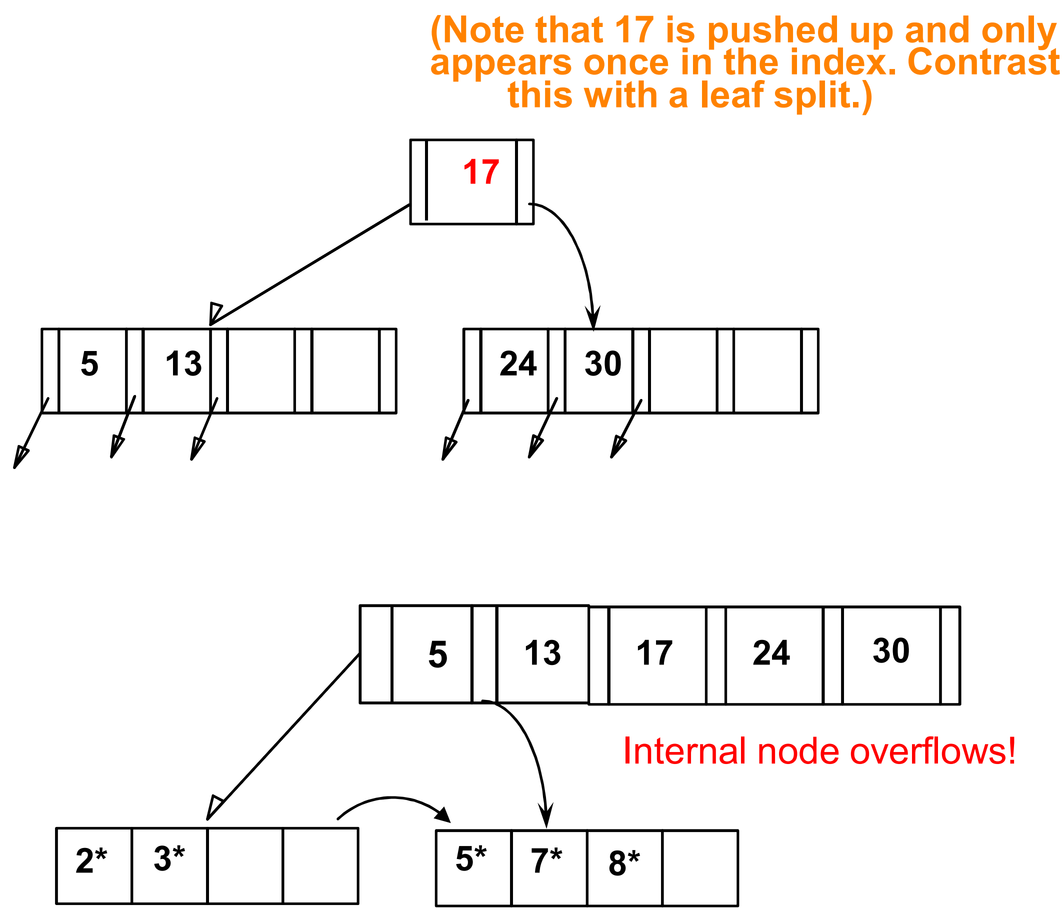


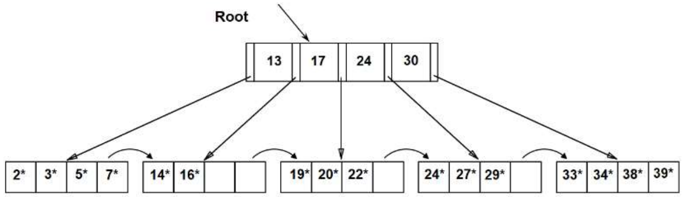
Inserting 8 into Example B+ Tree



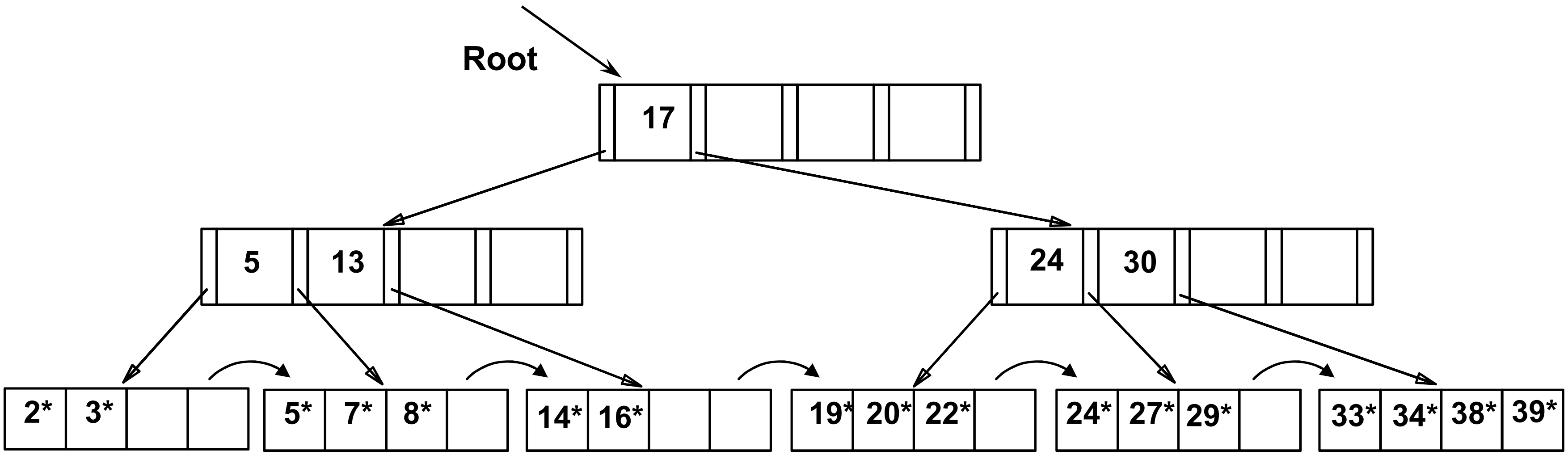
Insertion (Cont)

- Note difference between *copy-up (for leaf nodes)* and *push-up (for internal nodes)*; be sure you understand the reasons for this
- Observe how minimum occupancy is guaranteed in both leaf and index page splits





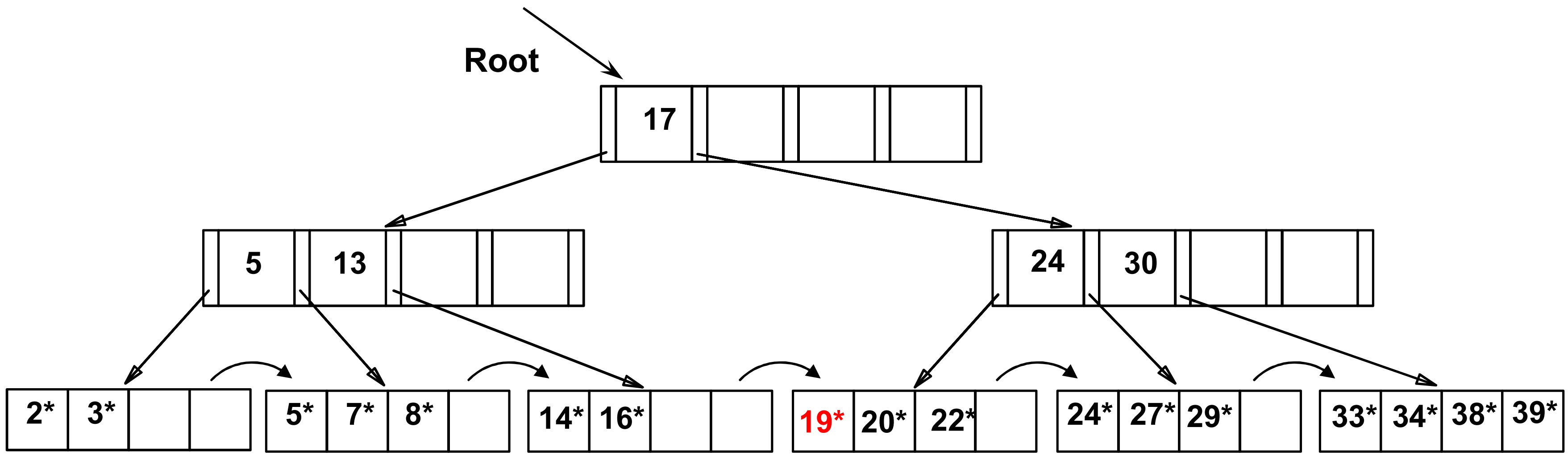
Example B+ Tree After Inserting 8



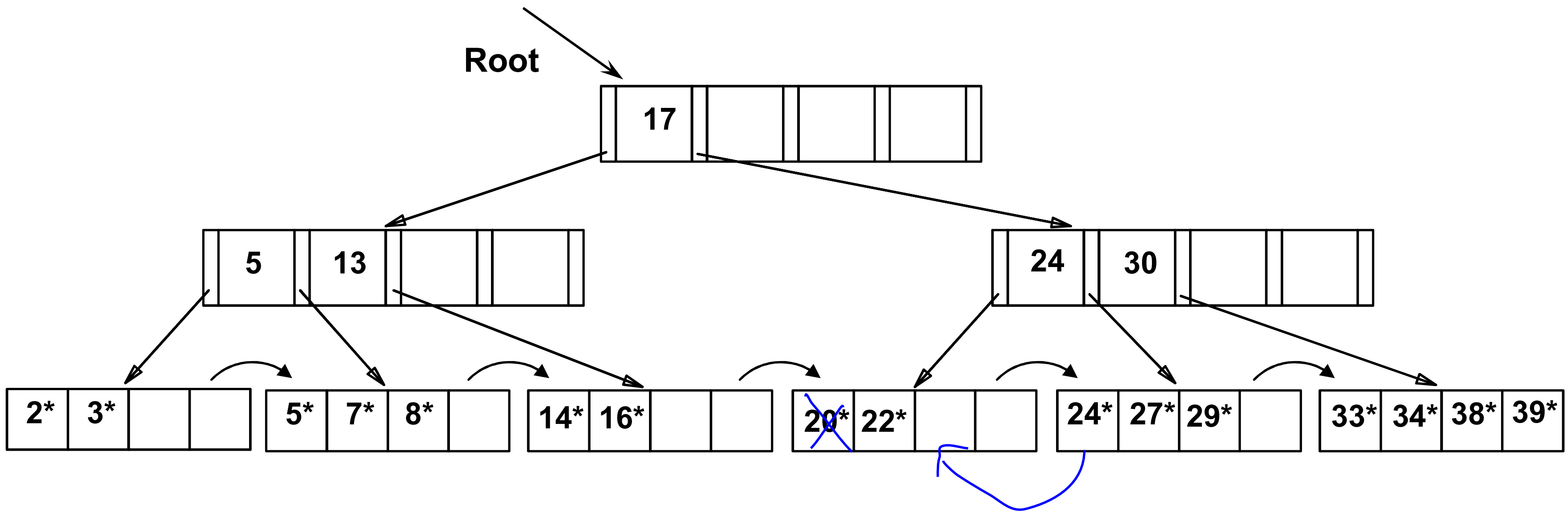
- Notice that root was split, leading to increase in height
- Tree gets wider and one level taller

thus after the insertion, there would be 2 new pages
- 1 is for the new leaf node
- 2 is for the new root node

Delete 19 from the example tree

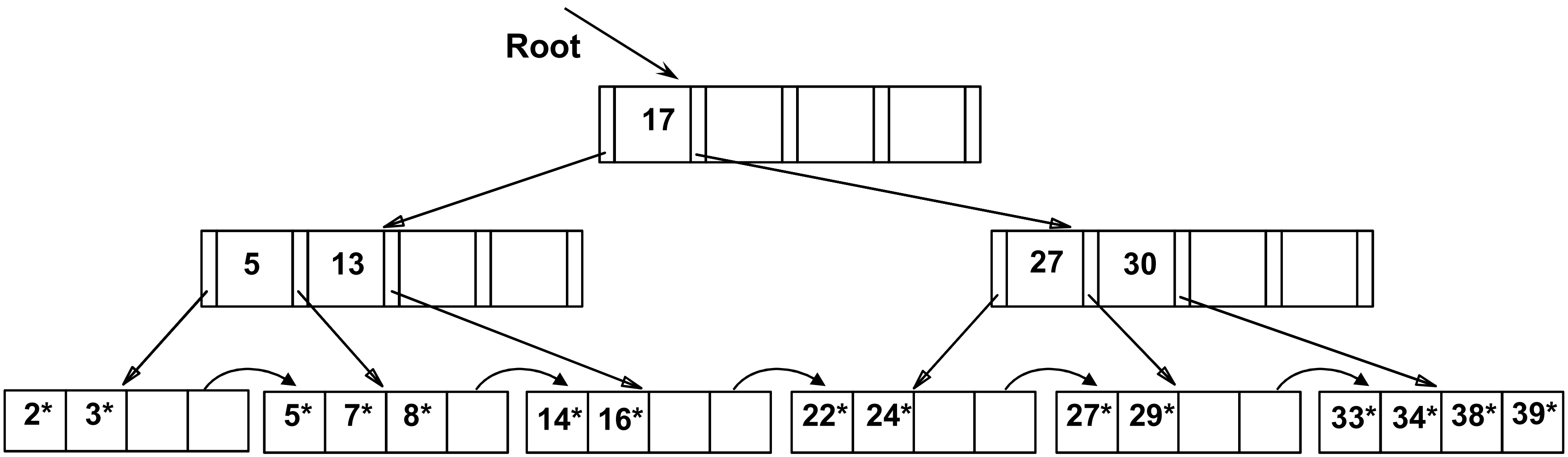


Delete 19 from the example tree



- Deleting 19 is easy
- Delete 20 next?

Example Tree After Deleting 20 ...

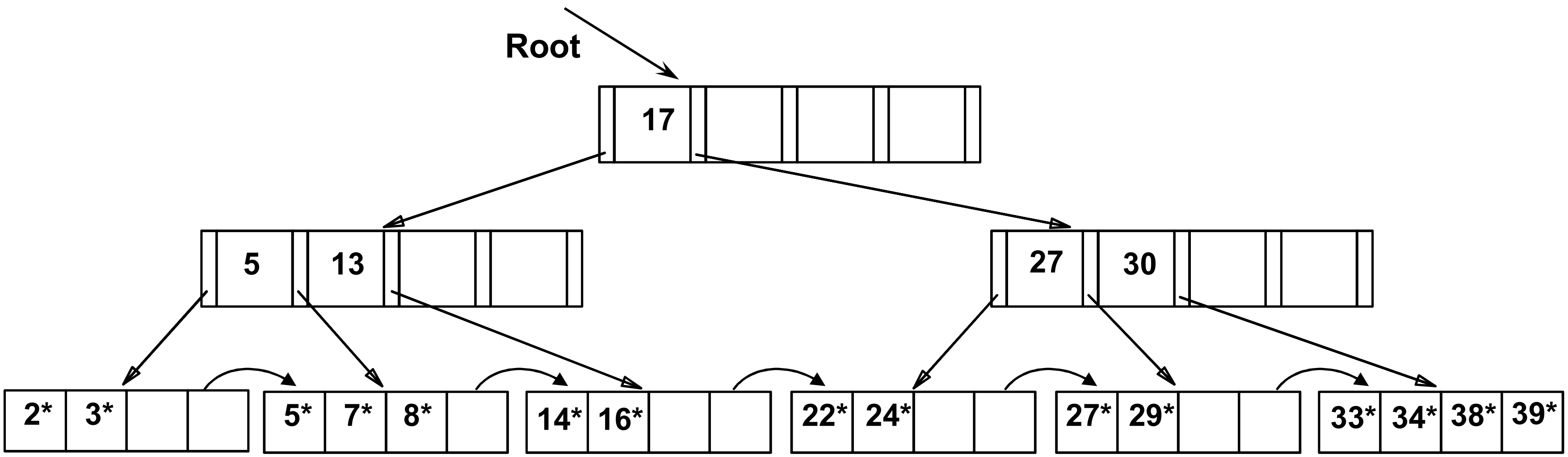


- Deleting 20 is done with re-distribution
- Notice how middle key is *copied up*

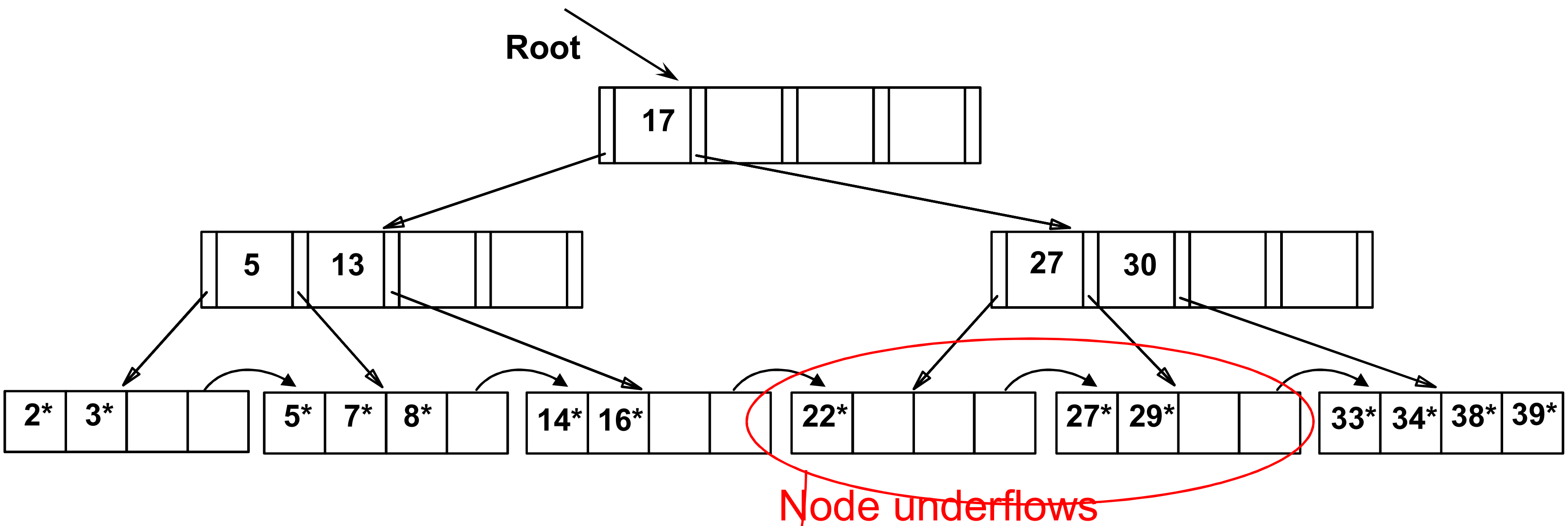
Deleting a Data Entry from a B+ Tree

- 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 same parent as L)*
    - If re-distribution fails, *merge L* and sibling
- If merge occurs, must delete entry (pointing to *L* or sibling) from parent of *L*
- Merge could propagate to root, decreasing height

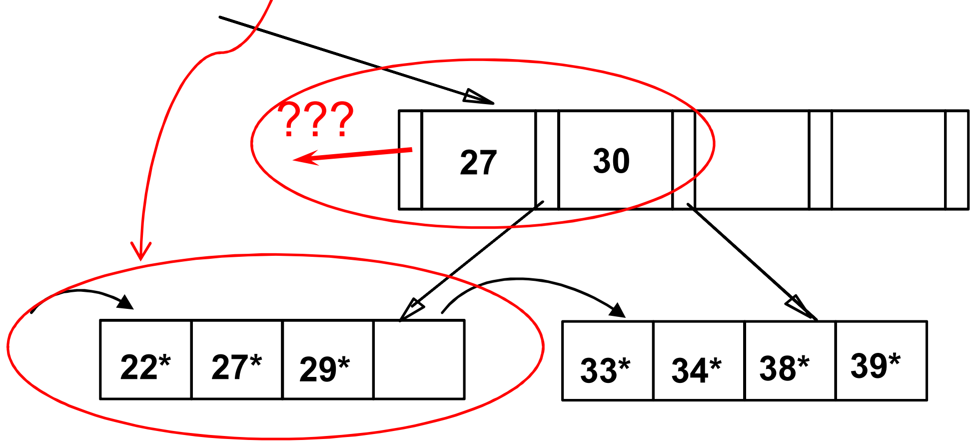
Example Tree



Example Tree After Deleting 24 ...

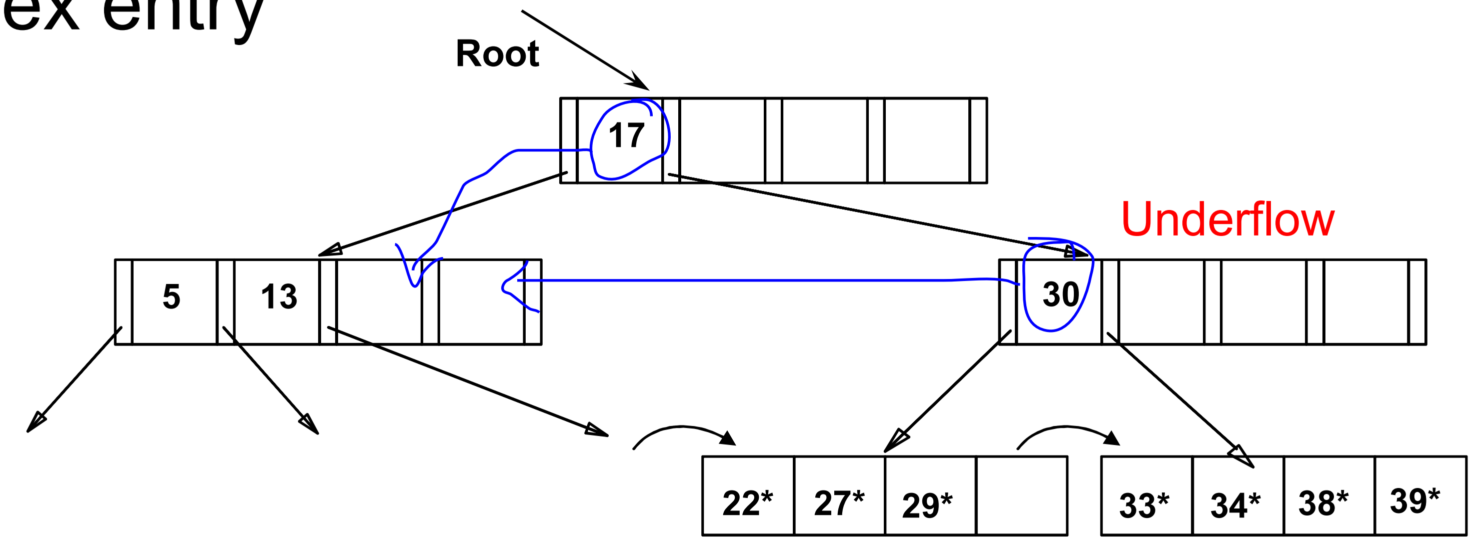


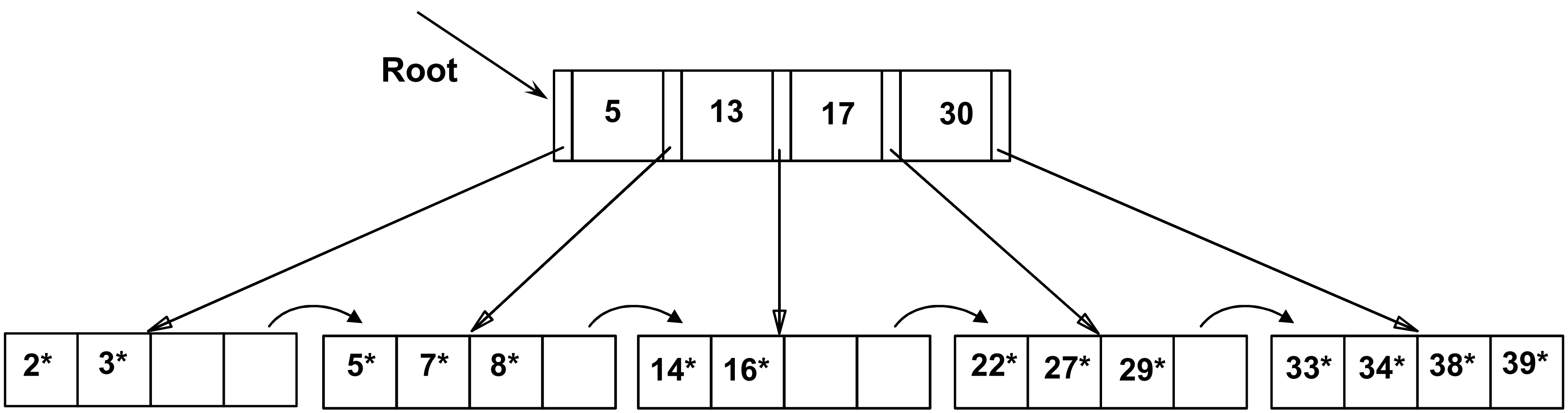
- Must merge



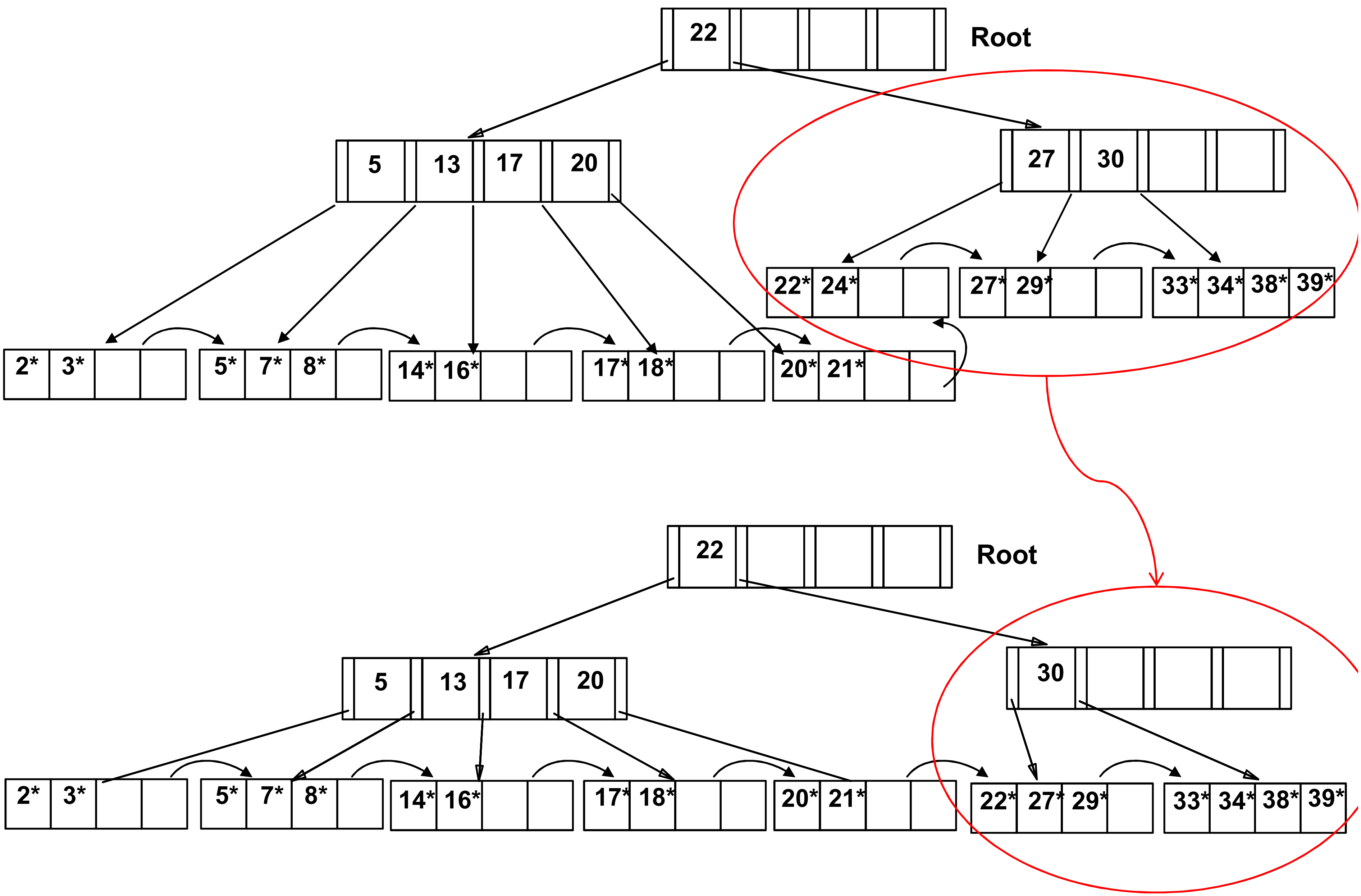
... And Then Deleting 24

- Observe *`toss'* of index entry, and *`pull down'* of index entry

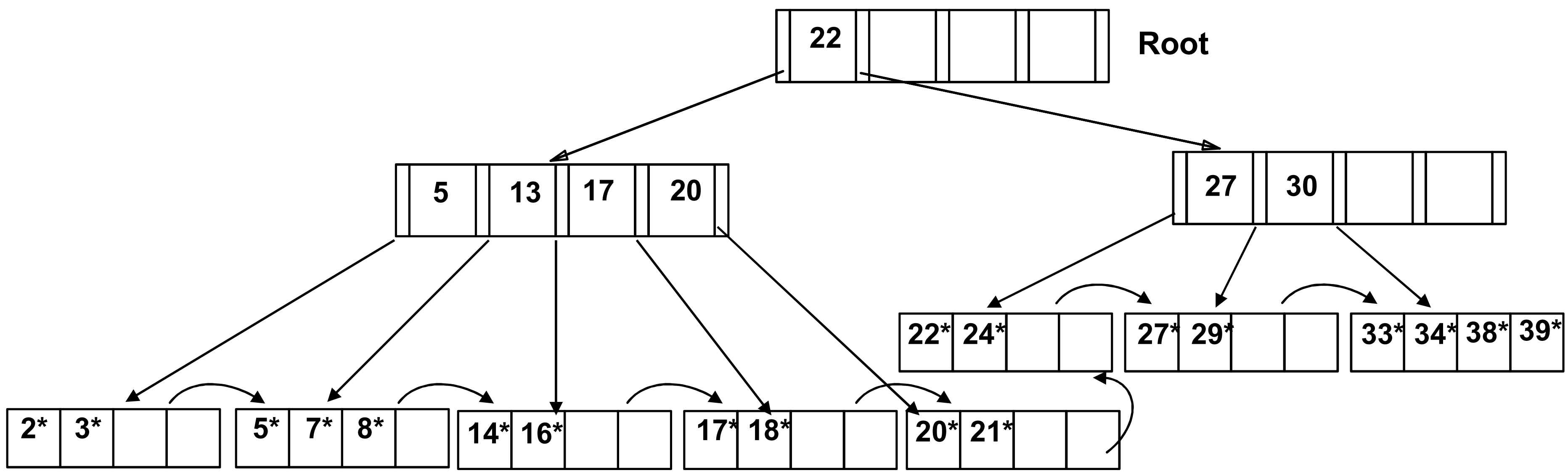




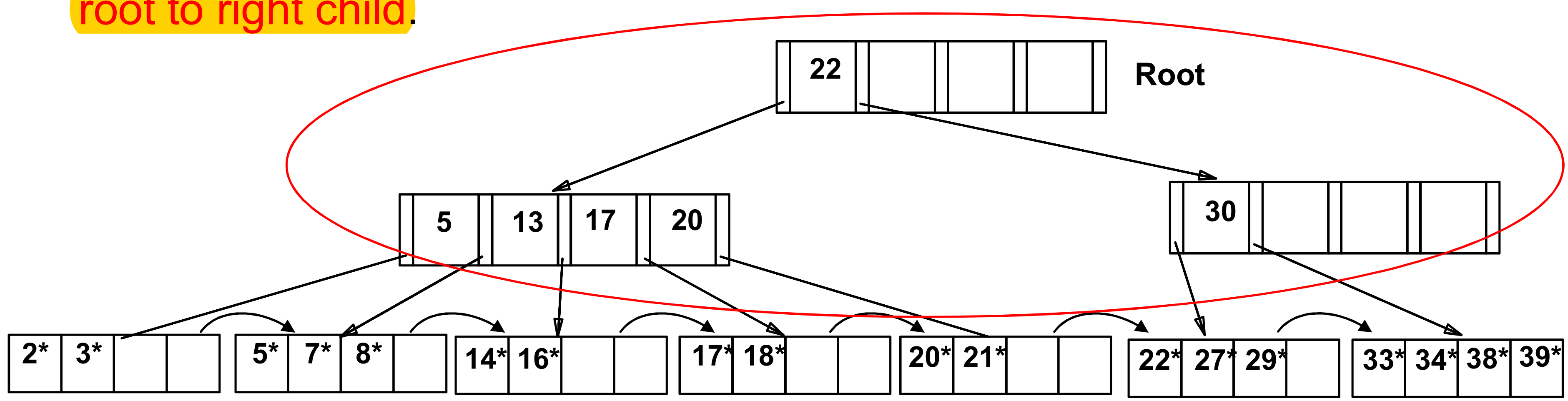
Example of Non-leaf Re-distribution (Delete 24)



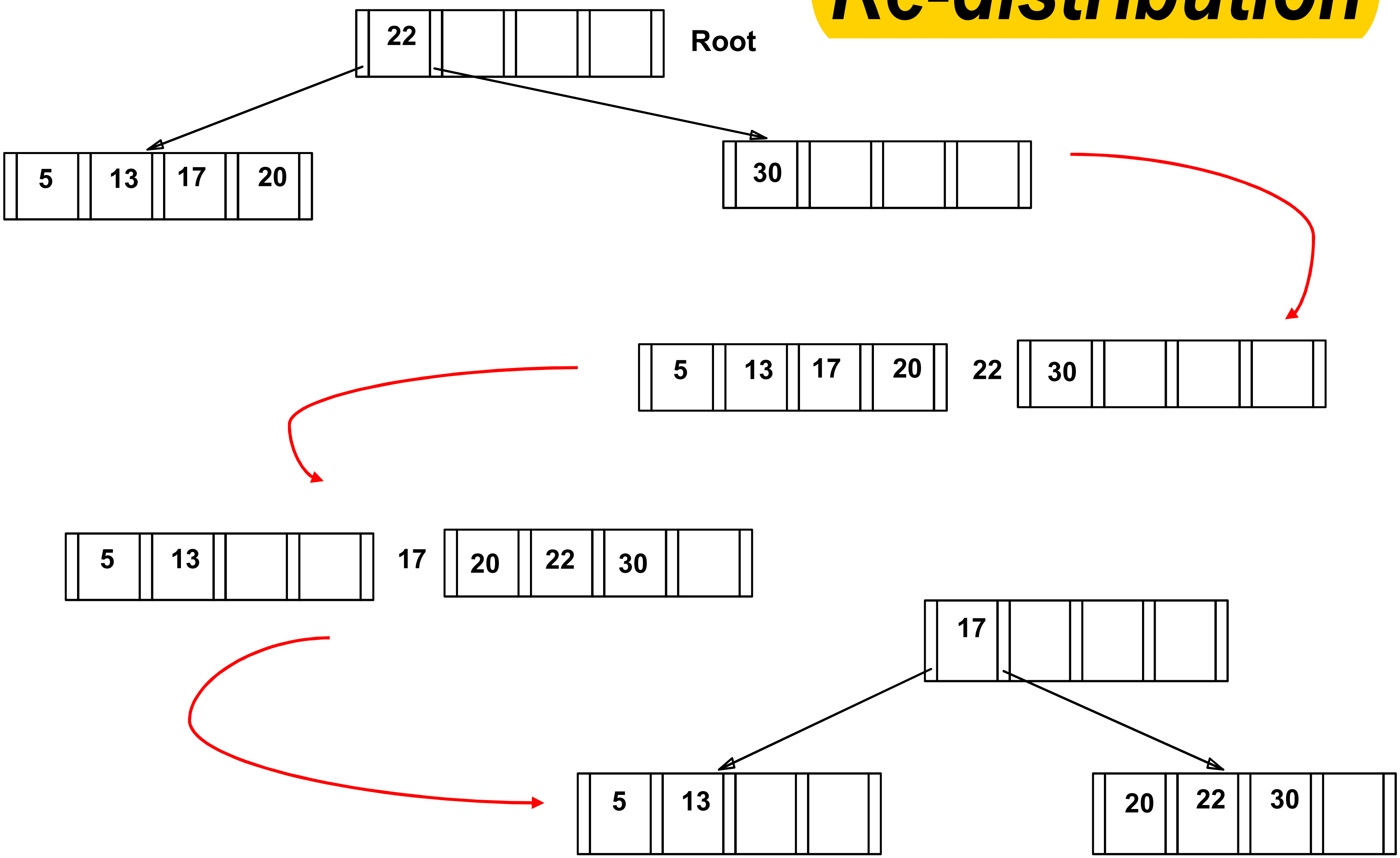
Example of Non-leaf Re-distribution (Delete 24)



- In contrast to previous example, can re-distribute entry from left child of root to right child.

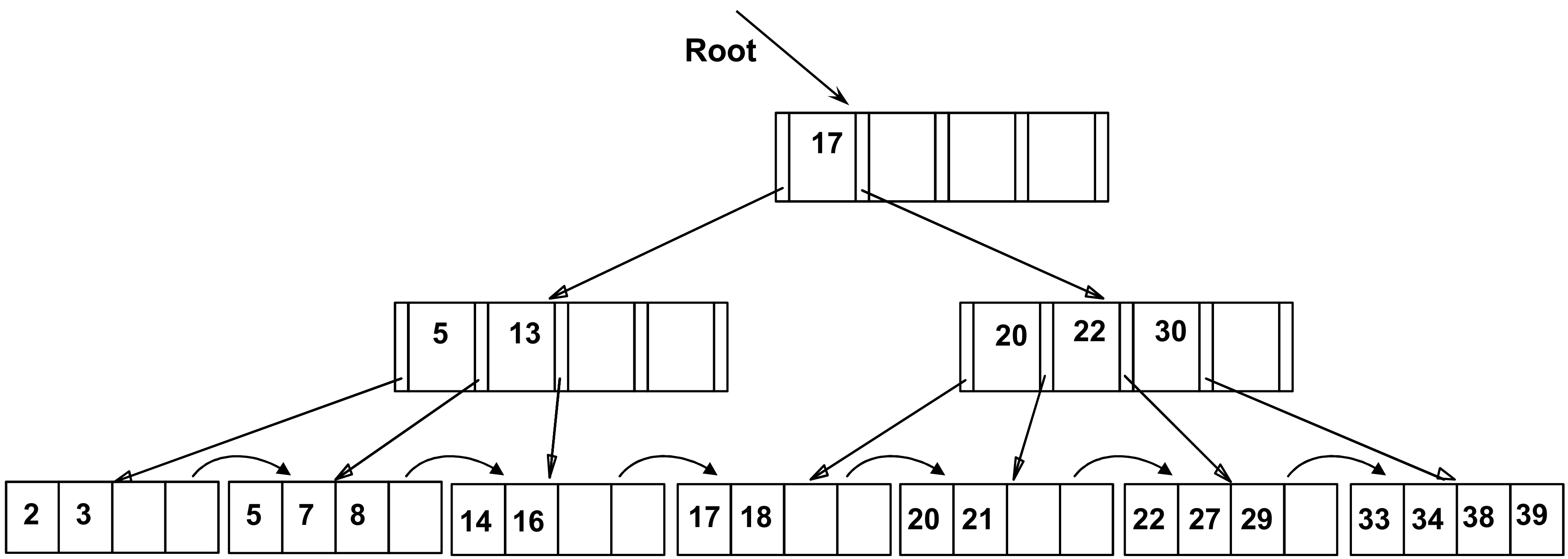


Re-distribution



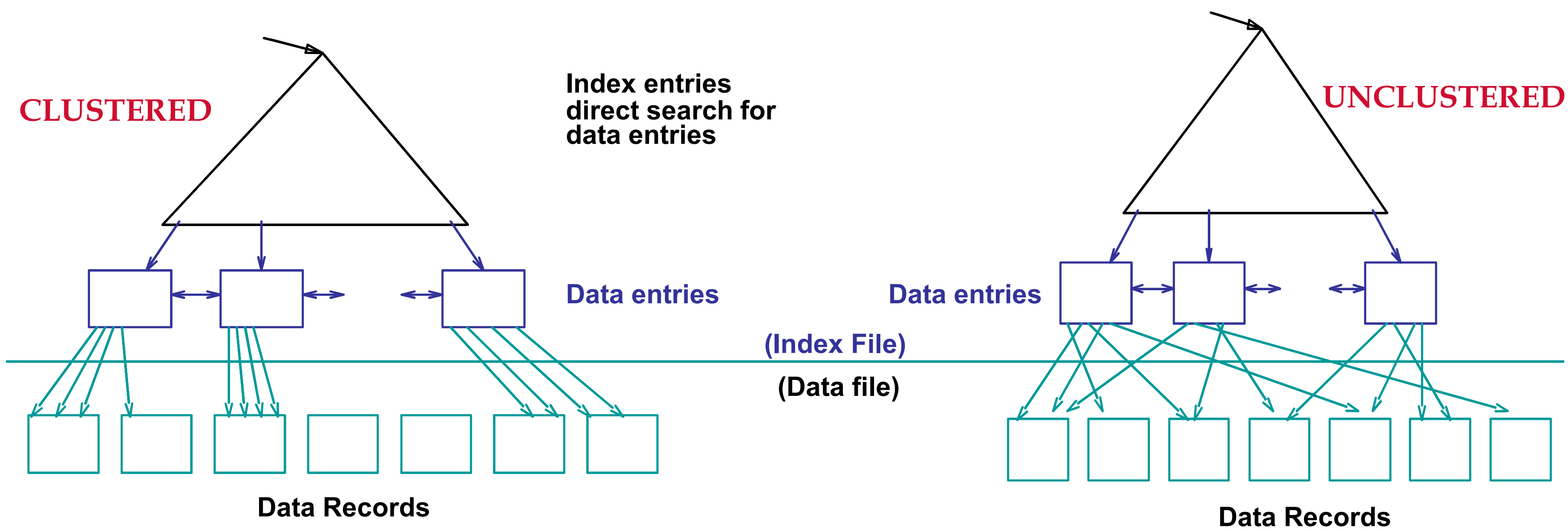
After Re-distribution

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



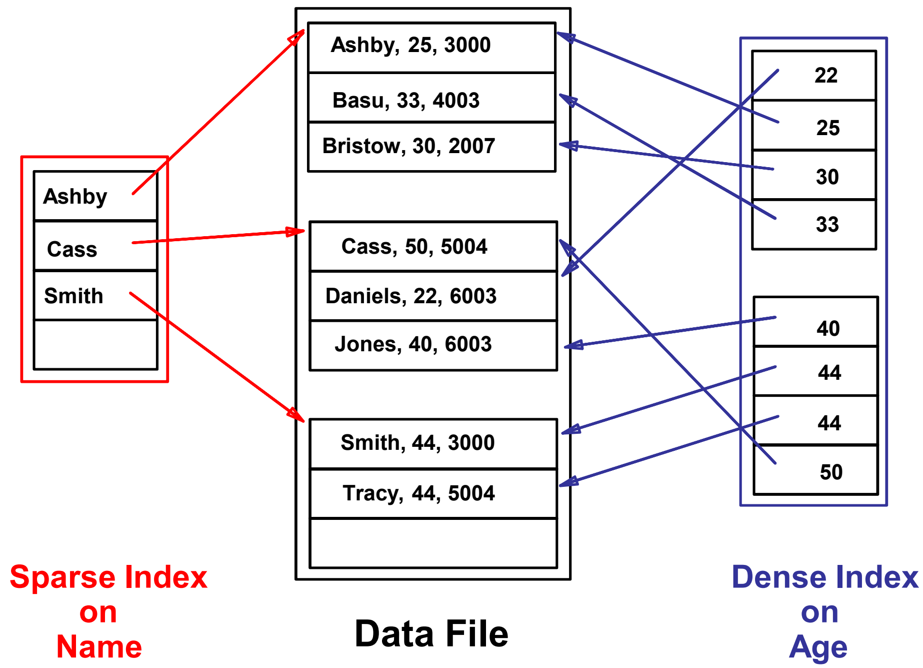
Clustered vs Unclustered Index

- An index is a clustered index if the order of its data entries is the same as or "close to" the order of the data records; otherwise, it is an unclustered index



Dense vs. Sparse

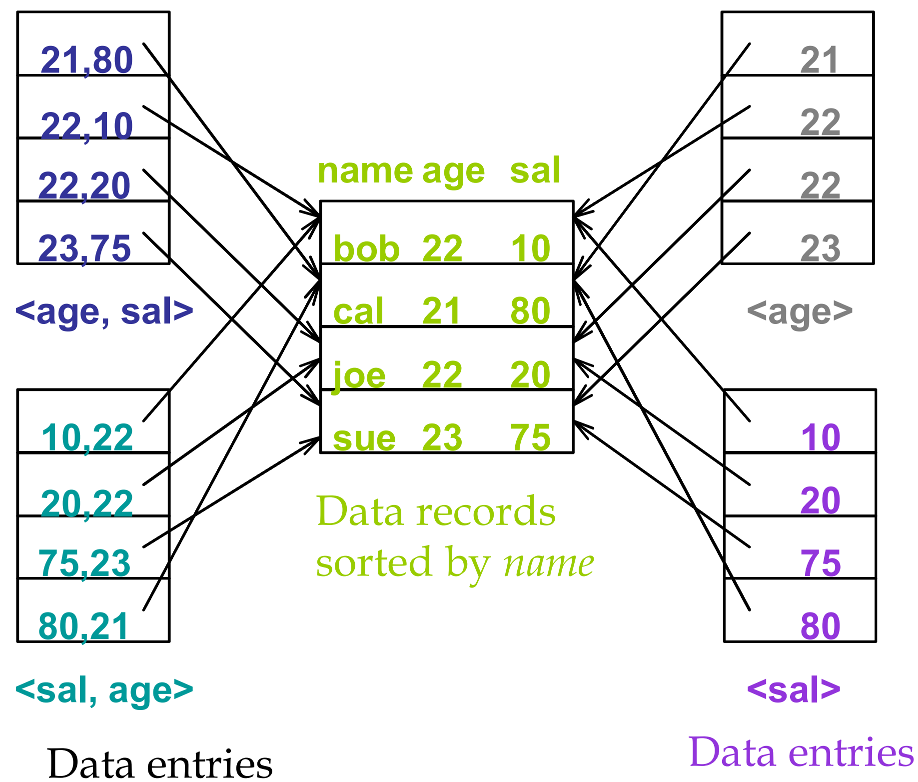
- Dense index
  - There is at least one data entry per search key value (in some data record)
  - B+-tree is a dense index
- Sparse index
  - Every sparse index is clustered
  - Sparse indexes are smaller
  - Sparse index cannot support “exists” search



Multi-attribute Indexes

- *Composite Search Keys*: Search on a combination of fields
  - Equality query: Every field value is equal to a constant value. E.g. wrt <age,sal> index:
    - age=22 & sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age=22 & sal > 10 (use <age, sal>)
    - age < 22 & sal = 10 (use <age,sal> may fetch more records than desired)
- There are also multi-attribute indexing structures (e.g., R-trees, Grid-file)

Examples of composite key indexes



Summary

- B+-tree index can facilitate fast search for both single record and range search
  - Storage-friendly (works well on any kind of storage)
  - good performance
  - universal applicability