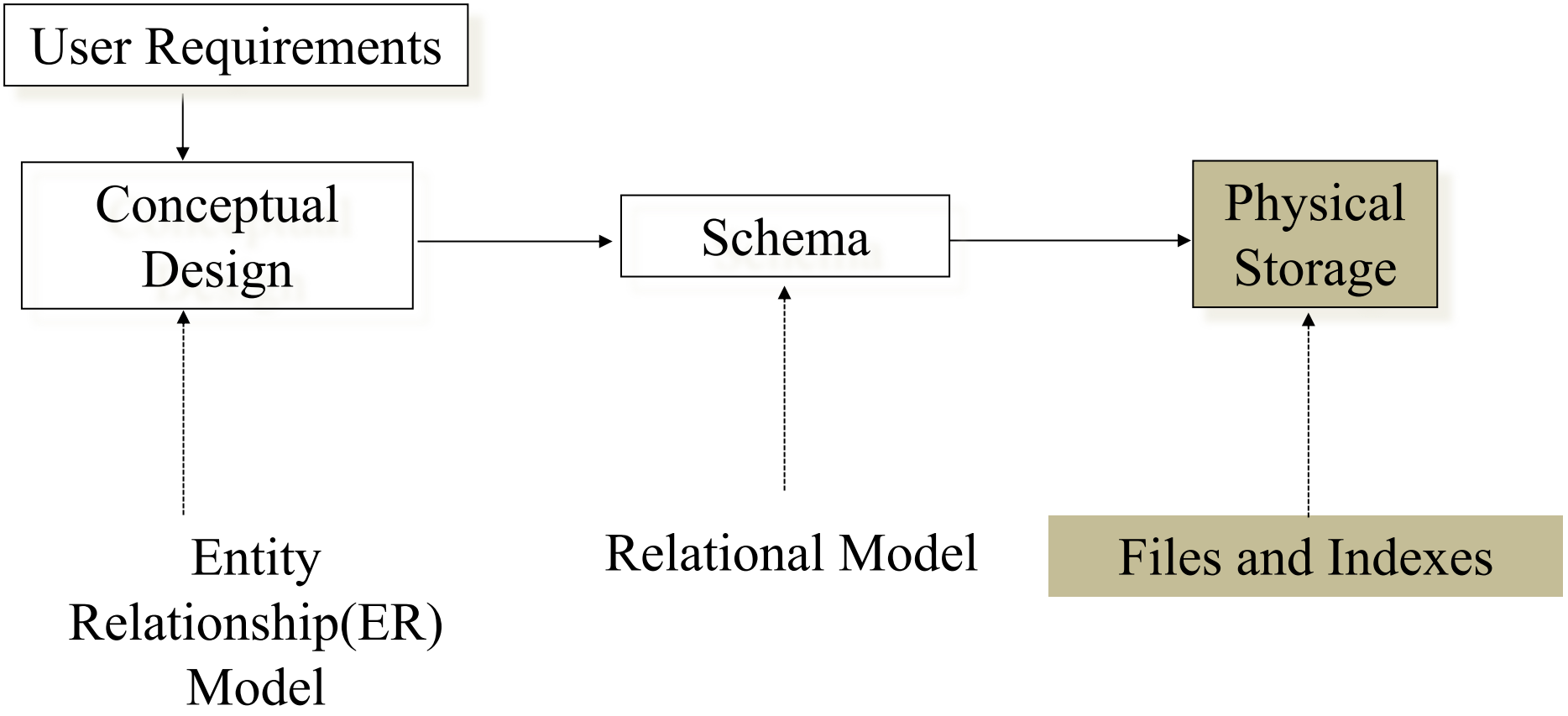
CS 540
Database Management Systems

Storage & indexing

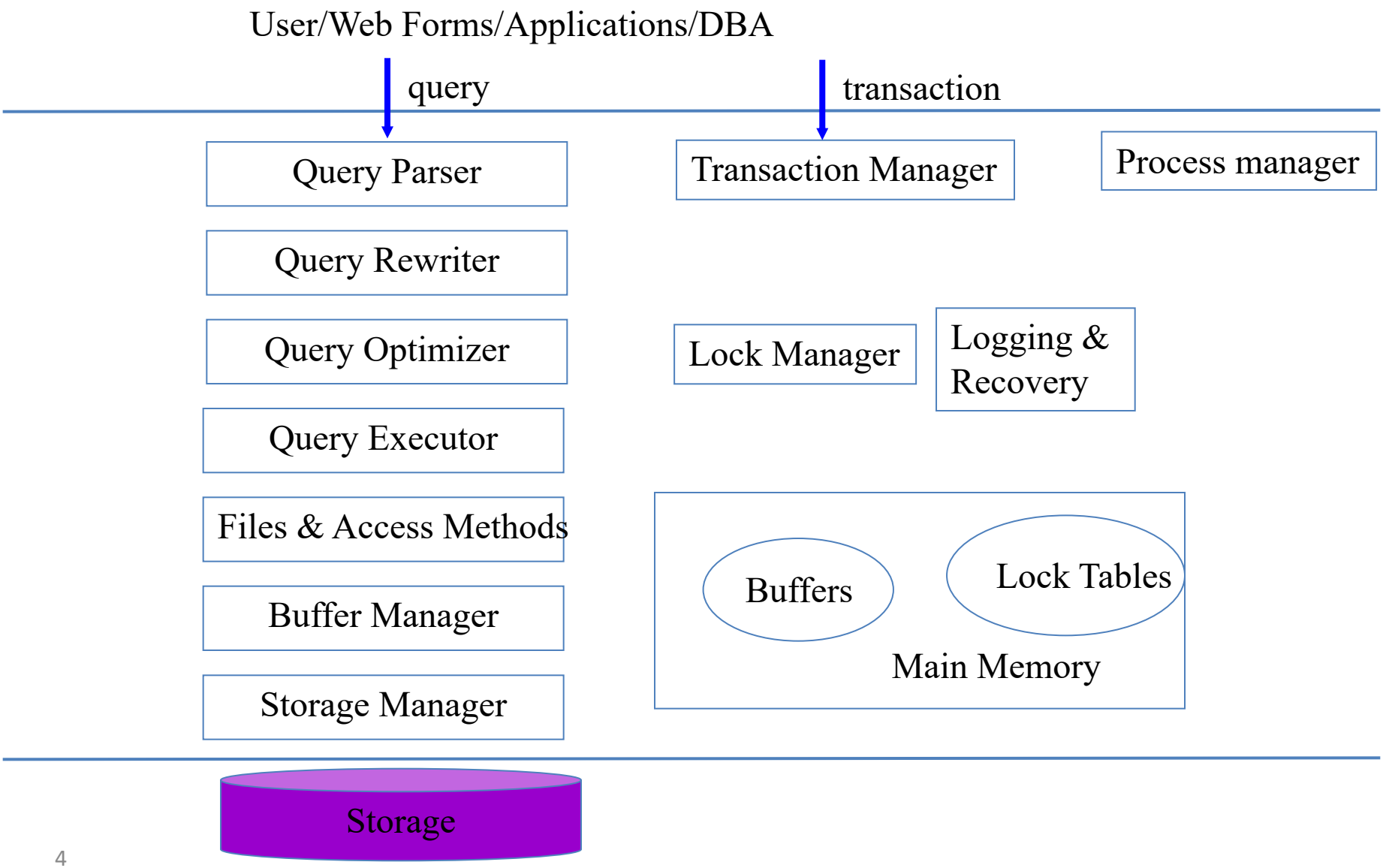
Database System Implementation



The advantage of RDBMS

- It separates logical level (schema) from physical level (implementation).
- Physical data independence
  - Users do not worry about how their data is stored and processes on the physical devices.
  - It is all SQL!
  - Their queries work over (almost) all RDBMS deployments.

DBMS Architecture



Challenges in physical level

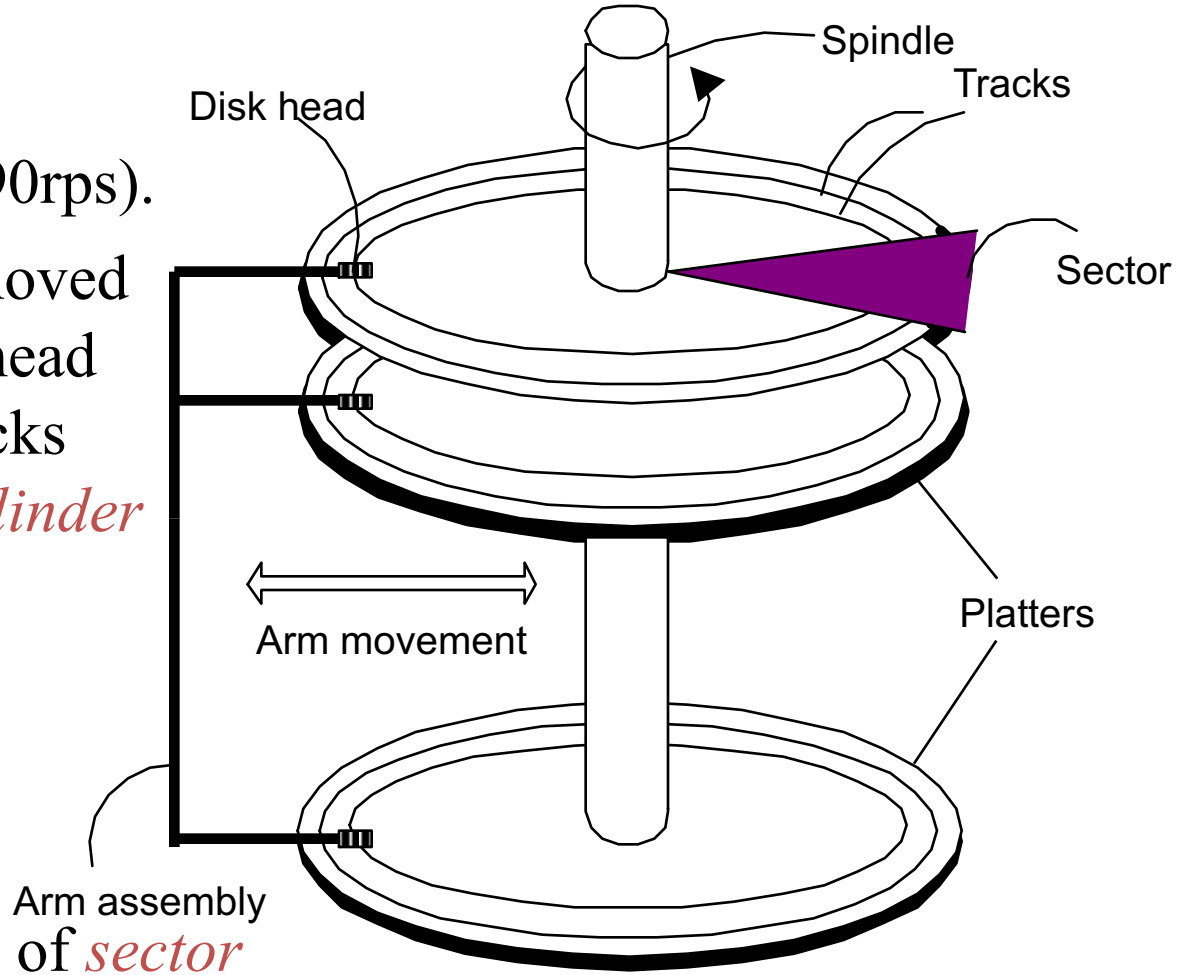
- Processor: 10000 – 100000 MIPS
- Main memory: around 10 Gb/ sec.
  - costs too much; volatile
- Secondary storage (hard disk, SSD, DNA, …)
  - higher capacity and durability
  - data is stored and retrieved in units: *blocks* or *pages*
  - disk access
    - seek time + rotational latency + transfer time
    - seek time: about 4 ms - 15 ms!
    - rotational latency: about 2 ms – 7 ms!
    - transfer time: about 1000 Mb/ sec
    - seek time and rotational delay dominate.

General approach

- Typical storage hierarchy:
  - main memory (RAM) for currently used data.
  - secondary storage (disk) for the main database.
- Main memory is getting cheaper
  - different storage hierarchy?
- Key to lower I/O cost: reduce seek/rotation delays! Hardware vs. software solutions?

Components of a disk

- The platters spin (say, 90rps).
- The arm assembly is moved in or out to position a head on a desired track. Tracks under heads make a *cylinder* (imaginary!).
- Only one head reads/writes at any one time.
- *Block size* is a multiple of *sector size* (which is fixed).



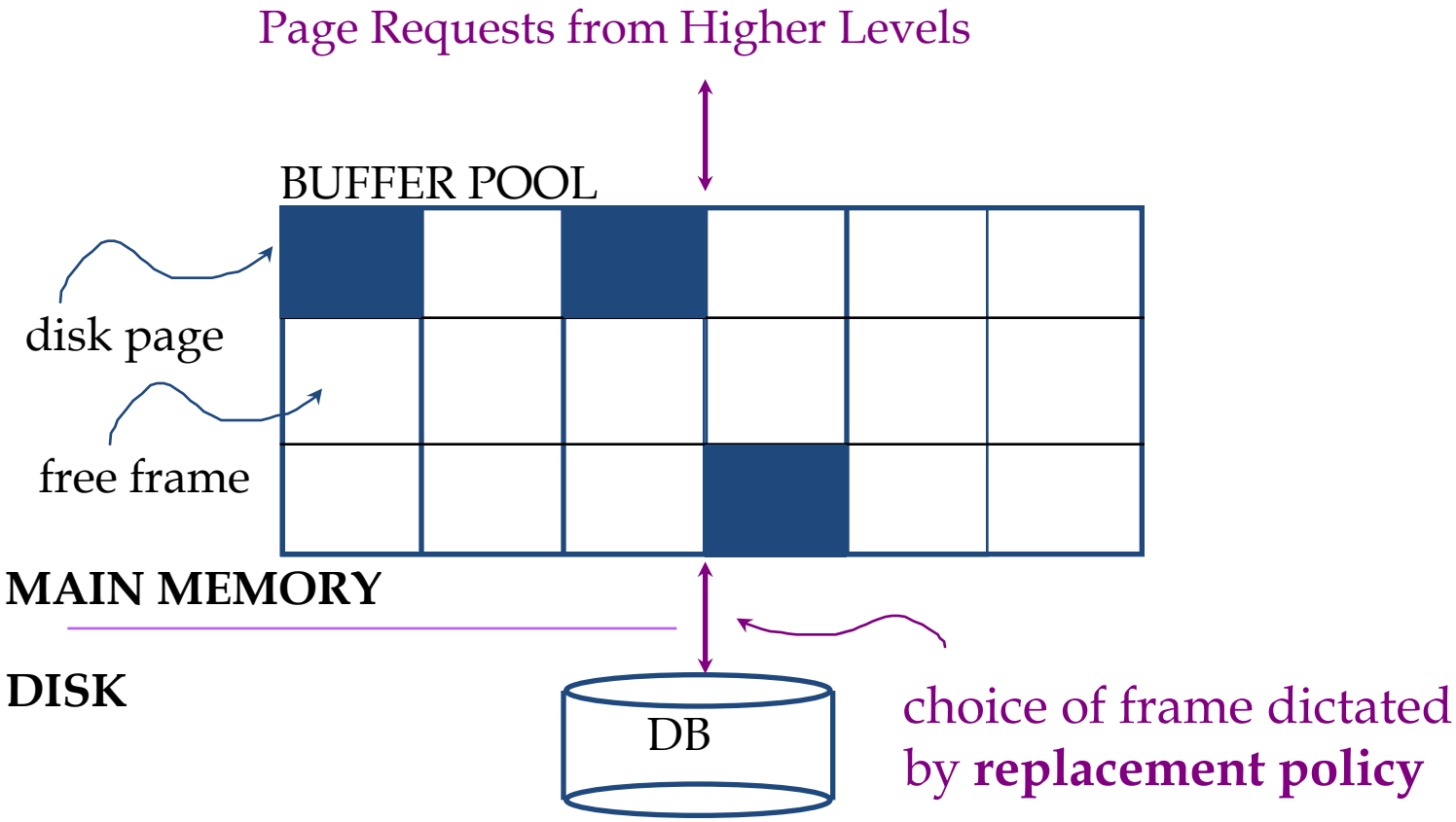
Random versus sequential access

- Disk random access (example ?)
  - seek time + rotational latency + transfer time.
- Disk sequential access (example ?)
  - *`next'* block concept:
    - blocks on same track, followed by
    - blocks on same cylinder, followed by
    - blocks on adjacent cylinder
  - blocks in a file should be arranged sequentially on disk (by `next'), to minimize seek and rotational delay.
- For a sequential scan, *pre-fetching* several pages at a time is a big win!

Storage management

- Lowest layer of DBMS software manages space on disk.
- Higher levels call upon this layer to:
  - allocate/de-allocate a page
  - read/write a page
- Request for a *sequence* of pages must be satisfied by allocating the pages sequentially on disk!
  - higher levels don't need to know how this is done, or how free space is managed.

Buffer management



- *Data must be in RAM for DBMS to operate on it!*
- *Table of <frame#, pageid> pairs is maintained.*

When a page is requested ...

- If requested page is not in pool:
  - Choose a frame for *replacement*
  - If frame is dirty, write it to disk
  - Read requested page into chosen frame
- *Pin* the page and return its address.

☛ *If requests can be predicted (e.g., sequential scans) pages can be pre-fetched several pages at a time!*

More on buffer management

- Requestor of page must unpin it, and indicate whether page has been modified:
  - *dirty* bit is used for this.
- Page in pool may be requested many times,
  - a *pin count* is used. A page is a candidate for replacement iff *pin count* = 0.
- CC & recovery may entail additional I/O when a frame is chosen for replacement. (*Write-Ahead Log* protocol; more later.)

Buffer replacement policy

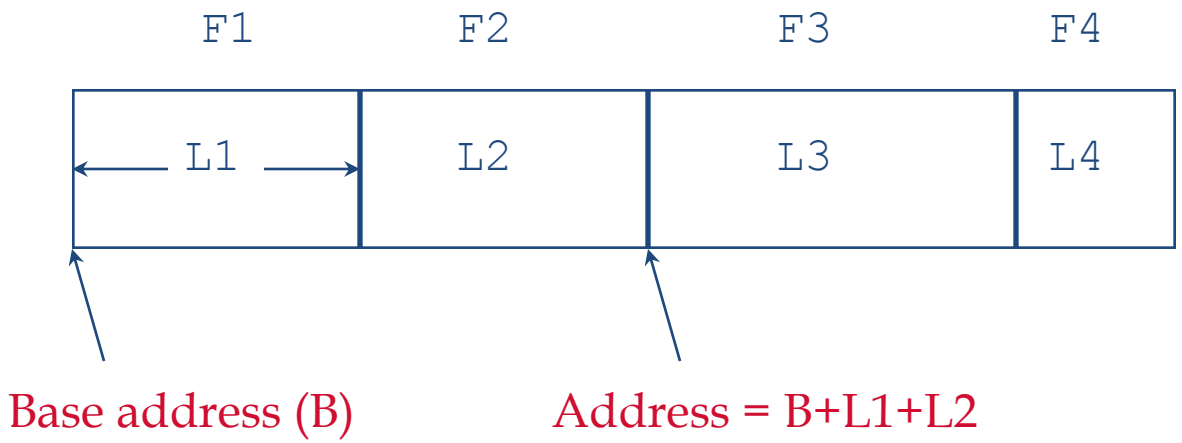
- Frame is chosen for replacement by a *replacement policy:*
  - Least-recently-used (LRU), Clock, MRU etc.
- Policy can have big impact on # of I/O's; depends on the *access pattern*.
- *Sequential flooding*: nasty situation caused by LRU + repeated sequential scans.
  - # buffer frames < # pages in file means each page request causes an I/O. MRU much better in this situation (but not in all situations, of course).

DBMS vs. OS File System

OS does disk space & buffer mgmt: why not let OS manage these tasks?

- Differences in OS support: portability issues
- Some limitations, e.g., files can't span disks.
- Buffer management in DBMS requires ability to:
  - pin a page in buffer pool, force a page to disk (important for implementing CC & recovery),
  - adjust *replacement policy,* and pre-fetch pages based on access patterns in typical DB operations.

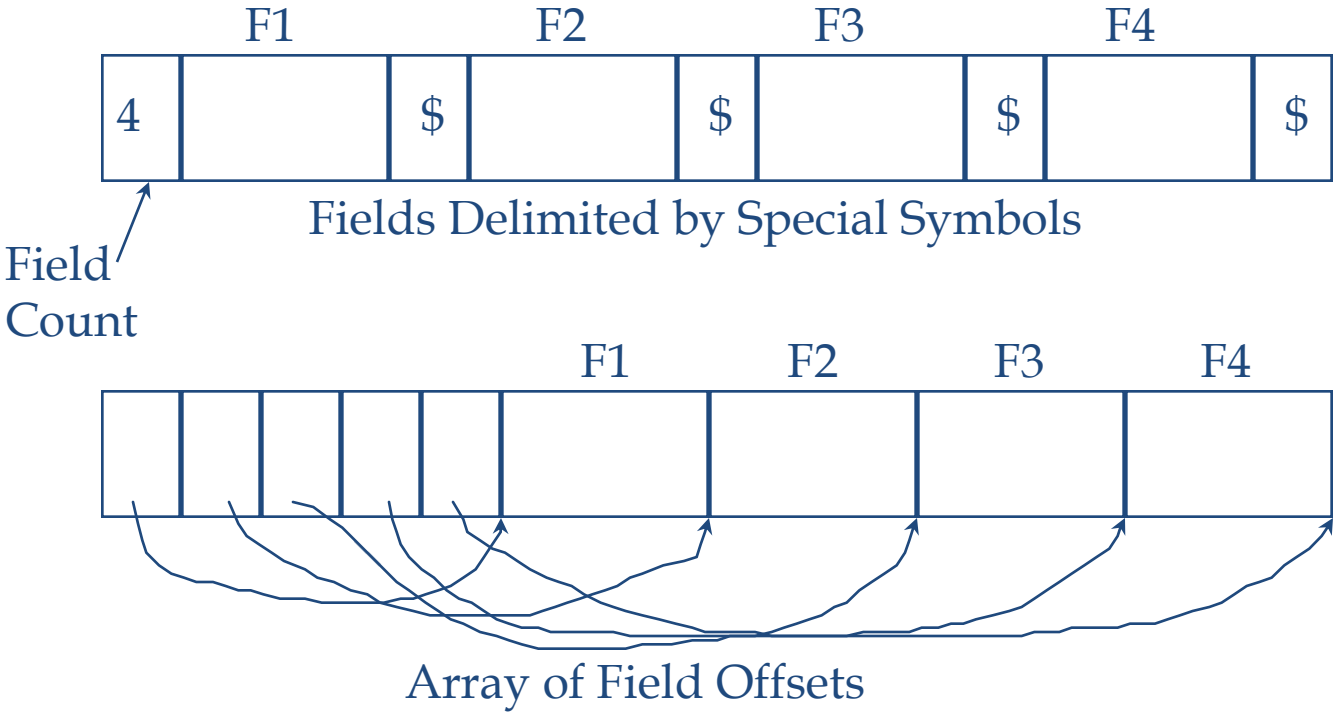
Record formats: fixed length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field does not require scan of record.

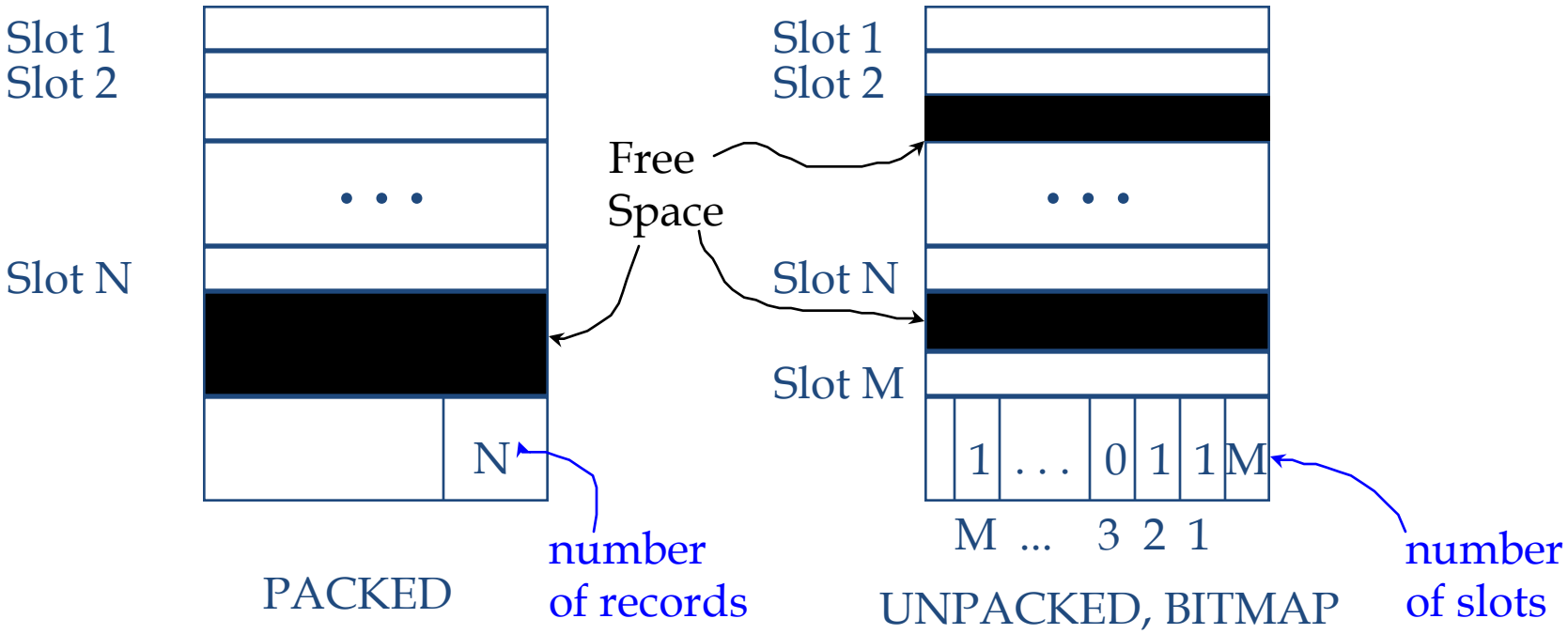
Record formats: variable length

- Two alternative formats (# fields is fixed):



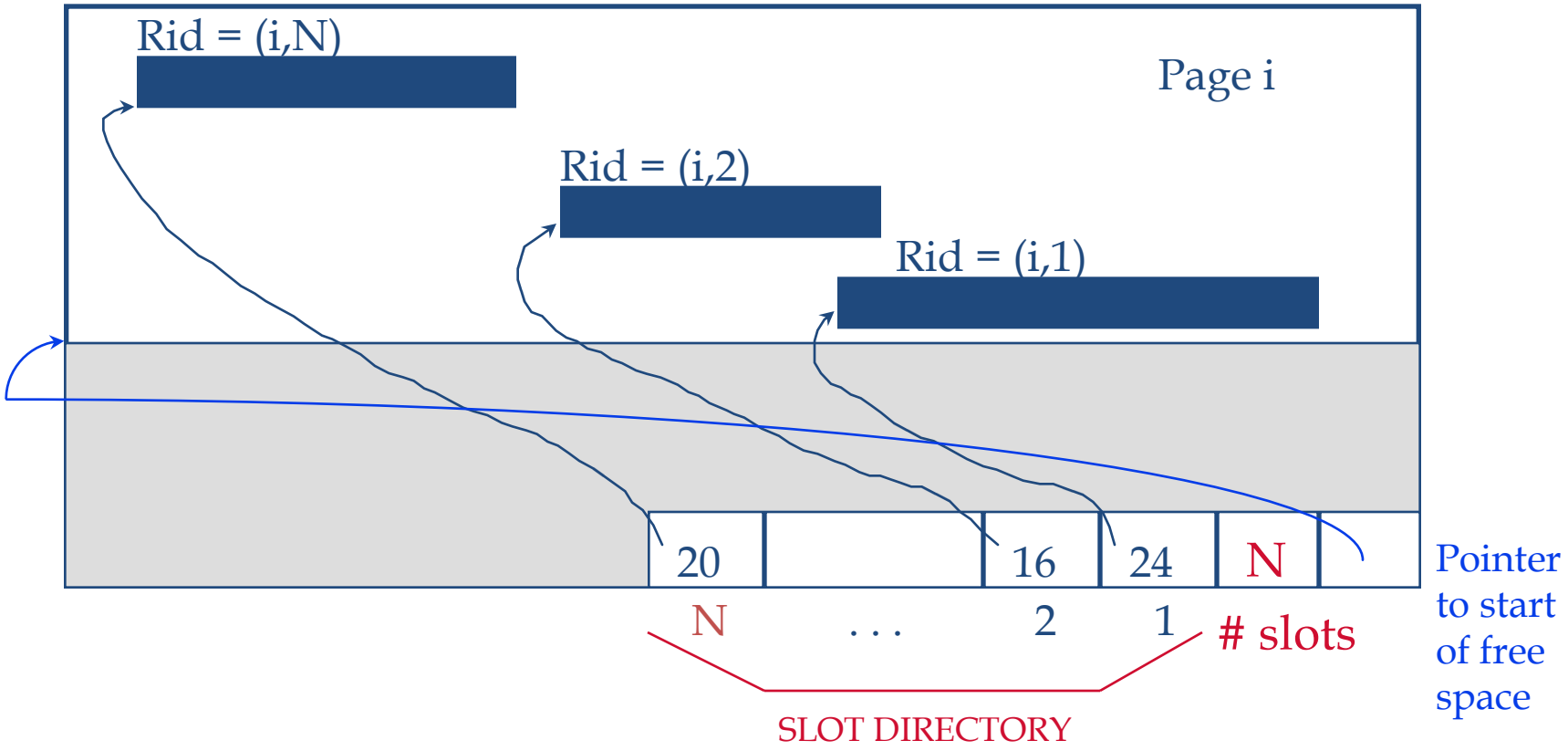
☛ Second offers direct access to i'th field, efficient storage of *nulls* (special *don't know* value); small directory overhead.

Page formats: fixed length records



☛ *Record id = <page id, slot #>. In first alternative, moving records for free space management changes rid; may not be acceptable.*

Page formats: variable length records



☛*Can move records on page without changing rid; so, attractive for fixed-length records too.*

Spanned versus un-spanned

- Un-spanned
  - Each records belongs to only one block
- Spanned
  - Records may be stored across multiple blocks
  - Saves space
  - The only way to deal with large records and fields: blob, image, …

Files of records

- Page or block is OK when doing I/O, but higher levels of DBMS operate on *records*, and *files of records*.
- FILE: A collection of pages, each containing a collection of records. Must support:
  - insert/delete/modify record
  - read a particular record (specified using *record id*)
  - scan all records (possibly with some conditions on the records to be retrieved)

System catalogs

- For each relation:
  - name, file name, file structure (e.g., Heap file)
  - attribute name and type, for each attribute
  - index name, for each index
  - integrity constraints
- For each view:
  - view name and definition
- Plus statistics, authorization, buffer pool size, etc.

☛ *Catalogs are themselves stored as relations*!

Access paths

- The methods that RDBMS uses to retrieve data.
- Attribute value(s) ➔ Tuple(s)
- Point query over *Coffee(<u>cname</u>, producer)*

```
Select *
 From coffee
 Where cname = 'Costa';
```

- Range query over *Sells(<u>sname</u>, <u>cname</u>, price)*

```
Select *
 From Sells
 Where price > 2 AND price < 10;
```

Types of access paths

- Heap files
  - there is not any order in the file
  - new blocks are inserted at the end of the file.
- Sorted files
  - order blocks (and records) based on some key.
  - physically contiguous or using links
- Average cost of heap versus sorted files
  - search?
  - insertion/update?
  - deletion?
- Middle ground?

Indexing

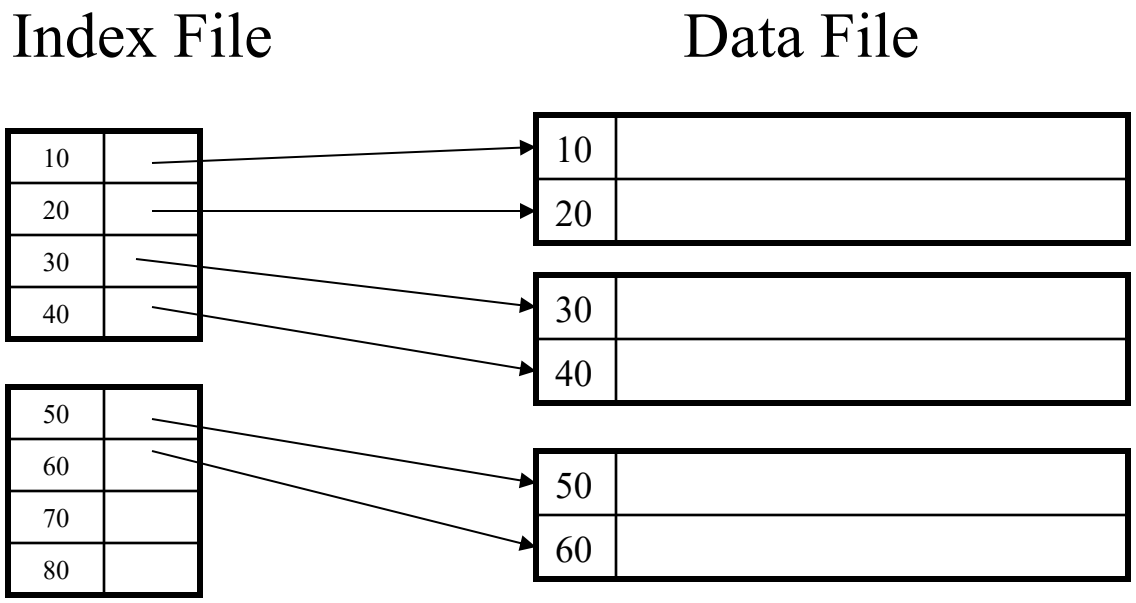
- An old idea

Index

- A data structure that speeds up selecting tuples in a relation based on some **search keys**.
- Search key
  - A subset of the attributes in a relation
  - May not be the same as the (primary) key
- Entries in an index
  - (k, r)
  - k is the search key.
  - r is the pointer to a record (record id).

Index

- **Data file** stores the table data.
- **Index file** stores the index data structure.



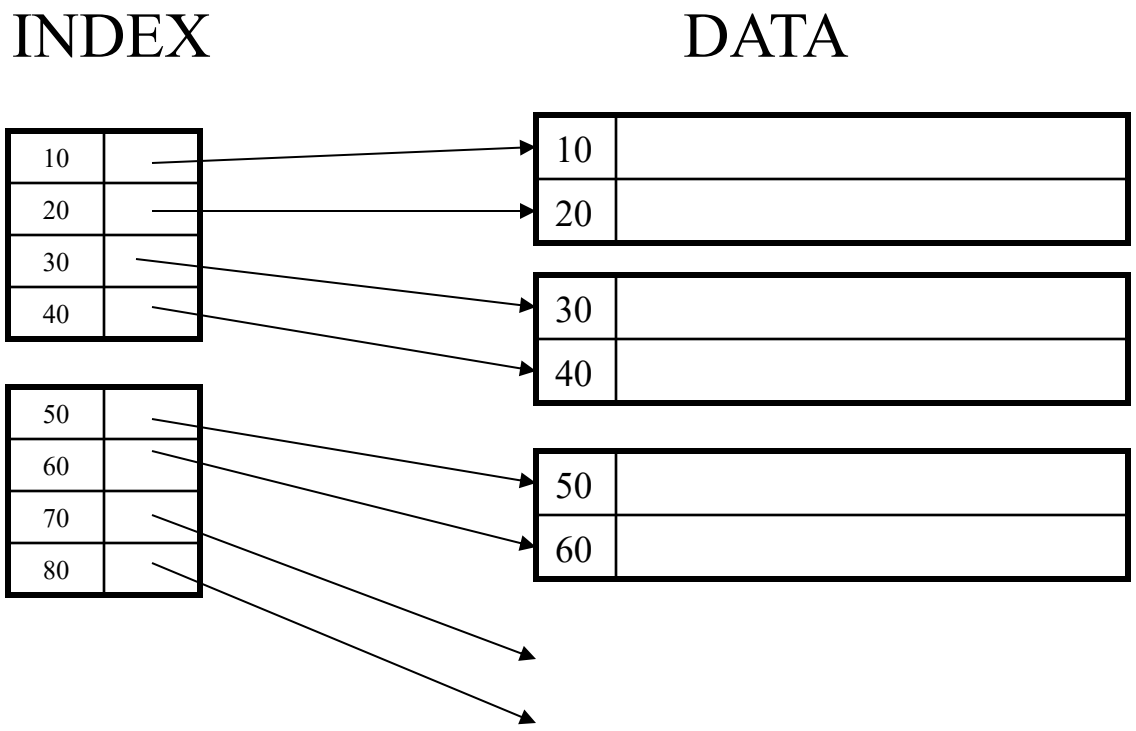
- Index file is smaller than the data file.
- Ideally, the index should fit in the main memory.

Index categorizations

- Clustered vs. unclustered
  - Records are stored according to the index order.
  - Records are stored in another order, or not any order.
- Dense vs. sparse
  - Each record is pointed by an entry in the index.
  - Each block has an entry in the index.
  - Size versus time tradeoff.
- Primary vs. secondary
  - Primary key is the search key
  - Other attributes.

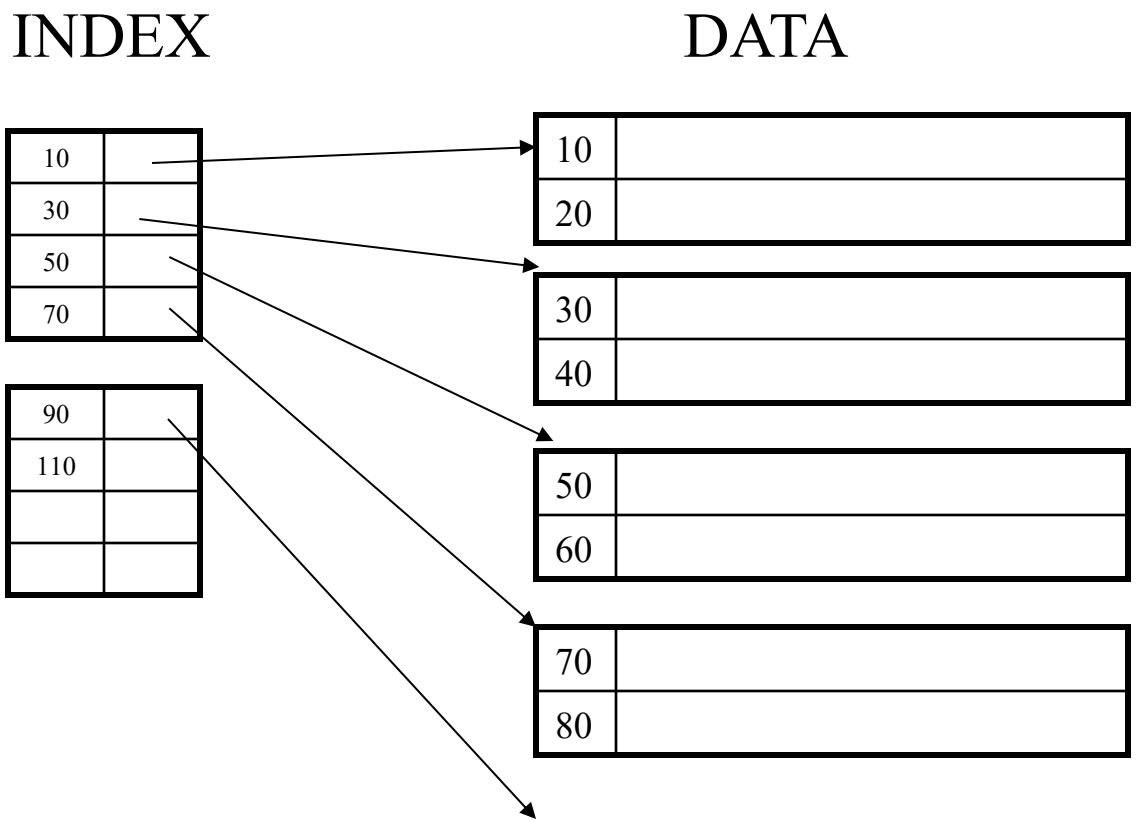
Index categorizations

- Clustered and dense



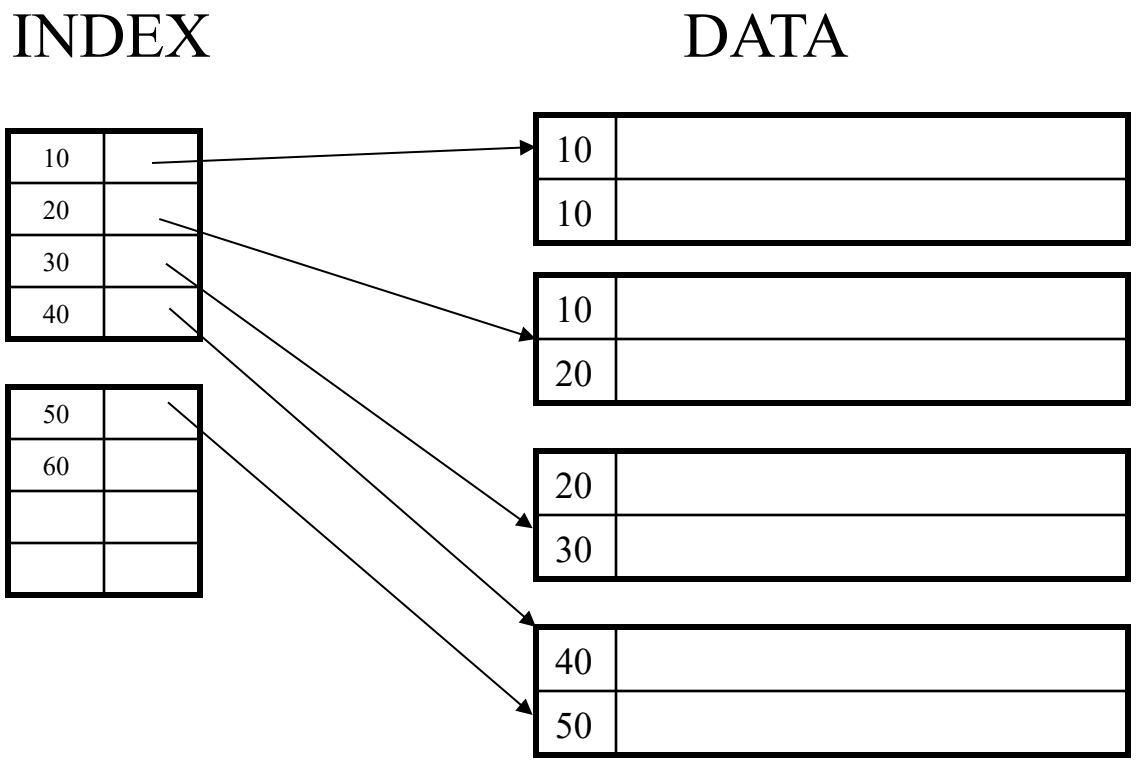
Index categorizations

- Clustered and sparse



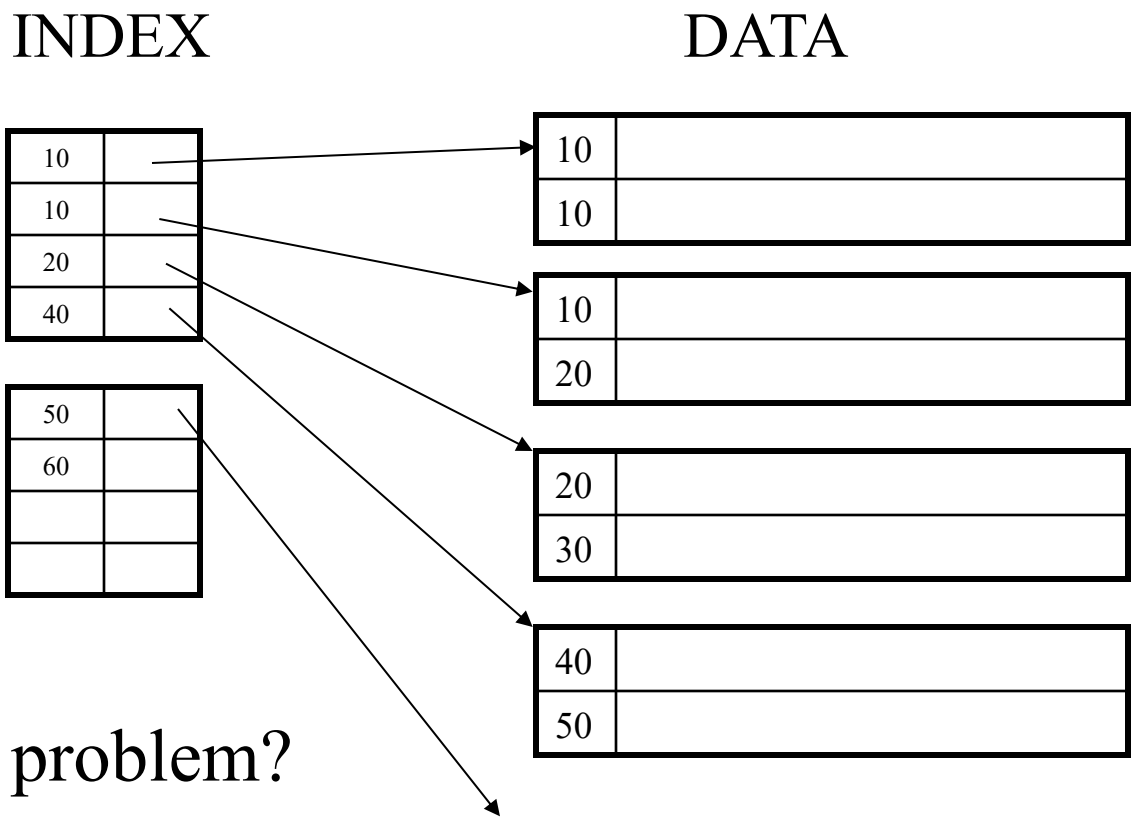
Duplicate search keys

- Clustered and dense



Duplicate search keys

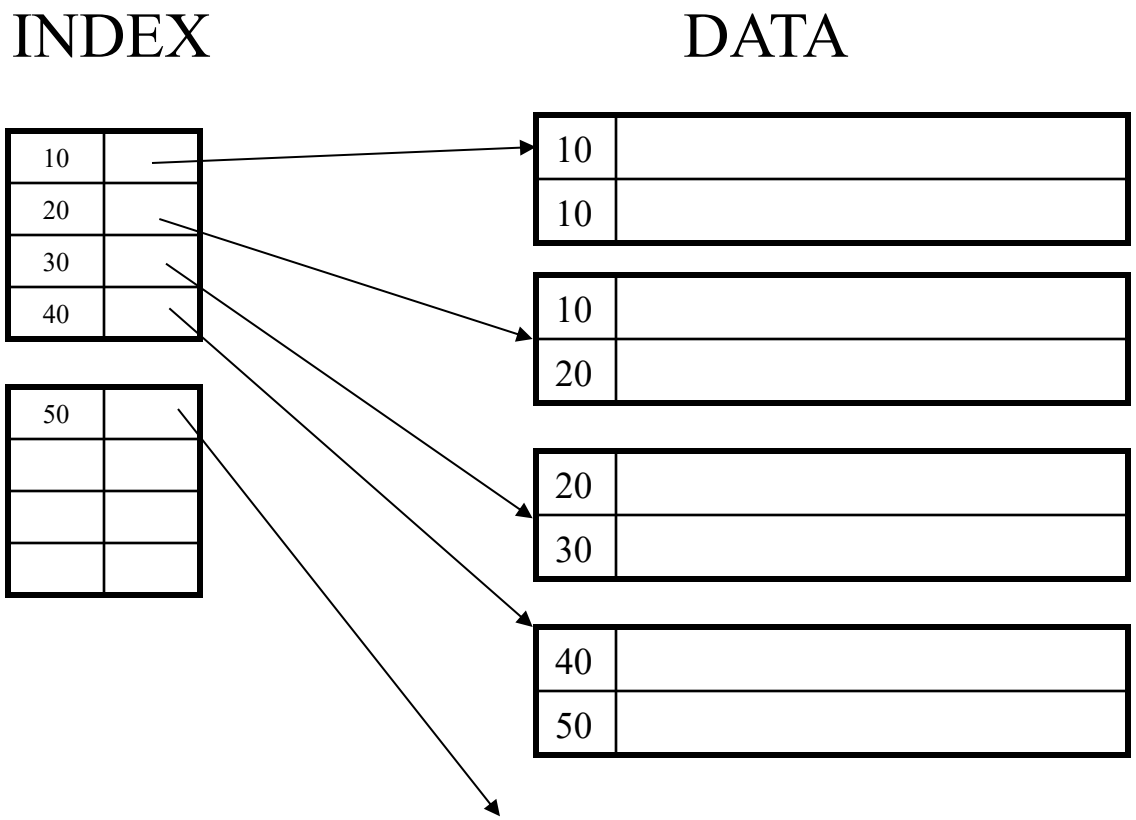
- Clustered and sparse:



  - Any problem?

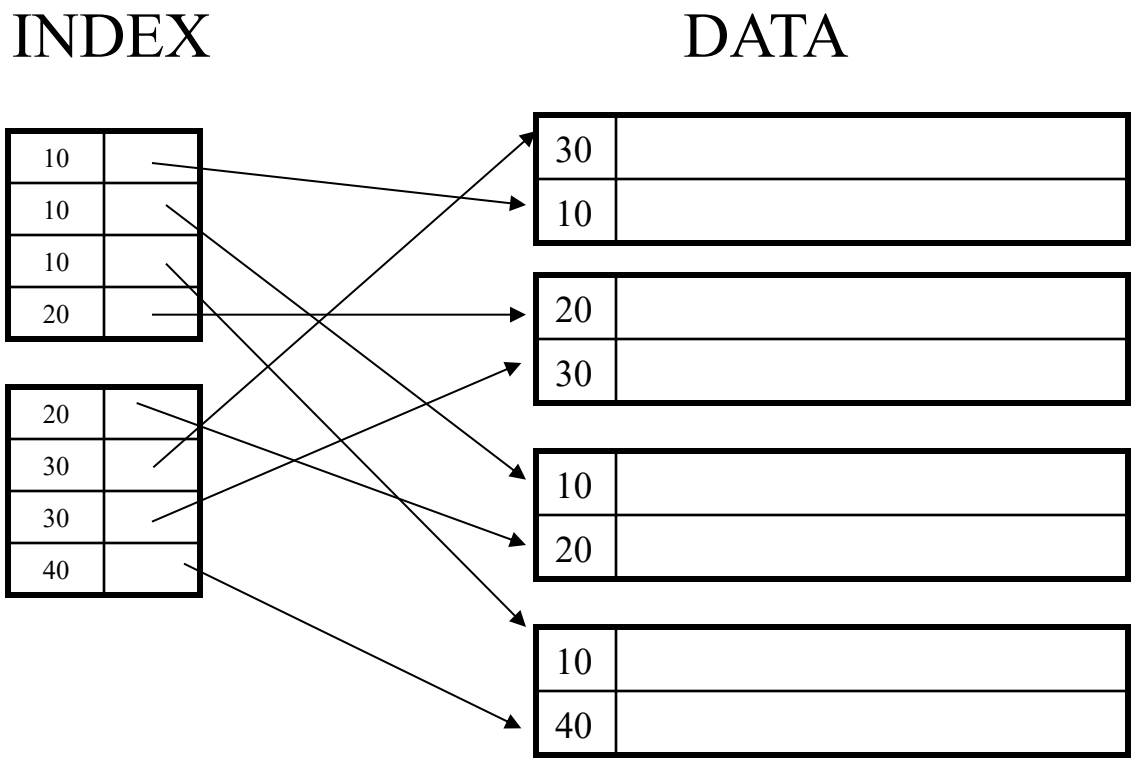
Duplicate search keys

- Clustered and sparse:
  - Point to the lowest new search key in every block



Unclustered Index

- Dense / sparse?



Index structures

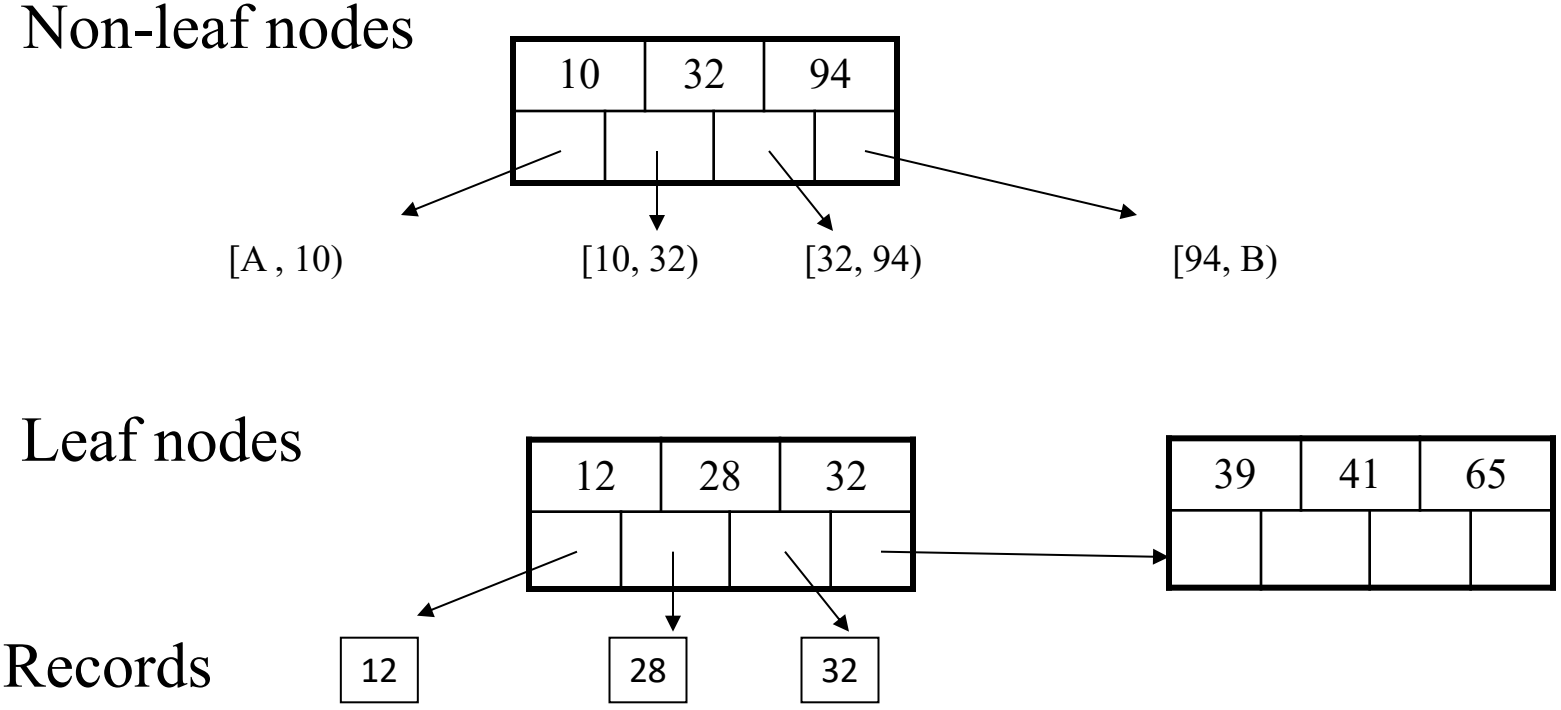
- Tree indexing
  - extends the idea of search trees in main memory
  - popular
  - most frequently used one is B+ tree
- Hash indexing
  - extends the idea of hash tables in main memory
  - less frequently used
- Other structures based on the type of data/ query
  - bitmap indexes, learned indexes, …

B+ trees

- The index of a very large data file gets too large.
- How about building an *index* for the index file?
  - A multi-level index, or a tree
- Insert/delete at $\log_F N$ cost; keep tree *height-balanced.* (F = fanout, N = # leaf pages)
- Supports point (equality) and range queries efficiently.

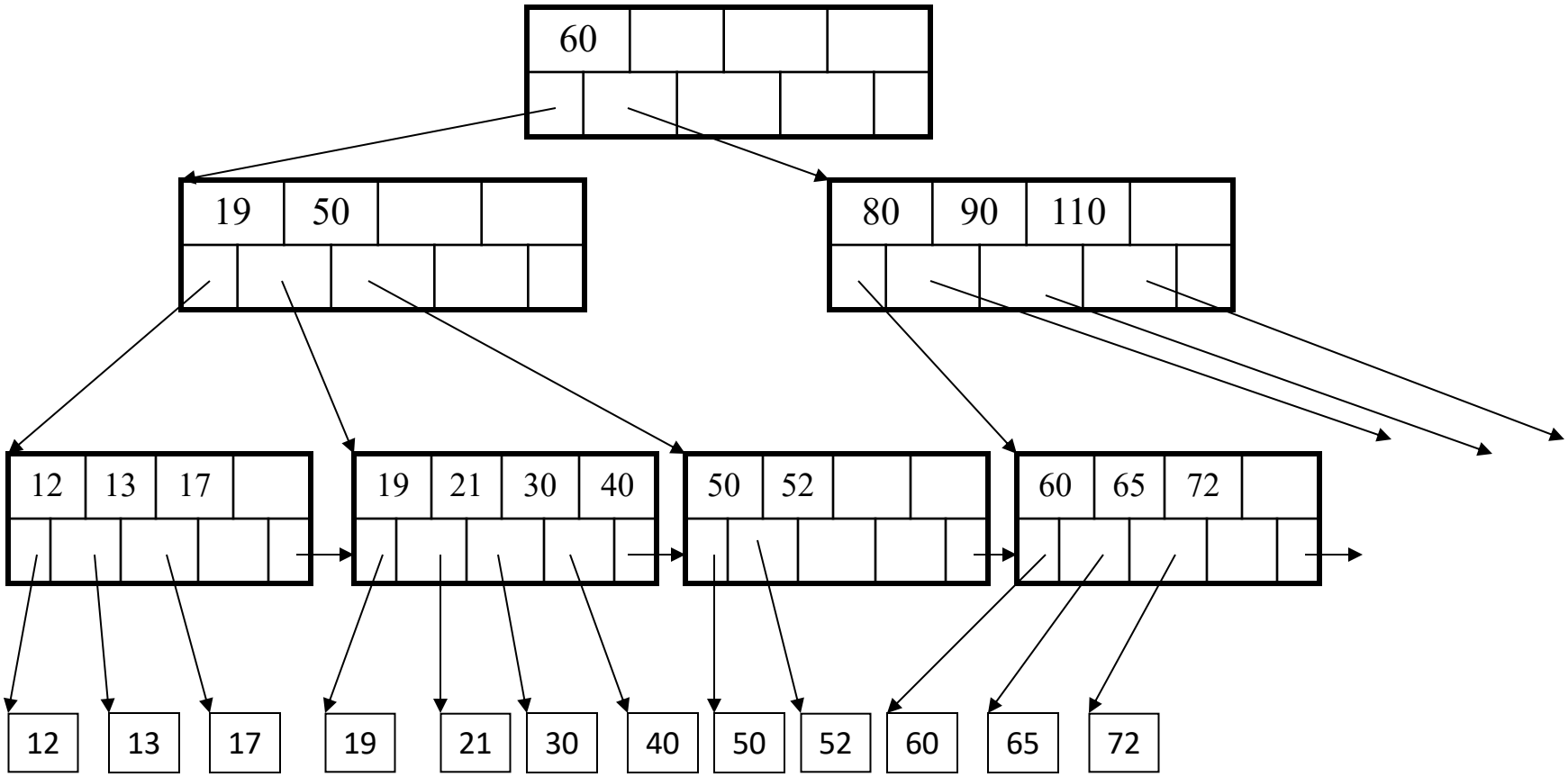
B+ trees

- Degree (order) of the tree: d
- Each node (except root) stores [d, 2d] keys
  - minimum 50% occupancy (except for root).



Example

d = 2


60
19
50
80
90
110
12
13
17
19
21
30
40
50
52
60
65
72
12
13
17
19
21
30
40
50
52
60
65
72

B+ tree tuning

- How to choose the value of d?
  - each node should fit in a block.
- Example
  - key value: 8 byte; record pointer: 16 bytes
  - block size: 4096 bytes
  - 2d * 8 + (2d + 1) * 16 <= 4096; d <= 85

B+ trees in practice

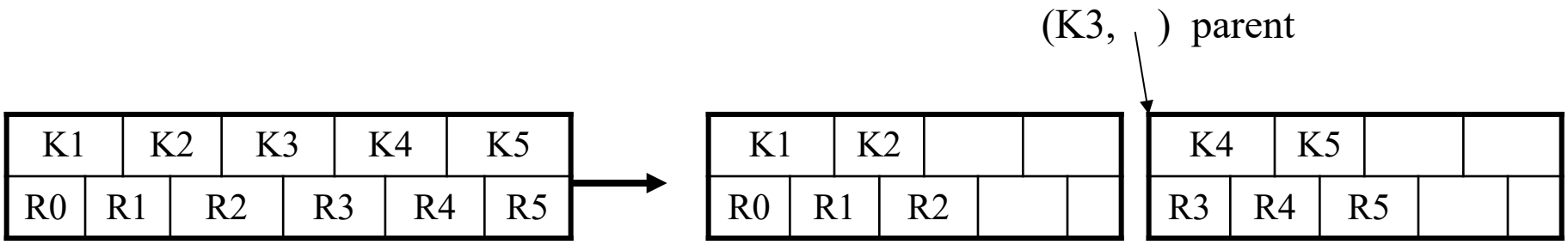
- Typical order: 100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Can often hold top levels in buffer pool:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Retrieving tuples

- Point queries
  - start from the root and follow the links to the leaf.
- Range queries
  - find the lowest point in the range; then, follow the links between the nodes.

Inserting a new key

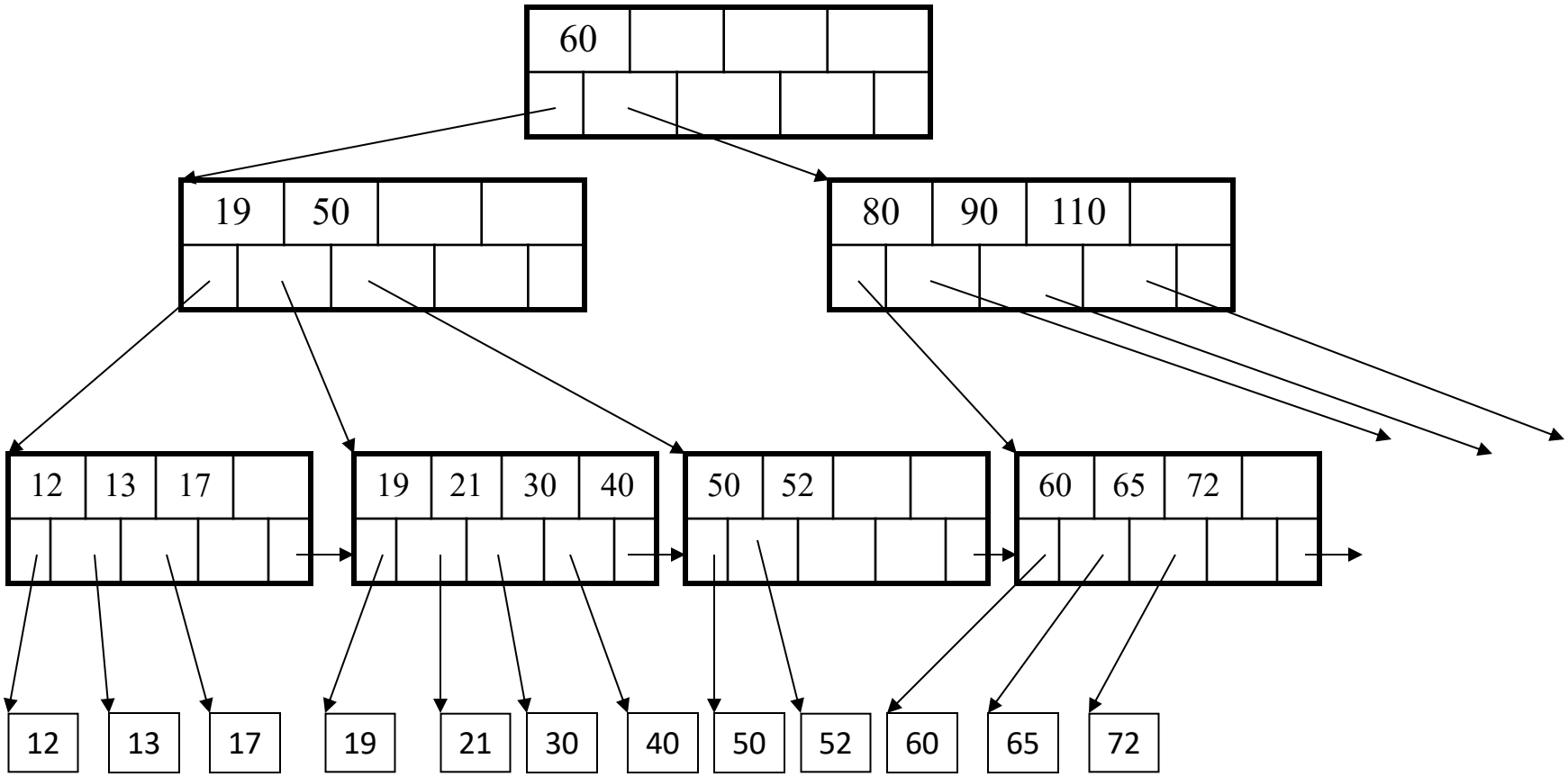
- Pick the proper leaf node and insert the key.
- If the node contains more than 2d keys, split the node and insert the extra node in the parent.



  - If leaf level, add K3 to the right node

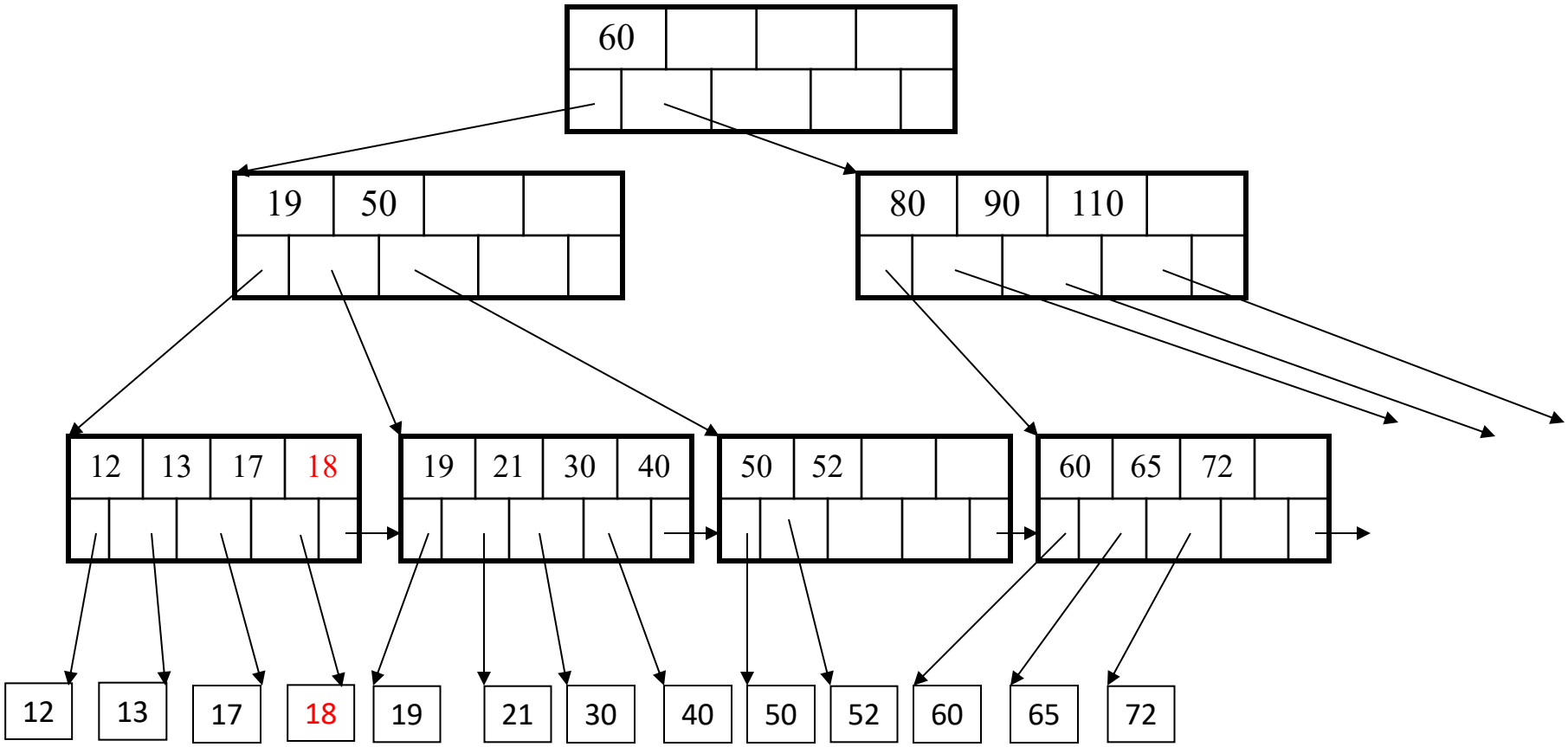
Insertion

Insert K = 18



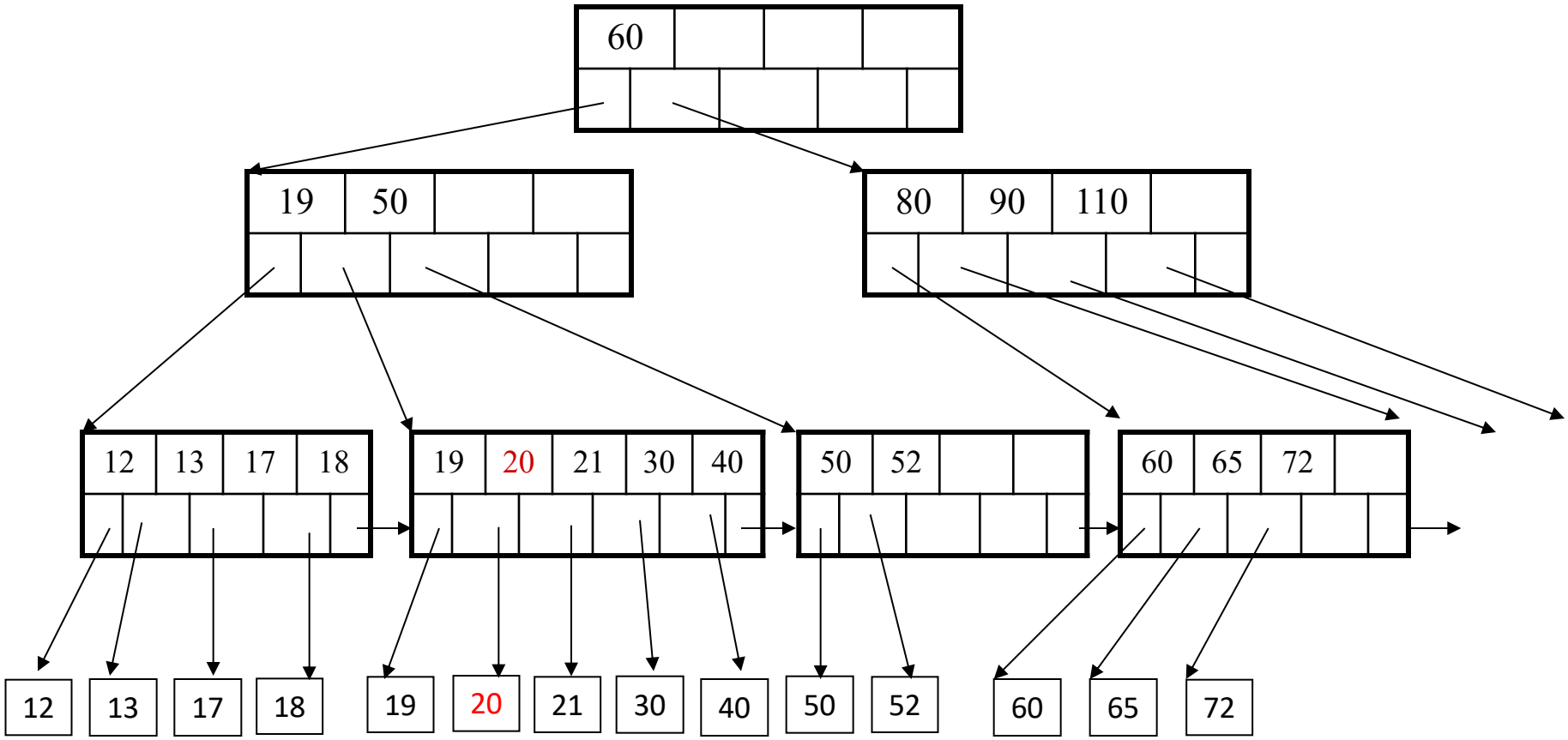
Insertion

Insert K = 18



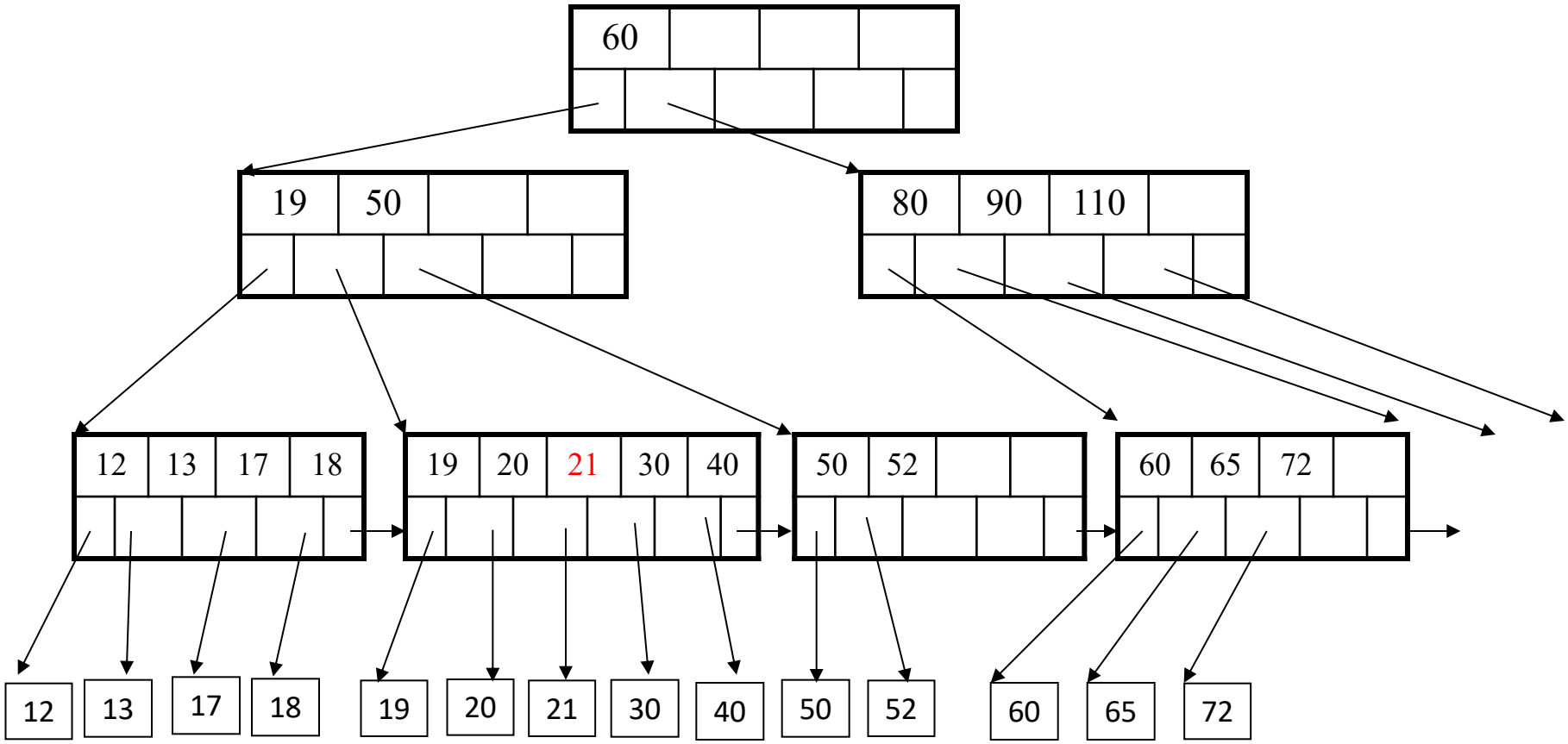
Insertion

Insert K= 20



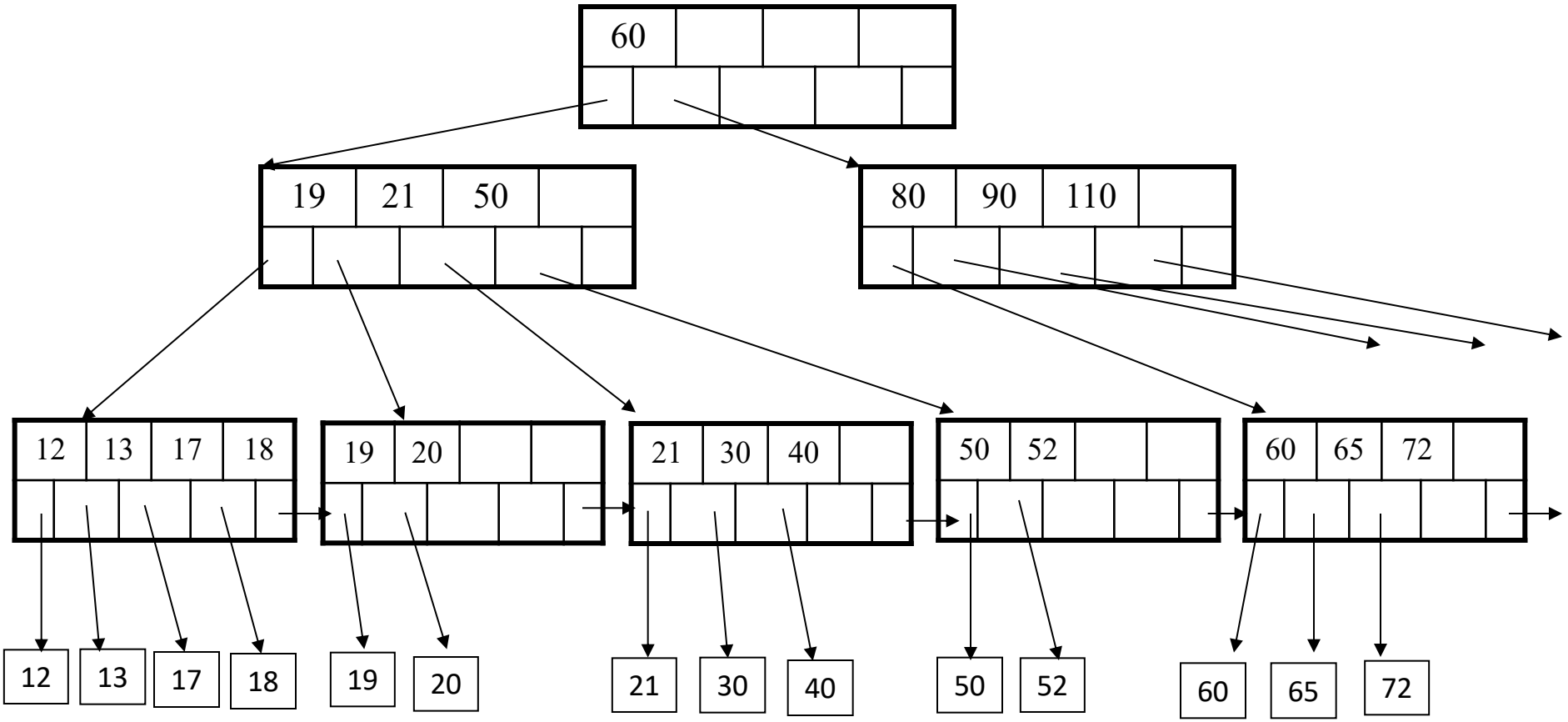
Insertion

Need to split the node



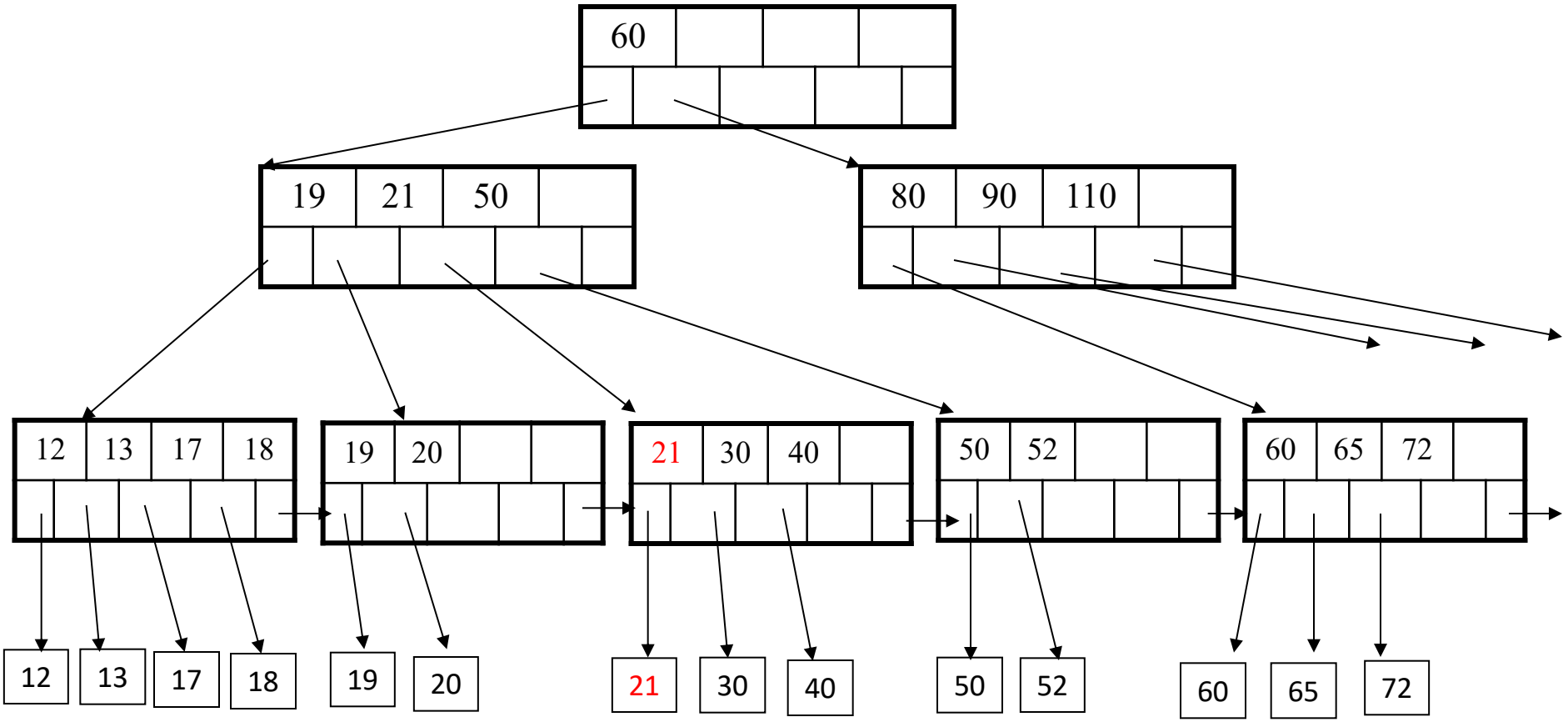
Insertion

Split and update the parent node.
What if we need to split the root?



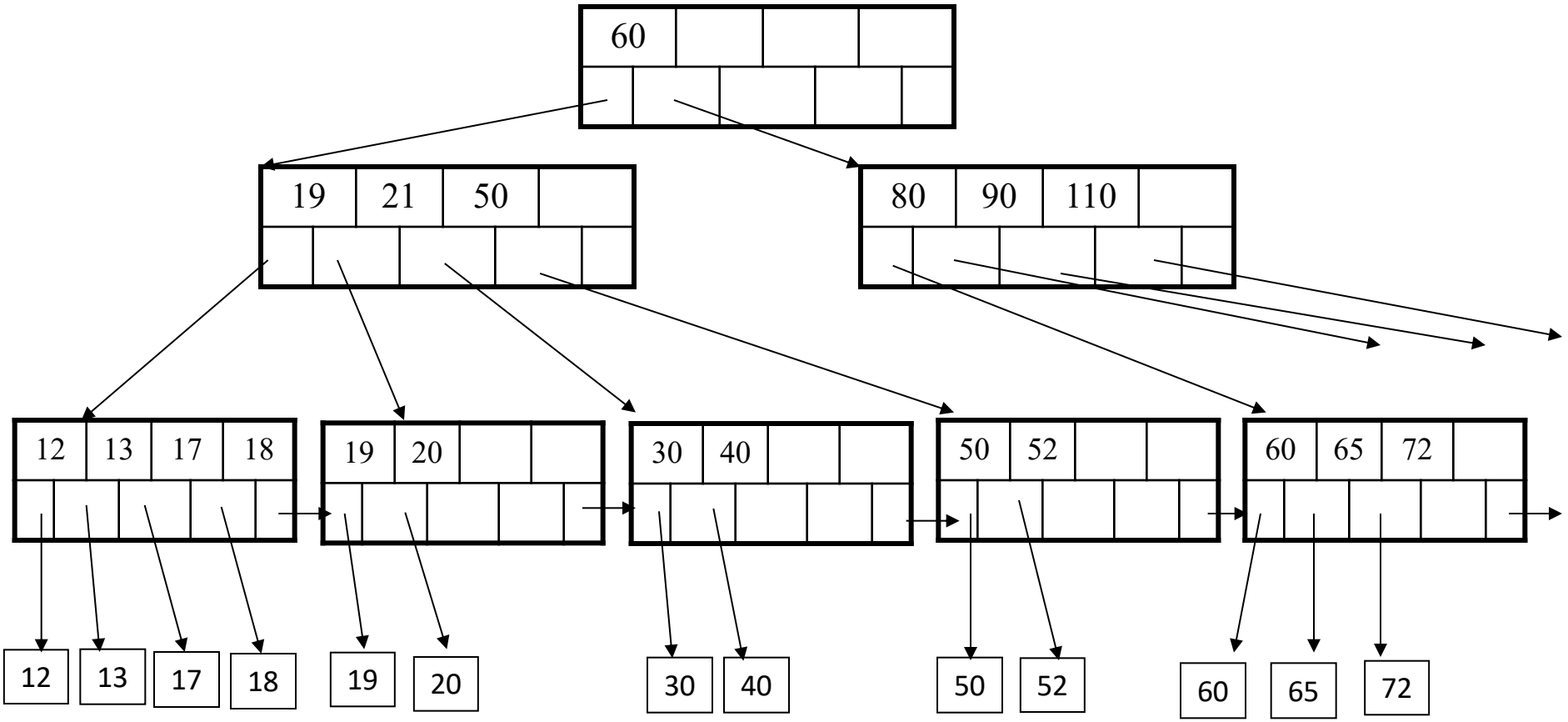
Deletion

Delete K = 21



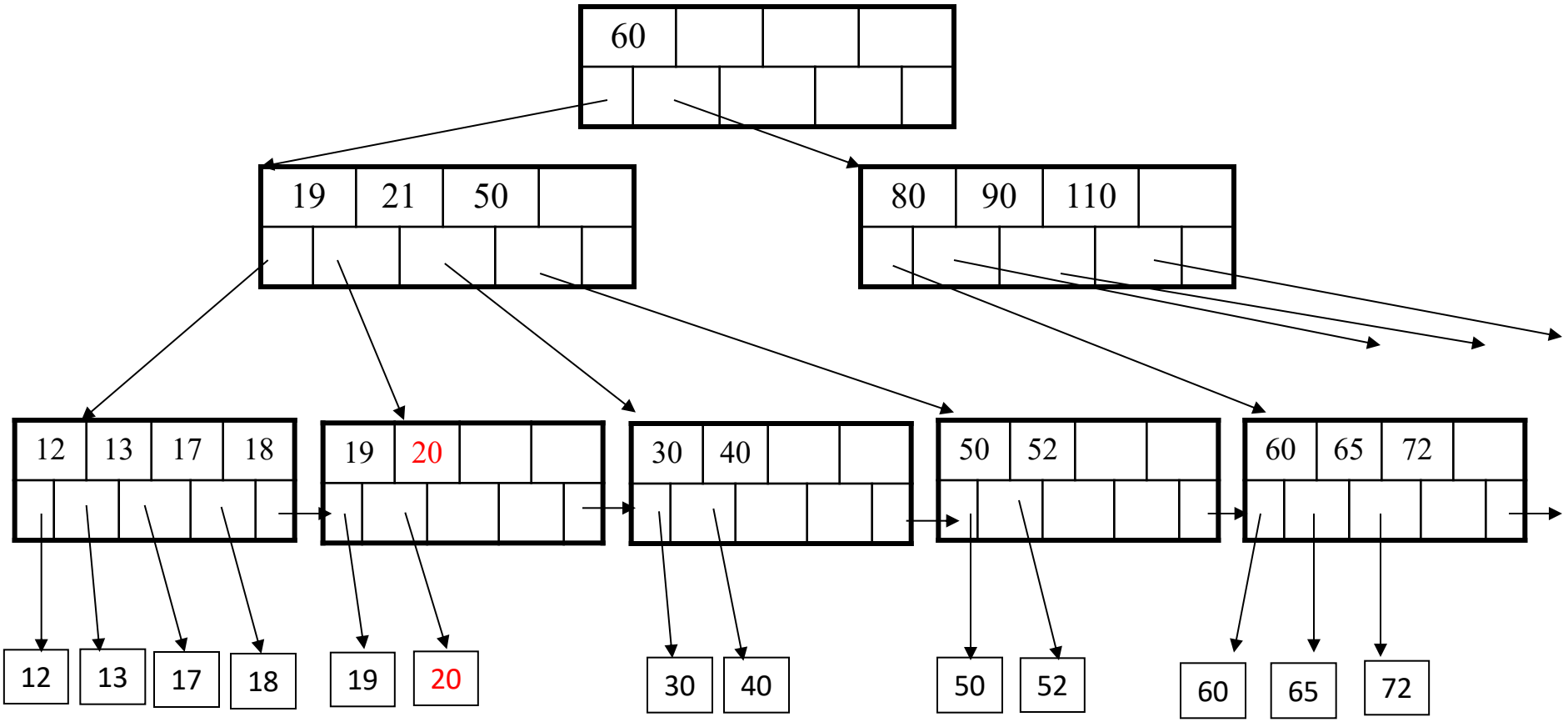
Deletion

Note: K = 21 may still remain in the internal levels



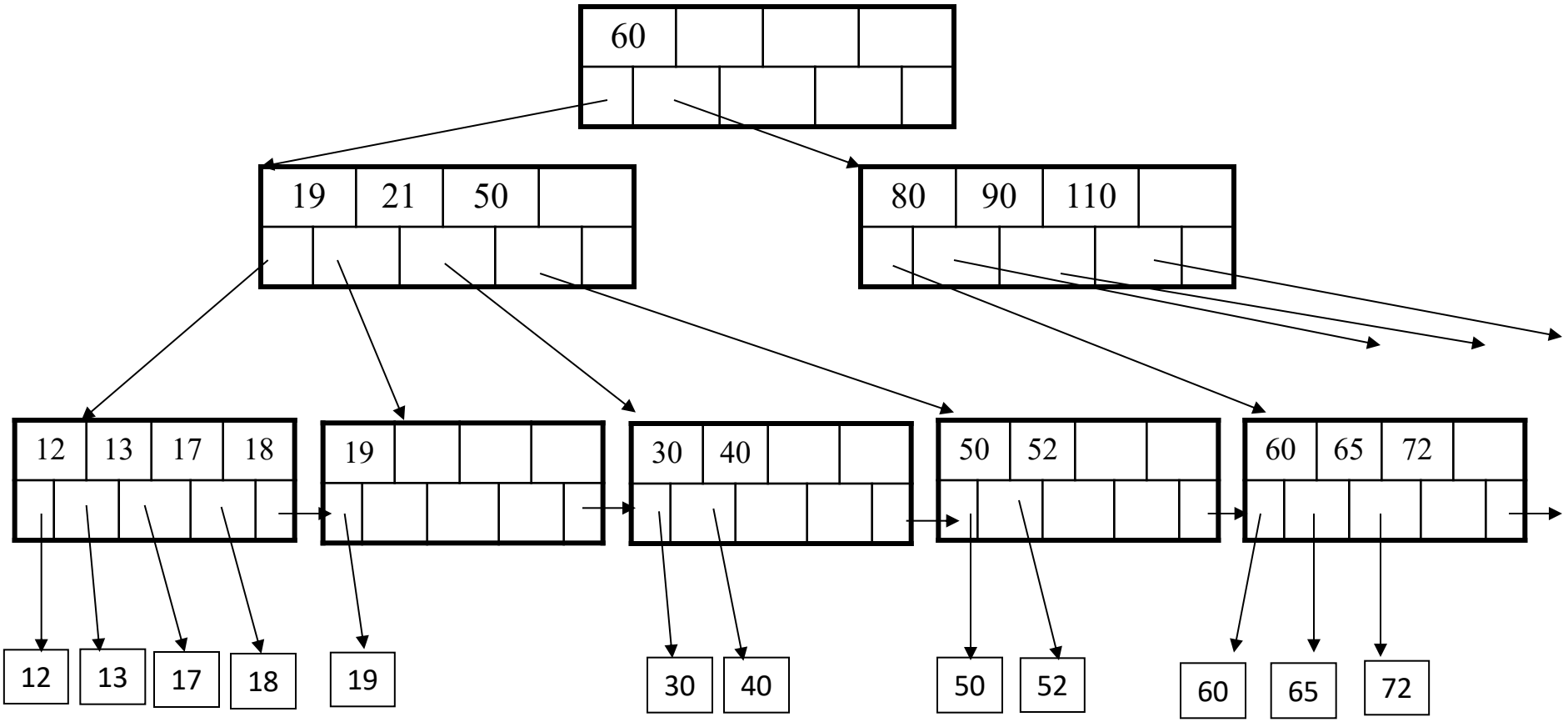
Deletion

Delete K = 20



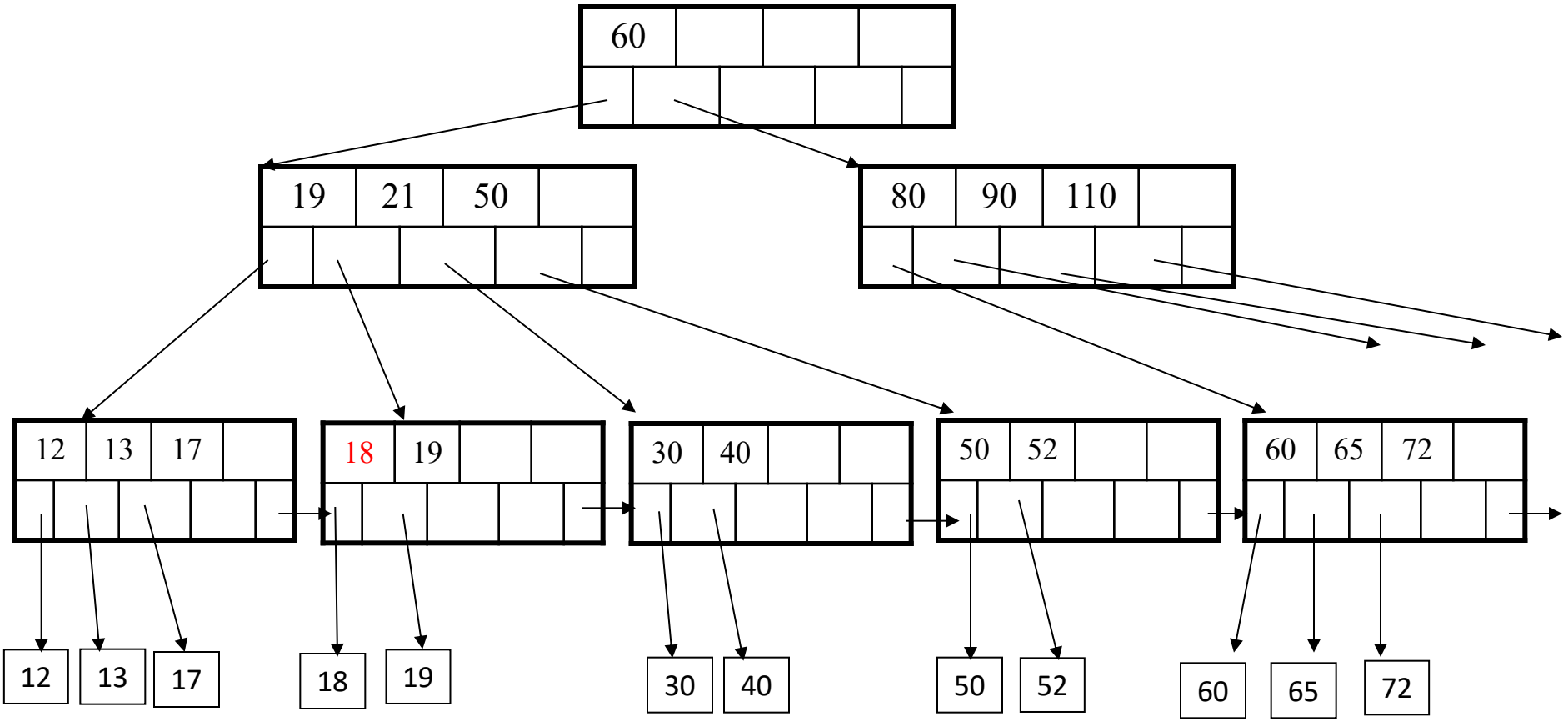
Deletion

We need to update the number of keys on the node:
Borrow from siblings: rotate



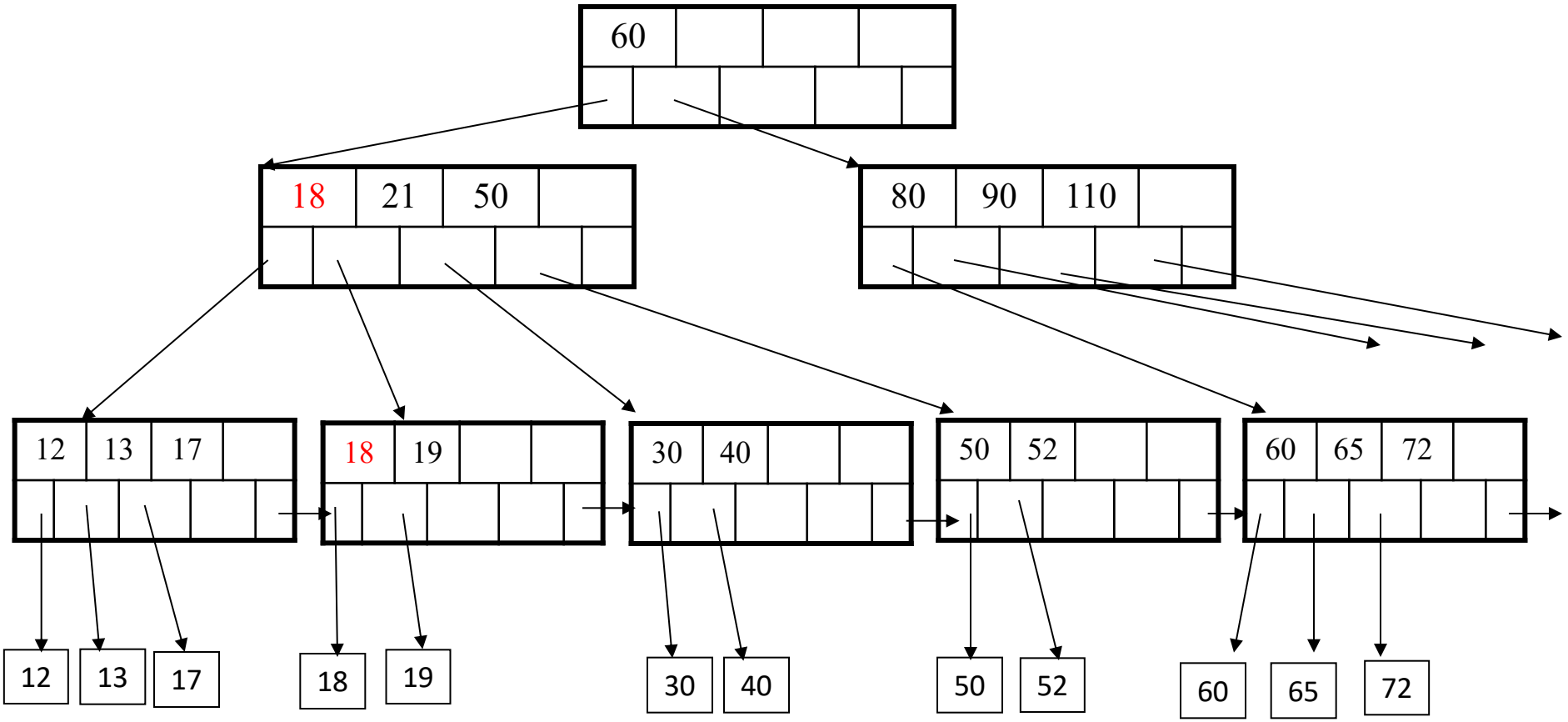
Deletion

We need to update the number of keys on the node:
Borrow from siblings: rotate



Deletion

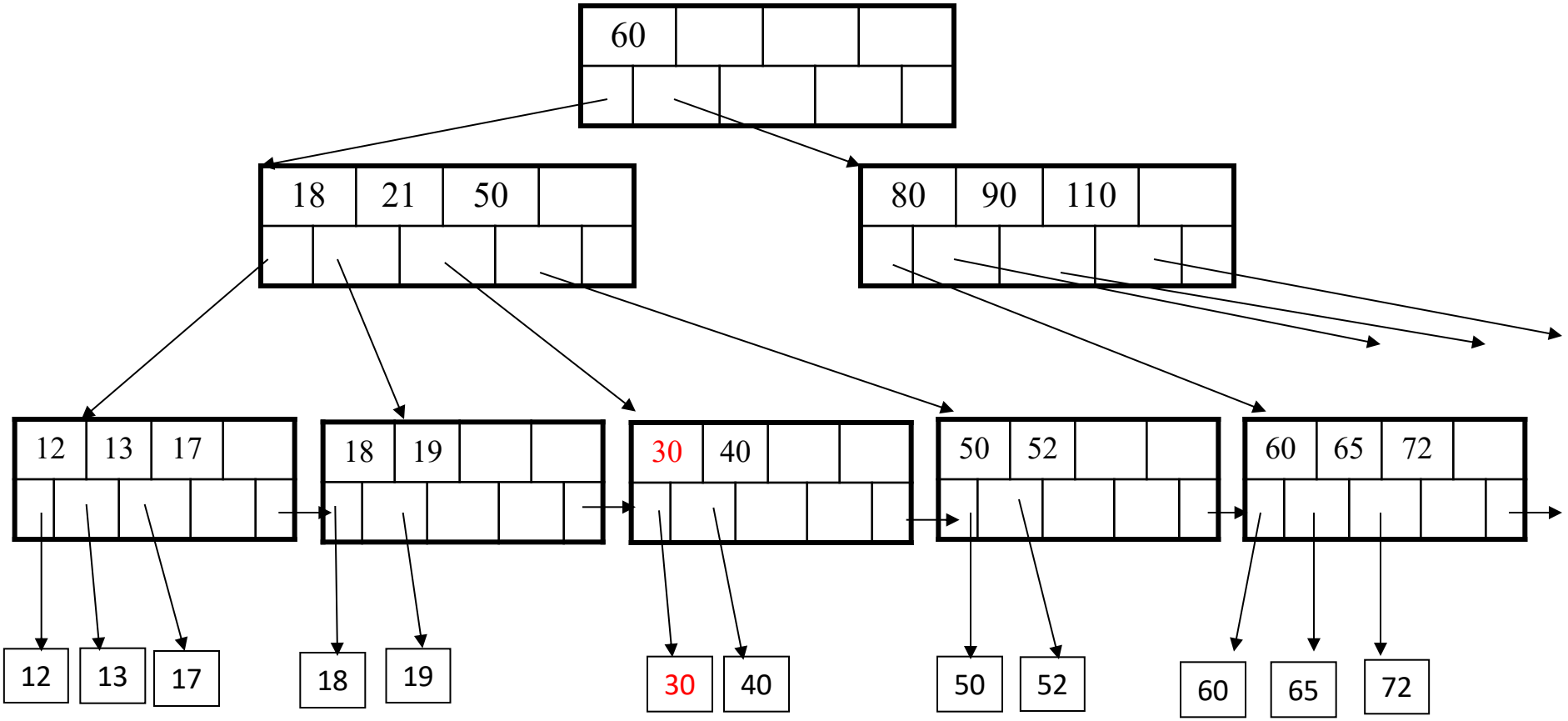
We need to update the number of keys on the node:
Borrow from siblings: rotate



Deletion

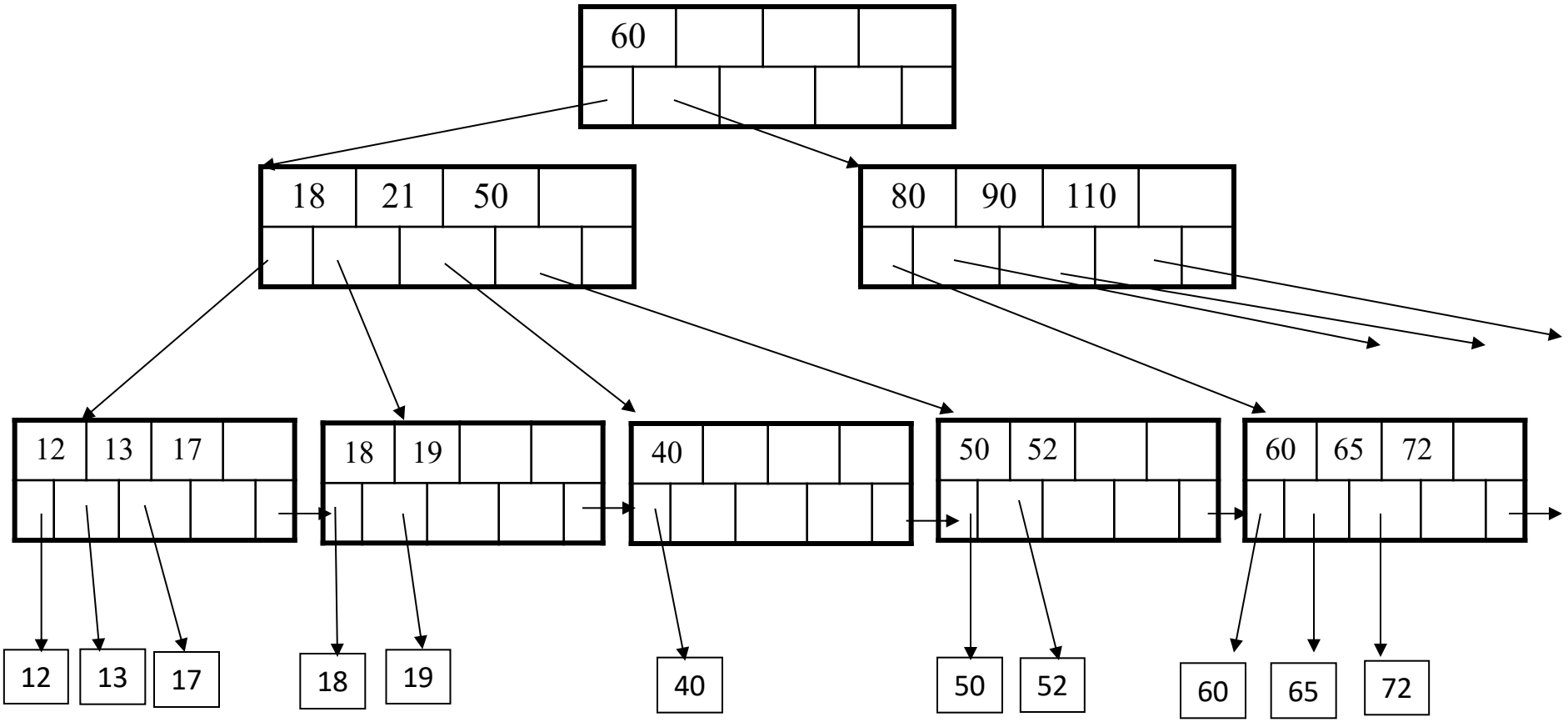
What if we cannot borrow from siblings?

Example: delete K = 30



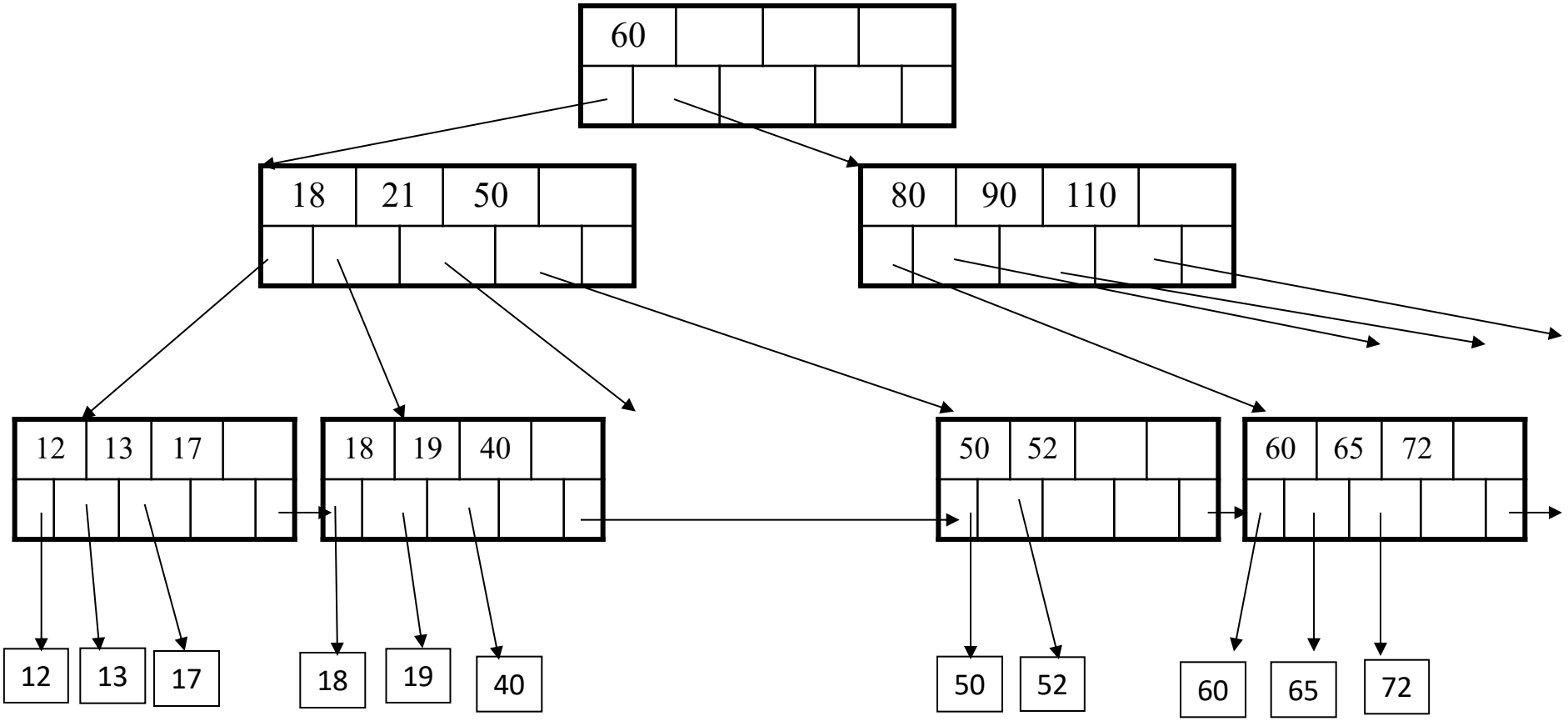
Deletion

What if we cannot borrow from siblings?
Merge with a sibling.



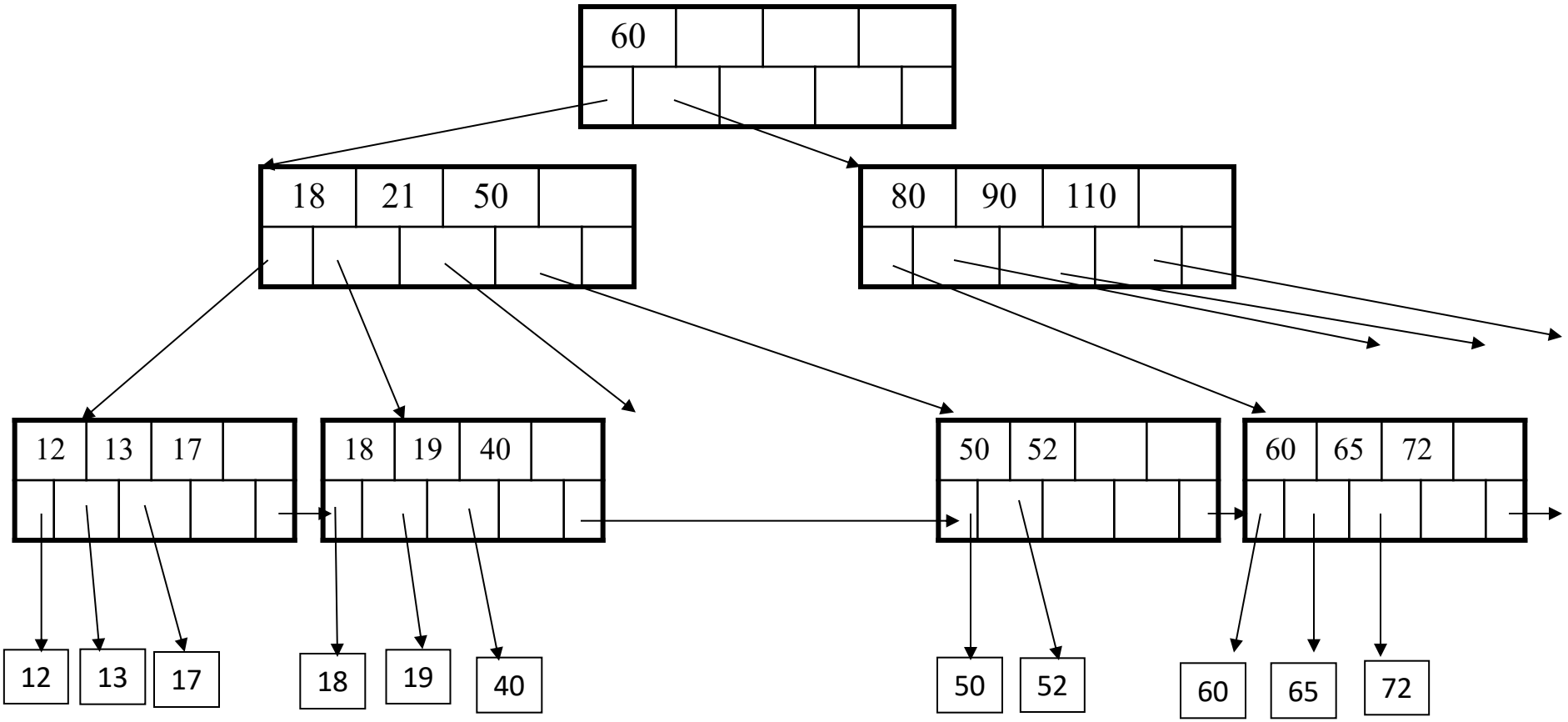
Deletion

What if we cannot borrow from siblings?
Merge siblings!



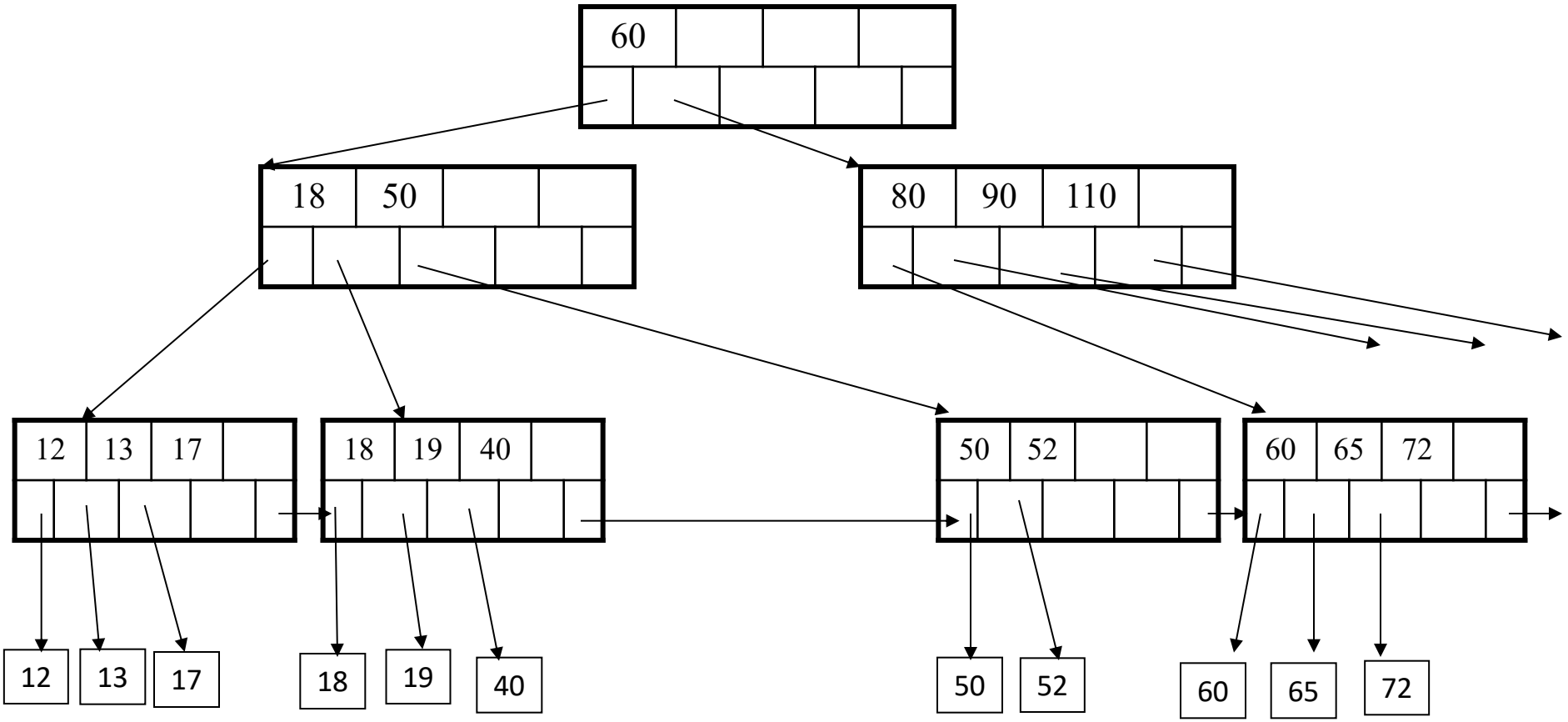
Deletion

What to do with the dangling key and pointer? simply remove them



Deletion

Final tree

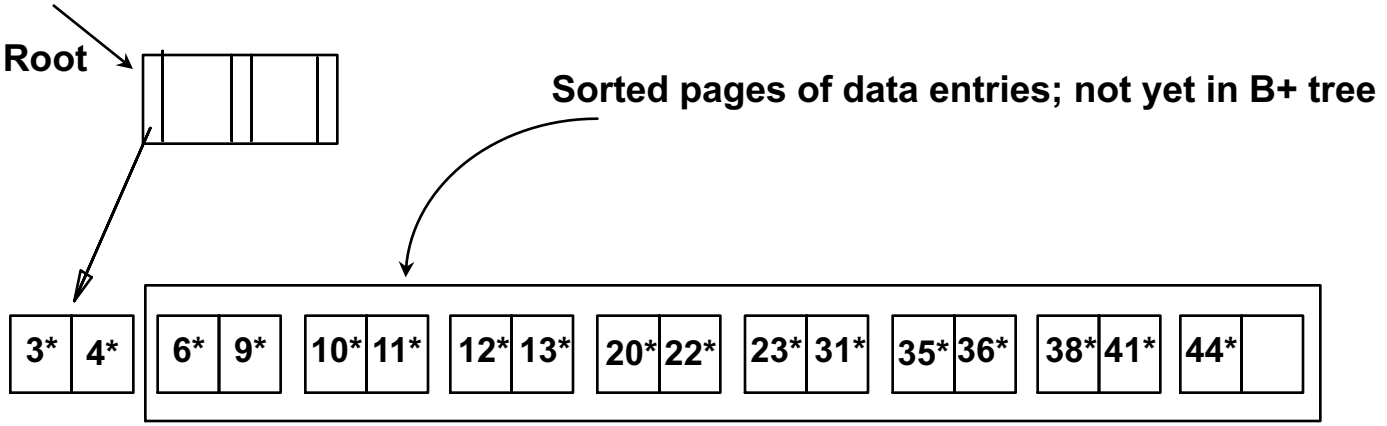


Prefix key compression

- Important to increase fan-out. (Why?)
- Key values in index entries only `direct traffic'; can often compress them.
  - E.g., If we have adjacent index entries with search key values *Dannon Yogurt*, *David Smith* and *Devarakonda Murthy*, we can abbreviate *David Smith* to *Dav*. (The other keys can be compressed too ...)
    - Is this correct? Not quite! What if there is a data entry *Davey Jones*? (Can only compress *David Smith* to *Davi*)
    - In general, while compressing, must leave each index entry greater than every key value (in any subtree) to its left.
- Insert/delete must be suitably modified.

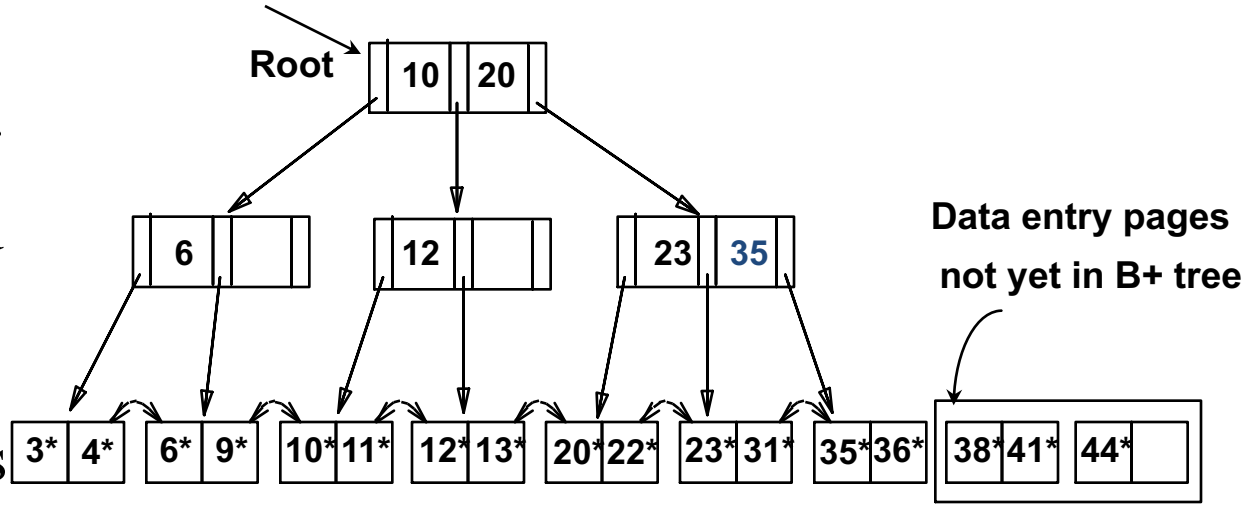
Bulk loading of a B+ tree

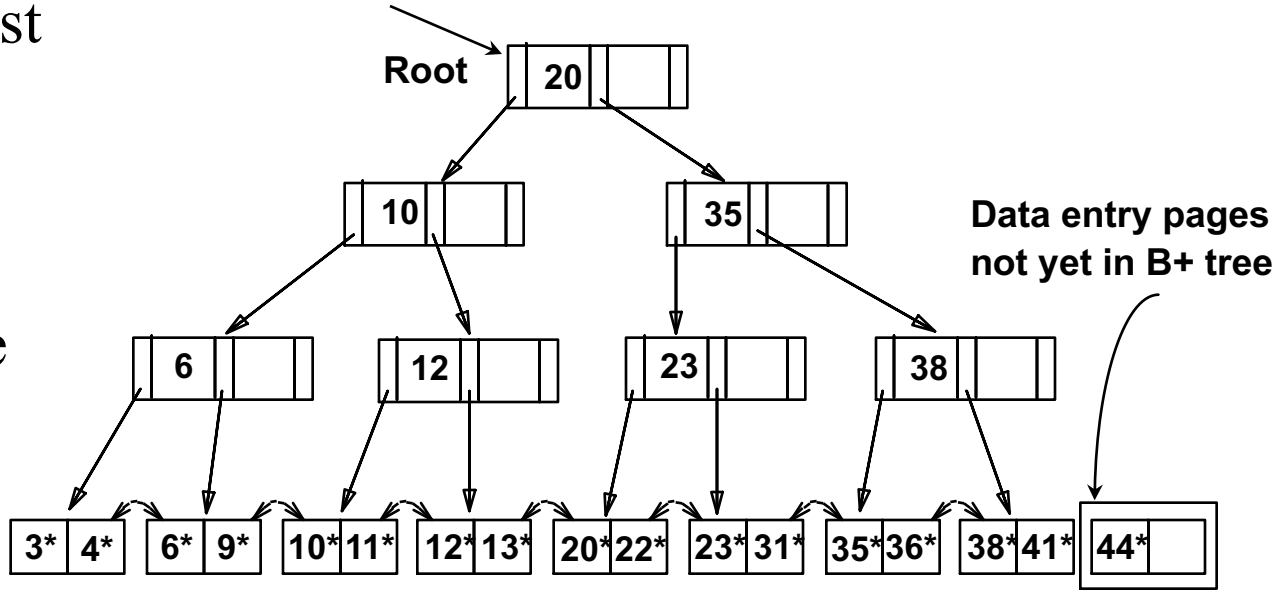
- If we have a large collection of records, and we want to create a B+ tree on some field, doing so by repeatedly inserting records is very slow.
- *Bulk Loading* can be done much more efficiently.
- *Initialization*: Sort all data entries, insert pointer to first (leaf) page in a new (root) page.



Bulk loading

- 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 of bulk loading

- Option 1: multiple inserts.
  - slow.
  - does not give sequential storage of leaves.
- Option 2: *Bulk Loading*
  - fewer I/Os during build.
  - leaves will be stored sequentially (and linked, of course).
  - can control "fill factor" on pages.