Indexing

What is an Index?

- A simple index is a table containing an ordered list of keys and reference fields.
  - e.g. the index of a book
- In general, indexing is another way to handle the searching problem.

Uses of an index

1. An index lets us **impose order on a file** without rearranging the file.
2. Indexes provide **multiple access paths** to a file.
   - e.g. library catalog providing search for author,book and title
3. An index can provide **keyed access to variable-length record** files.

A simple index for a pile file

	Label	ID	Title	Composer	Artist
17	LON	2312	Symphony No.9	Beethoven	Giulini
62	RCA	2626	Romeo and Juliet	Prokofiev	Maazel
117	WAR	23699	Nebraska	…	
152	ANG	3795	Violin Concerto	…	

Address of record (i.e. Byte offset)

Primary key = (Label, ID)

- Index is **sorted** (in main memory).
- Records appear in file in the order they are entered.

Index array:

Key	Reference
ANG3795	152
LON2312	17
RCA2626	62
WAR23699	117

- How to search for a recording with given LABEL ID?
  - **Binary search** in the index and then seek for the record in position given by the reference field.

Operations to maintain an indexed file

- Create the original empty index and data files.
- Load the index file into memory before using it.
- Rewrite the index file from memory after using it.
- Add data records to the data file.
- Delete records from the data file
- Update records in the data file.
- Update the index to reflect changes in the data file

Rewrite the index file from memory

- When the data file is closed, the index in memory needs to be written to the index file.
- An important issue to consider is what happens if the rewriting does not take place (e.g. power failures, turning machine off, etc.)
- Two important safeguards:
  - Keep a status flag in the header of the index file.
  - If the program detects the index is out of date it calls a procedure that reconstructs the index from the data file.

Record Addition

1. Append the new record to the end of the data file.
2. Insert a new entry to the index in the right position.
   – needs rearrangement of the index

Note: this rearrangement is done in the main memory.

Record Deletion

- Use the techniques for reclaiming space in files when deleting records from the data file.
- We must also delete the corresponding entry from the index in memory.

Record Updating

There are two cases to consider:

1. The update changes the value of the key field:
   – Treat this as a deletion followed by an insertion
2. The update does not affect the key field:
   – If record size is unchanged, just modify the data record. If record size is changed treat this as a delete/insert sequence.

Indexing by Multiple Keys

- We could build additional indexes for a file to provide multiple views of a data file.
  - e.g. Find all recordings of Beethoven's work.
- LABEL ID is a **primary key.**
- There may be **other search keys**: title, composer, artist.
- We can build **secondary indexes**.

Composer index:

Composer	Primary key
Beethoven	ANG3795
Beethoven	DG139201
Beethoven	DG18807
Beethoven	RCA2626
Corea	WAR23699
Dvorak	COL31809
Prokofiev	LON2312

- Note that reference is to the primary key rather than to the byte offset.

Retrieval using combinations of secondary keys

- Secondary indexes are useful in allowing the following kinds of queries:
  - Find all recordings of Beethoven's work.
  - Find all recordings titled "Violin concerto"
  - Find all recordings with composer Beethoven and title Symphony No.9.
- Boolean operators "and", "or" can be used to combine secondary search keys to qualify a request.

Example

- The last query is executed as follows:

Matches from composer index	**Matches from title index**	**Matched list (logical "and")**
ANG3795	ANG3795	ANG3795
DG139201	COL31809	DG18807
DG18807	DG18807	
RCA2626		

Problems with simple indexes

If index does not fit in memory:

1. Seeking the index is slow (binary search):
   – We don't want more than 3 or 4 seeks for a search.

N	**Log(N+1)**
15 keys	4
1000	~10
100,000	~17
1,000,000	~20

2. Insertions and deletions take O(N) disk accesses.

Indexes too large to fit into Memory

- Two main alternatives:
  1. Tree-structured (multi-level) index such as B+trees.
  2. Hashed organization (when access speed is a top priority)

Multilevel Indexing and B+ Trees

Outline

- Single-level index
- Multi-level index
- B+tree index

All can be classified as:

- Dense vs. sparse index
- Primary vs. secondary index
- Clustered vs. unclustered index

Indexed Sequential Access

Provide a choice between two alternative views of a file:

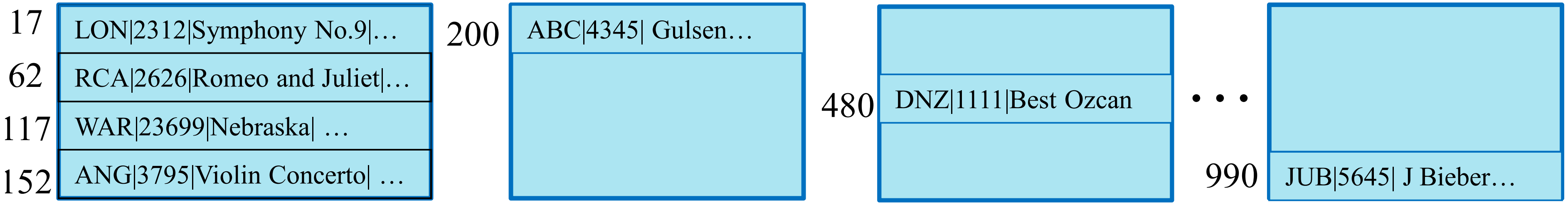
1. **Indexed**: the file can be seen as a set of records that is indexed by key; or
2. **Sequential:** the file can be accessed sequentially (physically contiguous records), returning records in order by key.

Example of applications

- Student record system in a university:
  - Indexed view: access to individual records
  - Sequential view: batch processing when posting grades
- Credit card system:
  - Indexed view: interactive check of accounts
  - Sequential view: batch processing of payments

Last week in Ceng351

- A **pile file** on disk:



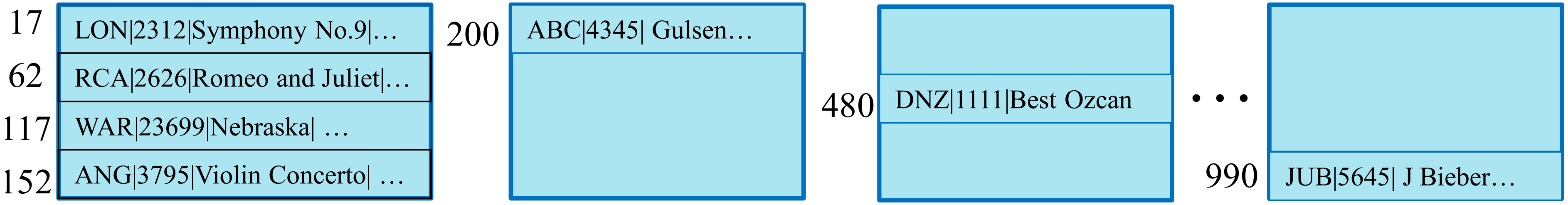
- An **in-memory** index:
  - Load to memory
  - Can do binary search
  - Can do add/del/update
  - (Re-)Write to disk

Key	Reference
ABC4345	200
ANG3795	152
DNZ1111	480
JUB5645	990
LON2312	17
RCA2626	62

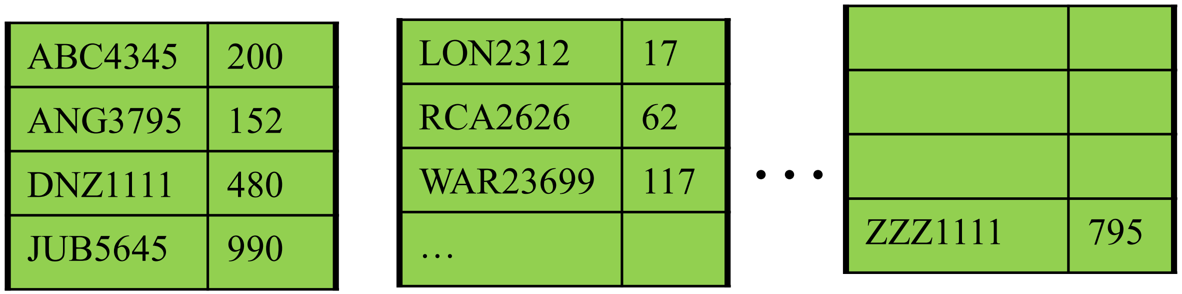
Problem: too large to fit into memory!

Last week in Ceng351

- A **pile file** on disk:



- Index as a **sorted sequential file**:



Key	Reference
ABC4345	200
ANG3795	152
DNZ1111	480
JUB5645	990
LON2312	17
RCA2626	62
WAR23699	117

Last week in Ceng351

- A **pile file** on disk:

$$T_F = (b/2) * \mathrm{btt}$$

Address	Record
17	LON|2312|Symphony No.9|…
62	RCA|2626|Romeo and Juliet|…
117	WAR|23699|Nebraska| …
152	ANG|3795|Violin Concerto| …

Address	Record
200	ABC|4345| Gulsen…

Address	Record
480	DNZ|1111|Best Ozcan

…

Address	Record
990	JUB|5645| J Bieber…

- Index as a **sorted sequential file**:

ABC4345	200
ANG3795	152
DNZ1111	480
JUB5645	990

LON2312	17
RCA2626	62
WAR23699	117
…	

…

ZZZ1111	795

TTT…	
GHR…	
AAB	
ZXE…	

…

BBB	
NNN	

Sorted area (x blocks) **Overflow area (y blocks)**

$$T_F = \log_2 x \; * (s + r + \mathrm{btt}) \; + s + r + (y/2) * \mathrm{btt}$$

Last week in Ceng351

- A **pile file** on disk:

$$T_F = (b/2) * \text{btt}$$

- Better fetch time (if no overflow)!
- How should we organize the index:
  - Based on underlying data file (pile or sorted seq. file)
  - Based on the properties of search key (indexing) field

- Index as a **sorted sequential file**:

ABC4345	200
ANG3795	152
DNZ1111	480
JUB5645	990

LON2312	17
RCA2626	62
WAR23699	117
…	

. . .

ZZZ1111	795

Sorted area (x blocks)

$$T_F = \log_2 x * (s + r + \text{btt})$$

The initial idea: Single level index

- The data file is ordered on a ***key field***.
  - The records are grouped into blocks in a sorted way.
- A single level index for these blocks:
  - Includes one index entry *for each block* in the data file; the index entry has the key field value, K(i), for the *first record* in the block, which is called the *block anchor*.
  - A similar scheme can use the *last record* in a block.
  - Index is an ordered file with entries (records) <K(i), P(i)>
    - We can still do binary search
    - Would be smaller than the data file

Single-level index on the ordering key field of the file.

- Data file is **ordered** on the (primary) **key field**, i.e, Name

Index:

K(i)	P(i)
Aaron	→ block 1
Adams	→ block 2
Alex	→ block 3
Allen, T	→ block 4
Ande Z	→ block 5
Arno	→ block 6
…	

Data file:

Name	Dept	ID
Aaron	2	e9
…		
Acosta	1	e7
Adams	5	e4
…		
Akers	1	e8
Alex…		
…		
Allen, S	1	e6
Allen, T…		
…		
Ande R	3	e21
Ande Z		
…		
Archer	4	e1
Arno	2	e117
…		

The **index** is an **ordered file** with two fields

- K(i): One index entry for each block → block anchors's PK value
- P(i): Block pointer

Single-level index on the ordering key field of the file.

- Data file is **ordered** on the (primary) **key field**, i.e, Name

K(i)	P(i)
Aaron	
Adams	
Alex	
Allen, T	

Ande Z	
Arno	
…	

Name Dept ID

Aaron	2	e9
…		
Acosta	1	e7

Adams	5	e4
…		
Akers	1	e8

Alex…		
…		
Allen, S	1	e6

Allen, T…		
…		
Ande R	3	e21

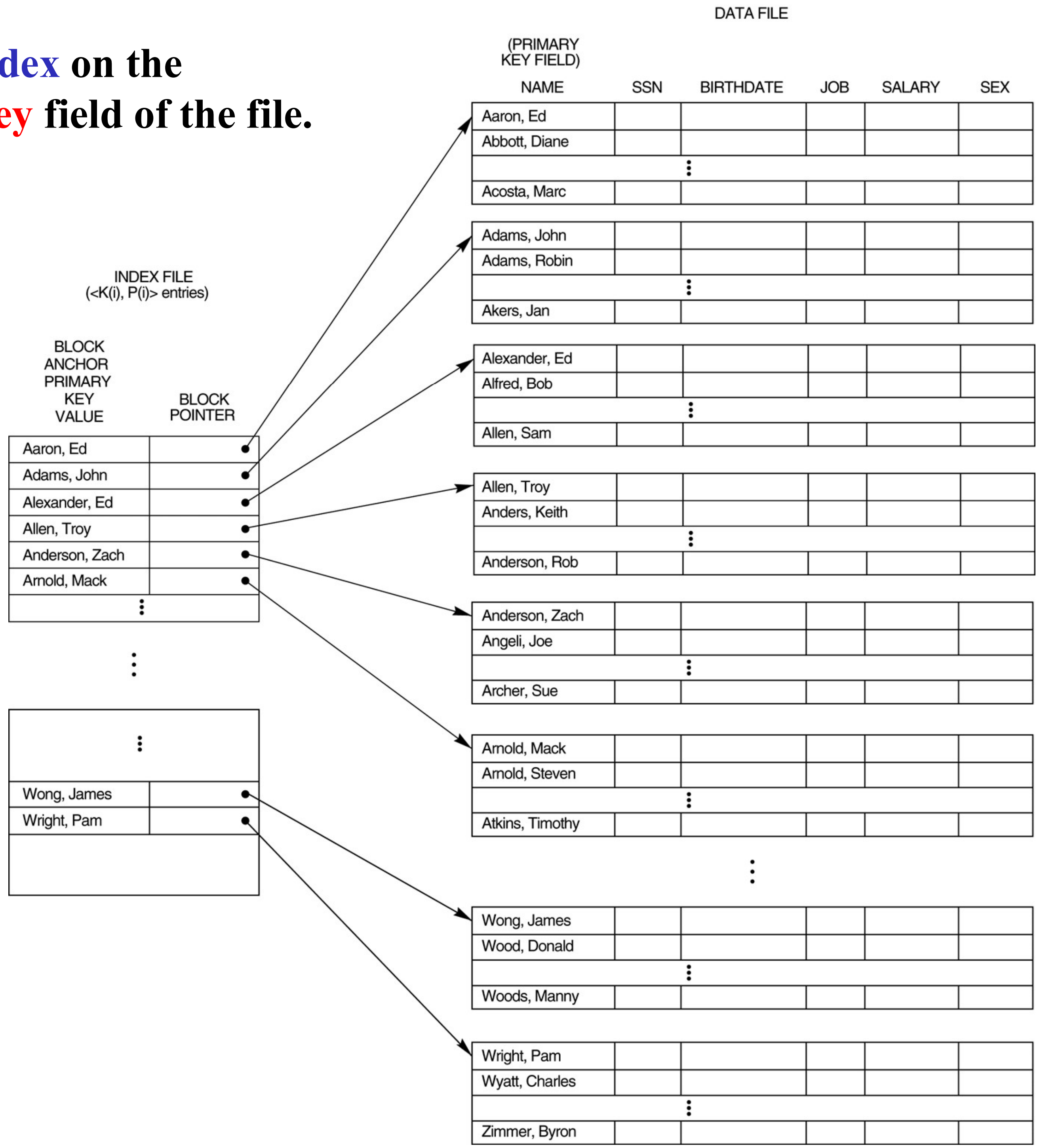
Ande Z		
…		
Archer	4	e1

Arno	2	e117
…		

The **index** is an **ordered file** with two fields

- K(i): One index entry for each block → block anchors's PK value
- P(i): Block pointer

Single-level index on the ordering key field of the file.



Important Concepts

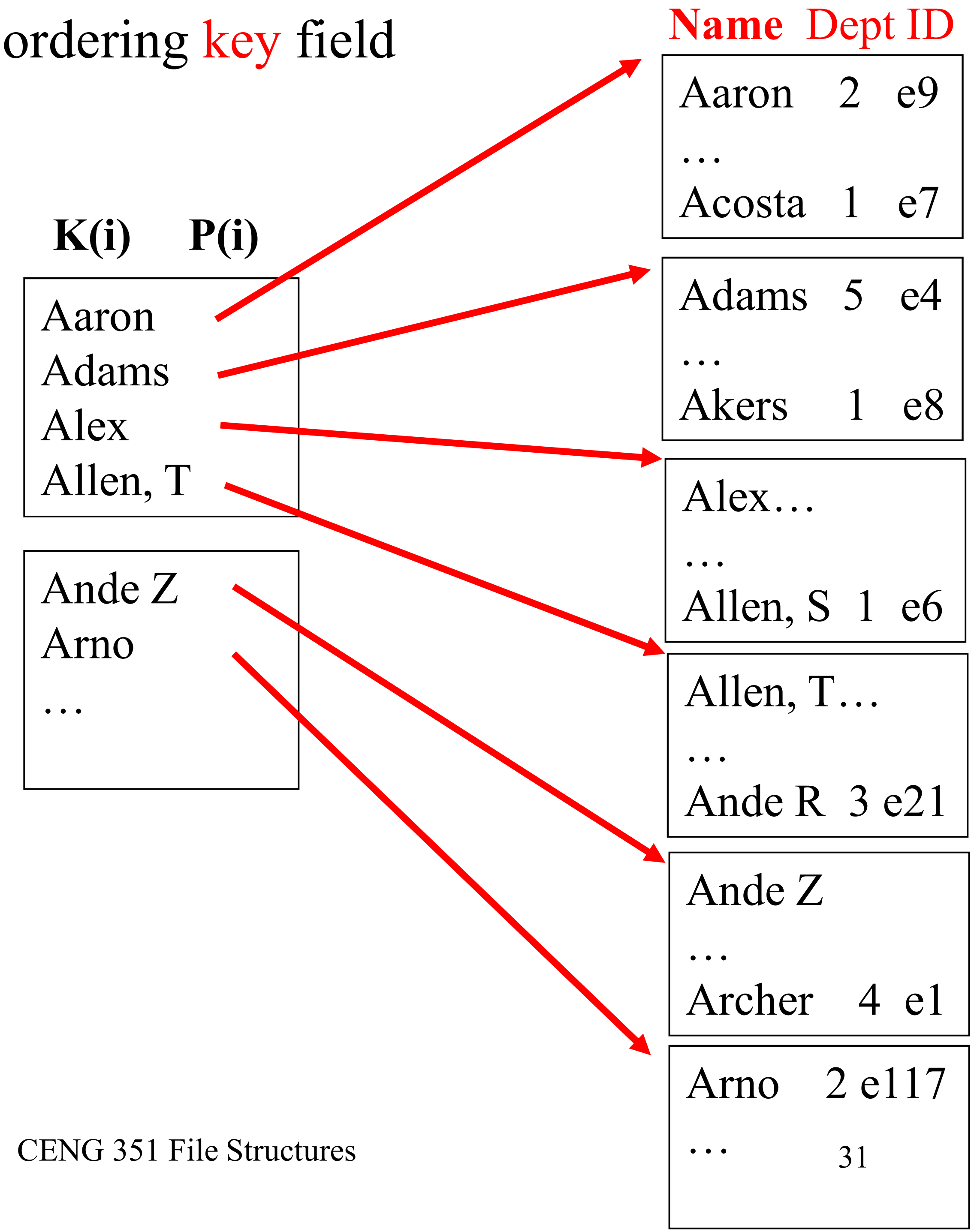
- A **primary index** is specified on the *ordering key field* of an ordered file of records
  - Alternative def (RB): An index on the primary key field is called a primary index
- If the ordering of the index and data records is the same (or, close), we call this a **clustering index**
  - So, a primary index is also clustering!
  - We can also have clustering index on an *ordering non-key* field
  - But, since a file can have only one physical ordering, it can have **at most one** primary/clustering index

Important Concepts

- A **dense index** has an index entry for *every* search key value (and hence, every record) in the data file. A **sparse (non-dense)** index has index entries for only *some* search values.

Single-level index on the ordering key field of the file.

- This is a **primary index,** because data is ordered on the primary key field (no duplicates).
- This is also a **clustered index** because the data is in the same order as the search key.
- This is also a **sparse (nondense)** index, since it includes an entry for each disk block of the data file (not for each record of the data file).



Another Example of a Clustered Single-Level Index

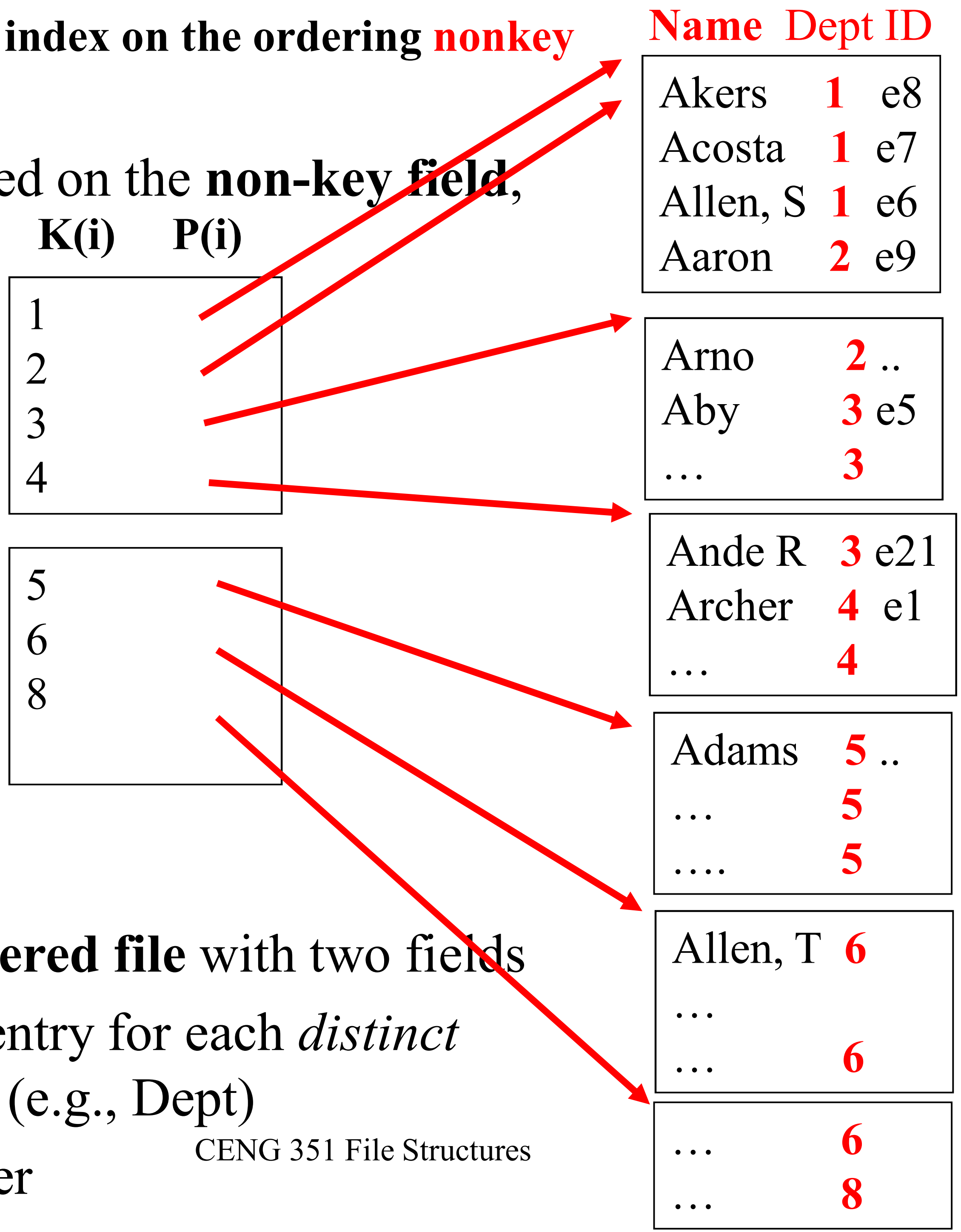
- The data file is ordered on a ***non-key field*** (unlike primary index, which requires that the ordering field of the data file have a distinct value for each record).

- Includes one index entry *for each distinct value* of the field; the index entry points to the *first data block* that contains records with that field value.

- It is another example of *sparse (nondense)* index.

Single-level clustering index on the ordering nonkey field

- Data file is ordered on the **non-key field**, i.e, **Dept**

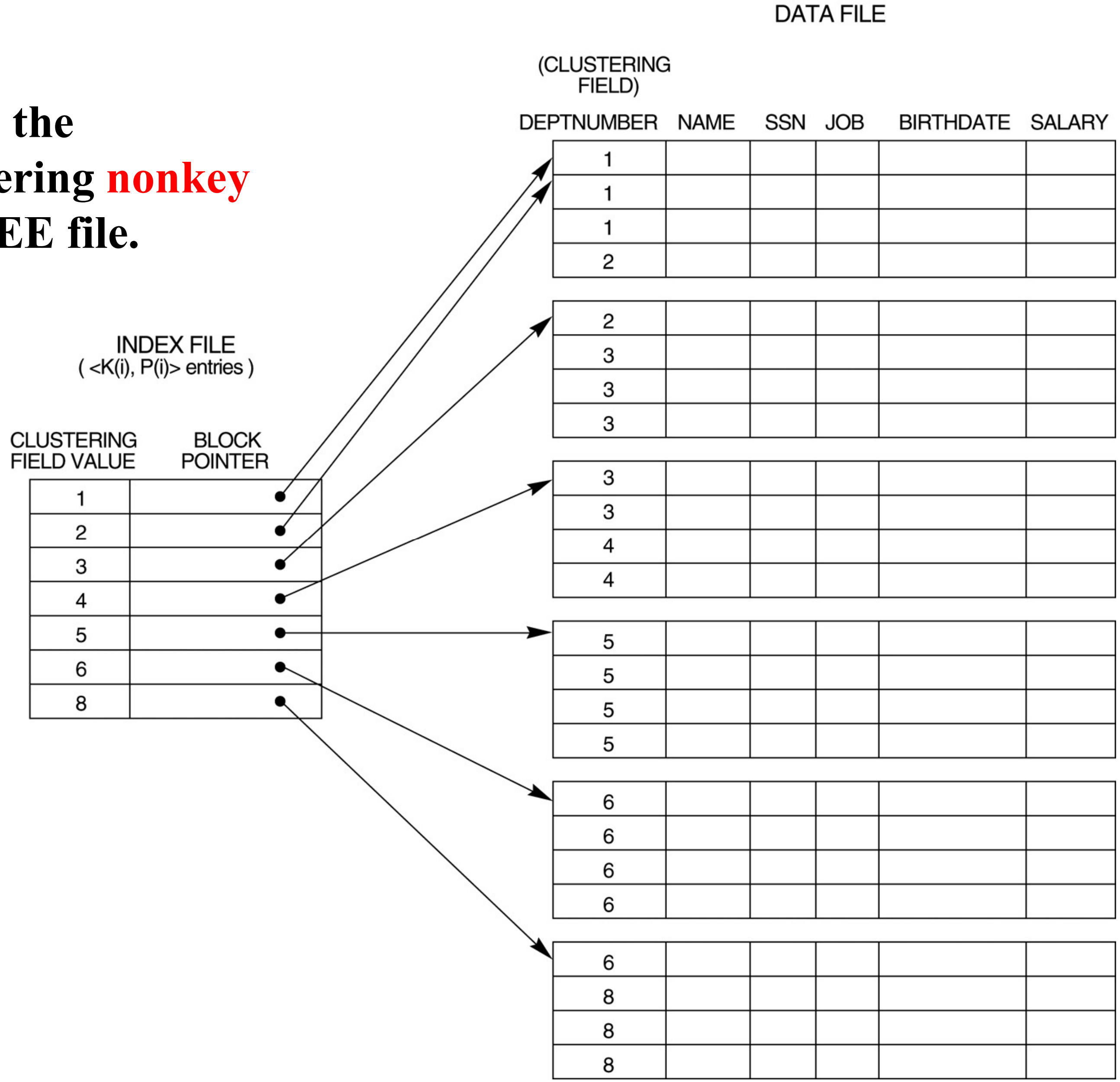


The **index** is an **ordered file** with two fields

- K(i): One index entry for each *distinct value* of the field (e.g., Dept)
- P(i): Block pointer

Single-level index

A clustering index on the DEPTNUMBER ordering nonkey field of an EMPLOYEE file.

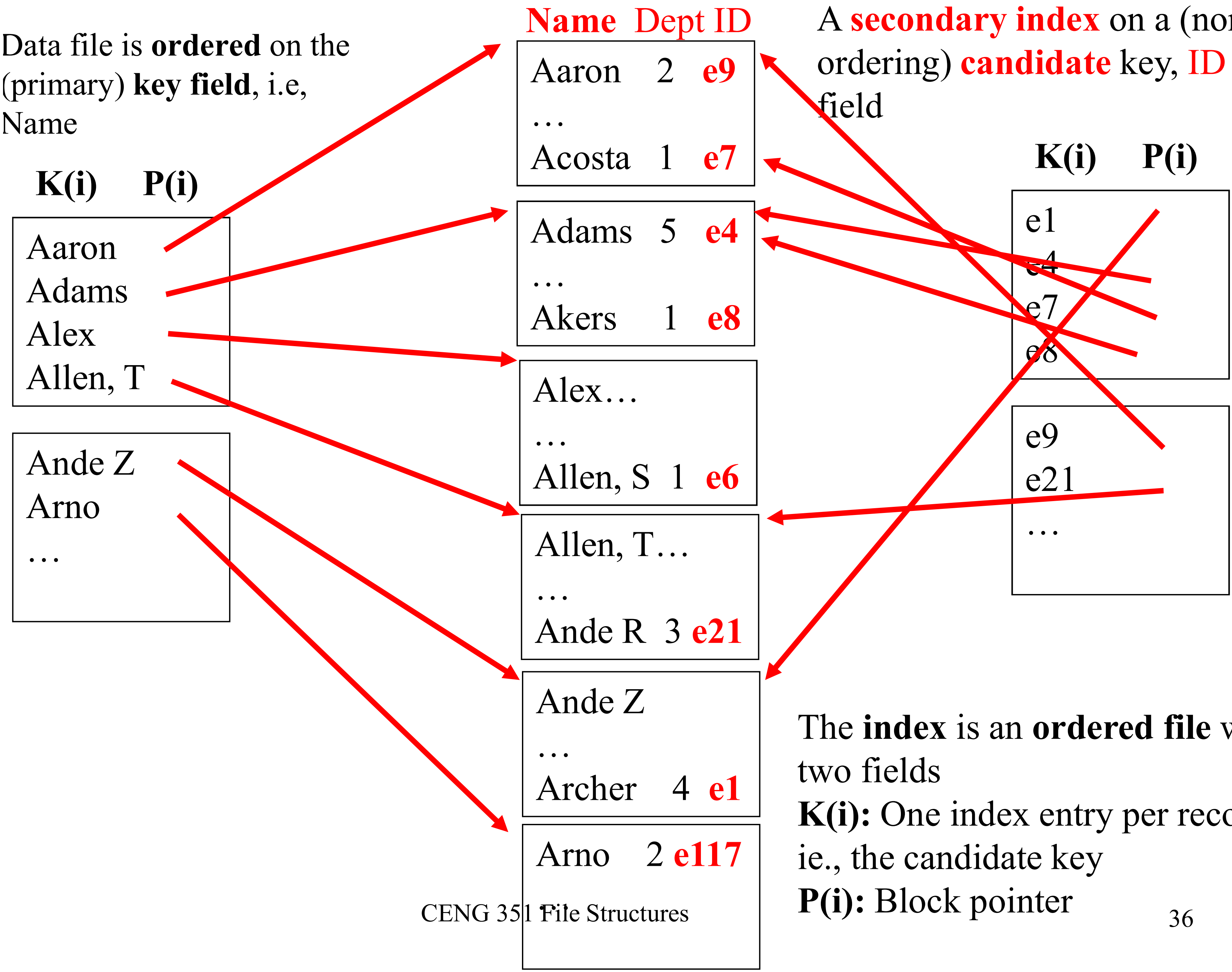


Single-Level Secondary Index

- A secondary index provides a secondary means of accessing a file for which some primary access already exists.
- The secondary index may be on a *non-ordering* field that is either
  - a candidate key and has a unique value in every record, or
  - a nonkey with duplicate values.
- The **index** is an **ordered file** with two fields:
  - The first field is the *indexing field*.
  - The second field is either a *block* pointer or a *record* pointer.
- Includes one entry *for each record* in the data file; hence, it is a ***dense index.***
- There can be *many* secondary indexes for the same file.

Data file is **ordered** on the (primary) **key field**, i.e, Name

A **secondary index** on a (non-ordering) **candidate** key, ID field



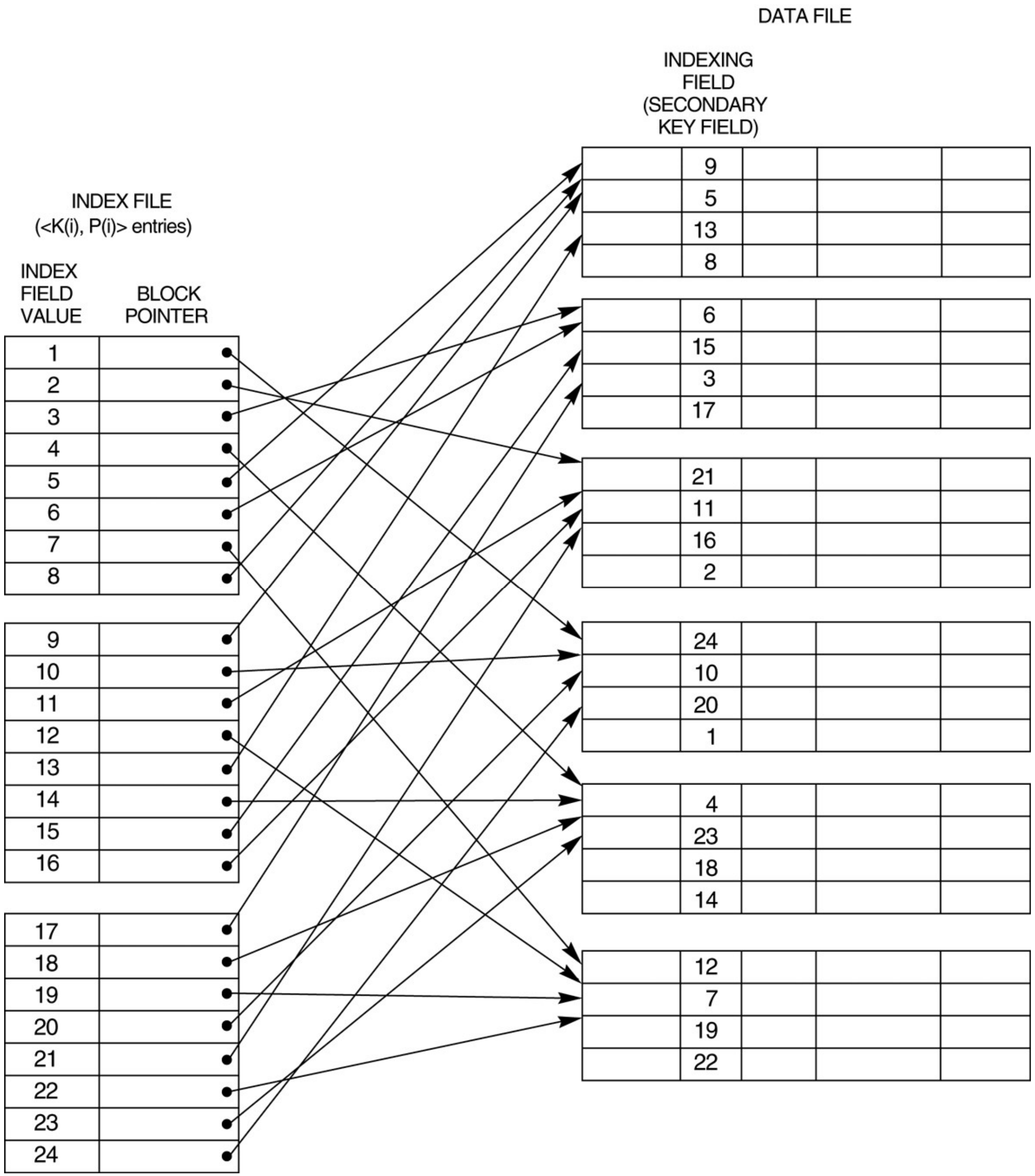
The **index** is an **ordered file** with two fields

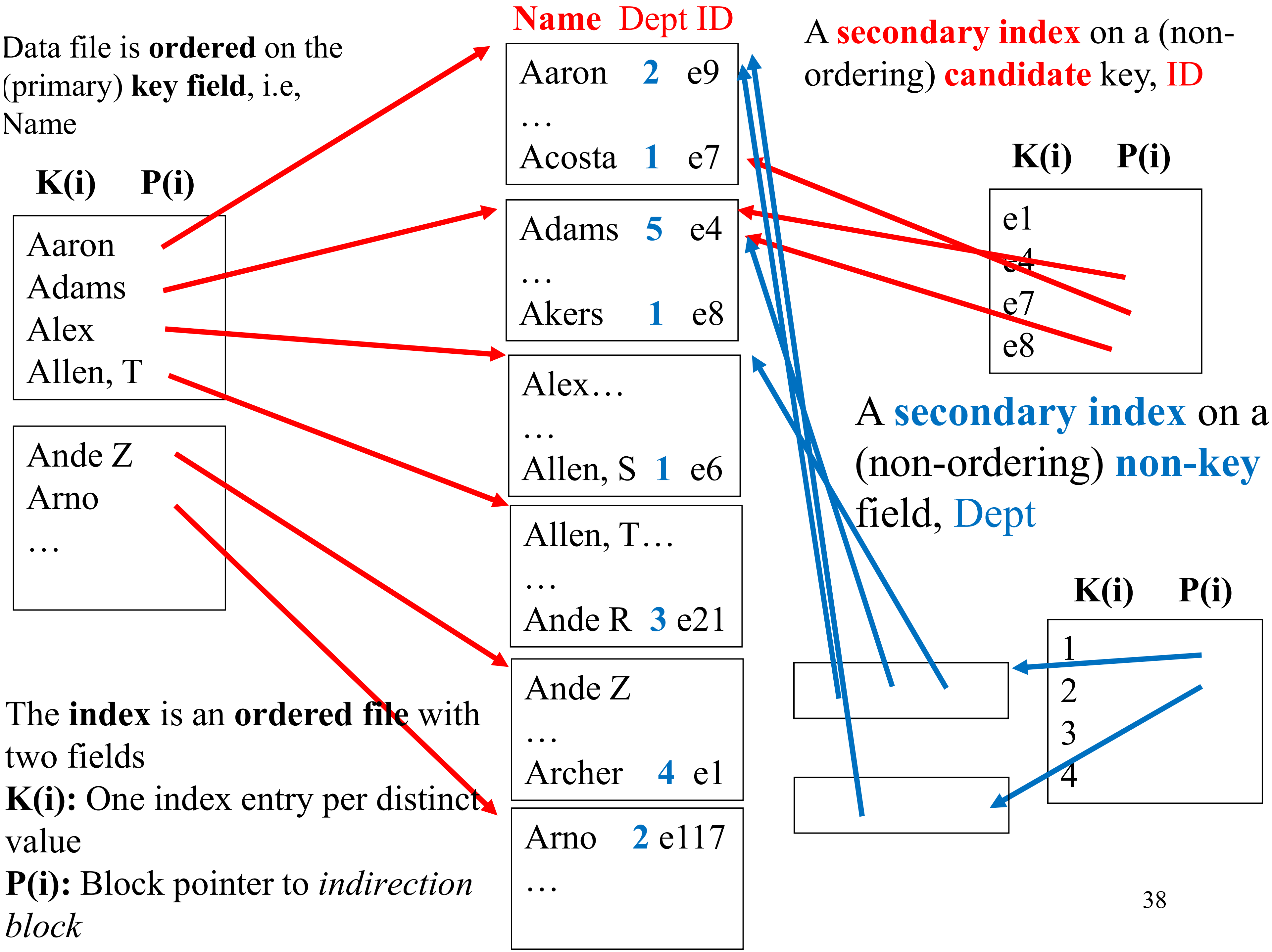
K(i): One index entry per record, ie., the candidate key

P(i): Block pointer

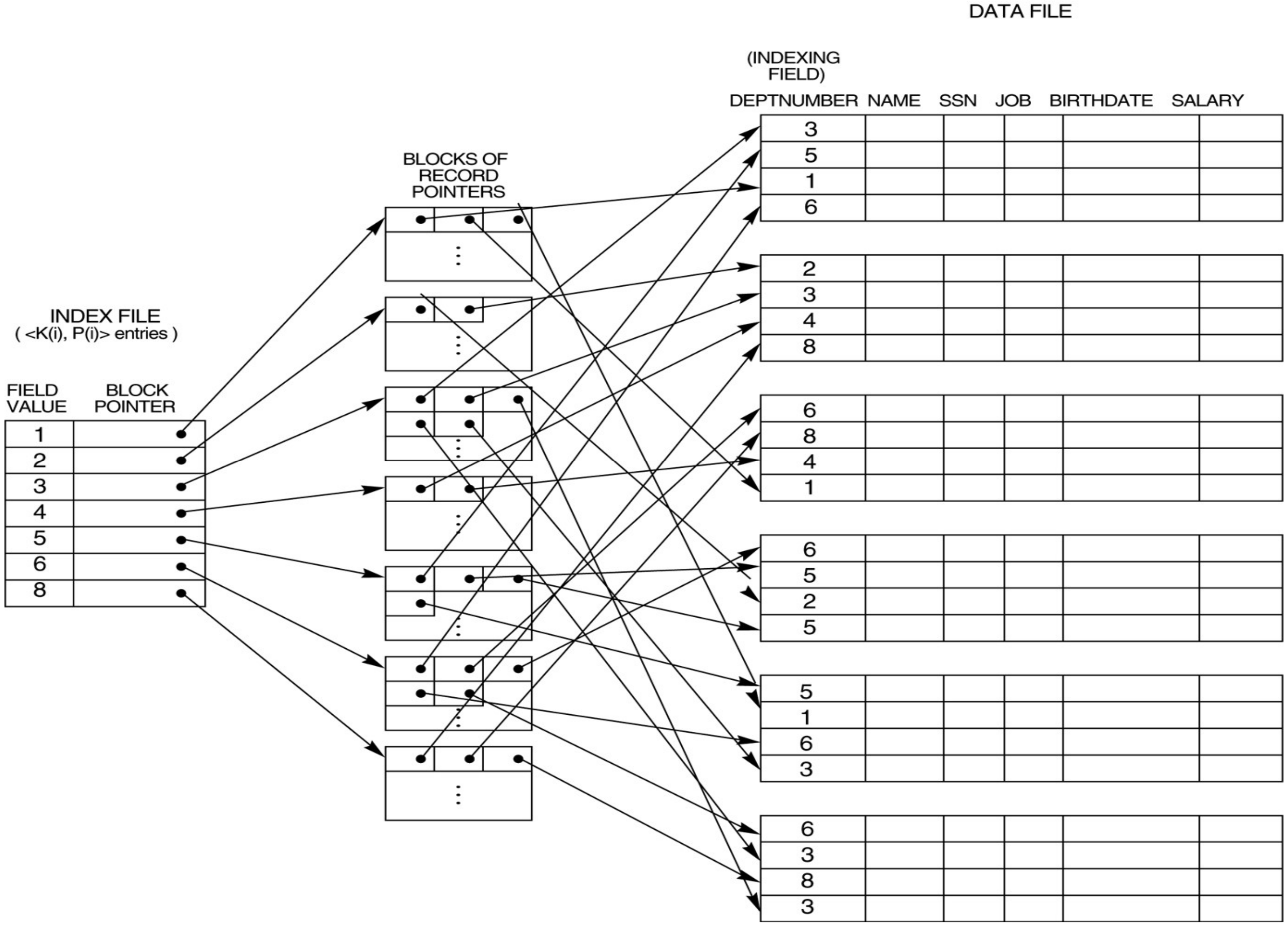
A **secondary index** on a candidate key (with block pointers) This is a **dense** index. No duplicates.

Note that the data file is *not* ordered according to the index field. Therefore it is an **unclustered** index



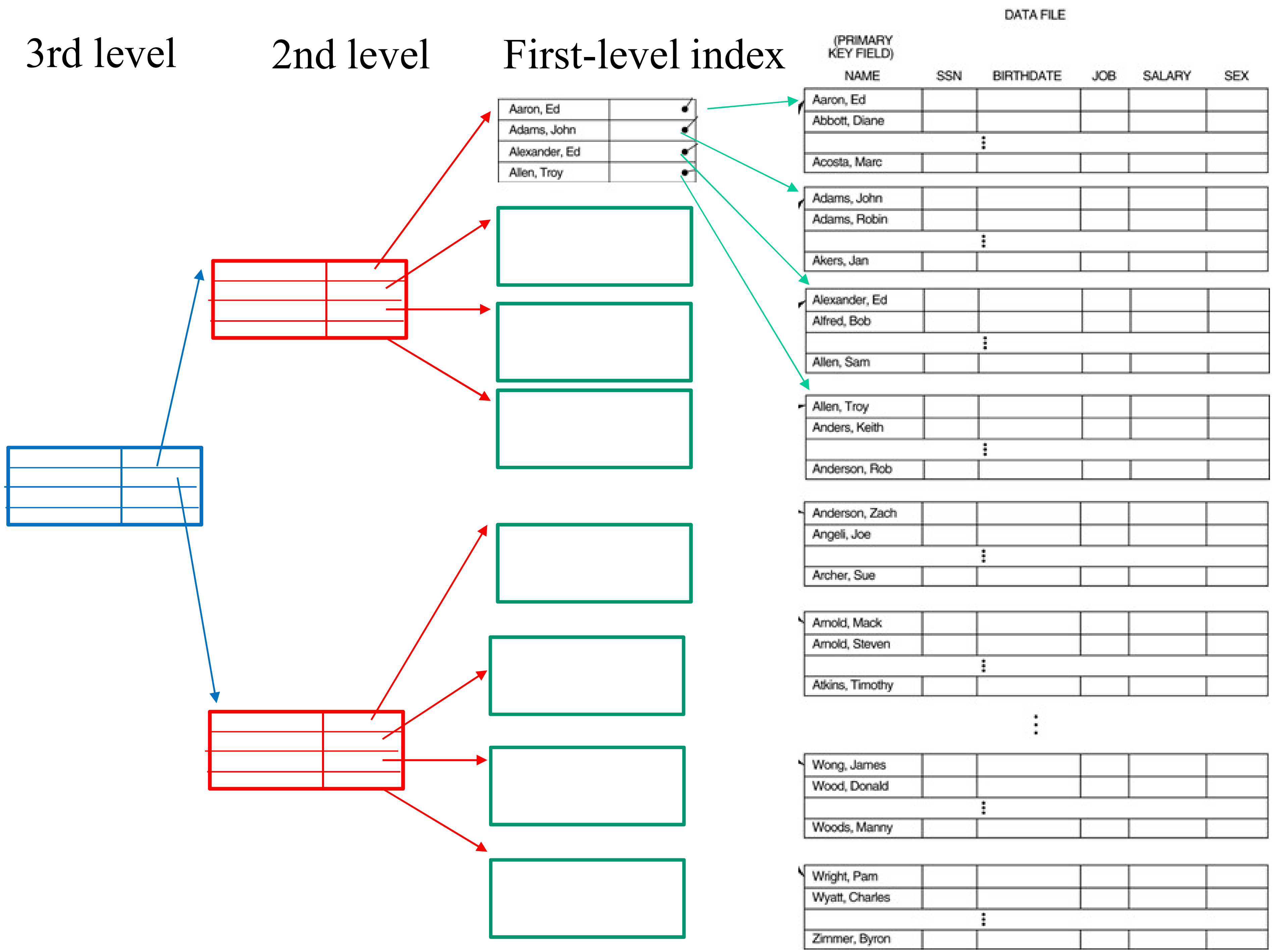

Data file is ordered on the (primary) key field, i.e, Name
K(i) P(i)
Aaron
Adams
Alex
Allen, T
Ande Z
Arno
…
Name Dept ID
Aaron 2 e9
…
Acosta 1 e7
Adams 5 e4
…
Akers 1 e8
Alex…
…
Allen, S 1 e6
Allen, T…
…
Ande R 3 e21
Ande Z
…
Archer 4 e1
Arno 2 e117
…
A secondary index on a (non-ordering) candidate key, ID
K(i) P(i)
e1
e4
e7
e8
A secondary index on a (non-ordering) non-key field, Dept
K(i) P(i)
1
2
3
4
The index is an ordered file with two fields
K(i): One index entry per distinct value
P(i): Block pointer to indirection block

A secondary index (with record pointers) on a nonkey field implemented using one level of indirection so that index entries are of fixed length and have unique field values. This is an unclustered index.

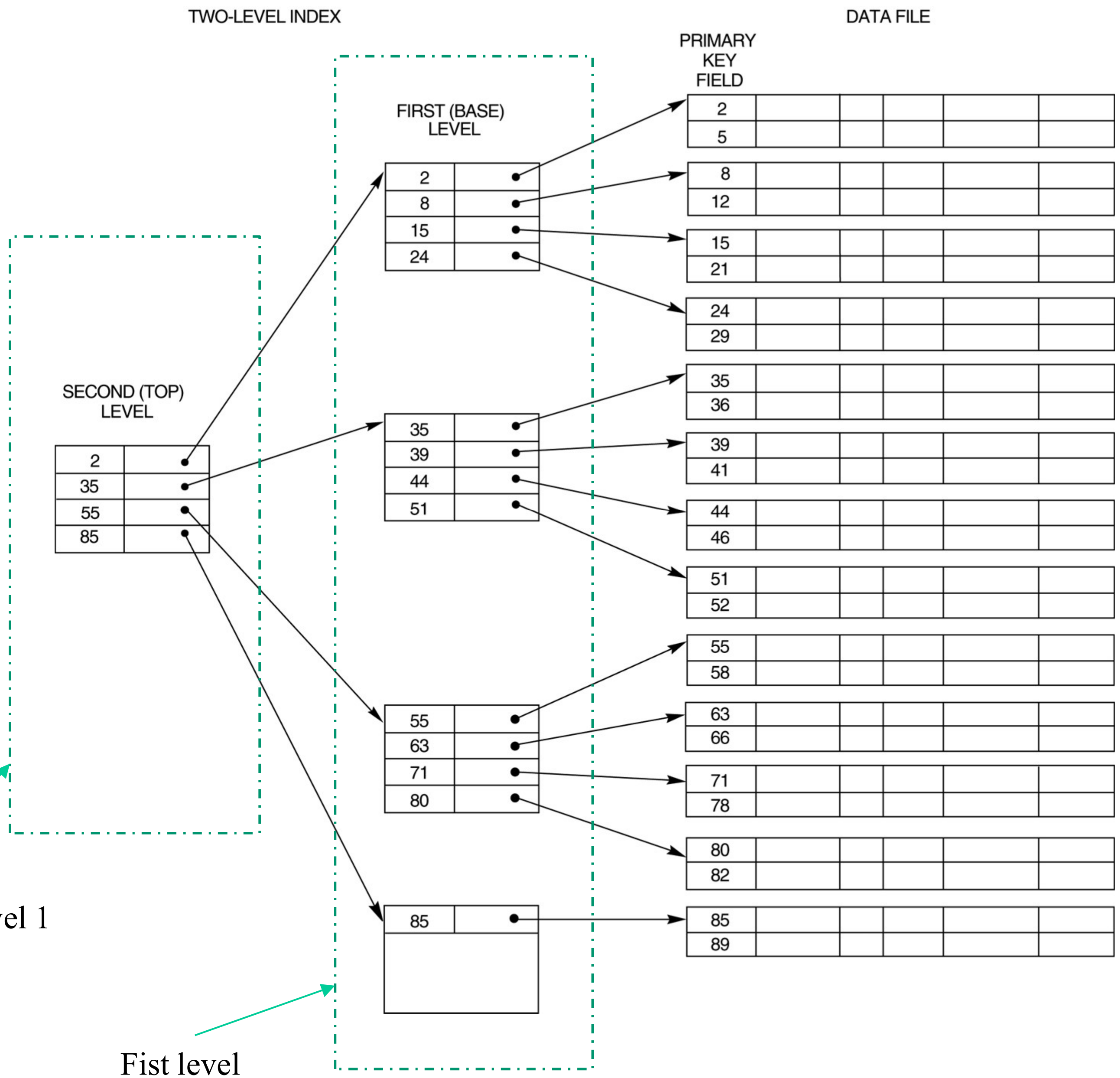


Multi-Level Indexes

- Because a single-level index is an ordered file, we can create an index *to the index itself* ; in this case, the original index file is called the *first-level index* and the index to the index is called the *second-level index*.
- We can repeat the process, creating a third, fourth, ..., top level until all entries of the *top level* fit in one disk block
- A multi-level index can be created for any type of first-level index (primary, secondary, clustering).


3rd level
2nd level
First-level index
DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
Aaron, Ed
Adams, John
Alexander, Ed
Allen, Troy
Aaron, Ed
Abbott, Diane
Acosta, Marc
Adams, John
Adams, Robin
Akers, Jan
Alexander, Ed
Alfred, Bob
Allen, Sam
Allen, Troy
Anders, Keith
Anderson, Rob
Anderson, Zach
Angeli, Joe
Archer, Sue
Arnold, Mack
Arnold, Steven
Atkins, Timothy
Wong, James
Wood, Donald
Woods, Manny
Wright, Pam
Wyatt, Charles
Zimmer, Byron

A two-level, clustered, sparse, primary index



Second level 1

Fist level

Multi-Level Indexes

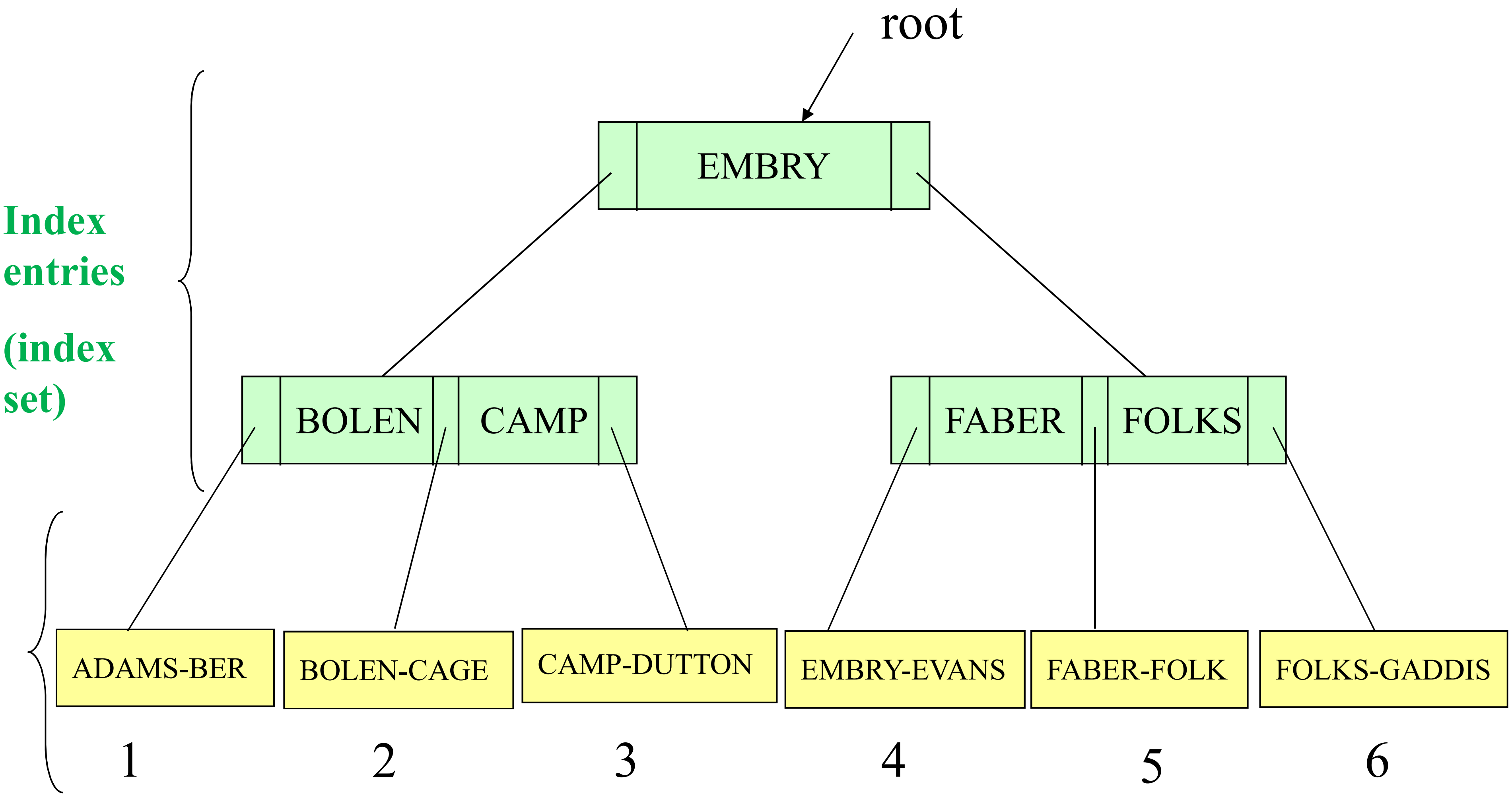
- Such a multi-level index is a form of *search tree*; however, insertion and deletion of new index entries is a severe problem because every level of the index is an *ordered file*.
- So this brings us to B+tree index structure.

Tree indexes

- If index doesn't fit in memory:
  - Divide the index structure into blocks,
  - Organize these blocks similarly building a tree structure.
- Tree indexes:
  - B Trees
  - B+ Trees
  - Simple prefix B+ Trees
  - …

B+ Trees

- B-tree is one of the most important data structures in computer science.
- What does B stand for? (Not binary!)
- B-tree is a **multiway search** tree.
- Several versions of B-trees have been proposed, but only B+ Trees have been used with large files.
- A B+tree is a B-tree in which data records are in leaf nodes, and faster sequential access is possible.



Index entries

(index set)

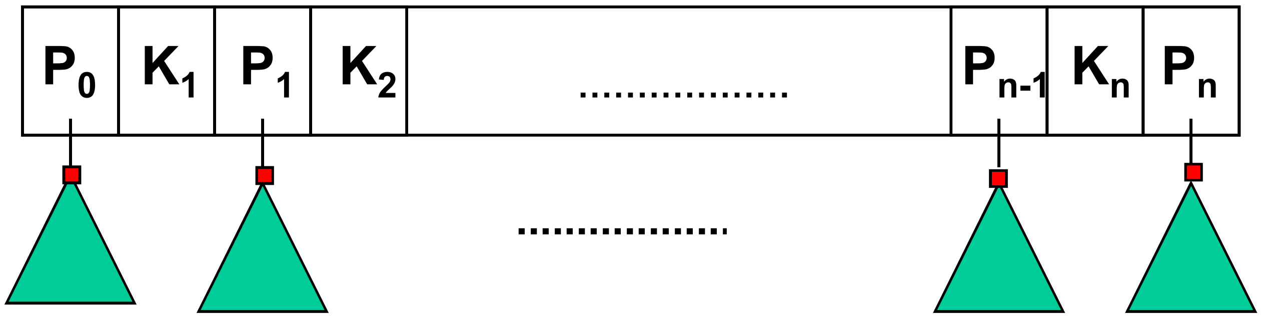
Data entries

(sequence set)

Formal definition of B+ Tree Properties

- Properties of a B+ Tree of order d:
  - All internal nodes (except root) have **at least d keys** and **at most 2d** keys.
  - Root can have at least **1 key** and **at most 2d** keys.
  - An internal node with **n** keys has **n+1** children
  - The root has at least 2 children unless it's a leaf.
  - All leaves are on the same level (balanced tree).

B+ tree: Internal/root node structure

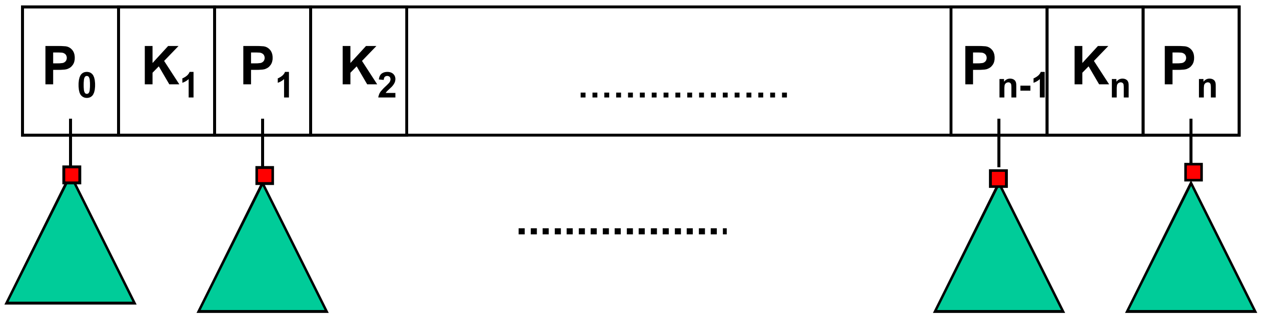


Each P_i is a pointer to a child node; each K_i is a search key value
\# of search key values = n, # of pointers = n+1

In a B+ Tree of order d:

- All internal nodes (except root) have **at least d keys** and **at most 2d** keys ($d \leq n \leq 2d$).
- Root can have at least **1 key** and **at most 2d** keys. ($1 \leq n \leq 2d$).
- An internal node with **n** keys has **n+1** children.
- The root has at least 2 children unless it's a leaf.

B+ tree: Internal/root node structure



Each P_i is a pointer to a child node; each K_i is a search key value
\# of search key values = n, # of pointers = n+1

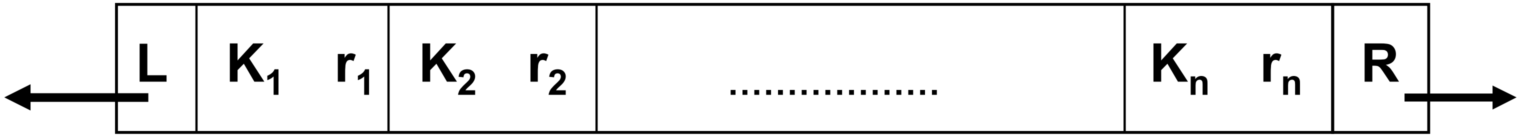
- Requirements:
- $K_1 < K_2 < \ldots < K_n$
- For any search key value K in the subtree pointed by Pi,

  If $P_i = P_0$, we require $K < K_1$

  If $P_i = P_n$, $K_n <= K$

  If $P_i = P_1, \ldots, P_{n-1}$, $K_i <= K < K_{i+1}$

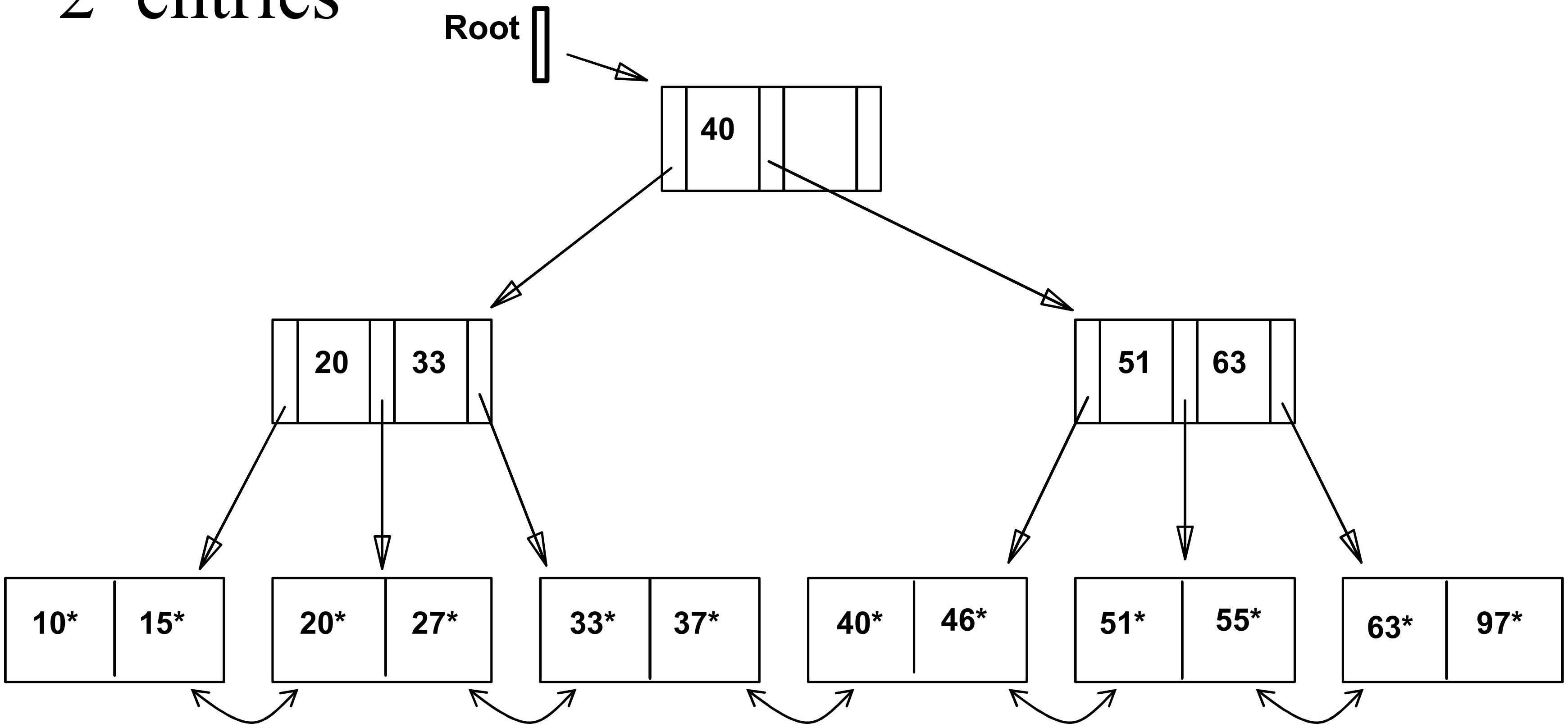
B+ tree: leaf node structure



- Pointer L points to the left neighbor; R points to the right neighbor (**doubly linked list**)
- $K_1 < K_2 < \ldots < K_n$
- $d \leq n \leq 2d$ (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
- All leaves are on the same level (balanced tree).

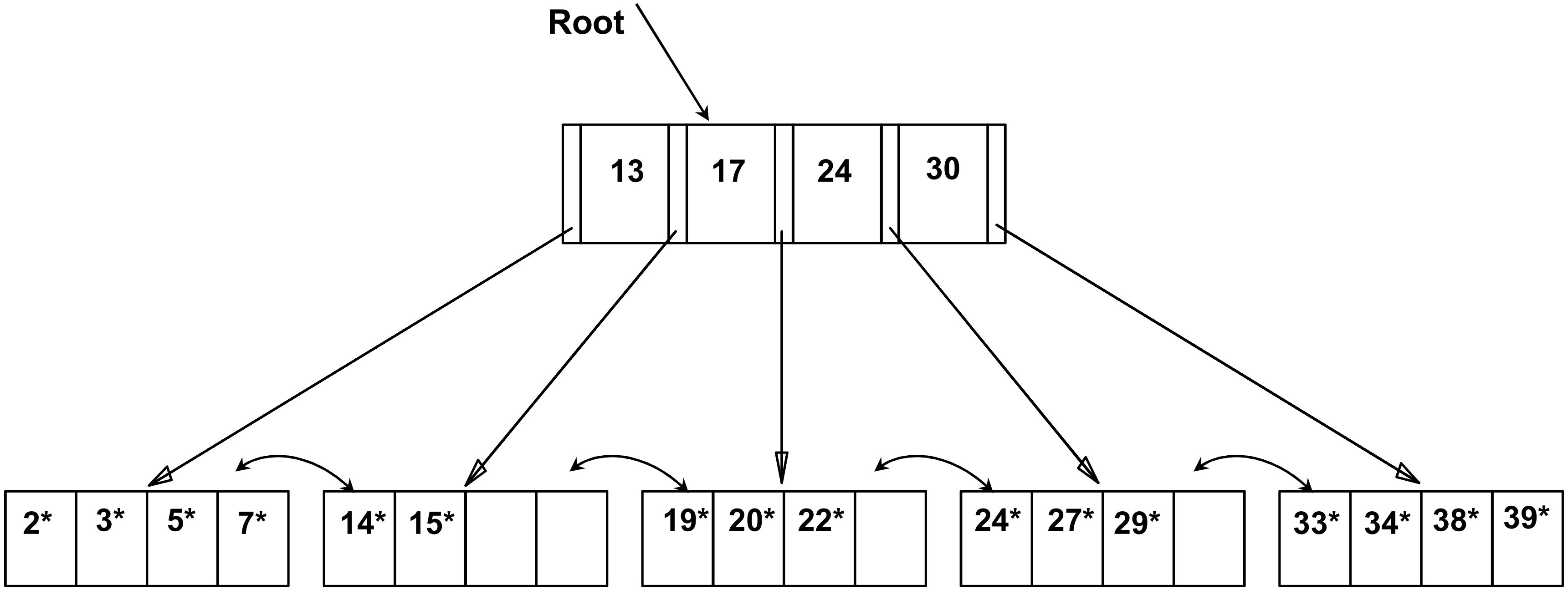
Example: B+ tree with order of 1

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



Example: Search in a B+ tree order 2

- Search: how to find the records with a given search key value?
  - Begin at root, and use key comparisons to go to leaf
- Examples: search for 5*, 16*, 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.

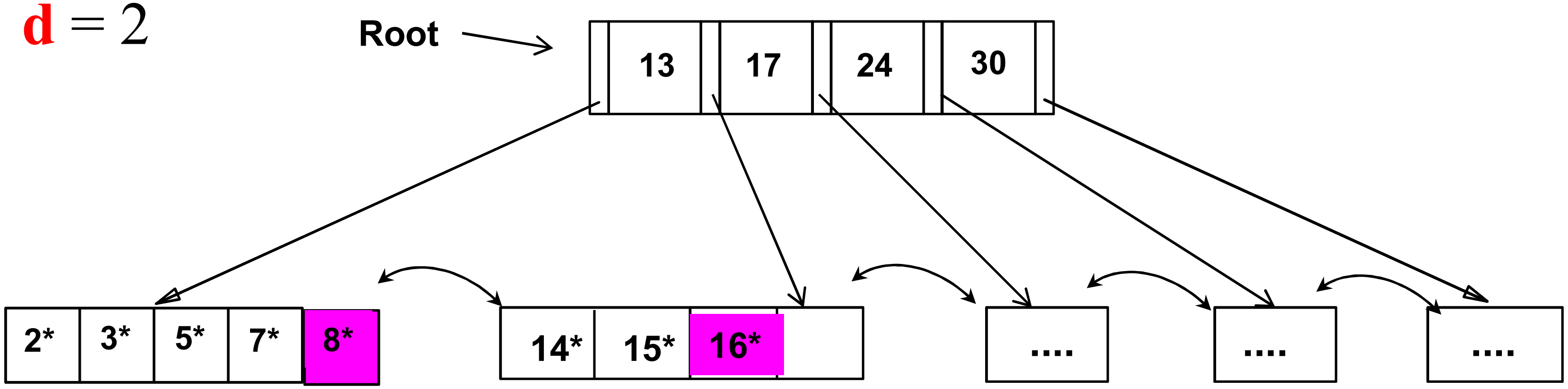


How to Insert a Data Entry into a B+ Tree?

- Let’s look at several examples first.

Inserting 16*, 8* into Example B+ tree

d = 2



Leaf node overflows!!!

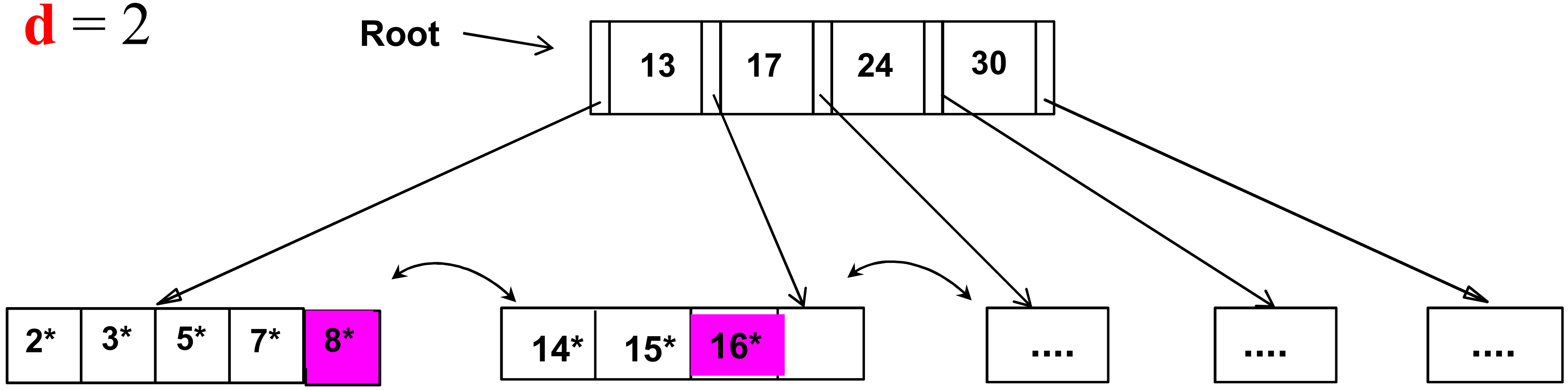
Leaf nodes:

$d \leq n \leq 2d$

$2 \leq n \leq 4$

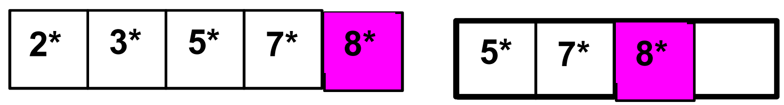
Inserting 16*, 8* into Example B+ tree

d = 2



Leaf node overflows!!!

When a leaf node overflows:

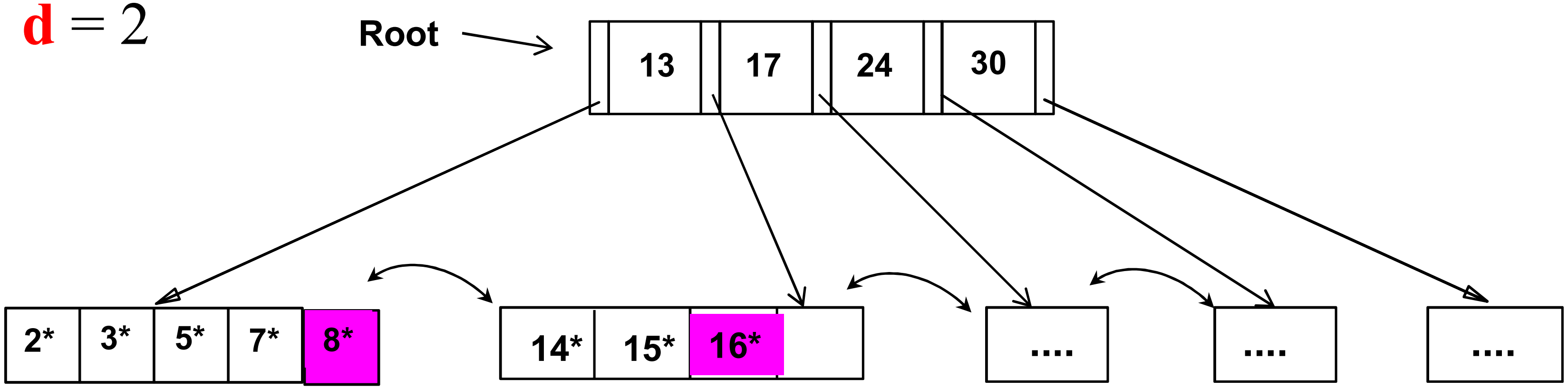


1) Split the node

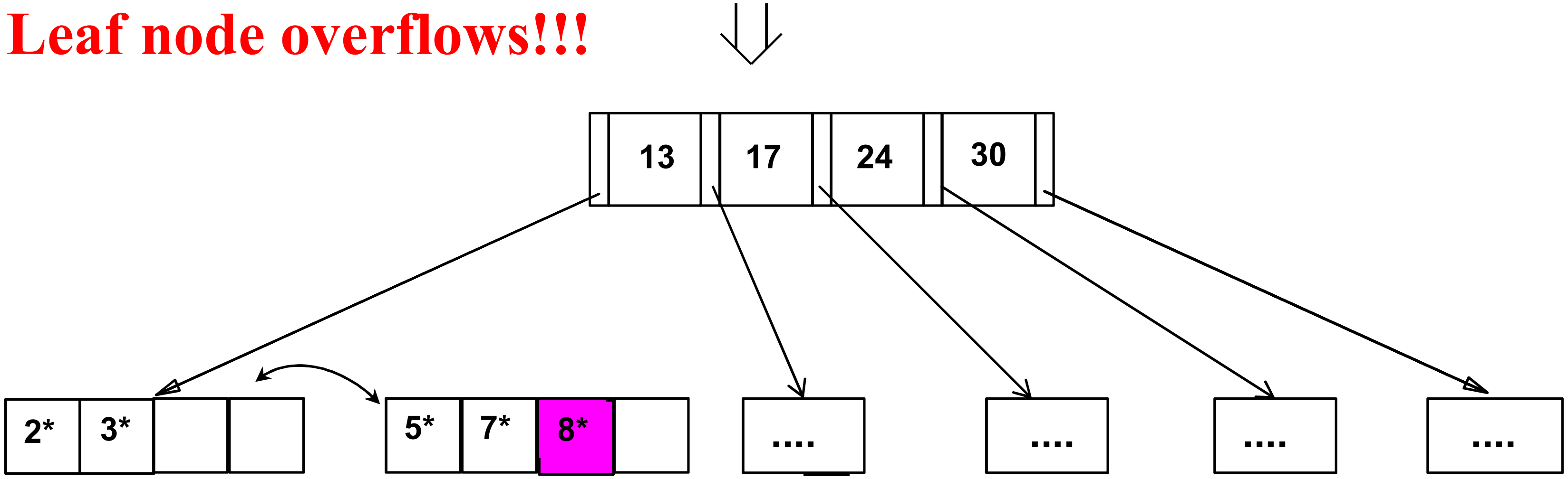
First **d** entries stay in old node , move rest of entries to new node

Inserting 16*, 8* into Example B+ tree

d = 2



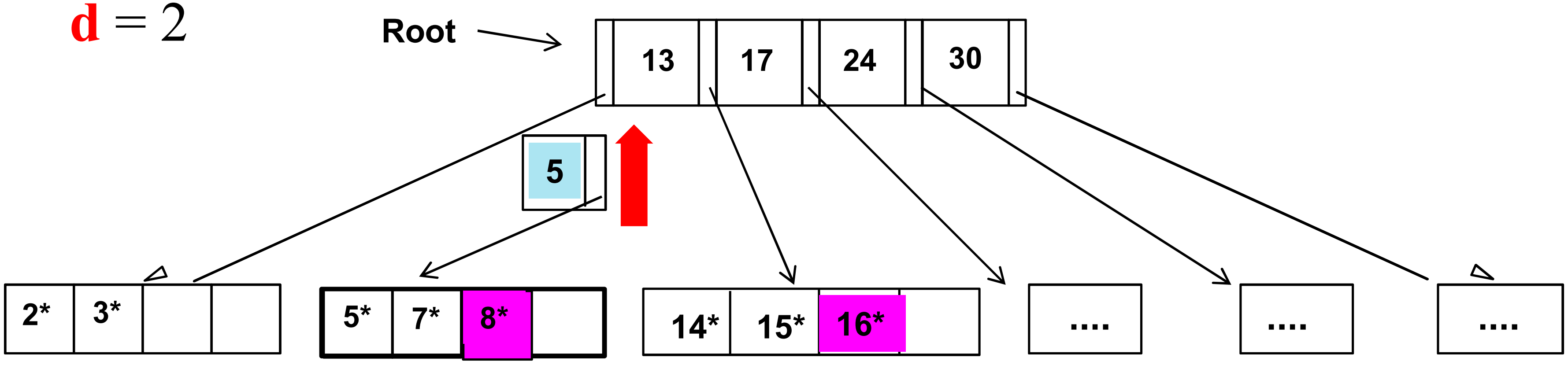
Leaf node overflows!!!



One new child (leaf node) generated; must add **one more pointer** to its parent, thus **one more key value** as well.

Inserting 16*, 8* into Example B+ tree

d = 2



Leaf node overflows!!!

When a <u>leaf node</u> overflows:

1) Split the node

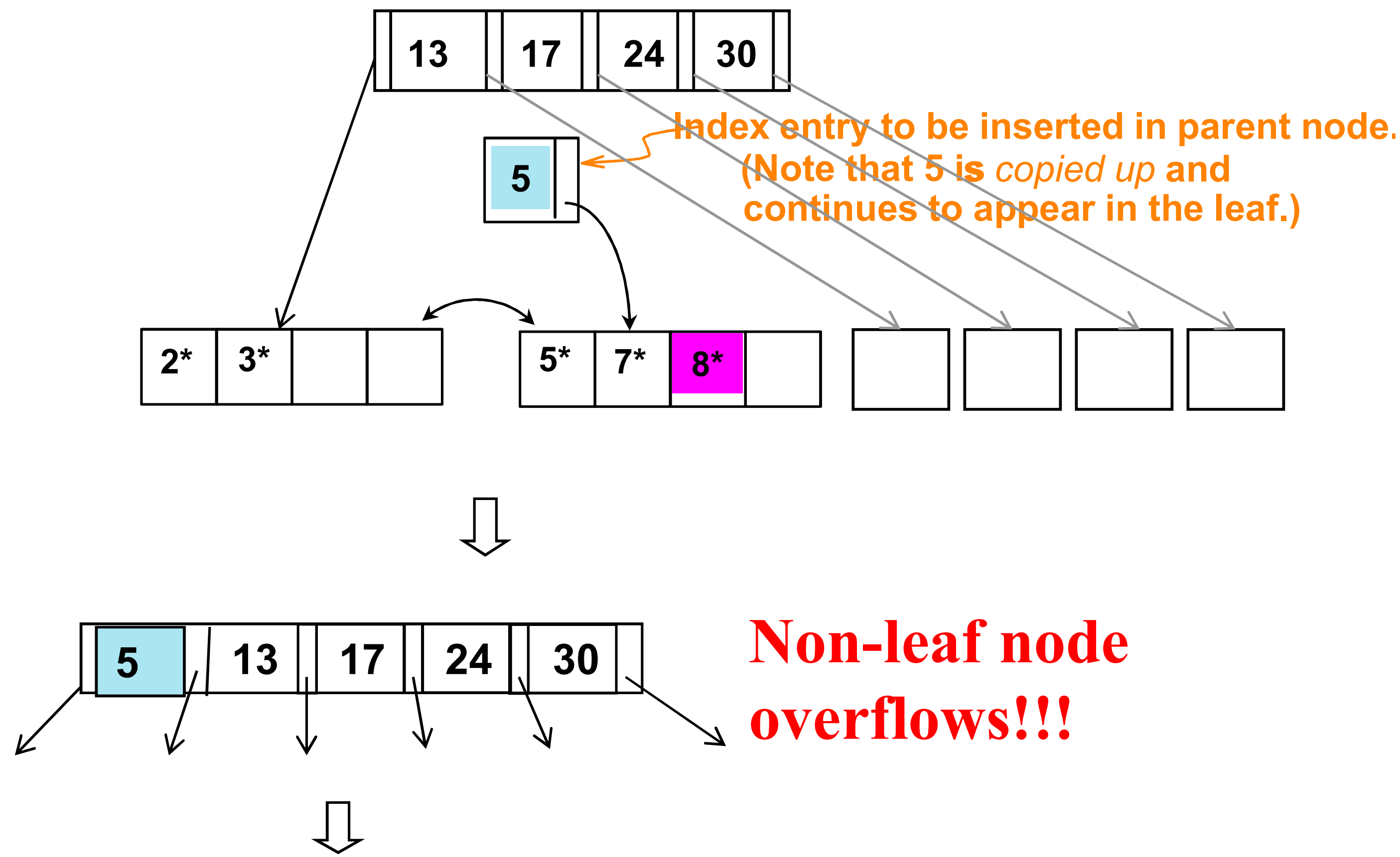
First **d** entries stay, move rest to new node

2) The new *middle key* to discriminate btwn old & and new nodes is: 5
Since data entry 5* must appear at the leaf, we COPY UP 5 & the ptr || to the new node!

Inserting 8* (cont.)

d = 2

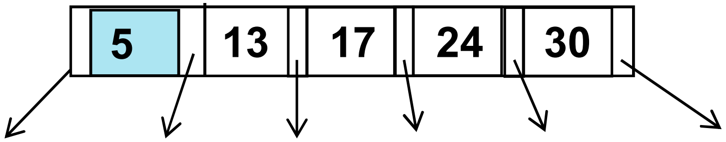
- Copy up the middle value (leaf split)



Non-leaf node overflows!!!

Inserting 8* (cont.)

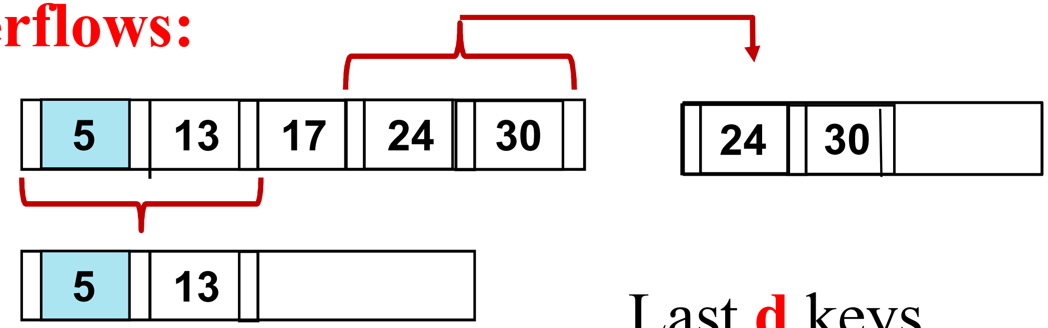
d = 2



Non-leaf node overflows!!!

When a non-leaf node overflows:

1) Split the node



First **d** keys
(and d+1 pointers)
stay in old node)

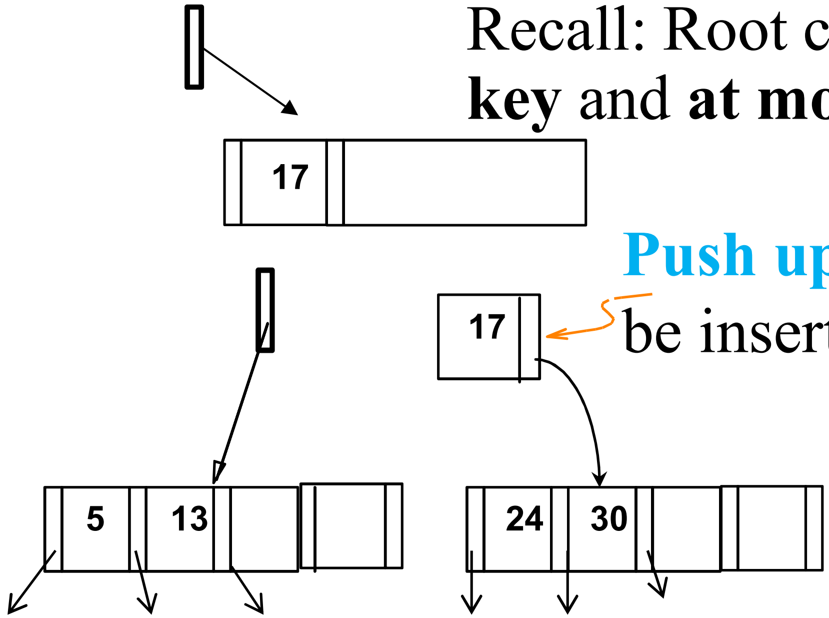
Last **d** keys
(and d+1 pointers)
move to new node

2) **PUSH UP** middle key (17)
and the ptr || to the new node)!

Insertion into B+ tree (cont.)

- Understand difference between copy-up and push-up
- Observe how minimum occupancy is guaranteed in both leaf and index node splits.

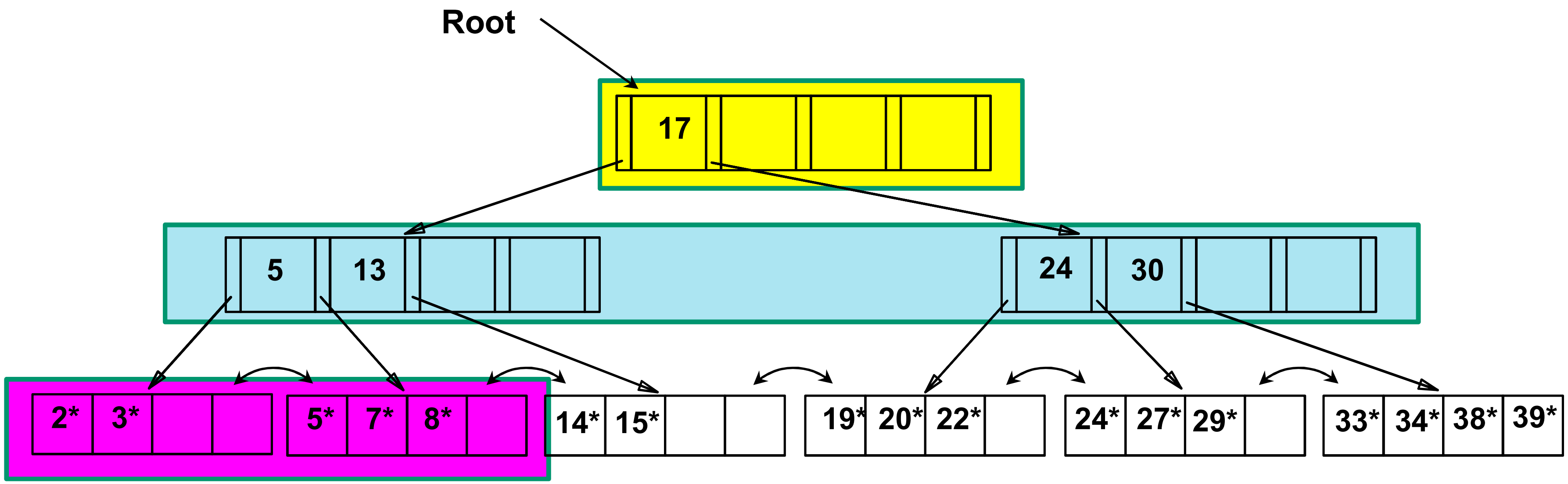
Recall: Root can have at least **1 key** and **at most 2d** keys.



Push up the index entry to be inserted in parent node

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

Example B+ Tree After Inserting 8*



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

B+ trees grow **bottom-up** dynamically!

Inserting a Data Entry into a B+ Tree: Summary

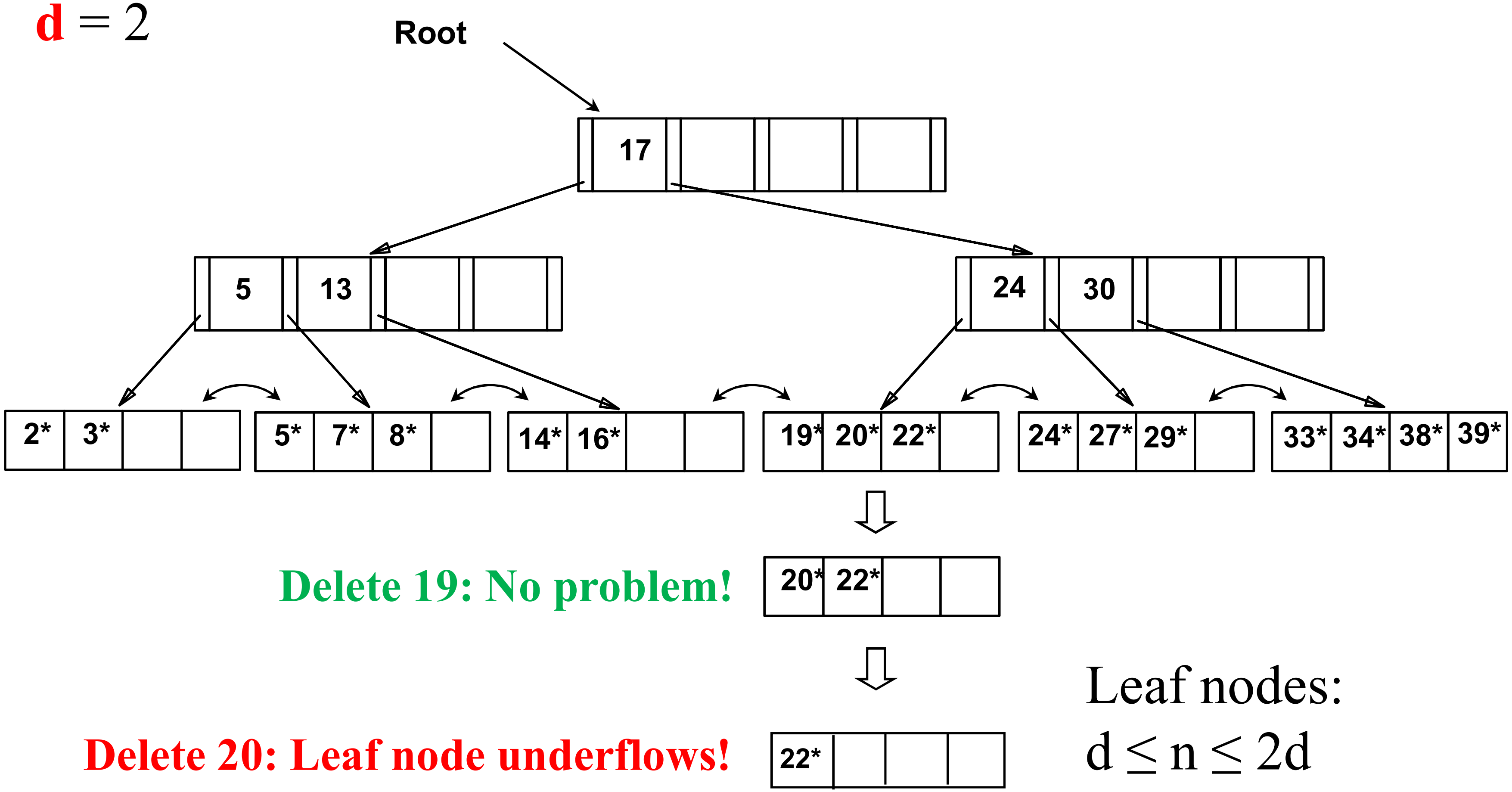
- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, put middle key in L2
    - **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can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

Deleting a Data Entry from a B+ Tree

- Examine examples first …

Delete 19* and 20*

d = 2



Delete 19: No problem!

Delete 20: Leaf node underflows!

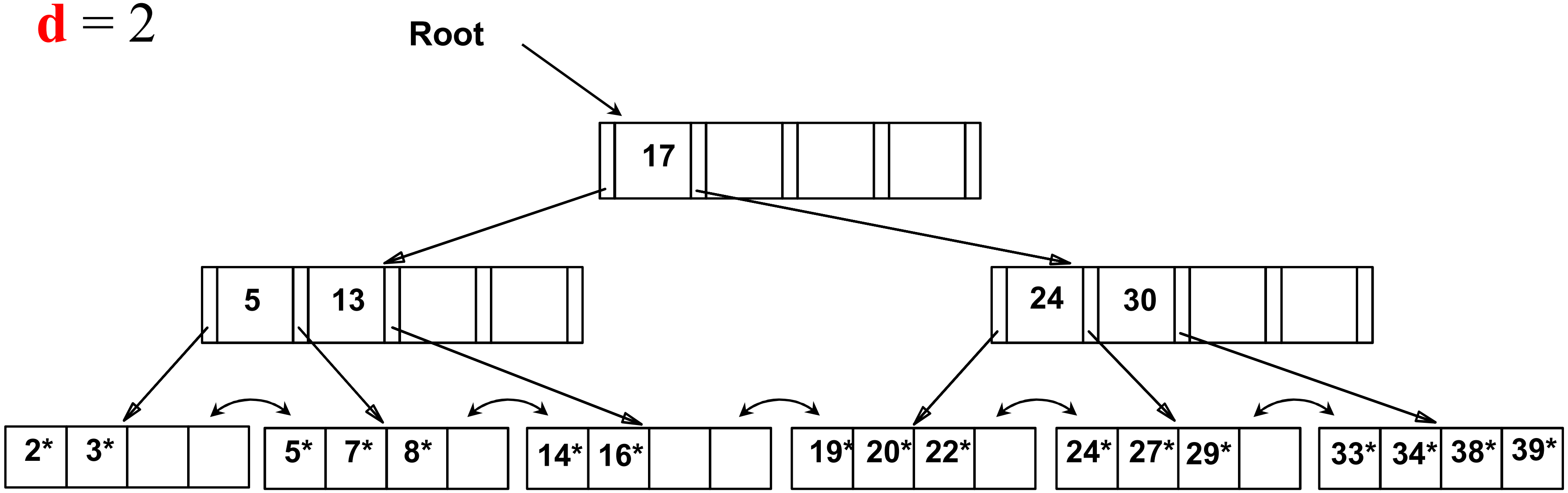
Leaf nodes:

$d \leq n \leq 2d$

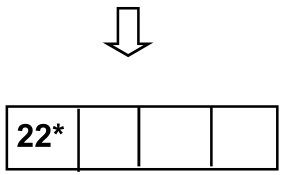
$2 \leq n \leq 4$

Delete 19* and 20*

d = 2



Leaf node underflows!



When a <u>leaf node</u> underflows:

Two options (try in order):

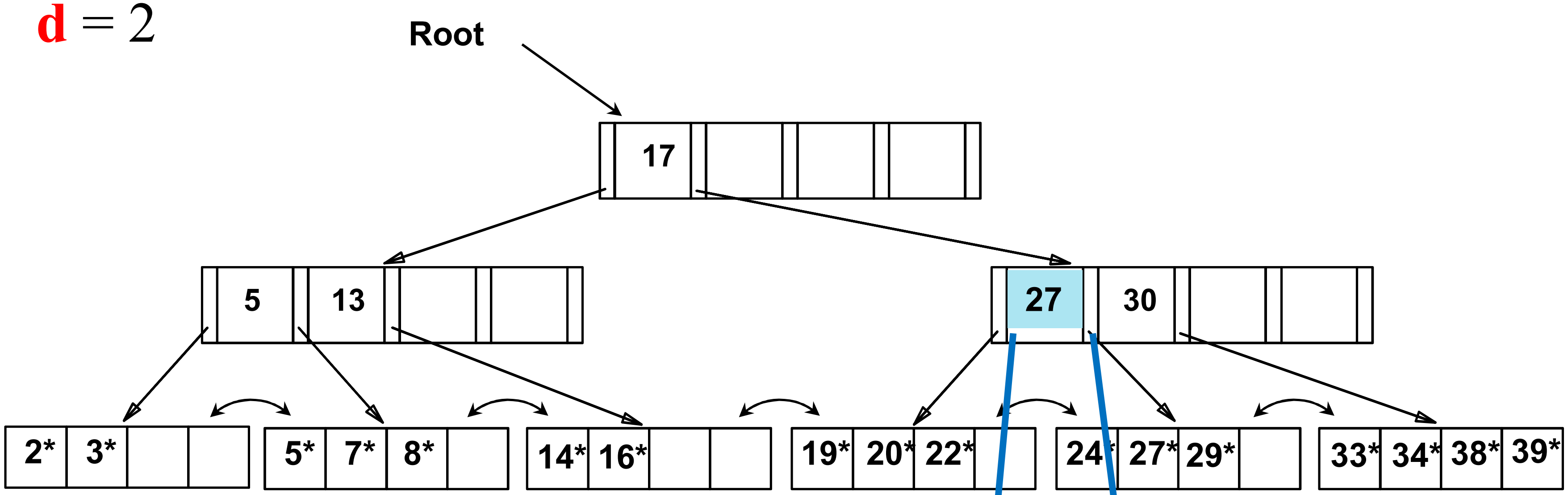
1- Redistribute evenly, and if this is not possible,

2- Merge nodes

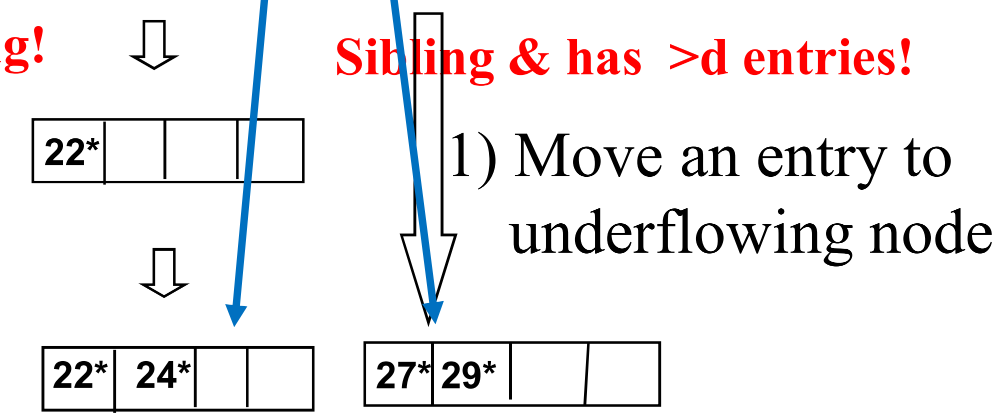
Let's redistribute!

Delete 19* and 20*

d = 2



X Not a sibling!

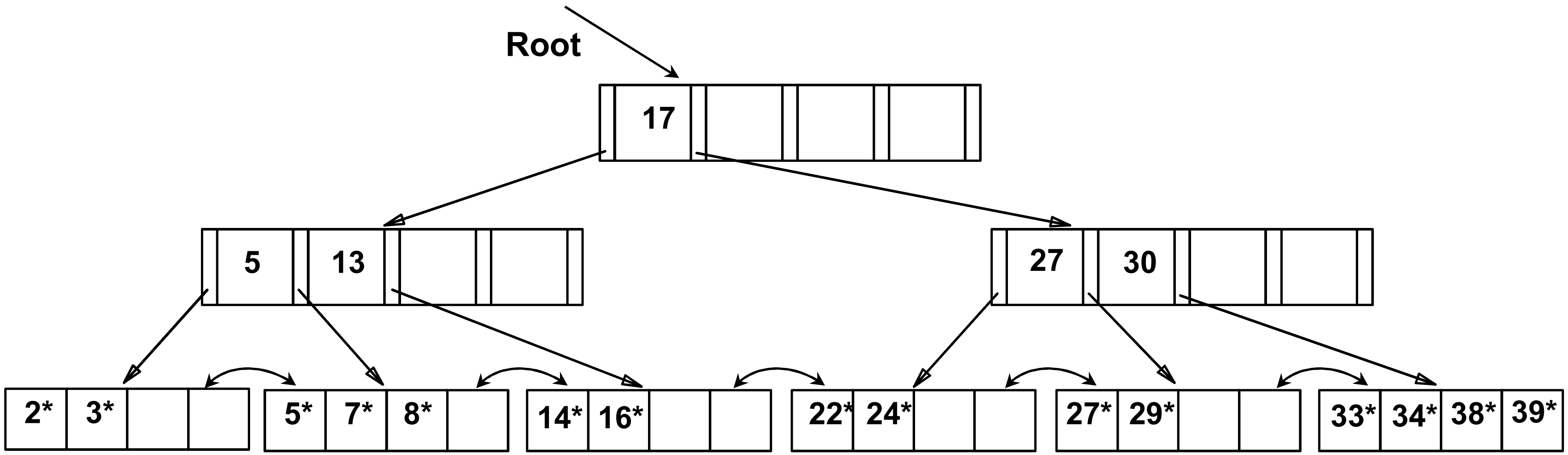


Redistribute for leaf nodes:

A **sibling** of a node O is the node that is adjacent (immediate to left or right) to O, and has the same parent as O!

2) **COPY-UP** new middle key (low key of RHS) → 27

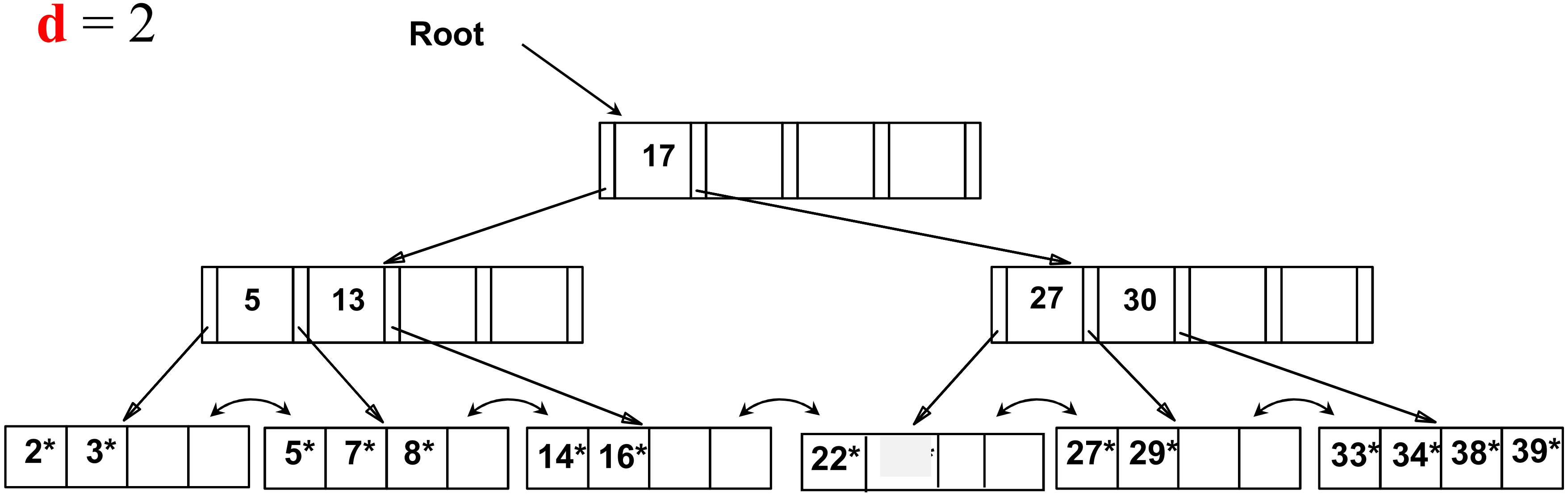
Deleting 19* and 20* (cont.)



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

Delete 24*

d = 2



When a leaf node underflows:

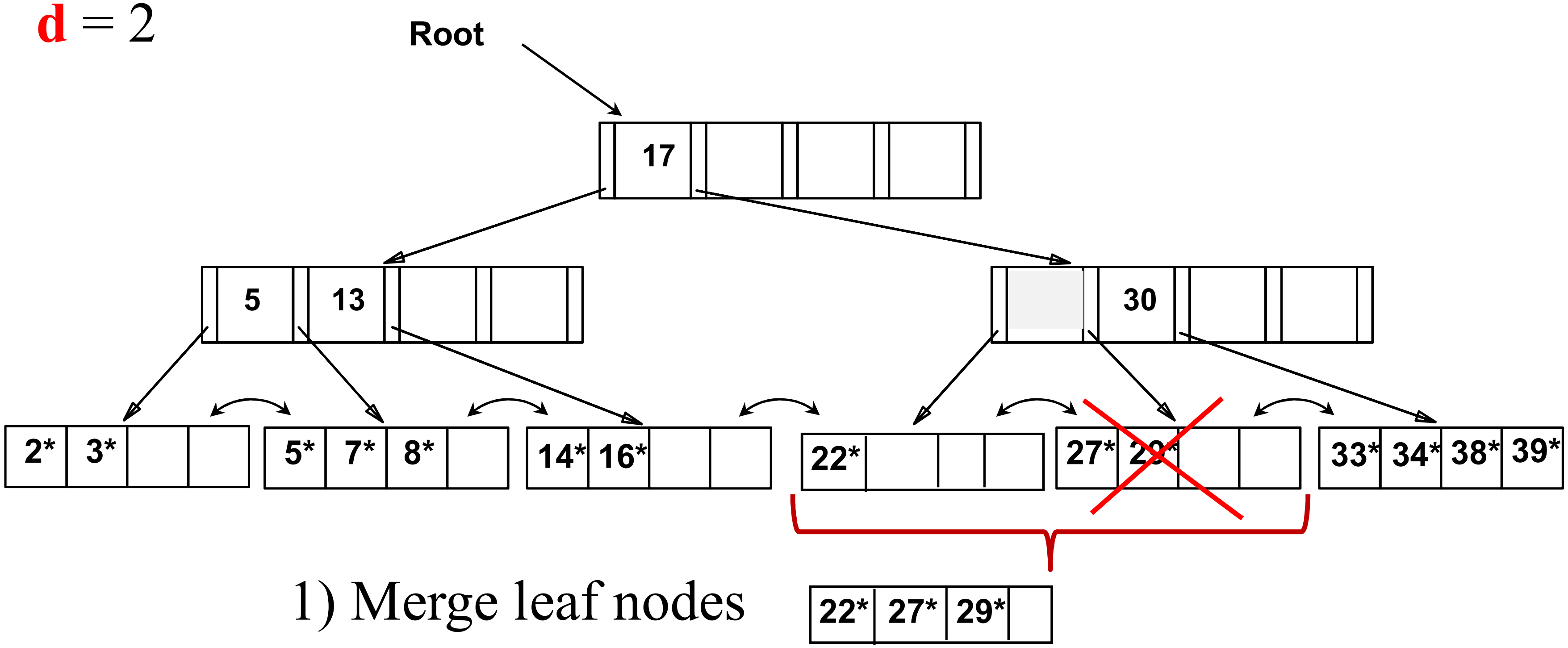
Two options (try in order): ⇩

1- Redistribute evenly, and if this is not possible, **IMPOSSİBLE!**

2- Merge nodes

Delete 24*

d = 2

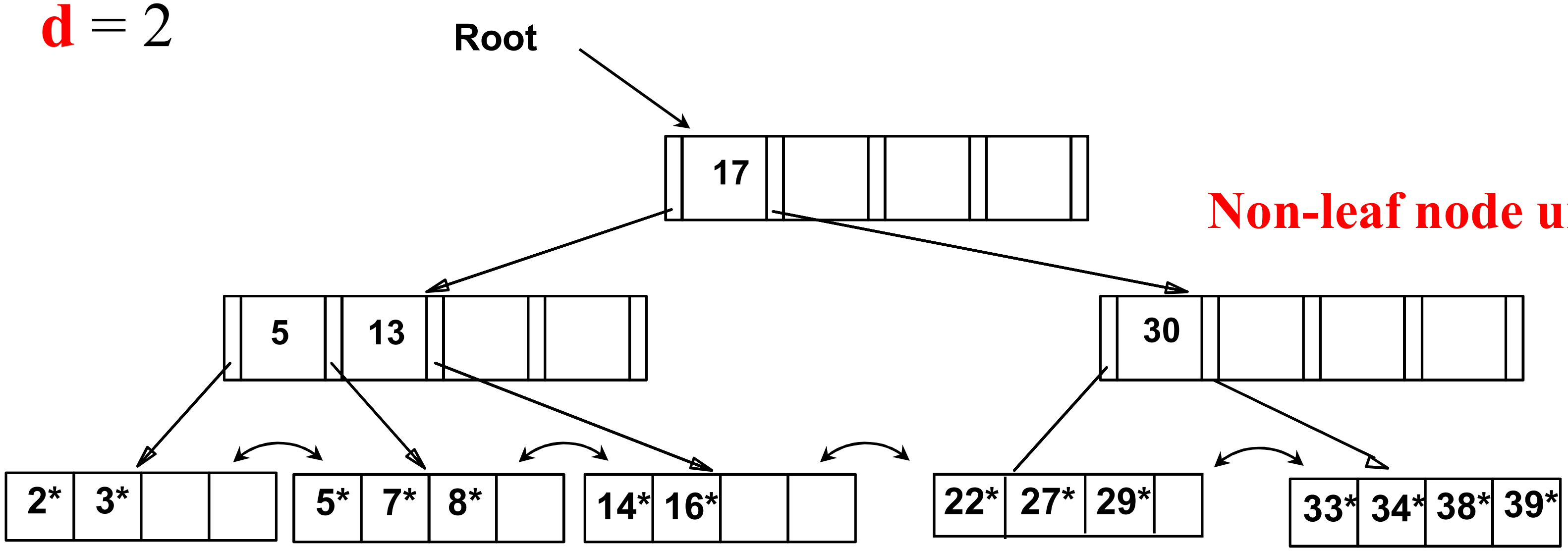


Merging leaf nodes

2) **TOSS** the index entry:
(27, || ptr to discarded node)

Delete 24*

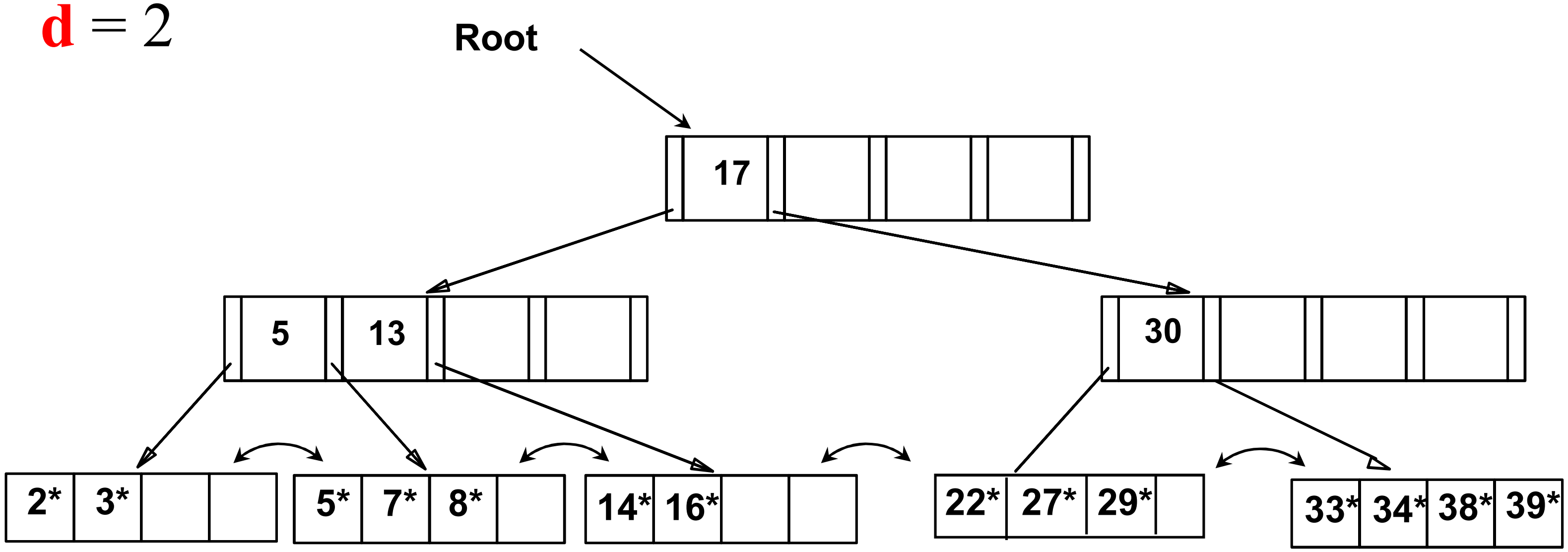
d = 2



Non-leaf node underflows!

Delete 24*

d = 2



When a <u>non-leaf node</u> underflows:

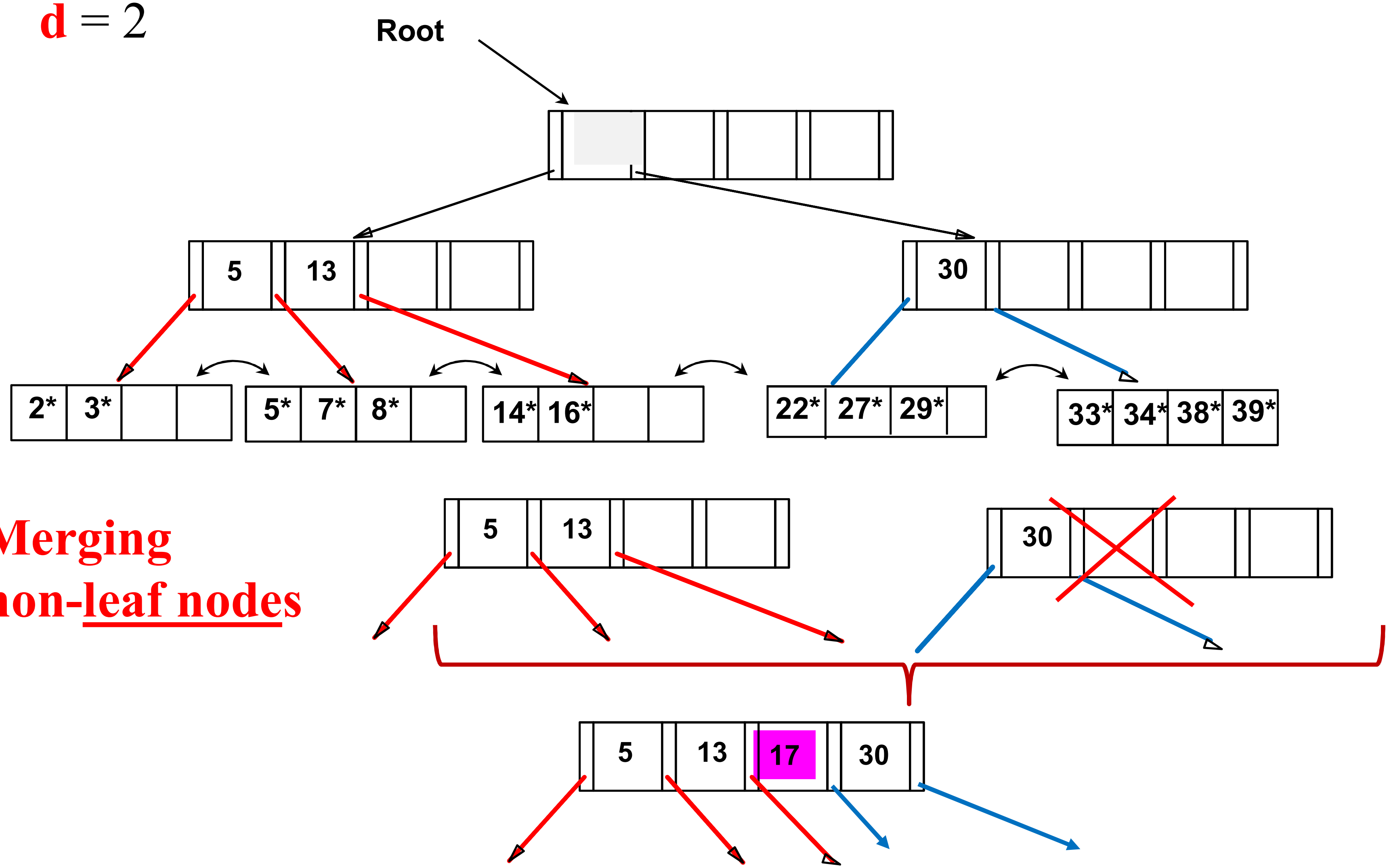
Two options (try in order):

1- Redistribute evenly, and if this is not possible, **IMPOSSİBLE!**

2- Merge nodes

Delete 24*

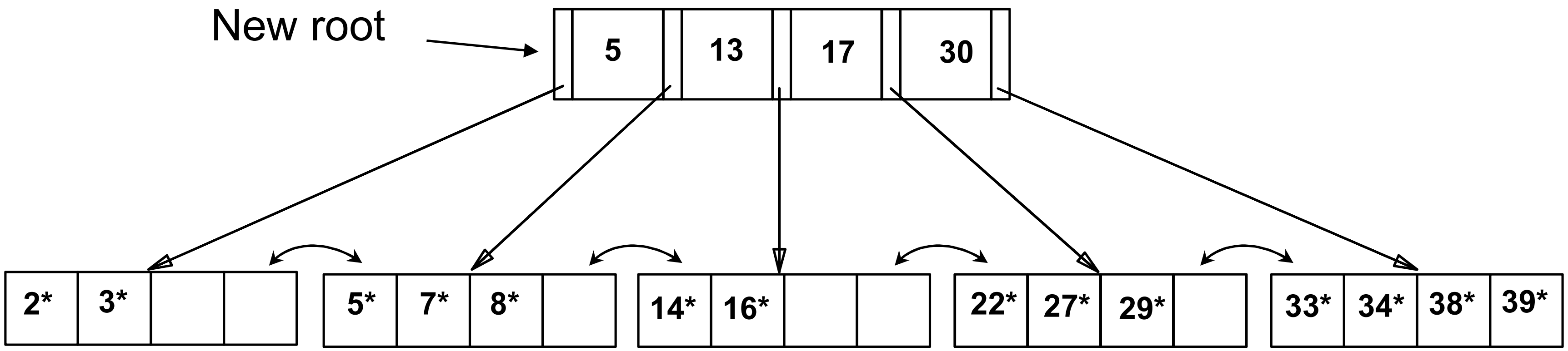
d = 2



Merging non-leaf nodes

Merge: Entries in first non-leaf node, **PULL DOWN** the splitting key (discard its pointer), followed by the entries in the second non-leaf node

Delete 24*



Deleting a Data Entry from a B+ Tree: Summary

- 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 occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Recall

When a <u>non-leaf node</u> underflows:

Two options (try in order):

1- Redistribute evenly, and if this is not possible,

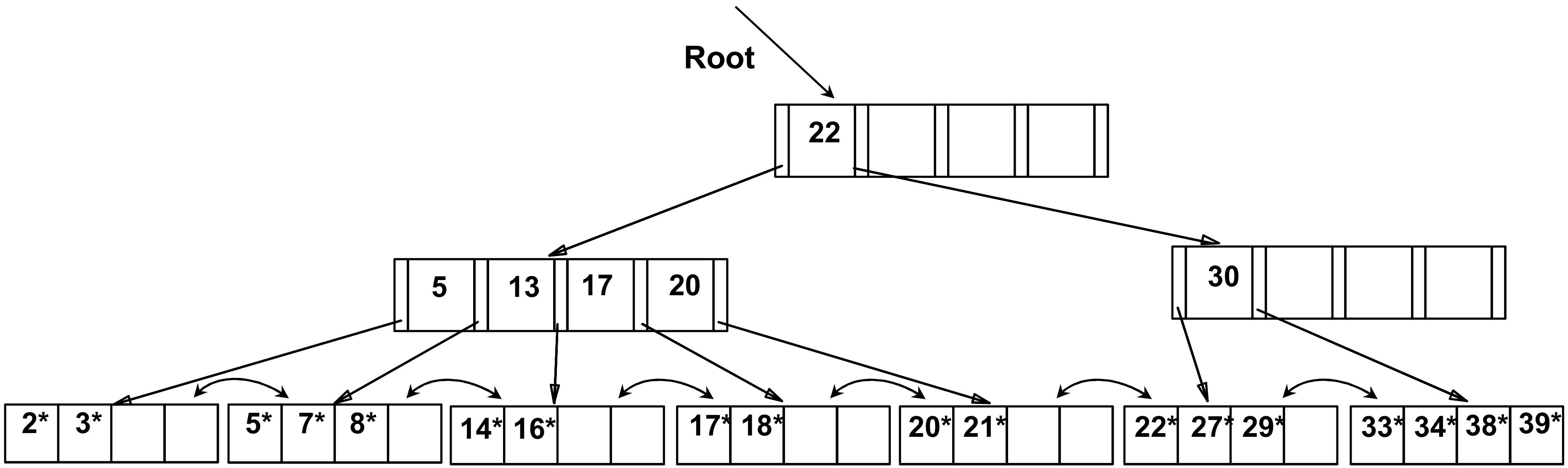
2- Merge nodes

You need an example?

Yes, you do! ☺

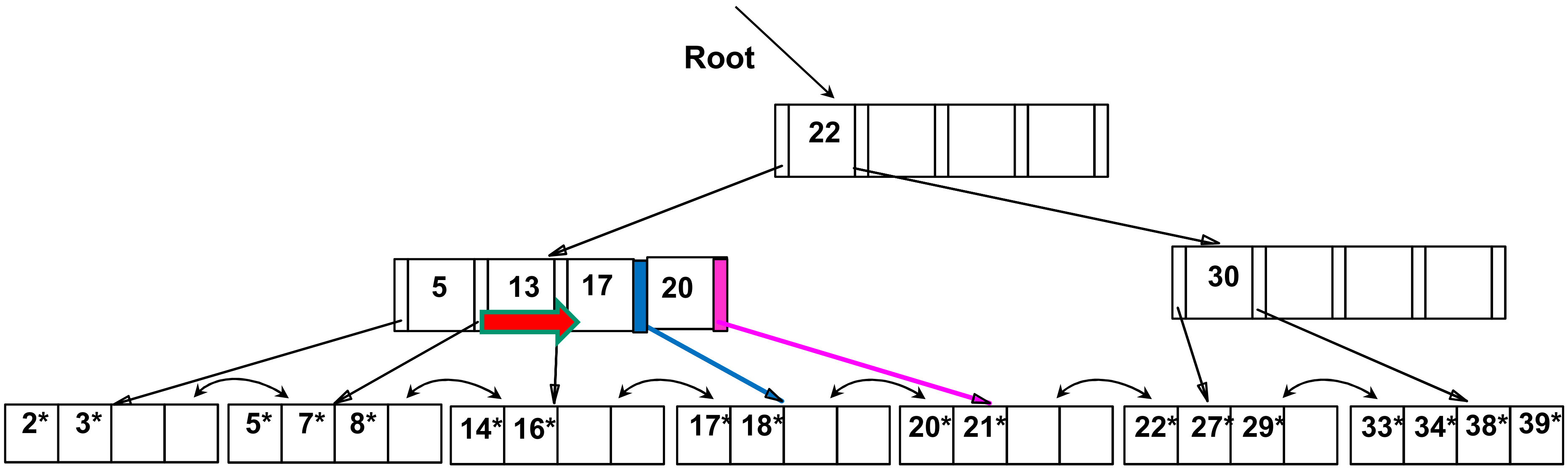
Example of Non-leaf Re-distribution

- Tree is shown below *during deletion* of 24*. (What could be a possible initial tree?)
- In contrast to previous example, can re-distribute entry from left child of root to right child.



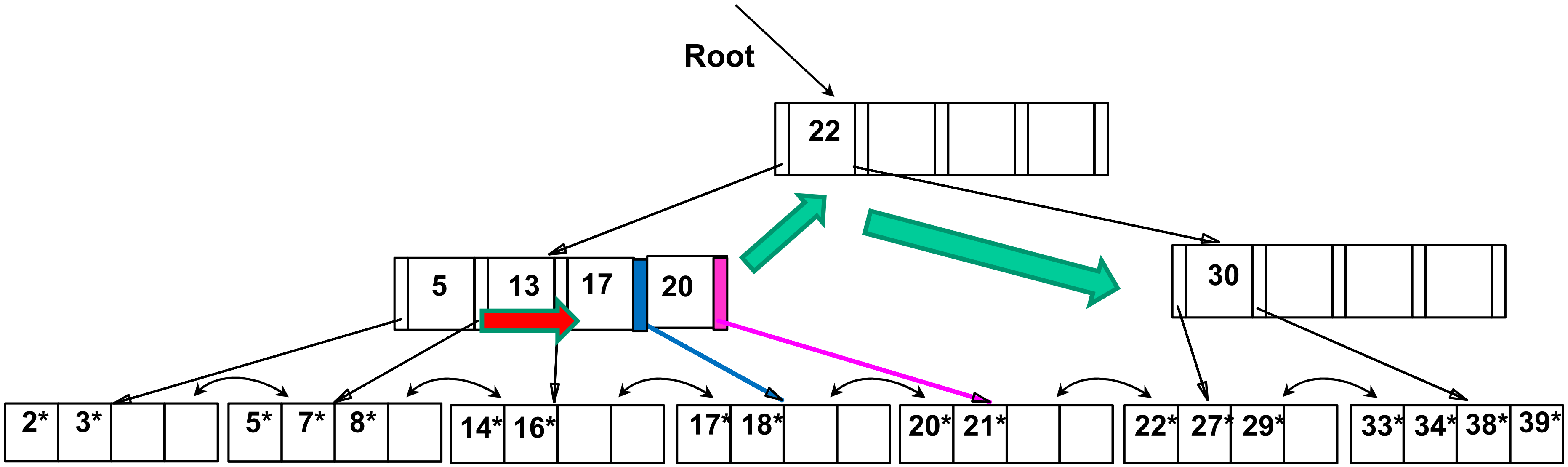
After Re-distribution

- 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.



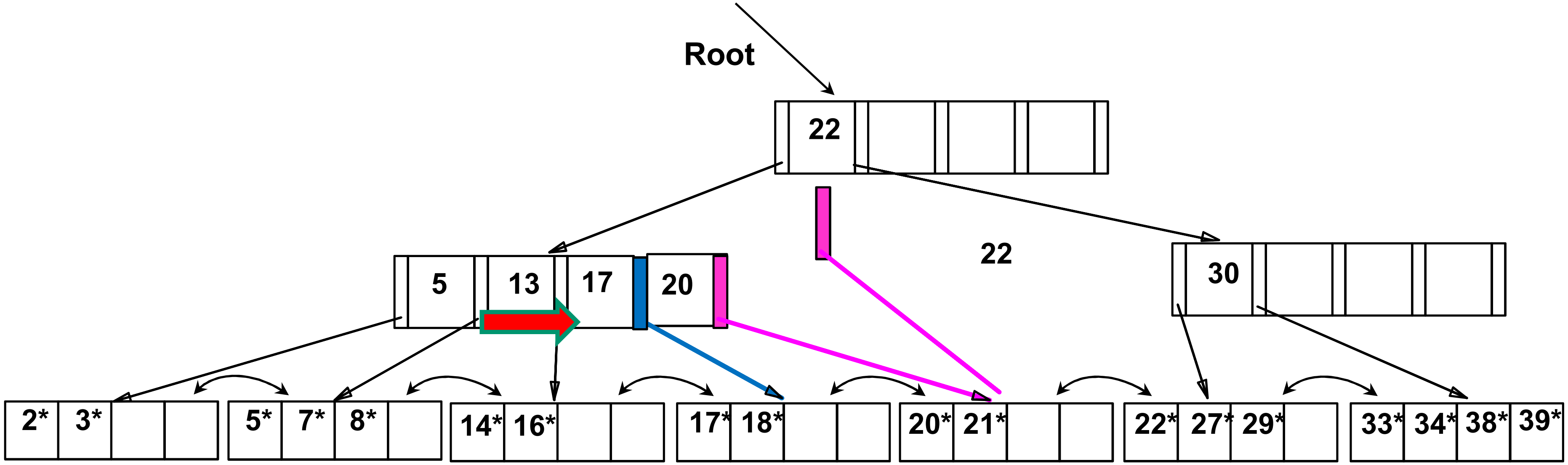
After Re-distribution

- 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.



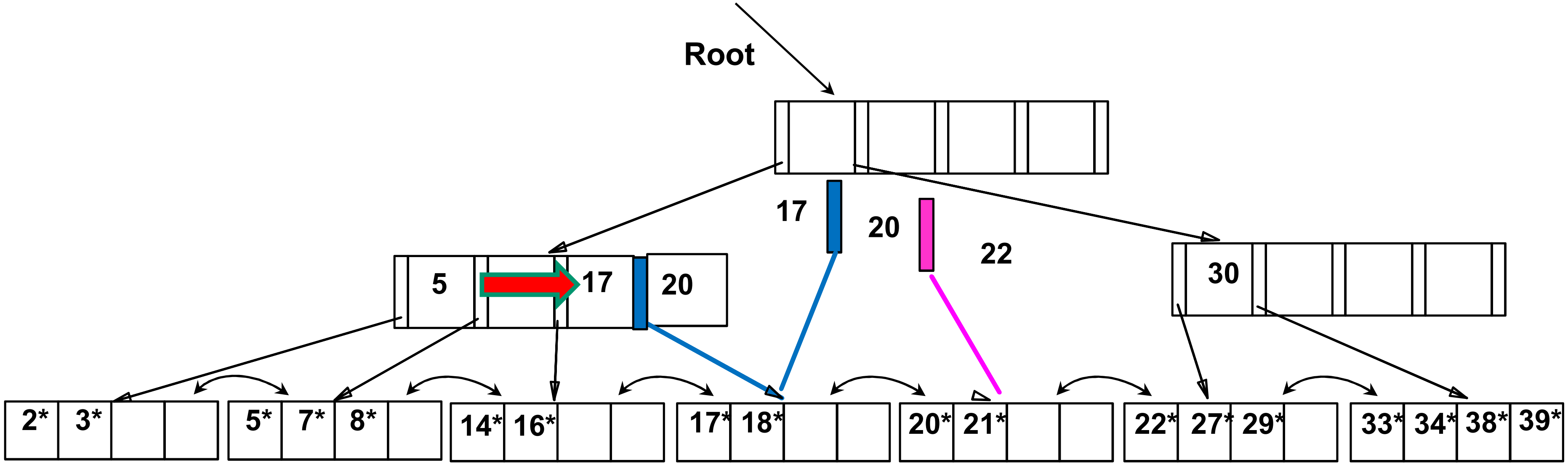
After Re-distribution

- 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.



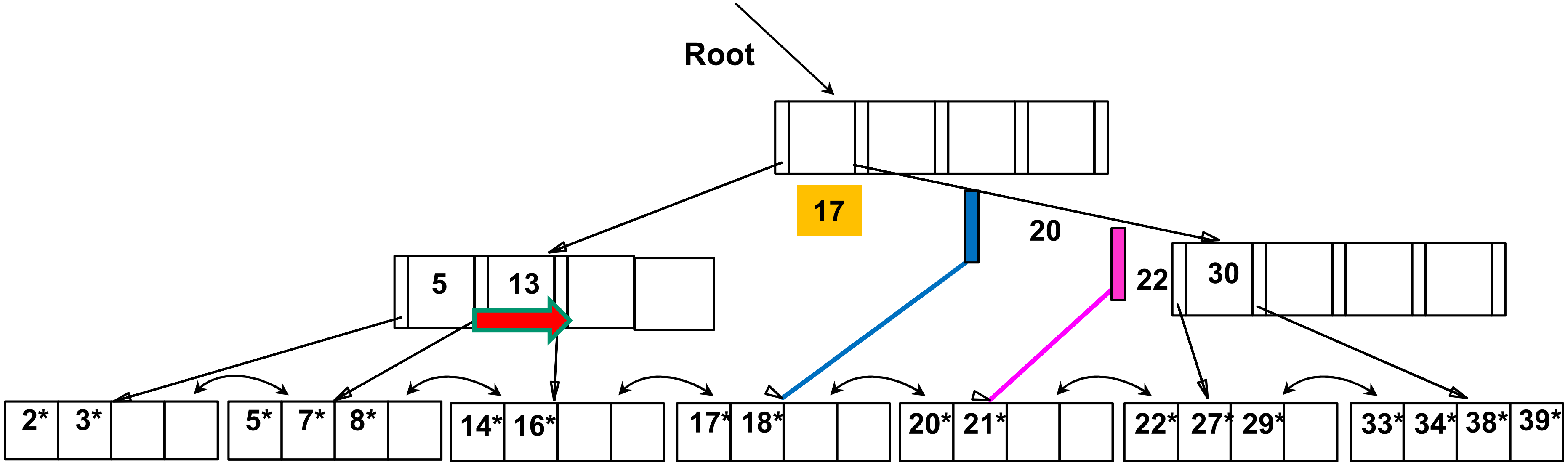
After Re-distribution

- 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.



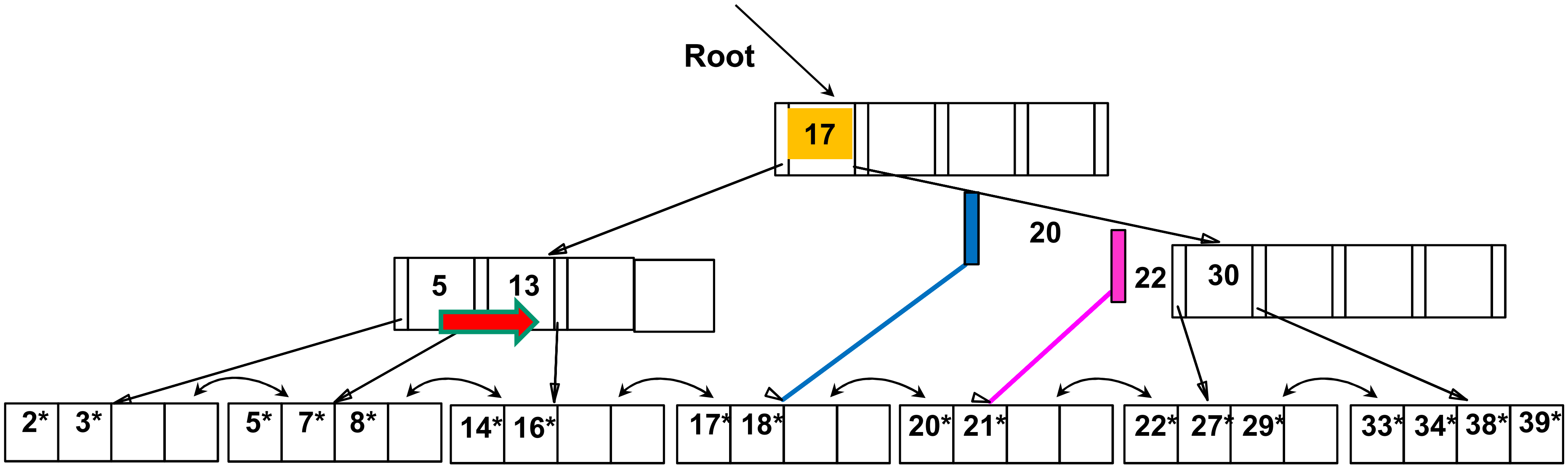
After Re-distribution

- 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.



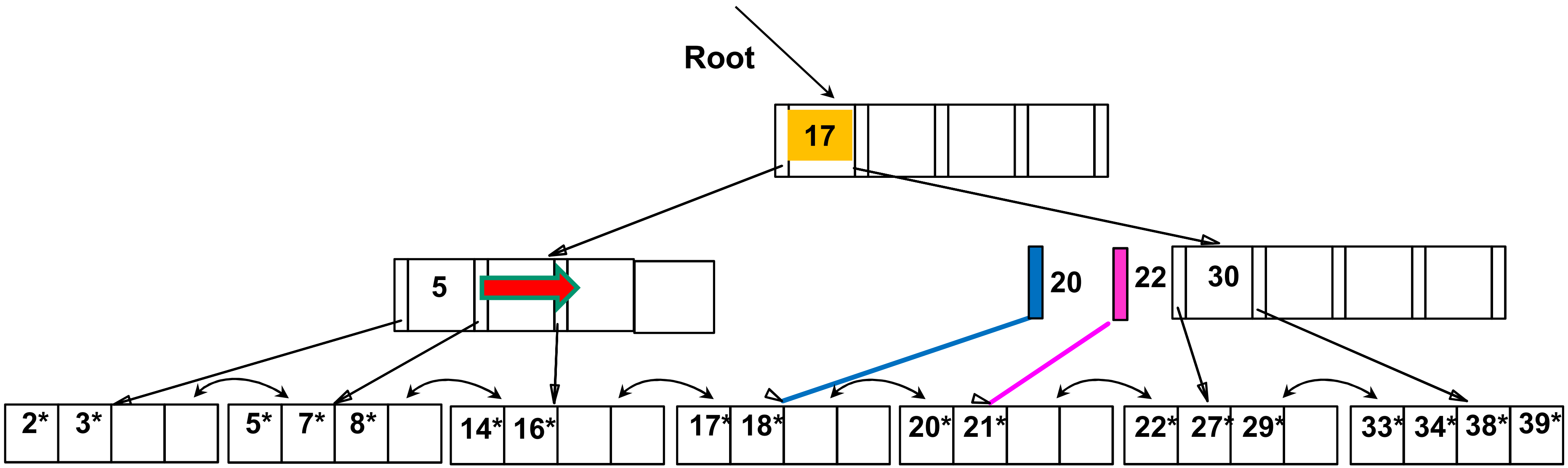
After Re-distribution

- 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.



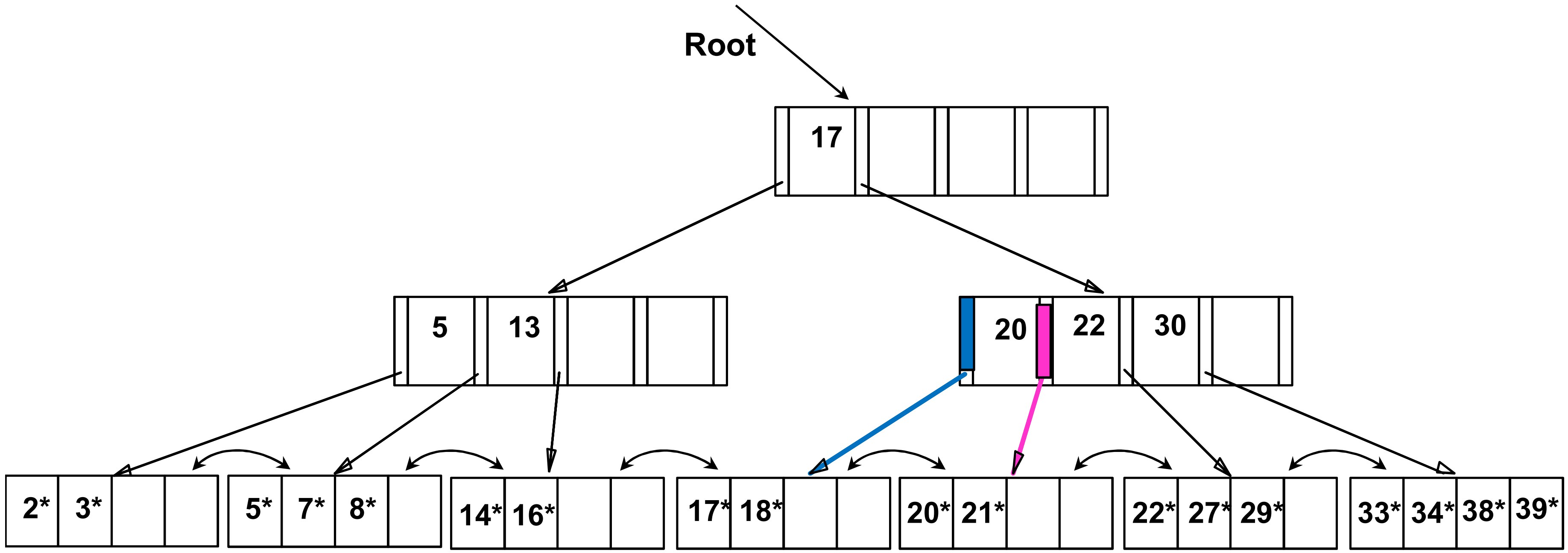
After Re-distribution

- 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.



After Re-distribution

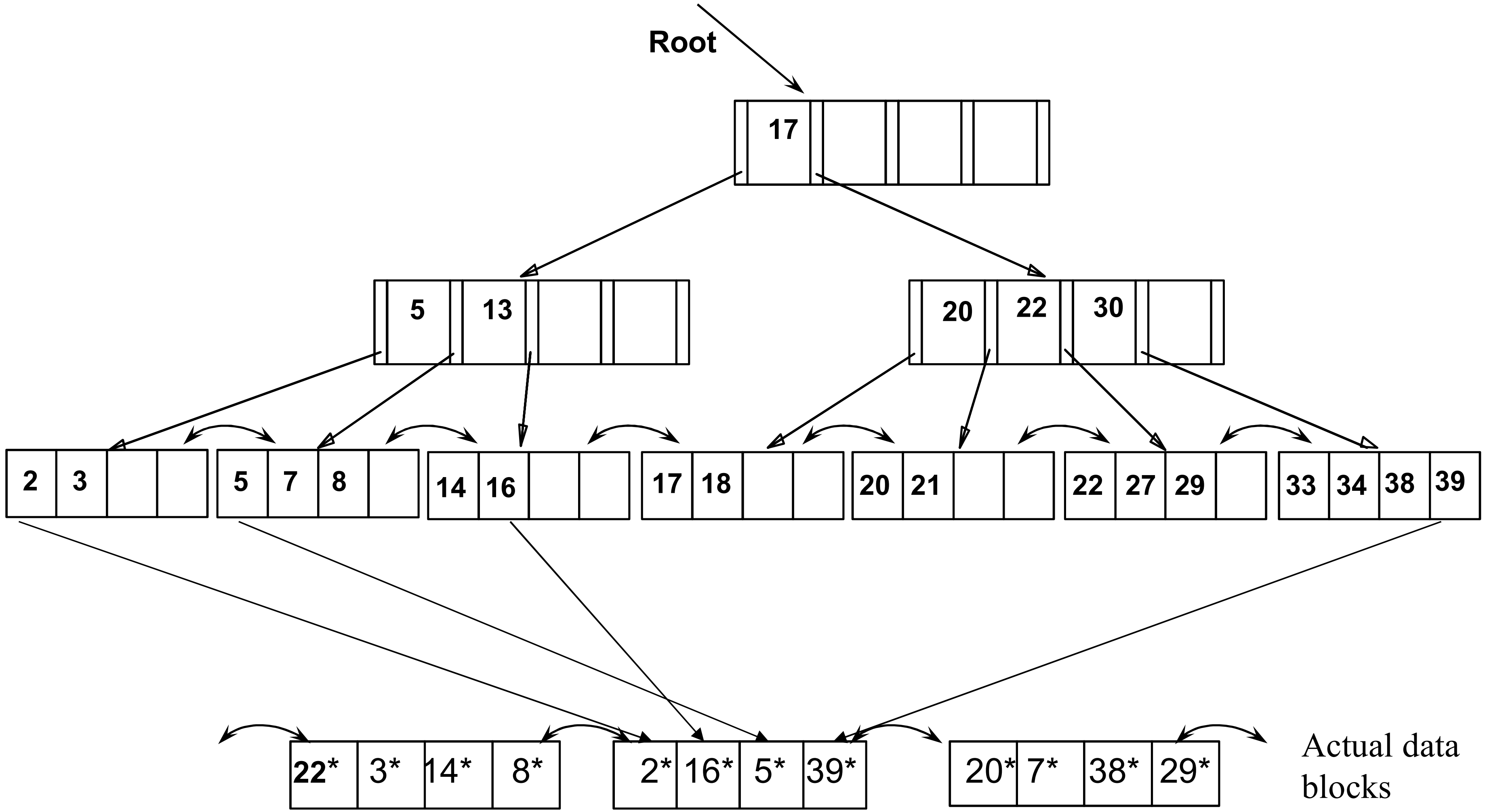
- Intuitively, entries are re-distributed by `*pushing through*' the splitting entry in the parent node.
- It suffices to re-distribute index entry with key 20; we've re-distributed 17 as well for illustration.



Primary vs Secondary Index

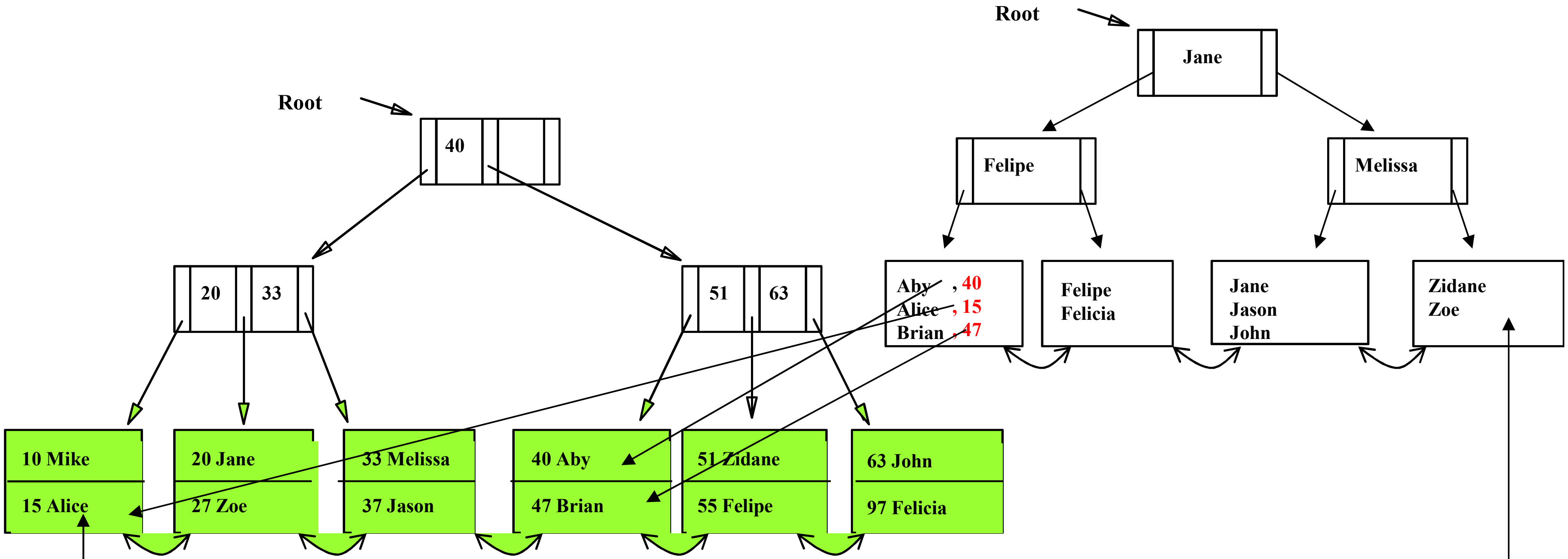
- Note: We were assuming the data items were in sorted order
  - This is called *primary/clustered B+tree* index
- *Secondary B+tree* index:
  - Built on an attribute that the file is not sorted on.
- Can have many different indexes on the same file.

A Secondary B+-Tree index



A file organized as (or, has) a **Primary B+-Tree** index on *ssn*

The same file also has a **Secondary B+-Tree** index on *name*

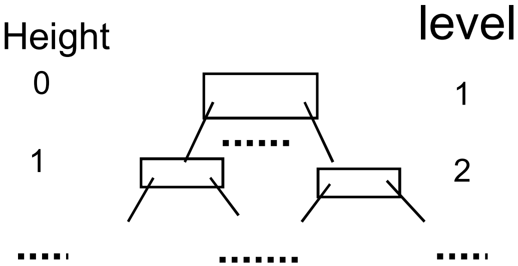


- As **15***, we store the **actual data record** with key value 15 (**Alternative-1**)
- In this case, the leaf nodes can be larger, say a few blocks (pages)

- We have k* as <key, rid> (**Alternative 2**)
- We can have rid's as pointers, or use PK as rid. We show both above

Cost for searching a value in B+ tree

- Assumptions:
  - Each interior node is a disk block
  - Each leaf node is also a disk block and data entries (K*) are of the form <key, ptr>. There are D data entries.
  - Let F be the average number of pointers in a node (for internal nodes, it is called *fanout,* **i.e.,** avg. number of children)
- Observe: Let H be the height of the B+ tree: we need to read H+1 nodes (blocks) to reach a data entry in a leaf node
- How do we find H?
  - Level 1 = 1 page = F^0 page
  - Level 2 = F pages = F^1 pages
  - Level 3 = F * F pages = F^2 pages
  - Level H+1 = …….. = F^H pages (i.e., leaf nodes)
  - F pointers → F-1 keys, so there must be D/(F-1) leaf nodes
  - D/(F-1) = F^H. That is, $H = \log_F(\frac{D}{F-1})$



B+ Trees in Practice

- Typical order: 100. Typical fill-factor: 66%.
  - average fanout = 133 (i.e, # of pointers in internal node)
- 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
- Suppose there are 1,000,000,000 data entries.
  - $H = \log_{133}(1000000000/132) < 4$
  - The cost is reading H+1 = 5 pages

Cost Computation: Another Example

Leaves would store the **actual records**

- A primary B+ tree index on key field giftID.
- 2.500.000 gift records, each record: 400 bytes.
- giftID: 12 bytes, address pointer: 4 bytes
- A bucket can hold 500 records
  - So we have larger leaf nodes (called **buckets**), as we store actual records
  - No claim for interior nodes, assume each is a block!
- B+ tree will have a fill factor of 50% [min occupancy]
- B (block size): 1600
- s: 10 ms, r: 5 ms, btt: 1 ms.

a) No of index nodes and their total size

We need to find i) fanout of the nodes, and ii) no of leaves.

i) fanout: Assume , **n keys (n+1) ptrs** can fit to an index node:

n x 12 + (n+1) x 4 = 1600 bytes → 16n = 1596 / 16 → n= 99

So at most 99 keys in a node (2d= 99, d (tree order) is floor(99/2))

Tree fill factor 50%; max 99 keys x 50% = 49 keys

fanout: 49 + 1 = 50 ptrs per node

ii) no of leaves:

500 rec/leaf * fill factor (50%) = 250 recs/leaf

2.5M records / 250 = 10000 leaf nodes (i.e., buckets)

a) No of index nodes and their total size

- Tree height = $\log_{50} 10000 = 3$
- So, there are H+1 = 4 levels

Level 4: 10000 leaf nodes (data buckets)

Level 3: ceil (10000 / 50 ptrs) = 200 nodes

Level 2: ceil (200/50) = 4 nodes

Level 1: ceil (4/50) = 1 node (root)

Index nodes: 1 + 4 + 200 = 205

Total Size: 205 x 1600 bytes

b) Time cost of reading an arbitrary record

- Three has H=3, so 4 levels
- At the first 3 levels, we fetch index nodes:

3 x (s + r+ btt) = 3x (10 +5+1) = 48 ms

- At the fourth level we fetch the leaf node (data bucket)
  - But how many blocks is a data bucket?
  - (500 recs x 400 bytes/rec) / 1600 = 125 blocks
  - So, cost s + r+ 125 x btt = 10+ 5+ 125 x 1 = 140 ms
- Total cost: 48 + 140 = 188 ms

c) Cost of reading all records in sorted manner

- Reach to leftmost leaf node, as before:
- at the first 3 levels, we fetch index nodes:

3 x (s + r+ btt) = 3x (10 +5+1) = 48 ms

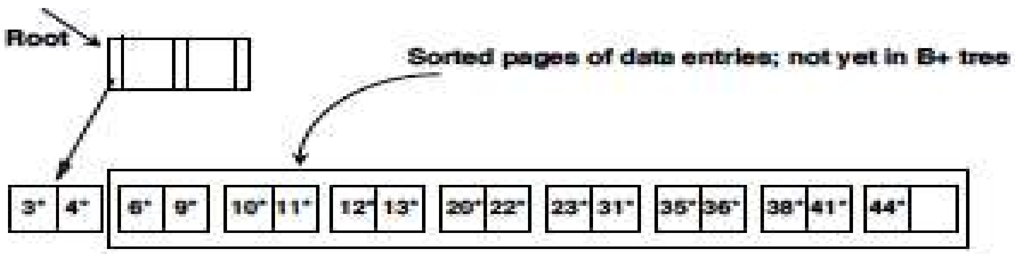
- Read all the leaf nodes (using doubly linked list pointers)
  - 10000 (s + r + 125 x btt)
- Think: What if this is a secondary B+ tree and we store <key, ptr> pairs at leaf nodes (data buckets)?

Terminology

- **Blocking Factor:** the number of records which can fit in a leaf node.
- **Fan-out :** the average number of children of an internal node.
- A B+tree index can be used either as a primary index or a secondary index.
  – **Primary index:** determines the way the records are actually stored (also called a sparse index, clustered index)
  – **Secondary index:** the records in the file are not grouped in blocks according to keys of secondary indexes (also called a dense index)

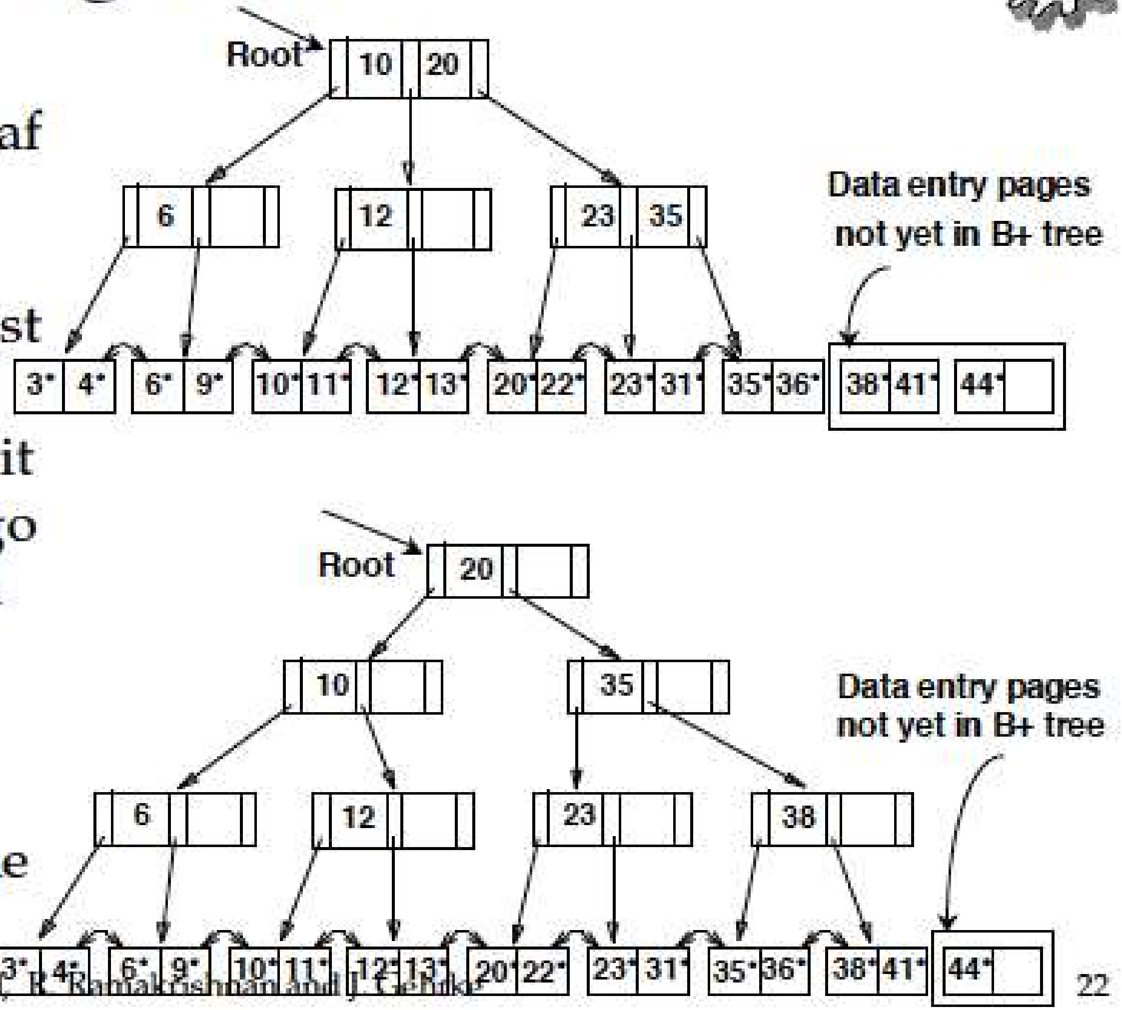
Bulk Loading of a B+ Tree

- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- Bulk Loading can be done much more efficiently.
  – Initialization: Sort all data entries, insert pointer to first (leaf) page in a new (root) page



Bulk Loading (Contd.)

- Index entries for leaf pages always entered into right-most index page just above leaf level. When this fills up, it splits. (Split may go up right-most path to the root.)
- Much faster than repeated inserts, especially when one considers locking!



Summary

- Tree-structured indexes are ideal for range-searches, also good for equality searches.
- B+ tree is a dynamic structure.
  - Inserts/deletes leave tree height-balanced; High fanout (**F**) means depth rarely more than 3 or 4.
  - Almost always better than maintaining a sorted file.
  - Typically, 67% occupancy on average.
  - If data entries are data records, splits can change rids!
- Most widely used index in database management systems because of its versatility. One of the most optimized components of a DBMS.

More…

- Hash-based Indexes
  - Static Hashing
  - Extendible Hashing
  - Linear Hashing
- Grid-files
- R-Trees
- etc…
- A nice animation site for B+ trees: https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html