Ceng 302
Database Management Systems

Database B+ Tree Index Structures

Prof. Dr. Adnan YAZICI

Department of Computer Engineering,

Middle East Technical University

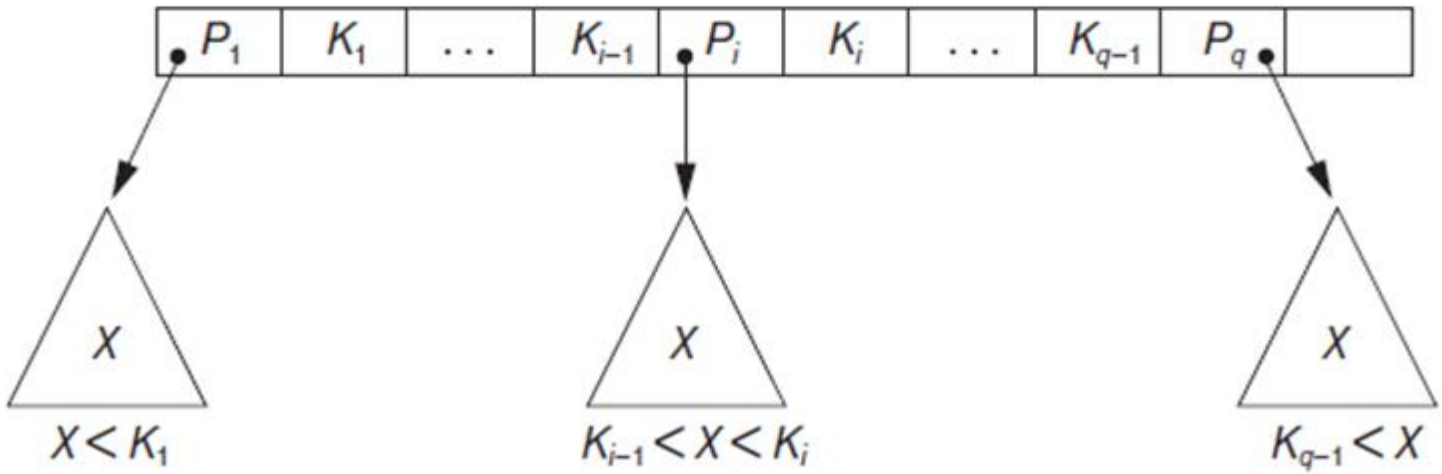
(**Fall 2021**)

Search Trees and 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** has 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.

Search Trees and B Trees

- Search tree used to guide search for a record
  - Given value of one of record's fields



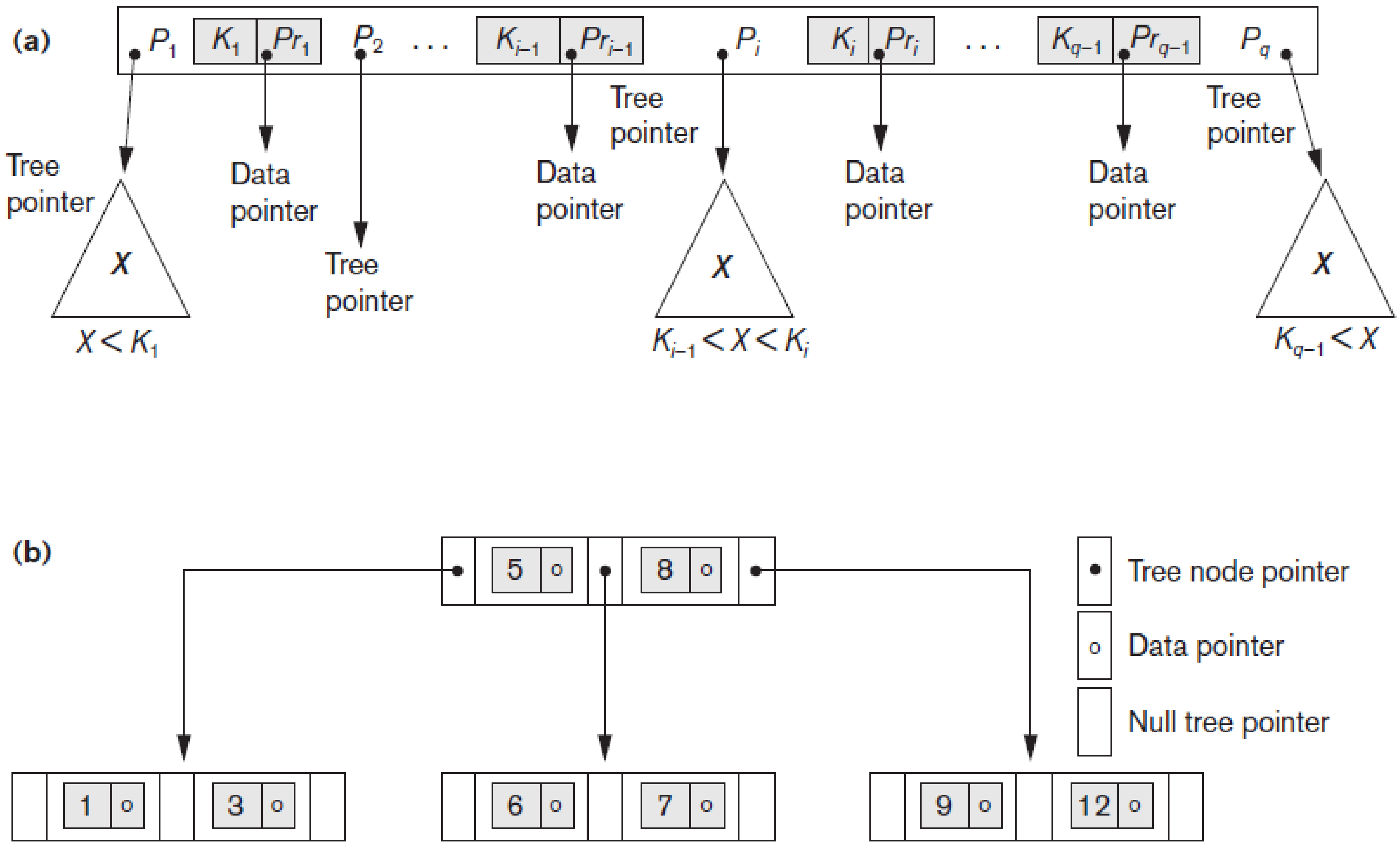
This restriction can be relaxed. If the index is on a nonkey field, duplicate search values may exist and the node structure and the navigation rules for the tree may be modified.

A node in a search tree with pointers to subtrees below it

B Trees

- Provide multi-level access structure
- Tree is always balanced
- Space wasted by deletion never becomes excessive
  - Each node is at least half-full
- Each node in a B-tree of **order *p*** can have at most ***p-1*** **search values** and ***p*** **pointers**.

B Tree Structures



B-tree structures (a) A node in a B-tree with *q−1* search values (b) A B-tree of **order *p=3***. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6

Formal definition of B+ Tree Properties

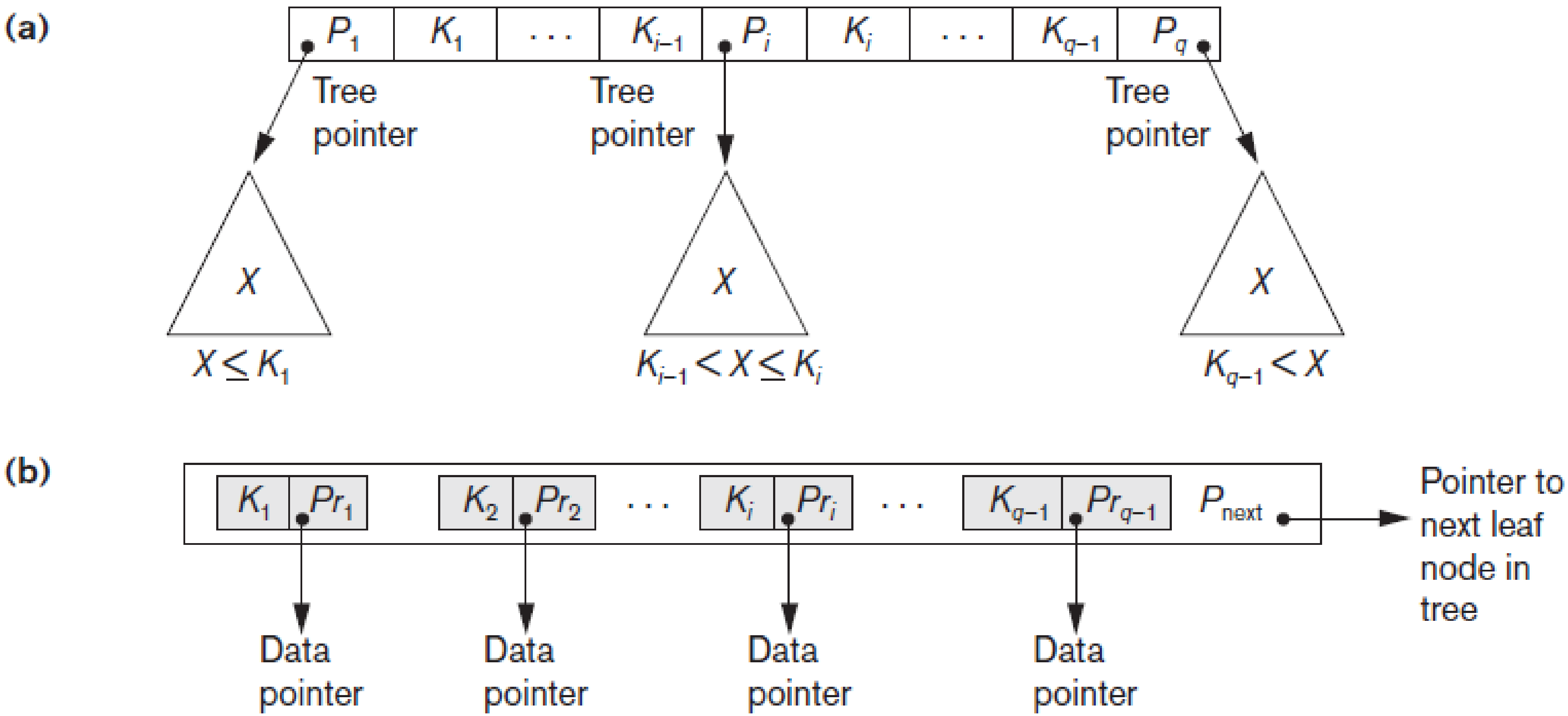
Properties of a B+ Tree of order n+1:

- All internal nodes (except root) has at least n/2 keys and at most n keys. That is, if the **order** of a B+-tree is **n+1**, each node (except for the root) must have between **n/2 and n keys.**
- The **root** has at least **2** children unless it's a **leaf**.
- All **leaves** are on **the same** level.
- An **internal node** with **n keys** has **n+1 pointers** or **children.**

Terminology

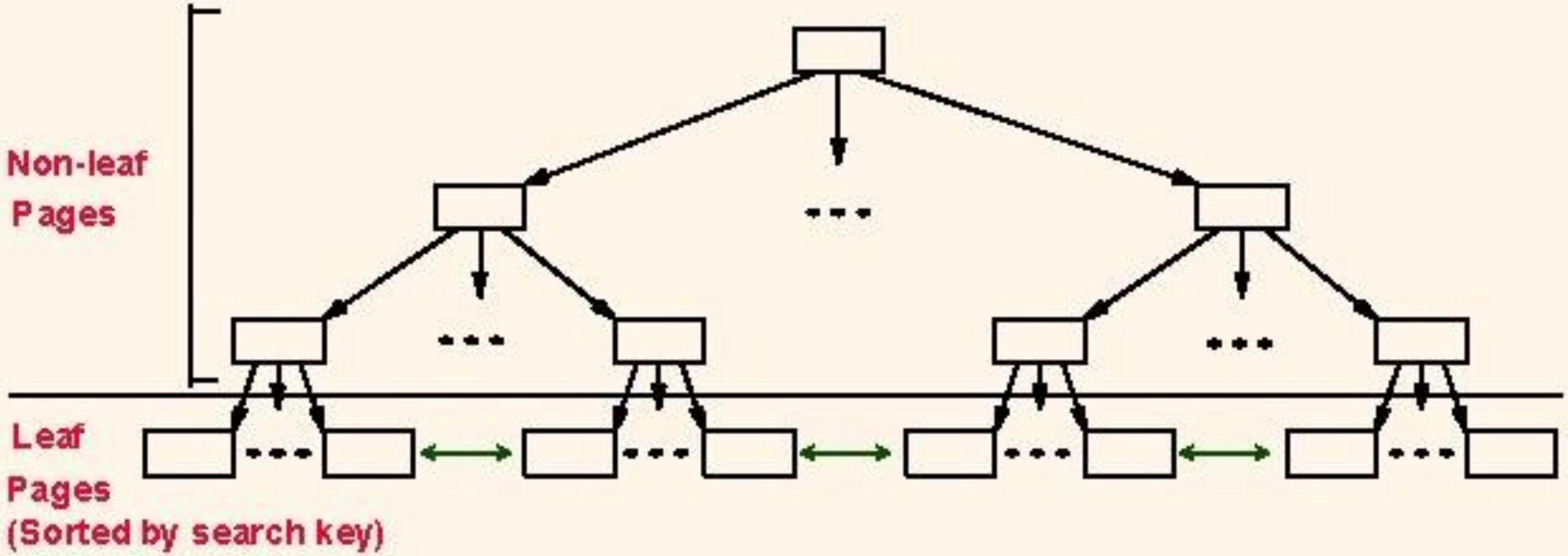
- **Bucket Factor:** the number of records which can fit in a leaf node.
- **Fan-out:** the average number of children of an internal node.
- A **B+-tree index** can be used either as a **primary** index or a **secondary** index.
  - **Primary index:** determines the way the records are actually stored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

B+ Trees (cont'd.)

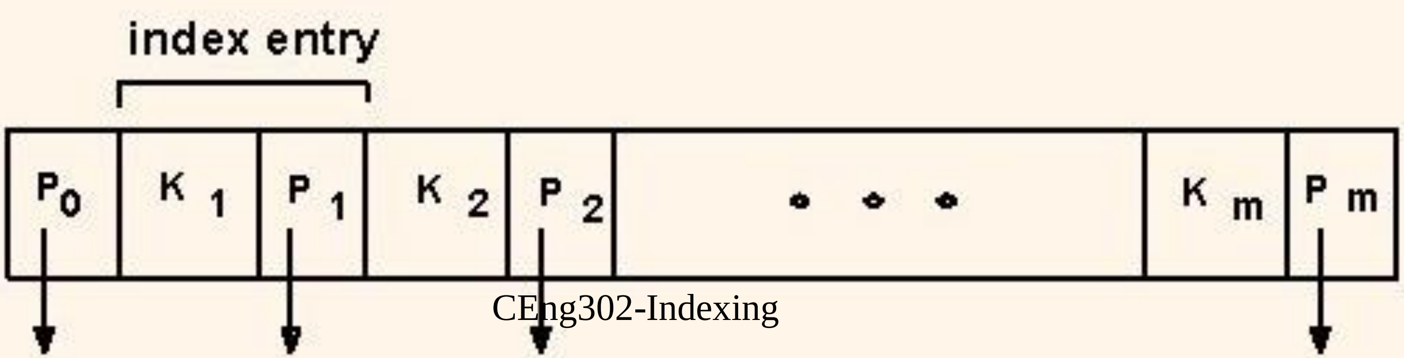


The nodes of a B+-tree (a) Internal node of a B+-tree with *q−1* search values (b) Leaf node of a B+-tree with *q−1* search values and *q−1* data pointers

B+ Tree Indexes

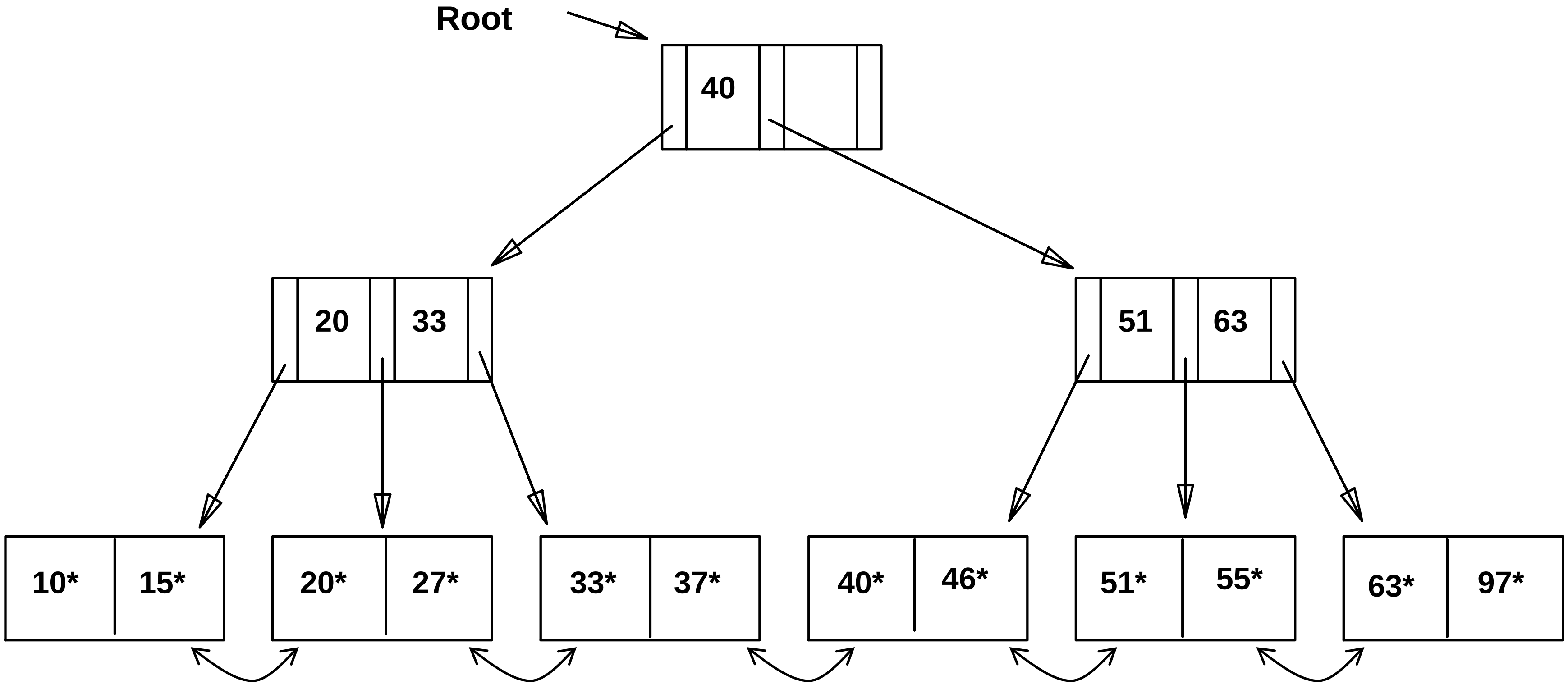


- Index leaf pages contain *data entries*, and are chained (prev & next)
- Index non-leaf pages have *index entries;* only used to direct searches:



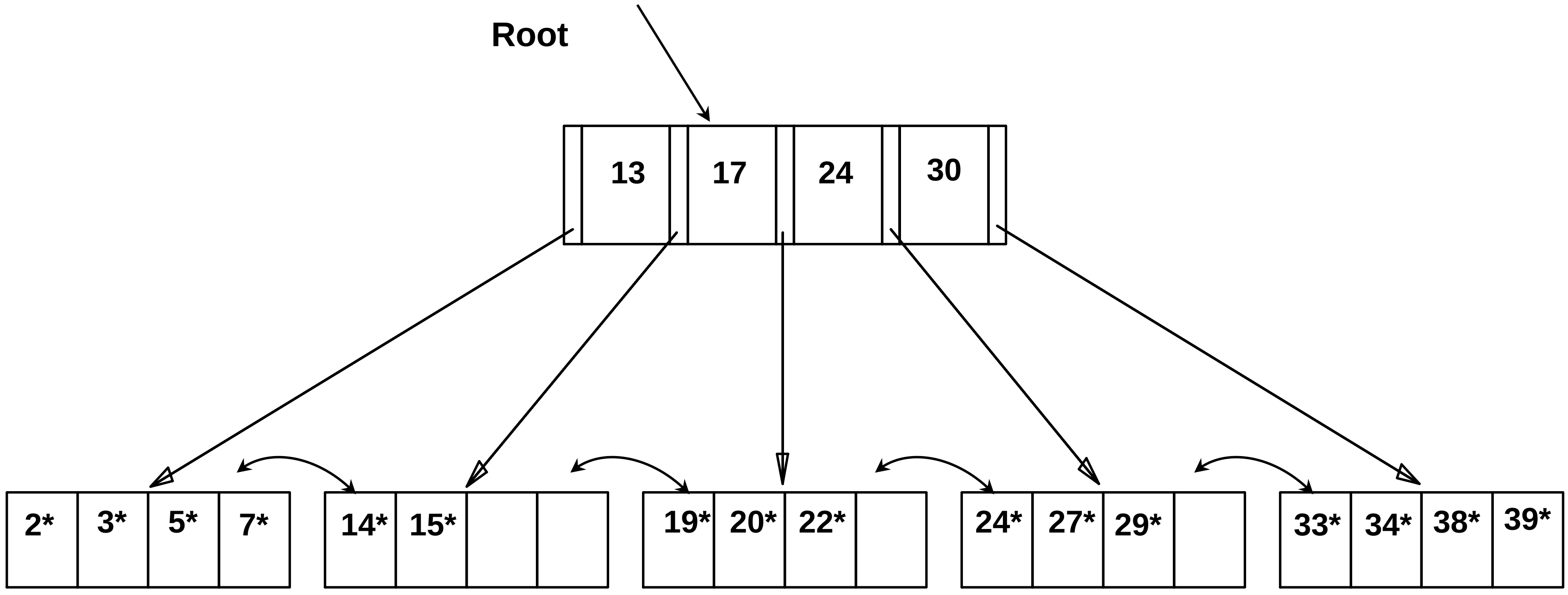
Example: B+ tree with order of 3

- Each node must hold at least 1 entry, and at most 2 entries, and 3 pointers.

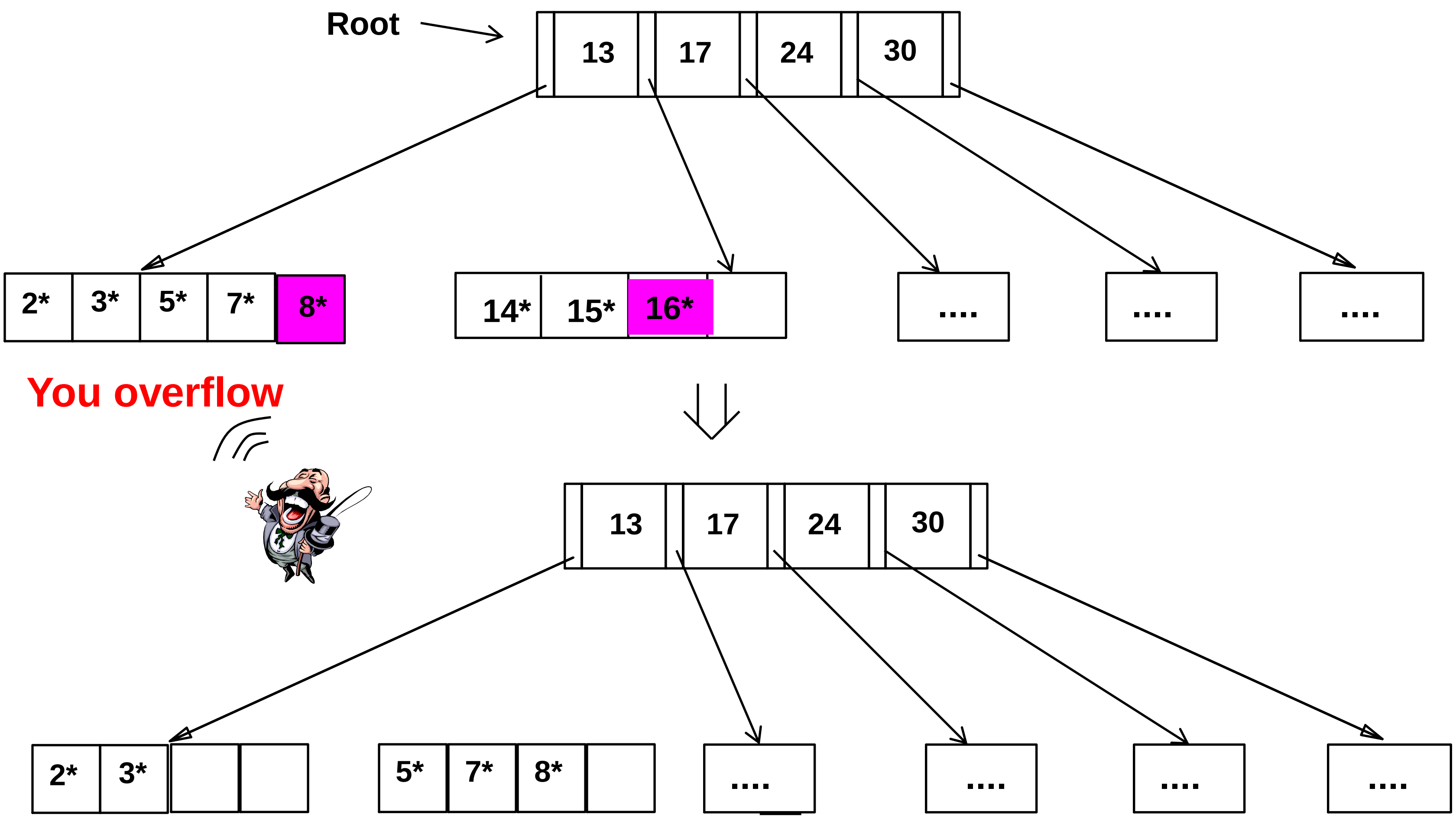


Example: Search in a B+ tree order 5

- **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.



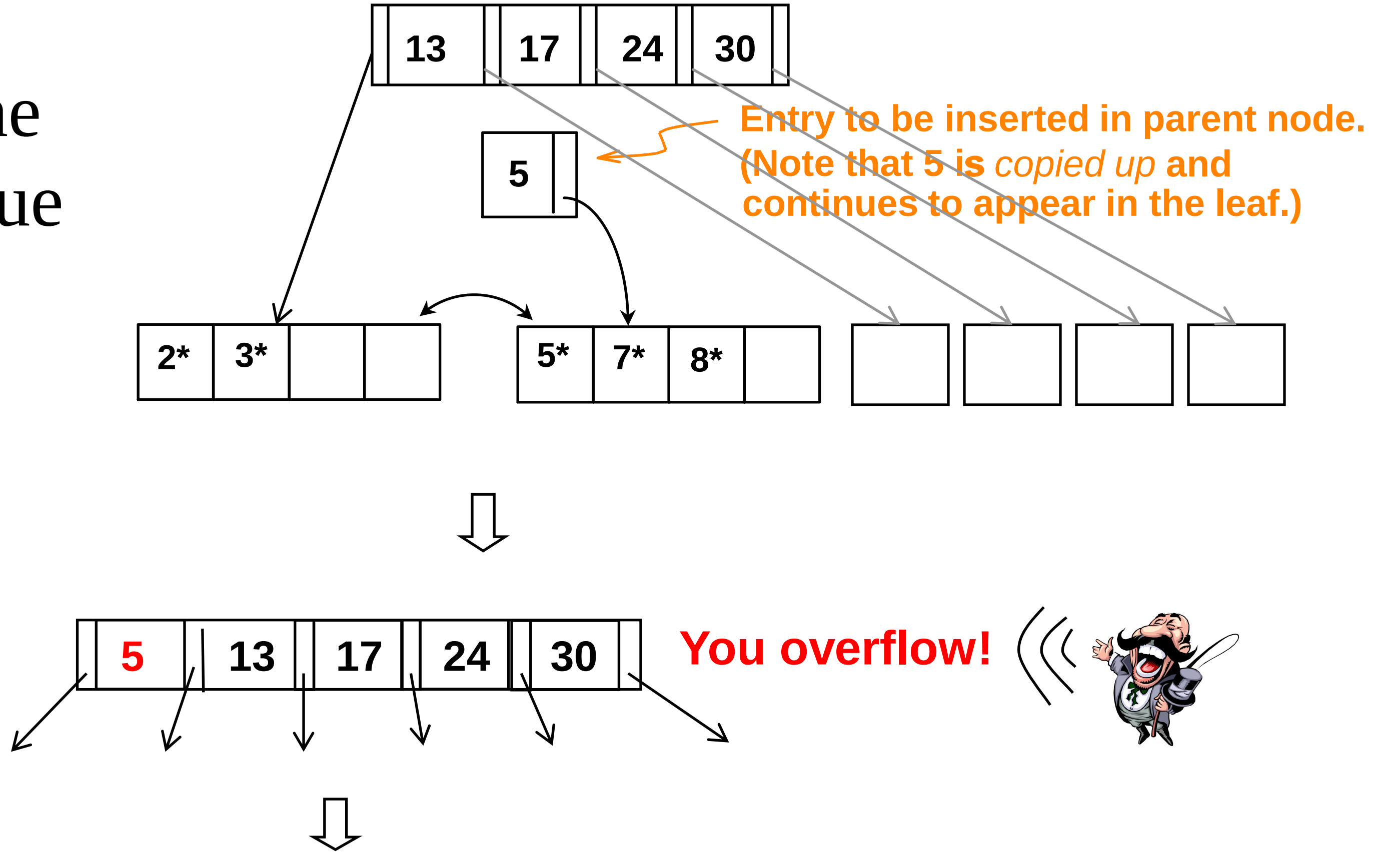
Inserting 16*, 8* into Example B+ tree



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

Inserting 8* (cont.)

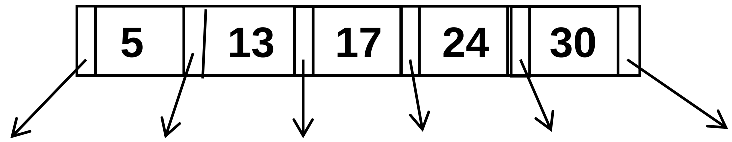
- Copy up the middle value (leaf split)



Insertion into B+ tree (cont.)

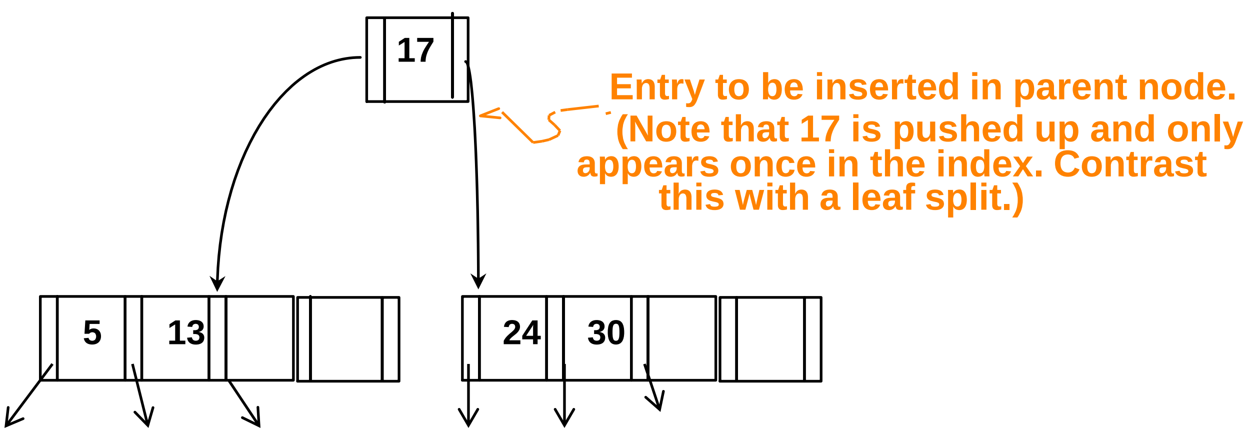
- Understand difference between copy-up and push-up

- Observe how minimum occupancy is guaranteed in both leaf and index pg splits.

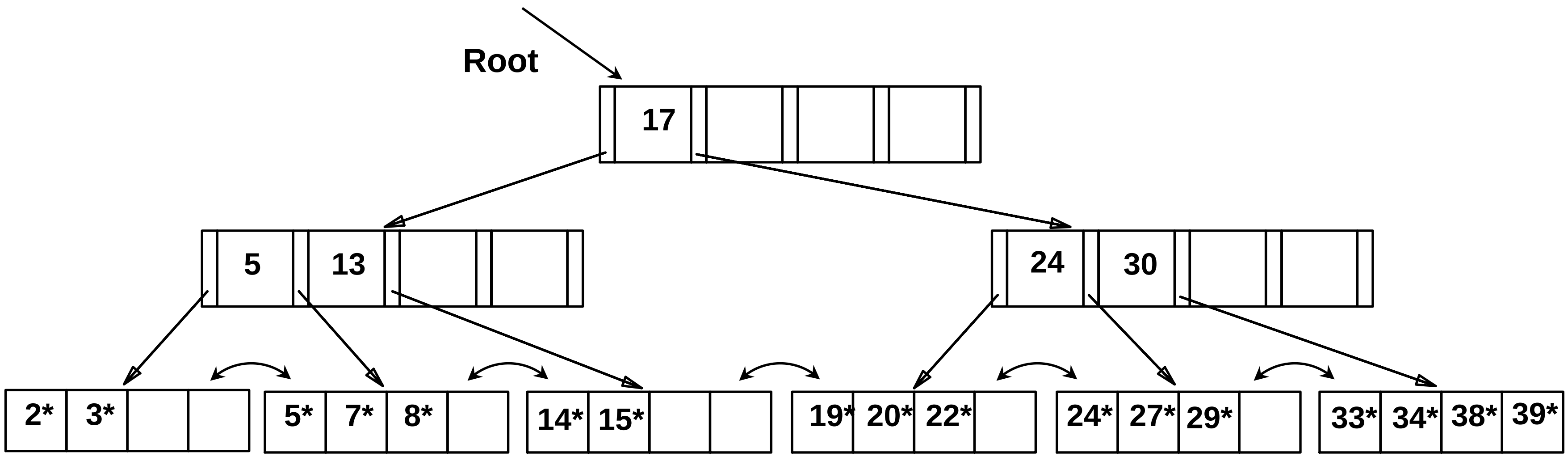


We split this node, redistribute entries evenly, and push up middle key.

⇓



Example B+ Tree After Inserting 8*

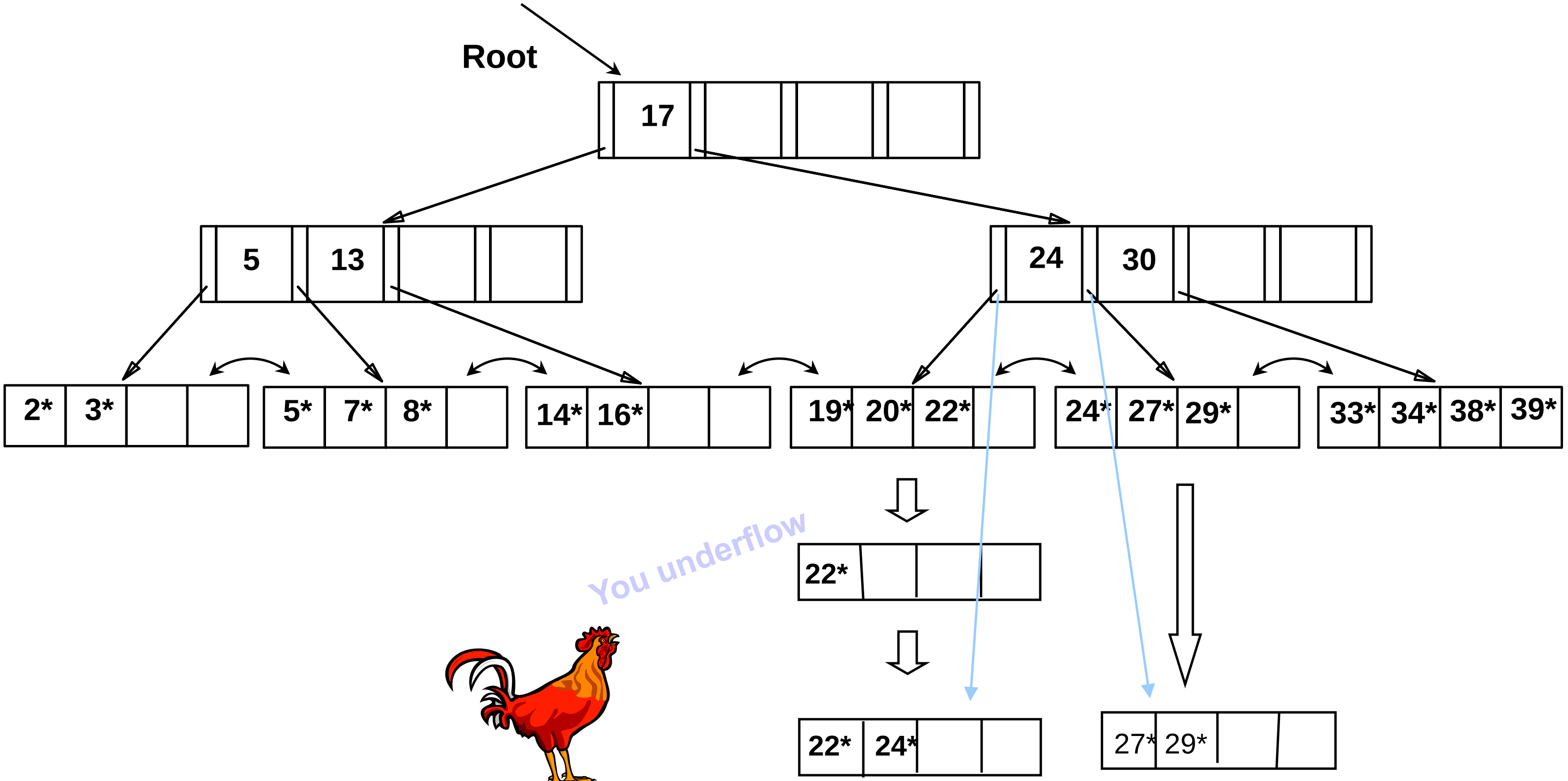


Notice that root was split, leading to increase in height.

Inserting a Data Entry into a B+ Tree: Summary

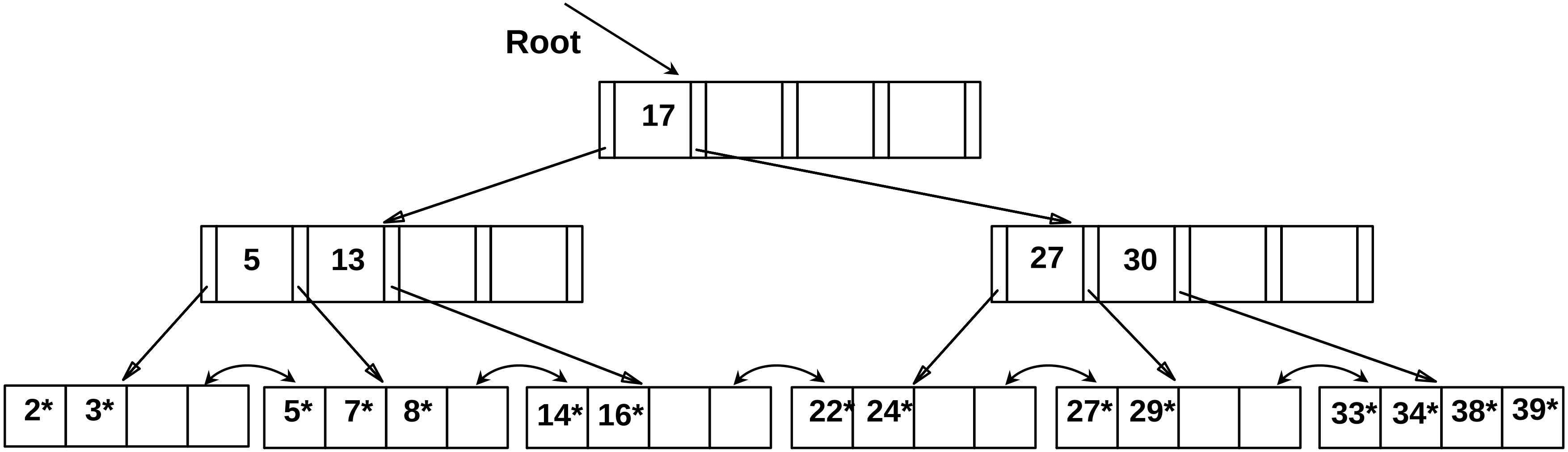
- Find correct leaf *L*.
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, put middle key in L2
    - **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can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

Delete 19* and 20*



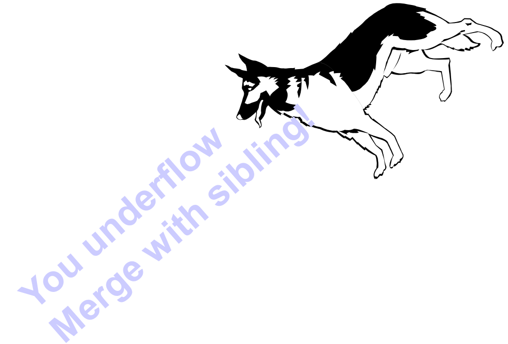
Have we still forgot something?

Deleting 19* and 20* (cont.)

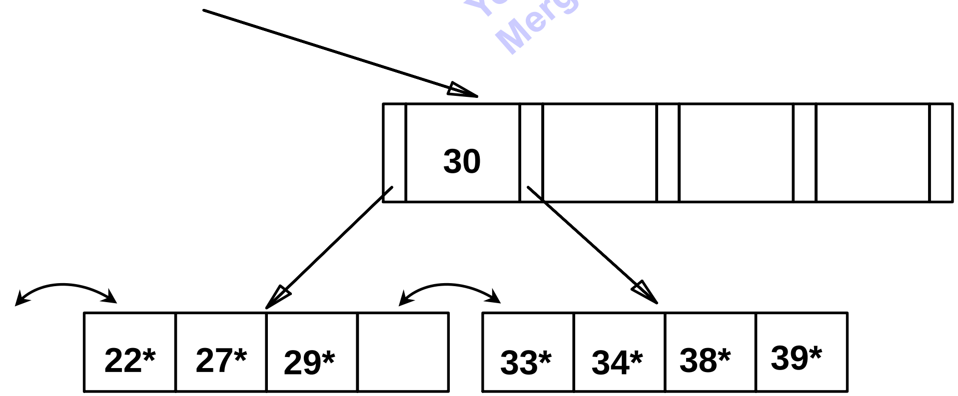


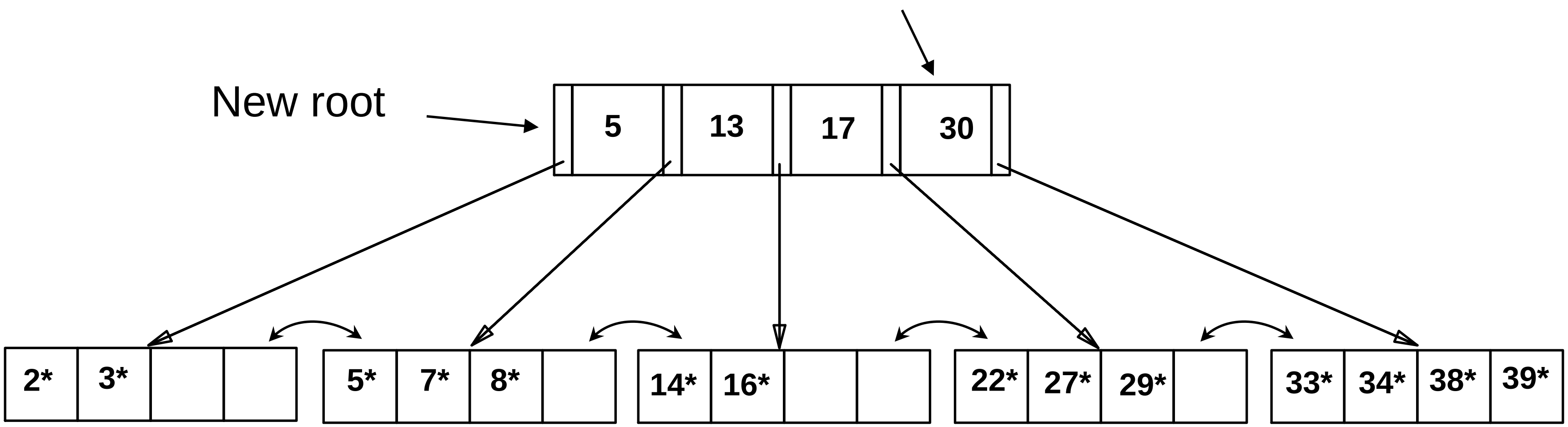
- Notice how 27 is *copied up*.
- But can we move it up?
- Now we want to delete 24
- Underflow again! But can we redistribute this time?

Deleting 24*



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



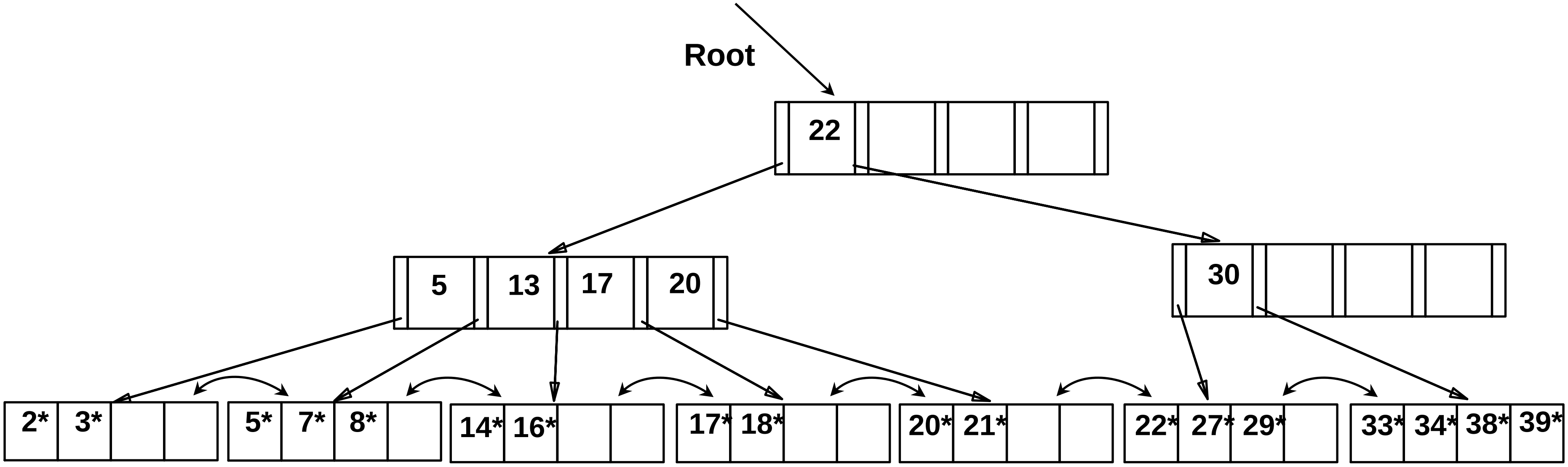


Deleting a Data Entry from a B+ Tree: Summary

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **(n/2)-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.

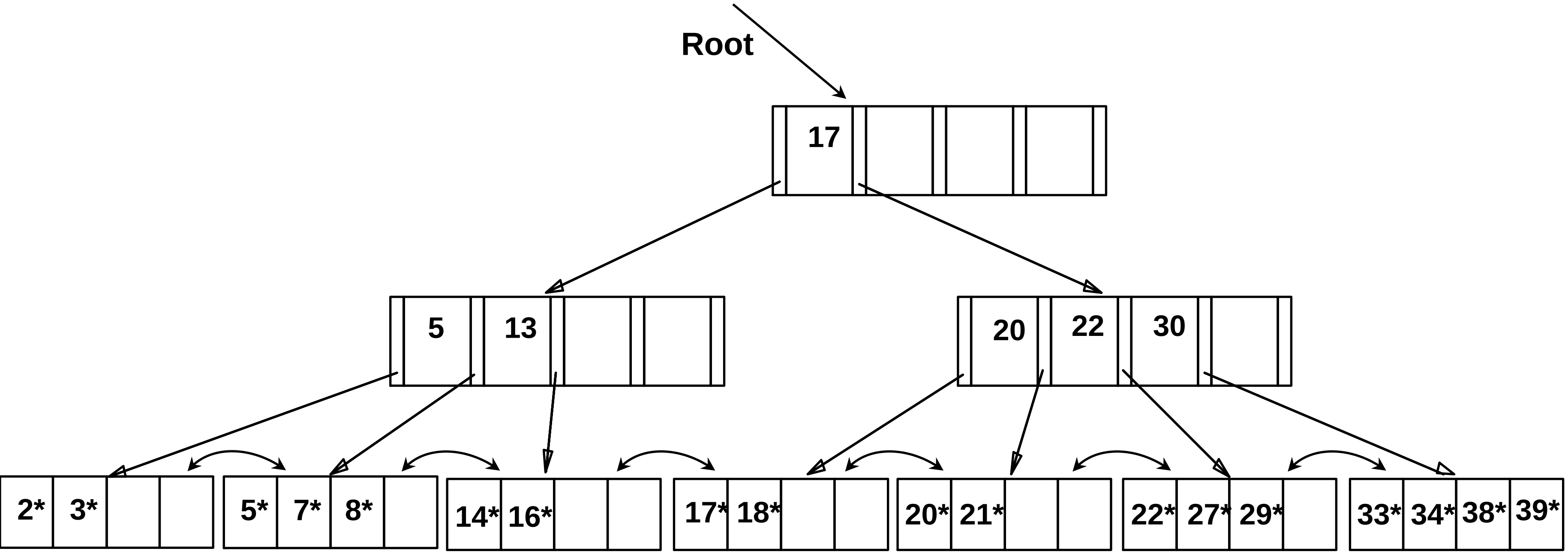
Example of Non-leaf Re-distribution

- 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.



B+-Tree Animation

Index Types

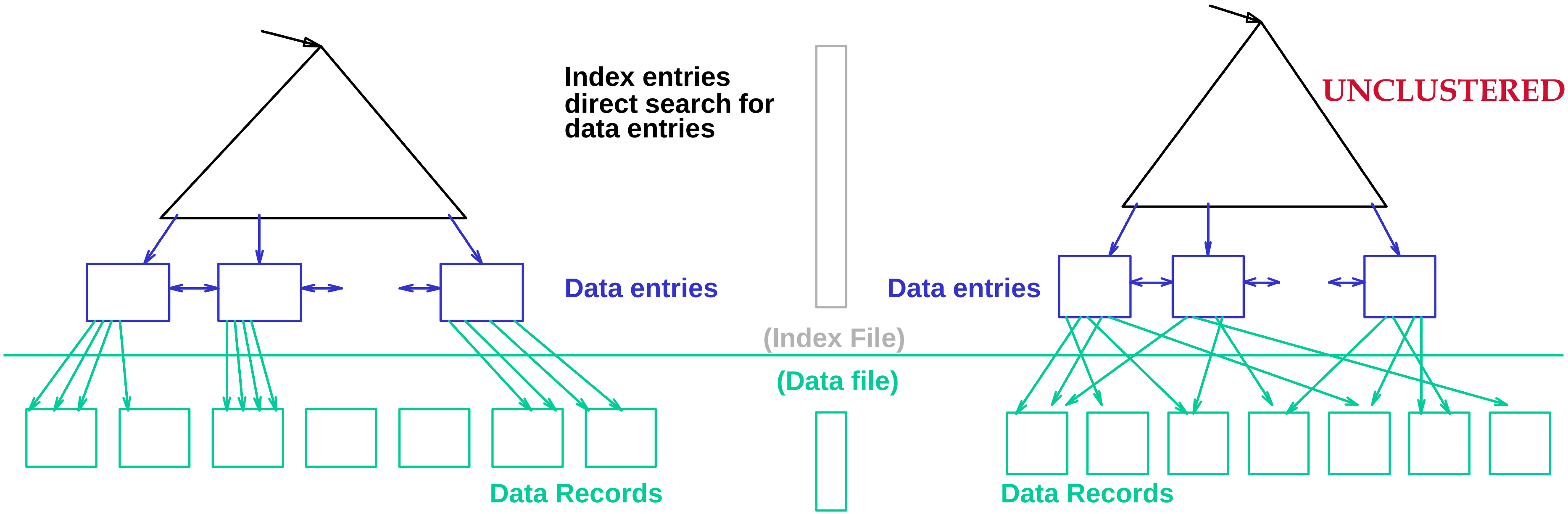
- **Primary and Clustered index**
  - Controls the physical order of rows
  - Does not require disk space
  - One per table (may inc. multiple columns)
  - Created by default on tables' Primary Key column

- **Secondary (Unclustered) Index**
  - Physical data structures that facilitate data retrieval
  - Can have many indexes
  - Indexes may include many columns

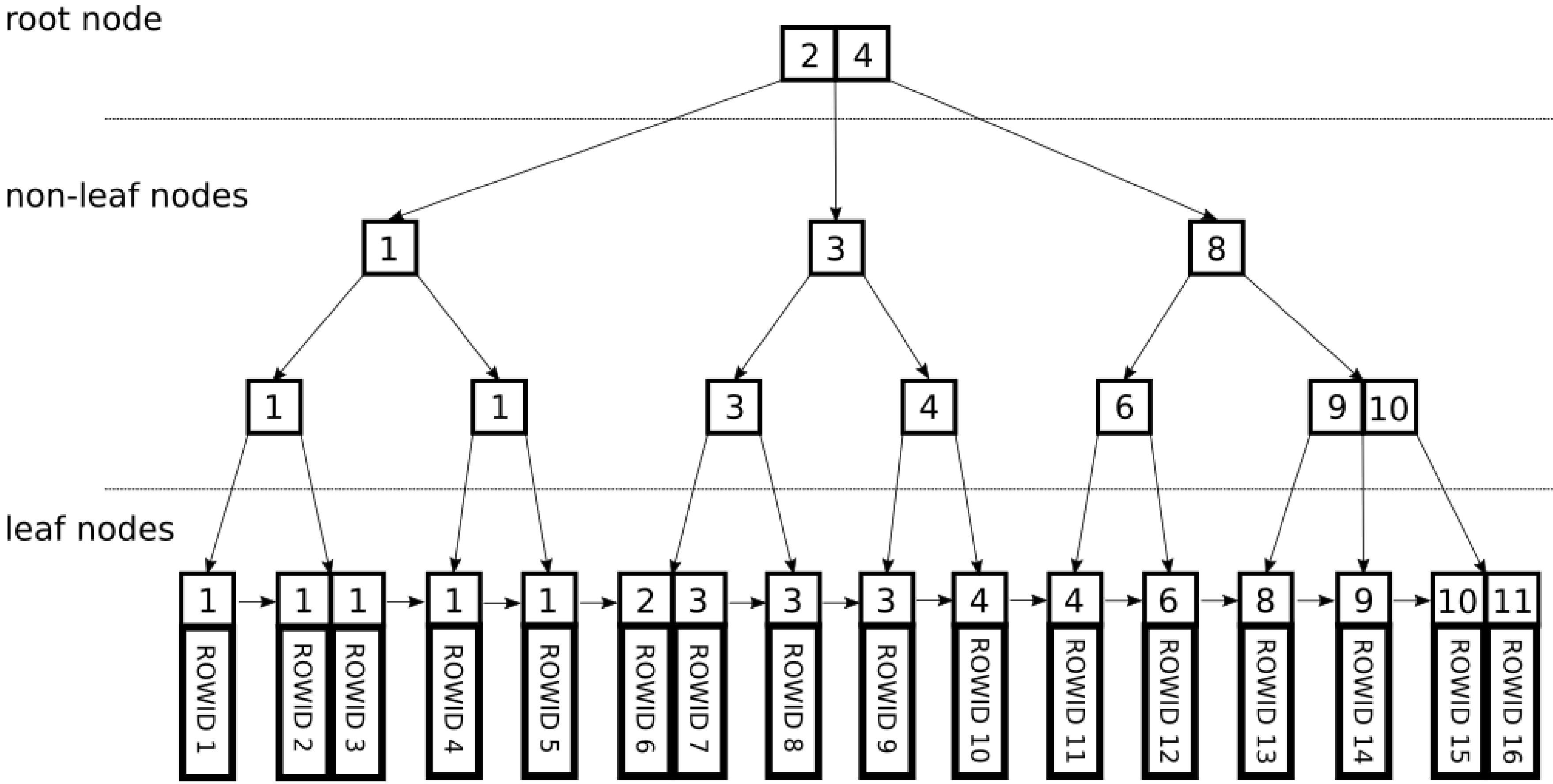
Primary and Clustered vs. Secondary (Unclustered) Index

Primary or clustered index

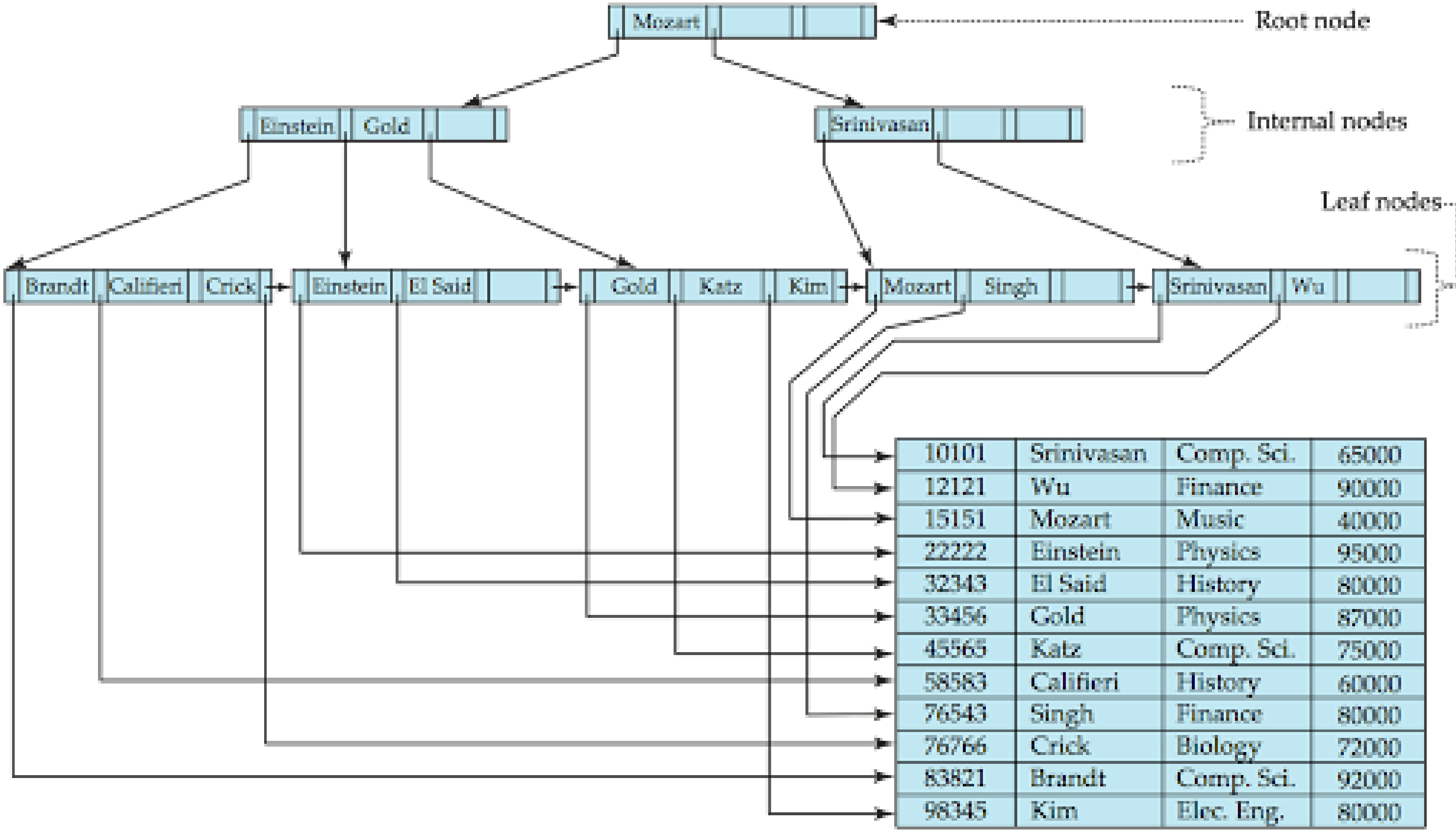
Secondary (Unclustered) index



B+ Trees (cont'd.)



B+ Trees (cont’d.)



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+ Trees (cont'd.)

- Properties
  - maximum branching factor of M
  - the root has between 2 and M children *or* at most L keys/values
  - other internal nodes have between $\lceil M/2 \rceil$ and M children
  - internal nodes contain only search keys (no data)
  - smallest datum between search keys x and y equals x
  - each (non-root) leaf contains between $\lceil L/2 \rceil$ and L keys/values
  - all leaves are at the same depth
- Result
  - height is $\Theta(\log_M n)$ between $\log_{M/2}(2n/L)$ and $\log_M(n/L)$
  - all operations run in $\Theta(\log_M n)$ time
  - operations get about $M/2$ to M or $L/2$ to L items at a time

Analysis of B+-Tree

- The maximum number of items in a B-tree of order m and height h:

root	$m - 1$
level 1	$m(m - 1)$
level 2	$m^2(m - 1)$
. . .	
level h	$m^h(m - 1)$

- So, the total number of items is

$$(1 + m + m^2 + m^3 + \ldots + m^h)(m - 1) =$$

$$[(m^{h+1} - 1)/ (m - 1)]\ (m - 1) = \boldsymbol{m^{h+1} - 1} = n \Rightarrow h = \log_m n$$

- When $m = 5$ and $h = 2$ this gives $5^3 - 1 = 124$

Analysis of B+-Tree

Time complexity in big O notation		
	Average	Worst case
Space	O(n)	O(n)
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Index Creation

- General form of the command to create an index

CREATE [UNIQUE] INDEX <index name>
ON <table name> (<column name> [<order>] { , <column name> [<order>] })
[CLUSTER] ;

  – Unique and cluster keywords are optional
  – Order can be ASC or DESC

- Secondary indexes can be created for any primary record organization
  – Complements other primary access methods

Definition of indexes in SQL

- The command for creating an index in SQL has the following format:

  CREATE INDEX IndexName
  ON RelationName (ColumnNameList) |ClusterName
  [ASC | DESC];

  where

  - IndexName is the name of the index being created
  - The ON clause specifies the object on which the index is allocated

  Ex:

  ```
  CREATE INDEX  nameIndex ON Person(name)
  ```

- An index created on more than one column is called *composite index*

Indexes on Multiple Keys

- Multiple attributes involved in many retrieval and update requests
- Composite keys
  - Access structure using key value that combines attributes
- Partitioned hashing
  - Suitable for equality comparisons

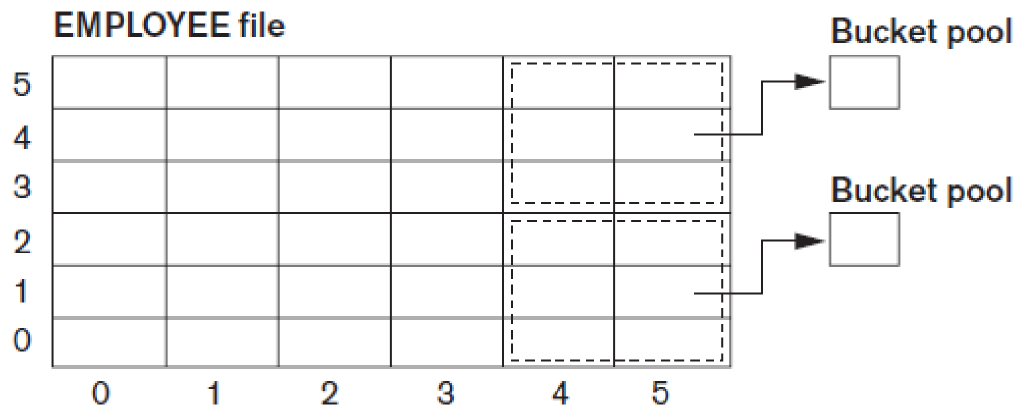
Indexes on Multiple Keys (cont'd.)

- Grid files
  - Array with one dimension for each search attribute

Dno

0	1, 2
1	3, 4
2	5
3	6, 7
4	8
5	9, 10

Linear scale
for Dno



Linear Scale for Age

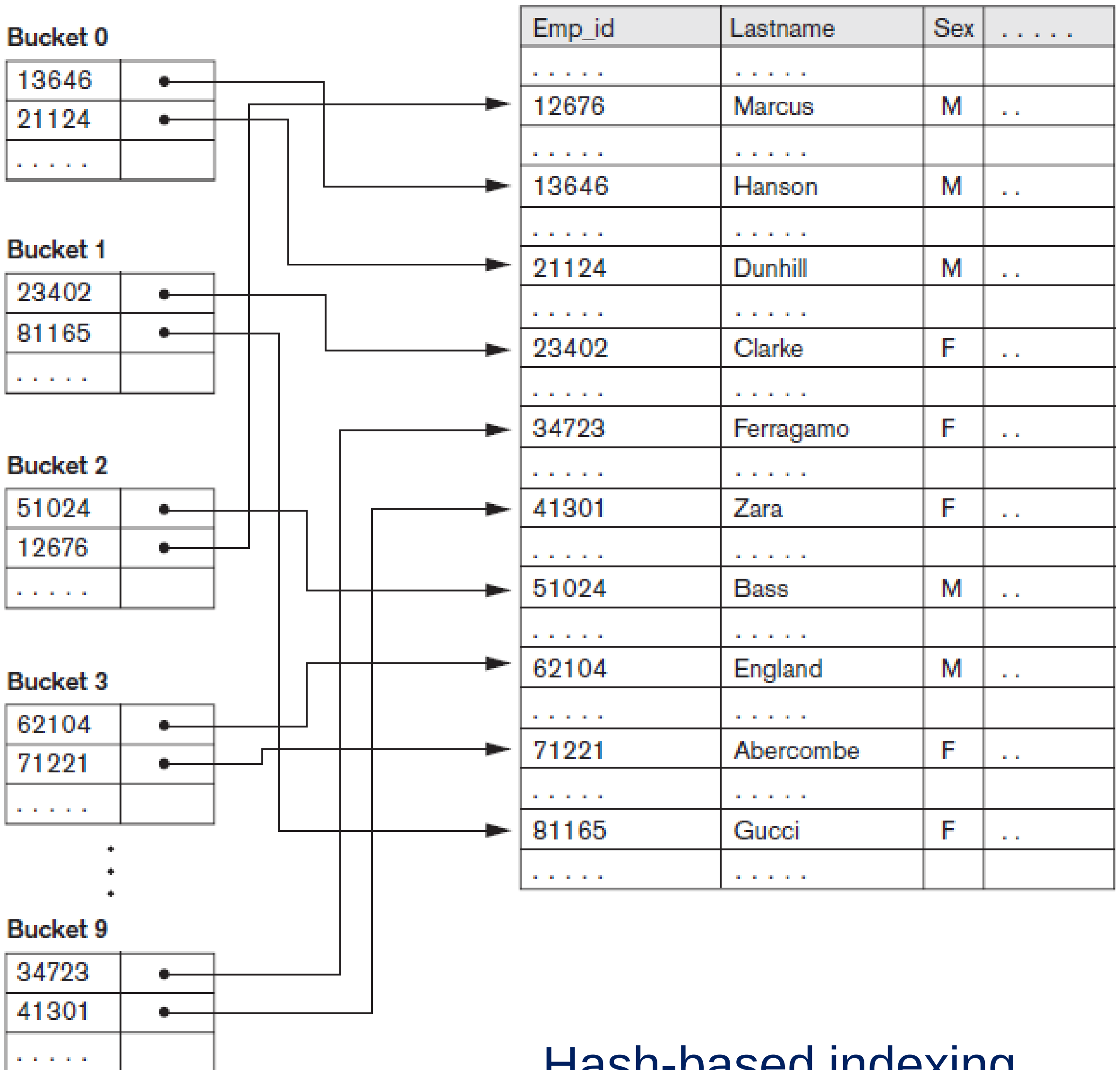
0	1	2	3	4	5
< 20	21–25	26–30	31–40	41–50	> 50

Example of a grid array on Dno and Age attributes

Other Types of Indexes

- Hash indexes
  - Secondary structure for file access
  - Uses hashing on a search key other than the one used for the primary data file organization
  - Index entries of form (K, P_r) or (K, P)
    - P_r: pointer to the record containing the key
    - P: pointer to the block (or bucket) containing the record for that key

Hash Indexes (cont'd.)



Hash-based indexing

Some General Issues Concerning Indexing

- **Physical index**
  - Pointer specifies physical record address
  - Disadvantage: pointer must be changed if record is moved
- **Logical index**
  - Used when physical record addresses expected to change frequently
  - Entries of the form (K, K_p)

Tuning Indexes

- Tuning goals
  - Dynamically evaluate requirements
  - Reorganize indexes to yield best performance
- Reasons for revising initial index choice
  - Certain queries may take too long to run due to lack of an index
  - Certain indexes may not get utilized
  - Certain indexes may undergo too much updating if based on an attribute that undergoes frequent changes

Physical Database Design in Relational Databases

- Physical design goals
  - Create appropriate structure for data in storage
  - Guarantee good performance
- Must know job mix for a particular set of database system applications
- Analyzing the database queries and transactions
  - Information about each retrieval query
  - Information about each update transaction

Physical Database Design in Relational Databases (cont'd.)

- Analyzing the expected frequency of invocation of queries and transactions
  - Expected frequency of using each attribute as a selection or join attribute
  - 80-20 rule: 80 percent of processing accounted for by only 20 percent of queries and transactions
- Analyzing the time constraints of queries and transactions
  - Selection attributes associated with time constraints are candidates for primary access structures

Physical Database Design Decisions

- Design decisions about indexing
  - Whether to index an attribute
    - Attribute is a key or used by a query
  - What attribute(s) to index on
    - Single or multiple
  - Whether to set up a clustered index
    - One per table
  - Whether to use a hash index over a tree index
    - Hash indexes do not support range queries
  - Whether to use dynamic hashing
    - Appropriate for very volatile files

Summary

- Indexes are access structures that improve efficiency of record retrieval from a data file
- Ordered single-level index types
  - Primary, clustering, and secondary
- Multilevel indexes can be implemented as B-trees and B+ -trees
  - Dynamic structures
- Multiple key access methods
- Logical and physical indexes

Summary (Cont.)

- 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 much 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.