CSE 544: Lecture 10

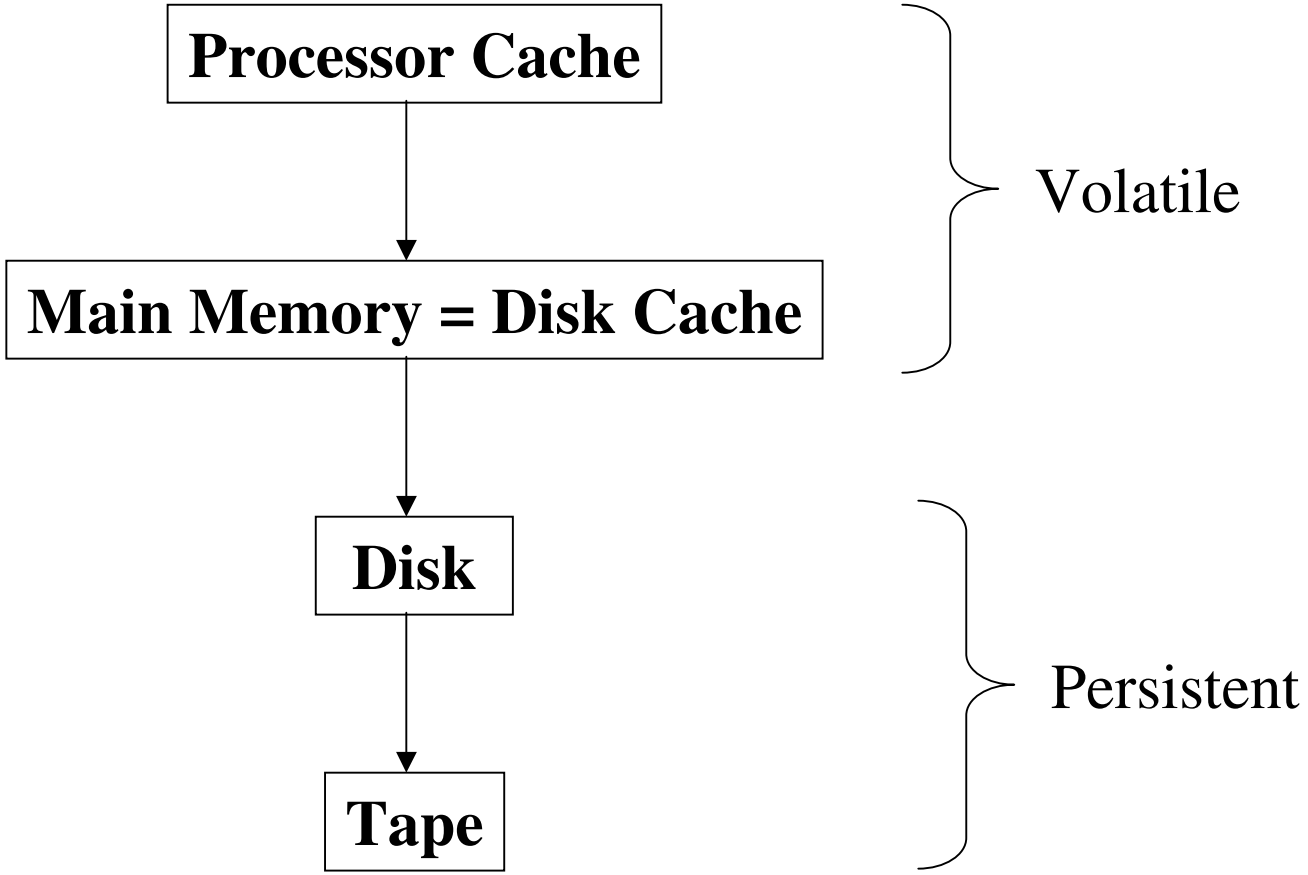
Storing Data, Indexes

Wednesday, 5/1/2002

Outline

- Storing Data:
  - Disks (7.1-7.3)
  - Buffer manager (7.4)
  - Page formats (7.6)
  - Record formats (7.7)
- Indexes:
  - Types of indexes (8.3, 8.4)
  - B+ trees (Chapter 9)

The Memory Hierarchy



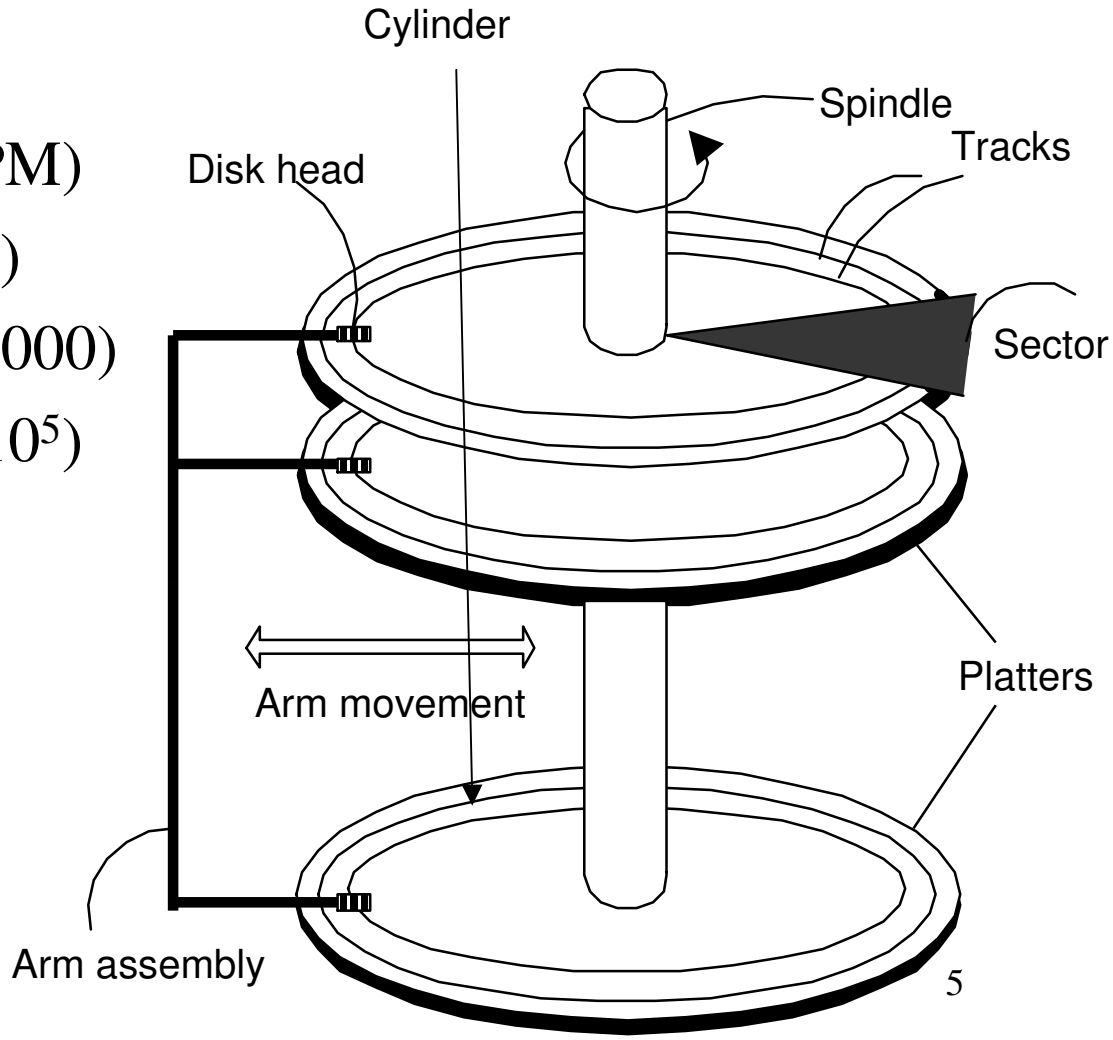
Disk

- The unit of disk I/O = ***block***
  - Typically 1 block = 4k
- Used with a main memory buffer

The Mechanics of Disk

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platers (1-30)
- Number of tracks (<=10000)
- Number of bytes/track(10^5)



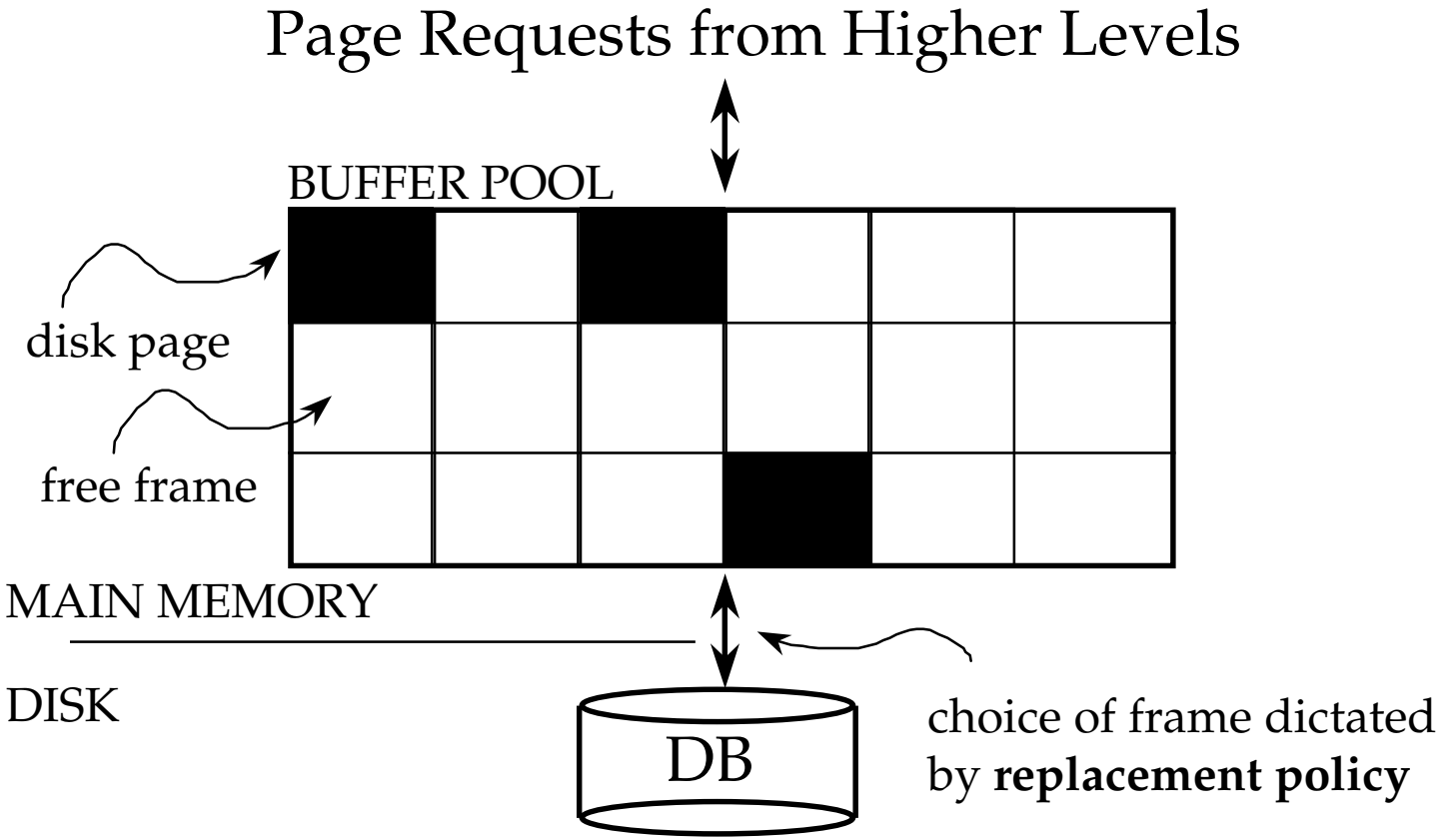
Important Disk Access Characteristics

- Disk latency = seek time + rotational latency
- Seek time = time for the head to reach cylinder
  - 10ms – 40ms
- Rotational latency = time for the sector to rotate
  - Rotation time = 10ms
  - Average latency = 10ms/2
- Transfer time = typically 5-10MB/s
- Disks read/write one block at a time (typically 4kB)

RAIDs

- = "Redundant Array of Independent Disks"
  - Was "inexpensive" disks
- Idea: use more disks, increase reliability
- Recall:
  - Database *recovery* helps after a systems crash, not after a disk crash
- 6 ways to use RAIDs. More important:
  - Level 4: use N-1 data disks, plus one parity disk
  - Level 5: same, but alternate which disk is the parity
  - Level 6: use Hamming codes instead of parity

Buffer Management in a DBMS



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

Buffer Manager

- When a page is first requested, it is read in a free frame
- The DBMS typically requests it to be *pinned*:
  - pin_count = how many processes requested it pinned
- When the DBMS writes to it, the buffer manager marks it *dirty*
- When the DBMS doesn't need it any more, *un-pinned*:
  - pin_count is decremented

- Typical replacement policy (always chooses among *un-pinned* frames):
  - LRU
  - Clock
  - MRU

Buffer Manager

Why not use the Operating System for the task??

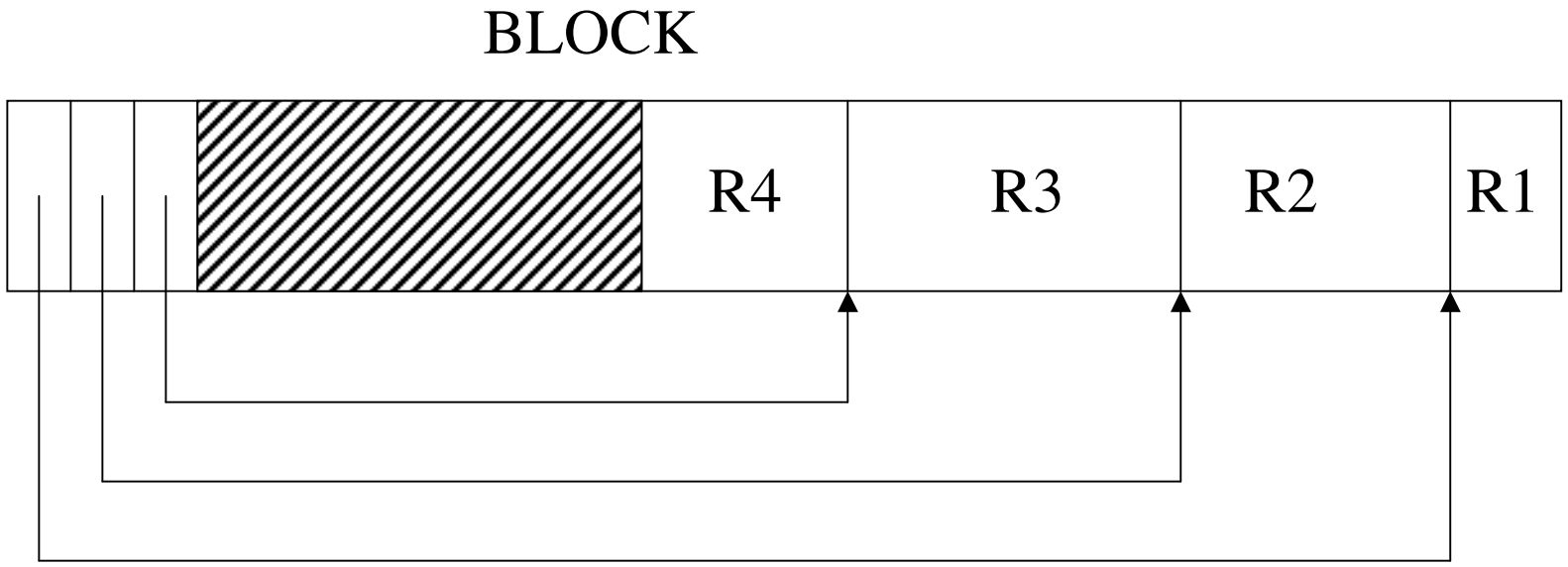
- DBMS may be able to anticipate access patterns
- Hence, may also be able to perform prefetching
- DBMS needs the ability to force pages to disk.

Managing Free Blocks

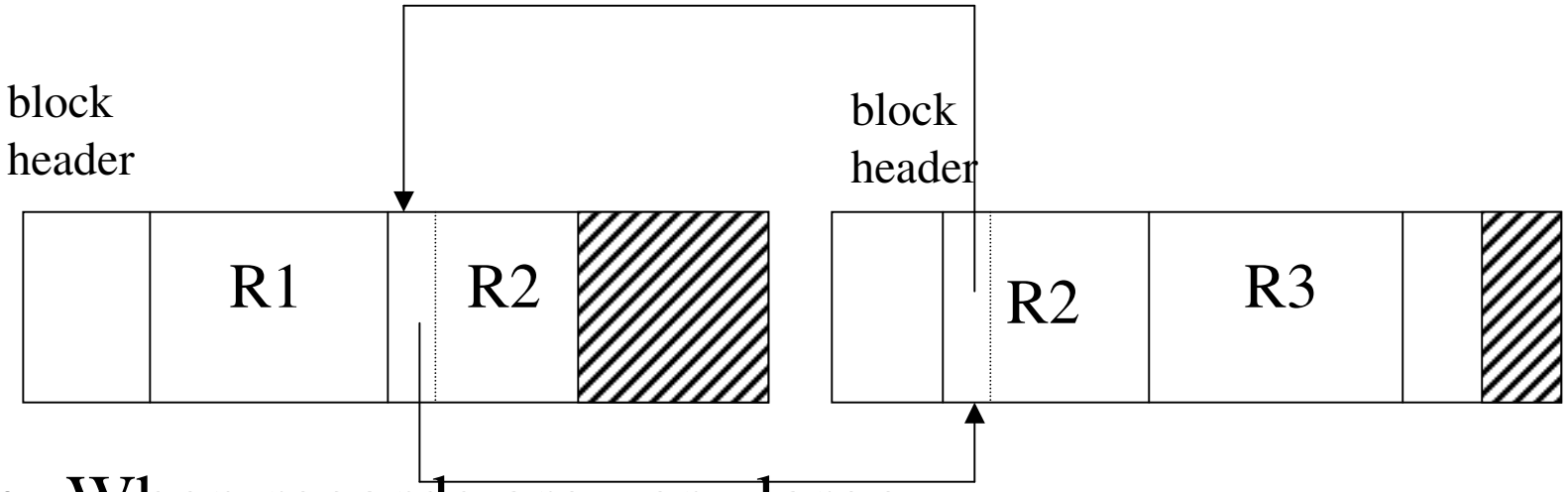
- By the OS
- By the RDBMS (typical)
  - Linked list of free blocks
  - Bit map

Storing Records in Blocks (Page Format)

- Blocks have fixed size (typically 4k)



Spanning Records Across Blocks



- When records are very large
- Or even medium size: saves space in blocks
- Commercial RDBMS avoid this

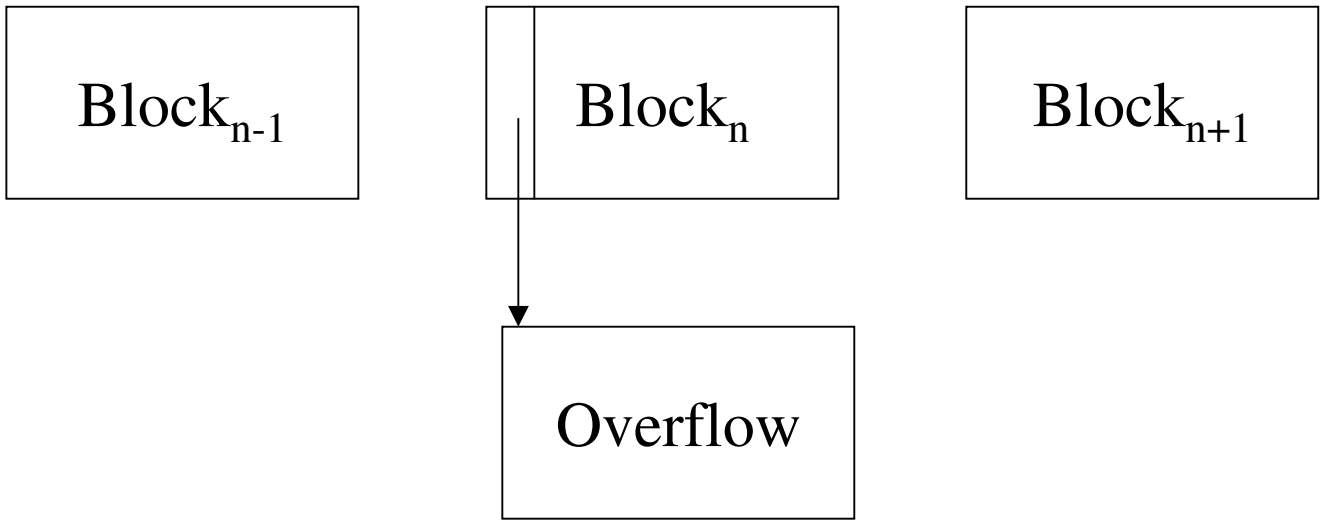
BLOB

- Binary large objects
- Supported by modern database systems
- E.g. images, sounds, etc.
- Storage: attempt to cluster blocks together

Modifications: Insertion

- File is unsorted (= ***heap file***)
  - add it to the end (easy ☺)
- File is sorted:
  - Is there space in the right block ?
    - Yes: we are lucky, store it there
  - Is there space in a neighboring block ?
    - Look 1-2 blocks to the left/right, shift records
  - If anything else fails, create *overflow block*

Overflow Blocks



- After a while the file starts being dominated by overflow blocks: time to reorganize

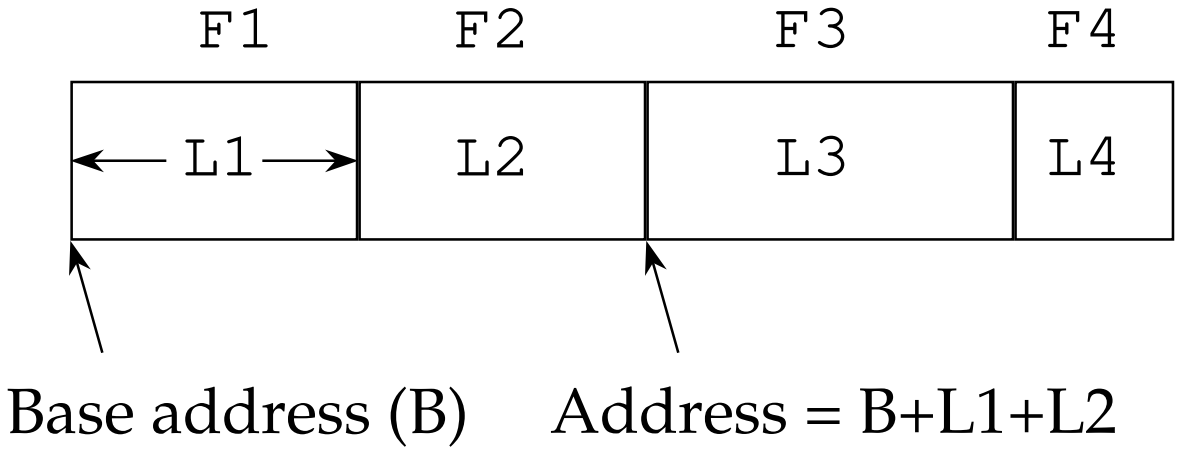
Modifications: Deletions

- Free space in block, shift records
- Maybe be able to eliminate an overflow block
- Can never really eliminate the record, because others may point to it
  - Place a tombstone instead (a NULL record)

Modifications: Updates

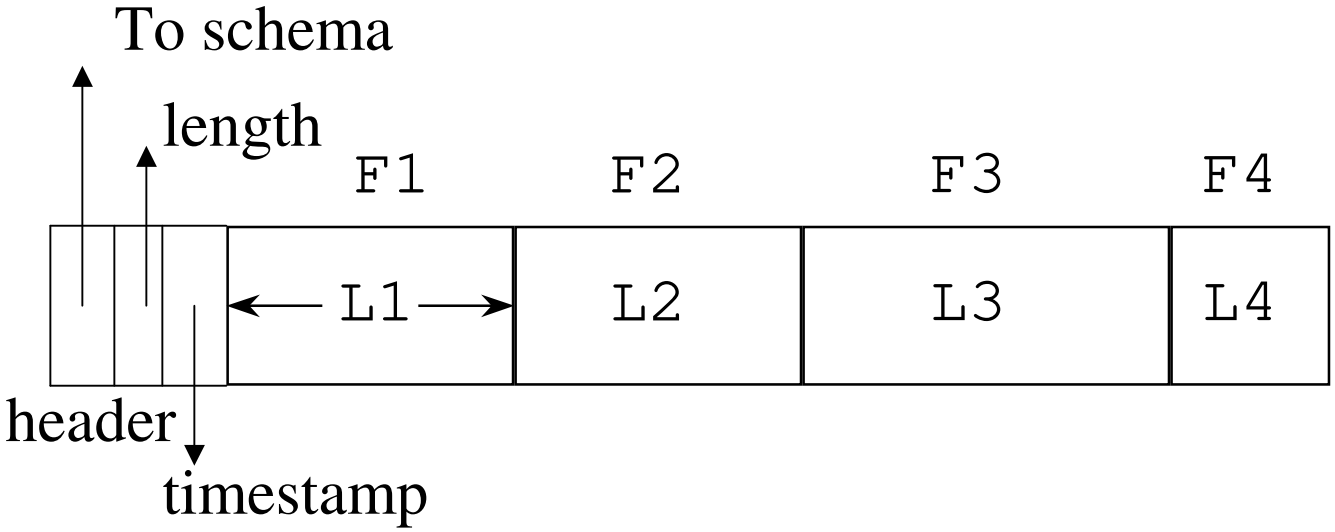
- If new record is shorter than previous, easy ☺
- If it is longer, need to shift records, create overflow blocks

Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.
- **Note the importance of schema information!**

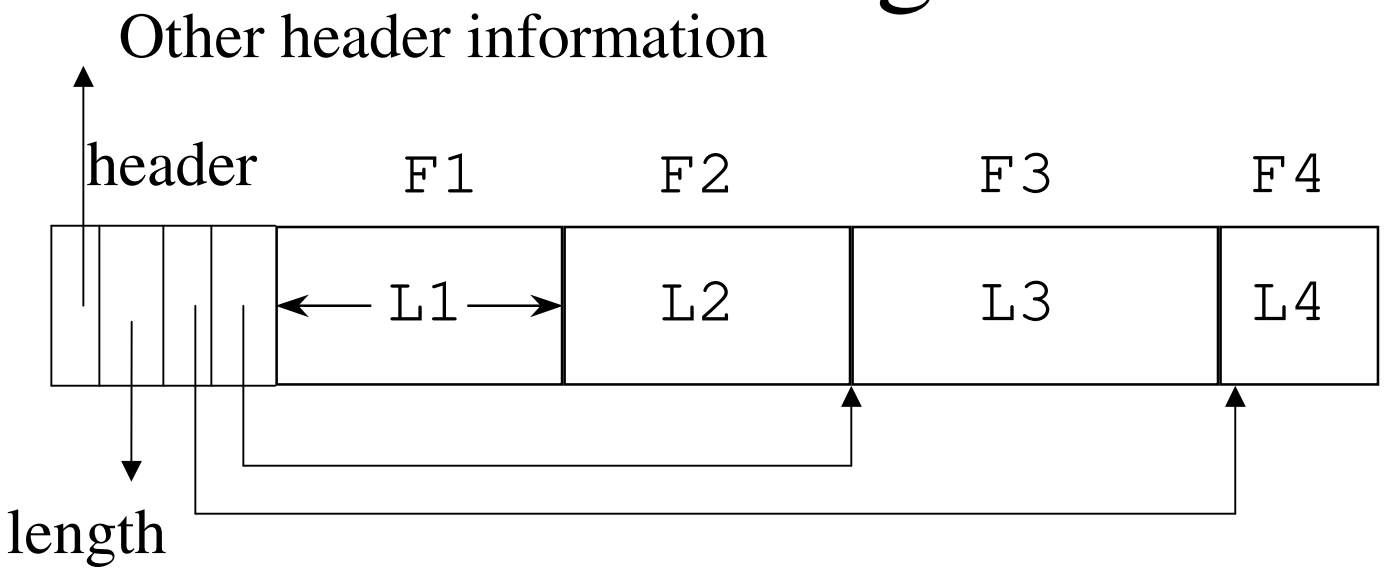
Record Header



Need the header because:

- The schema may change
  for a while new+old may coexist
- Records from different relations may coexist

Variable Length Records



Place the fixed fields first: F1, F2
Then the variable length fields: F3, F4
Null values take 2 bytes only
Sometimes they take 0 bytes (when at the end)

Indexes

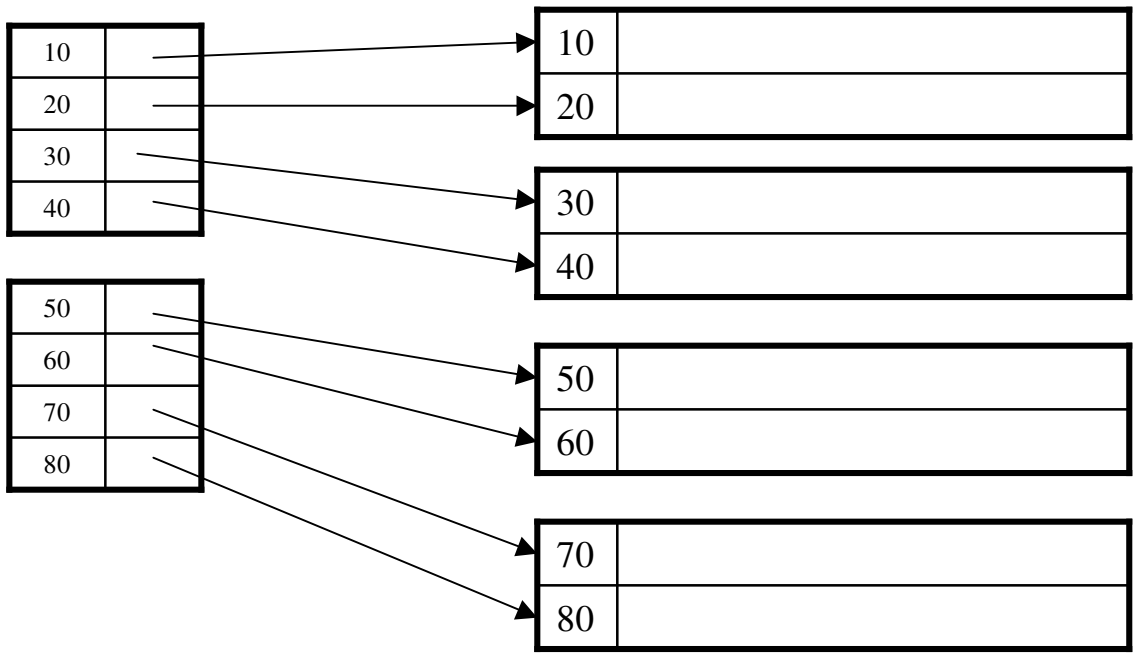
- An *index* on a file speeds up selections on the *search key field(s)*
- Search key = any subset of the fields of a relation
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- Entries in an index: (k, r), where:
  - k = the key
  - r = the record OR record id OR record ids

Index Classification

- Clustered/unclustered
  - Clustered = records sorted in the key order
  - Unclustered = no
- Dense/sparse
  - Dense = each record has an entry in the index
  - Sparse = only some records have
- Primary/secondary
  - Primary = on the primary key
  - Secondary = on any key
- B+ tree / Hash table / …

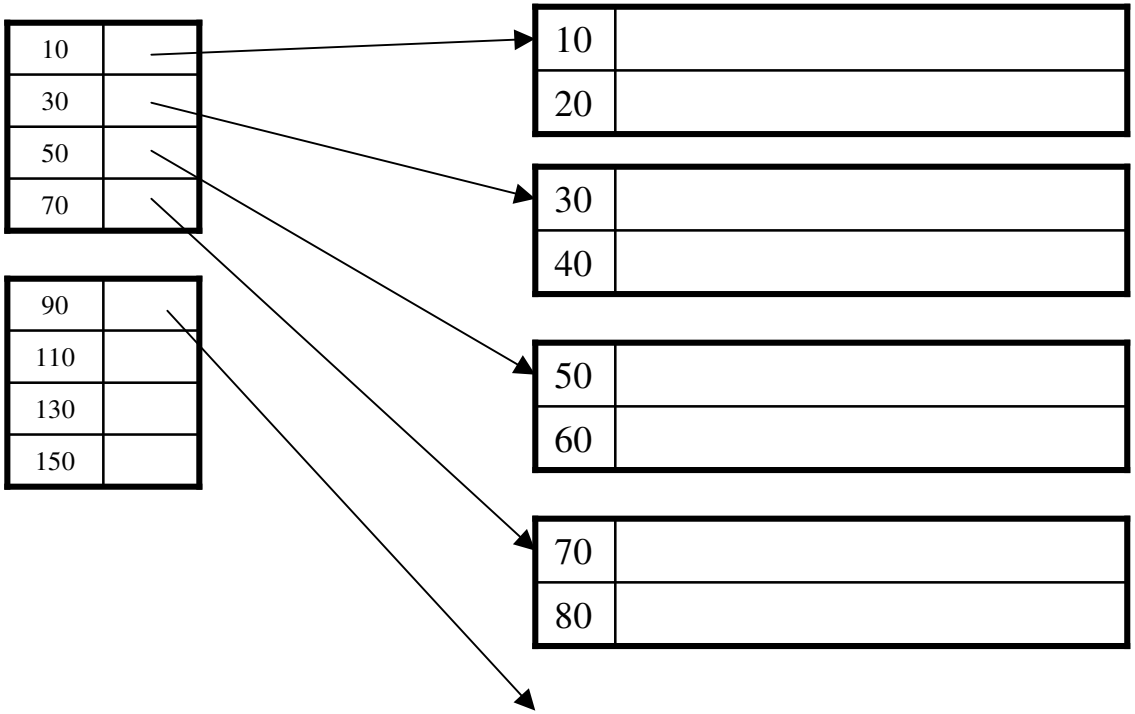
Clustered Index

- File is sorted on the index attribute
- *Dense* index: sequence of (key,pointer) pairs



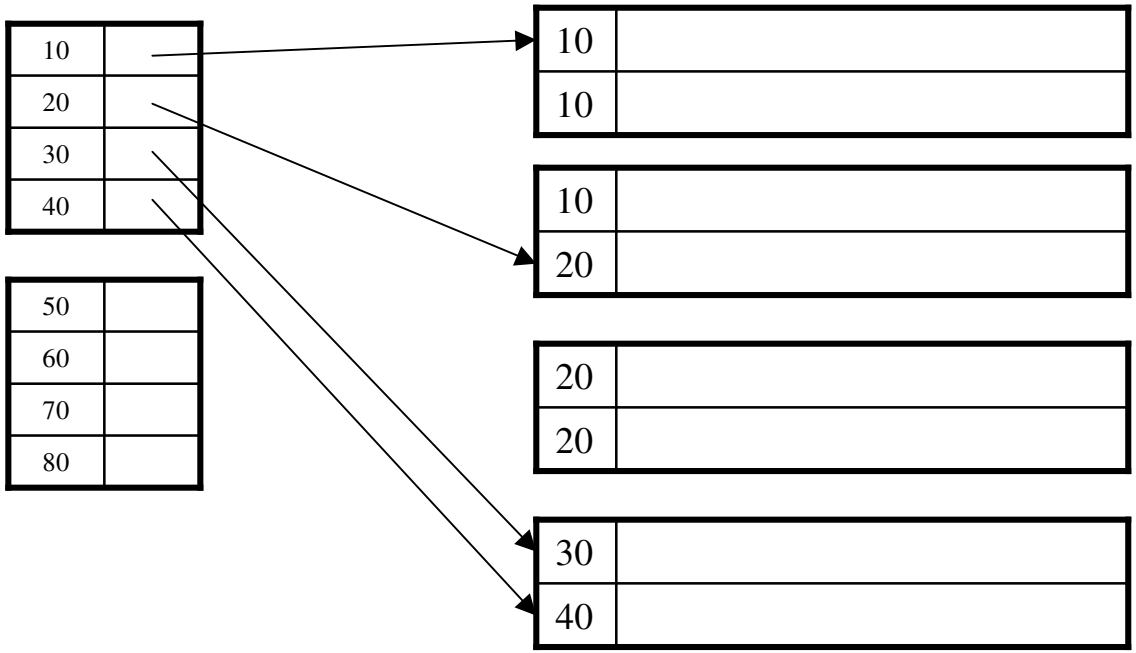
Clustered Index

- *Sparse* index



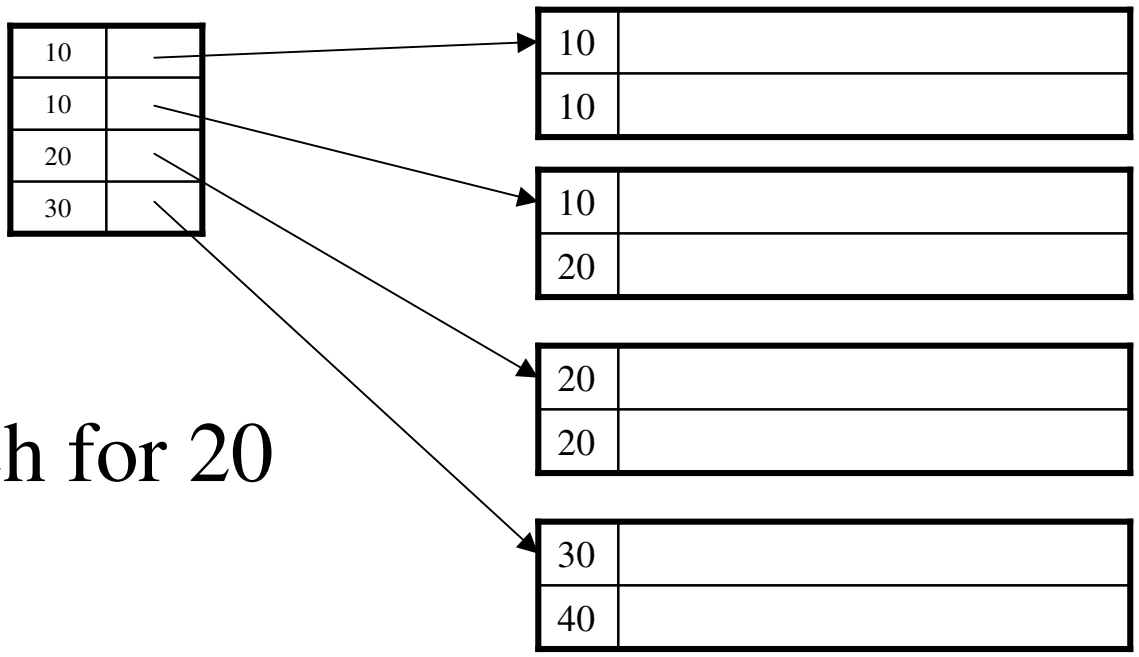
Clustered Index with Duplicate Keys

- Dense index:



Clustered Index with Duplicate Keys

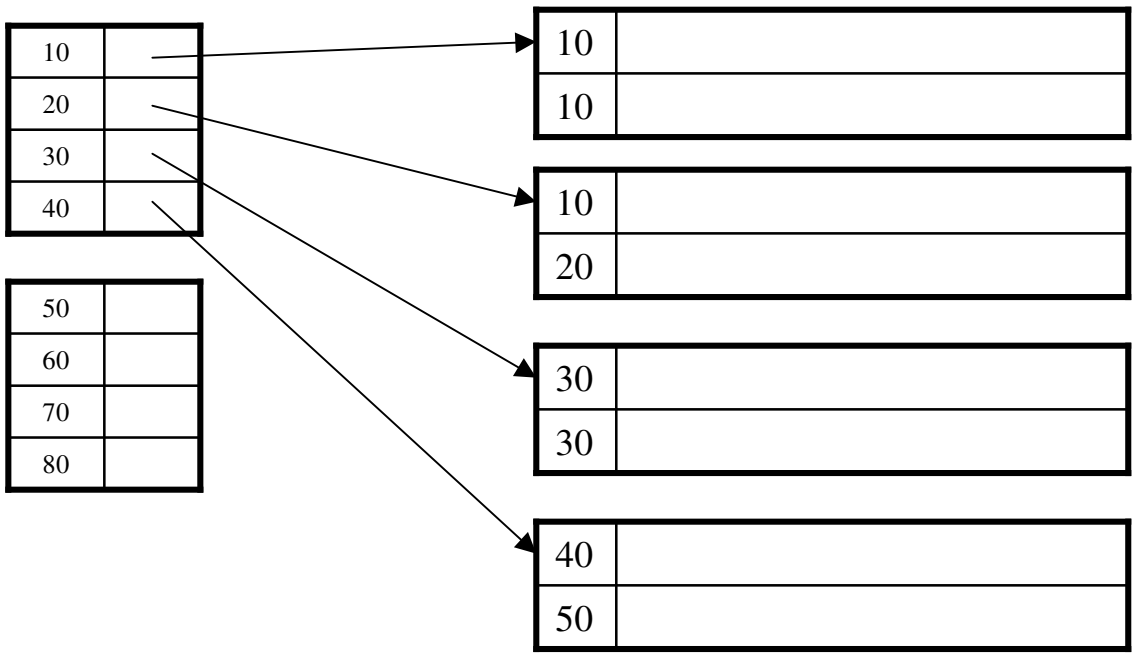
- Sparse index: pointer to lowest search key in each block:



- Search for 20

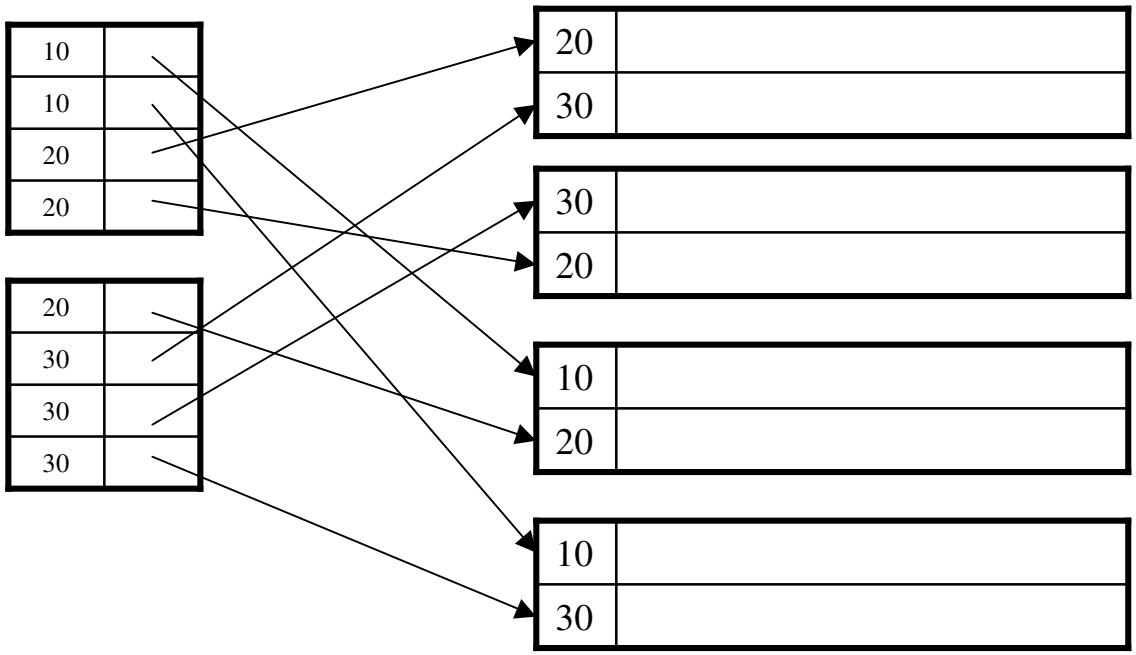
Clustered Index with Duplicate Keys

- Better: pointer to lowest new search key in each block:
- Search for 20

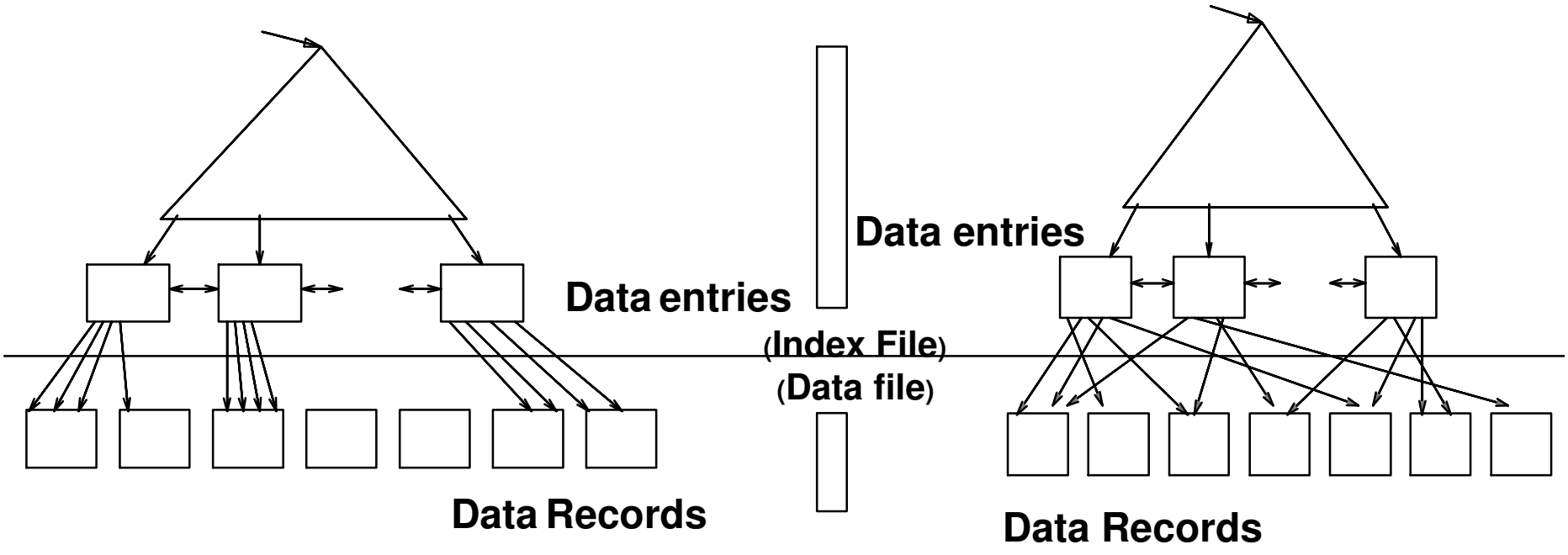


Unclustered Indexes

- To index other attributes than primary key
- Always dense (why ?)



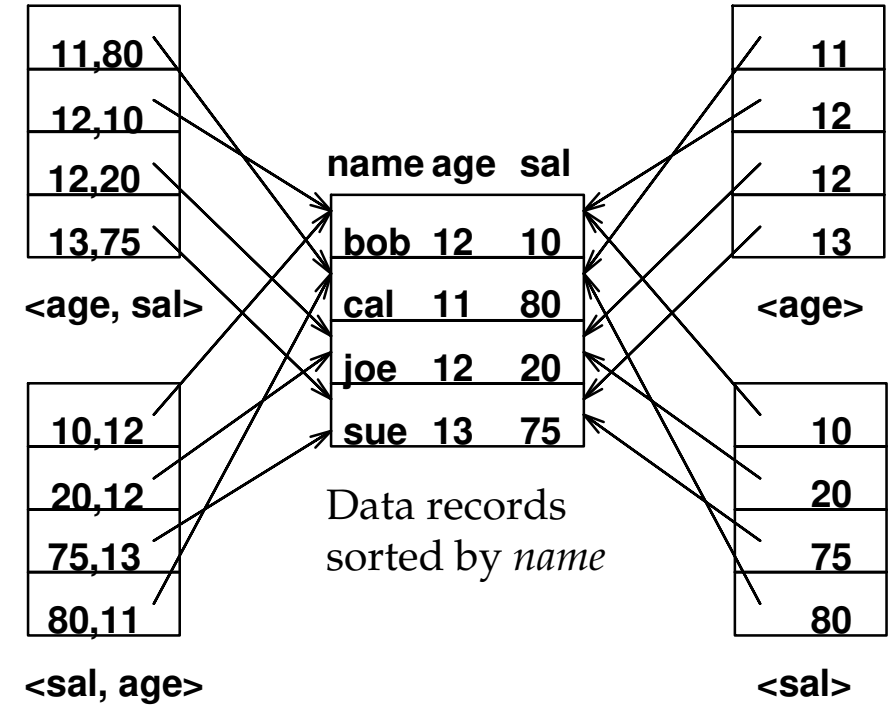
Summary Clustered vs. Unclustered Index



Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=20 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =20; or age=20 and sal > 10

Examples of composite key indexes using lexicographic order.



Data entries in index sorted by <*sal,age*>

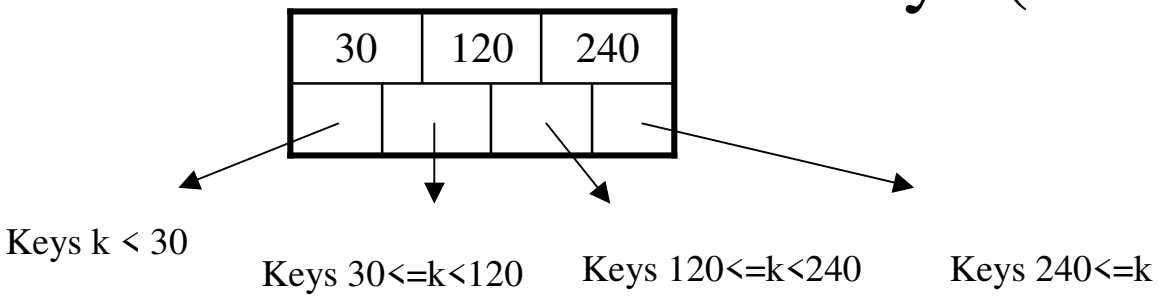
Data entries sorted by <*sal*>

B+ Trees

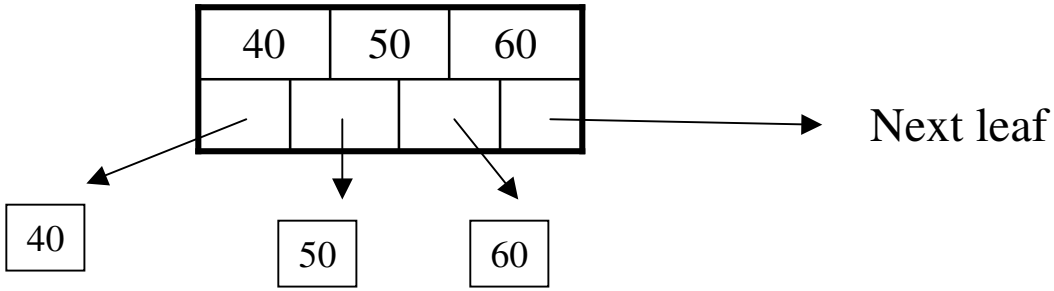
- Search trees
- Idea in B Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - Make leaves into a linked list (range queries are easier)

B+ Trees Basics

- Parameter d = the *degree*
- Each node has >= d and <= 2d keys (except root)

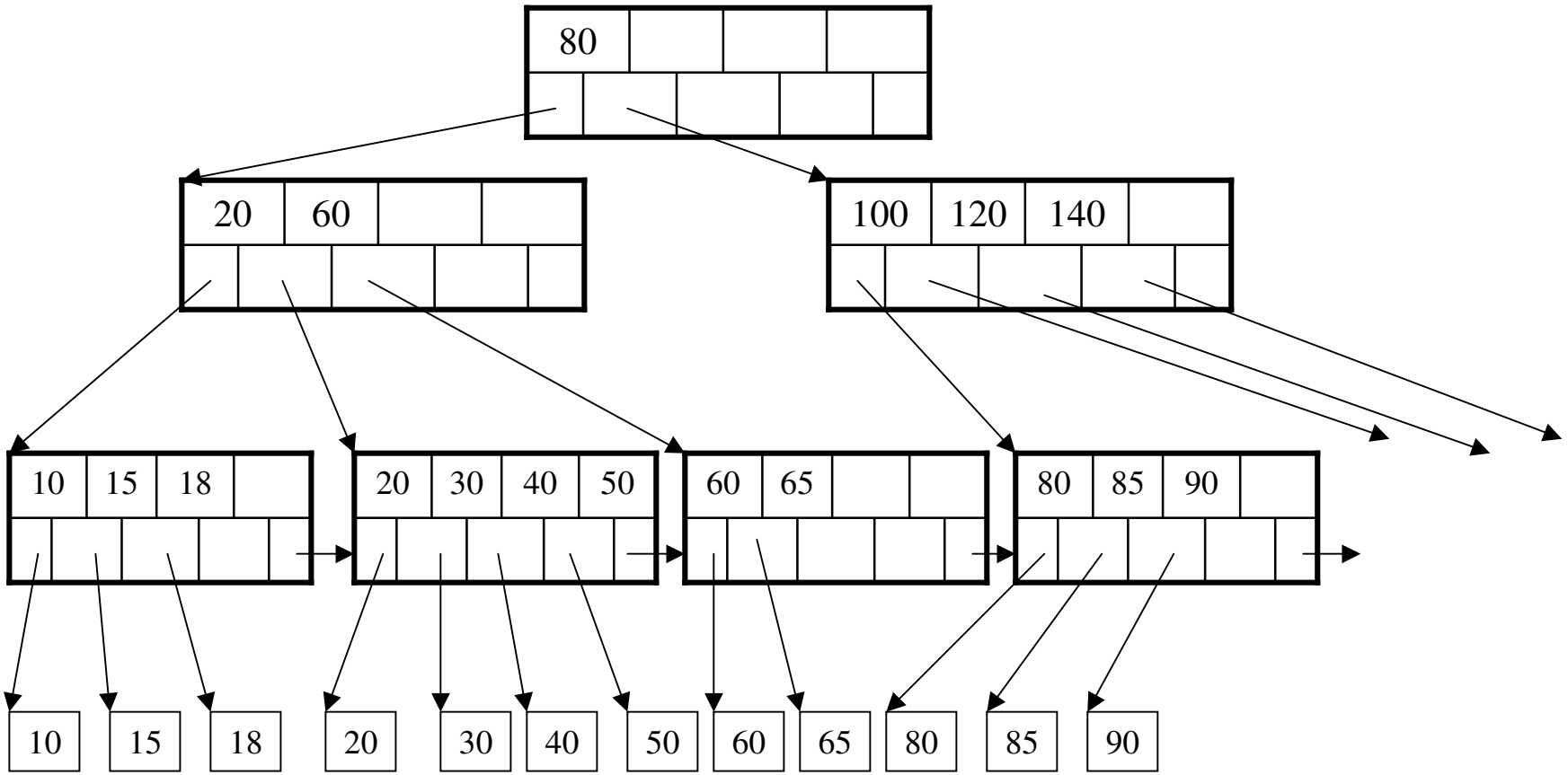


- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2



B+ Tree Design

- How large d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 byes
- 2d x 4 + (2d+1) x 8 <= 4096
- d = 170

Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
Select name
From people
Where age = 25
```

- Range queries:
  - As above
  - Then sequential traversal

```
Select name
From people
Where 20 <= age
  and  age <= 30
```

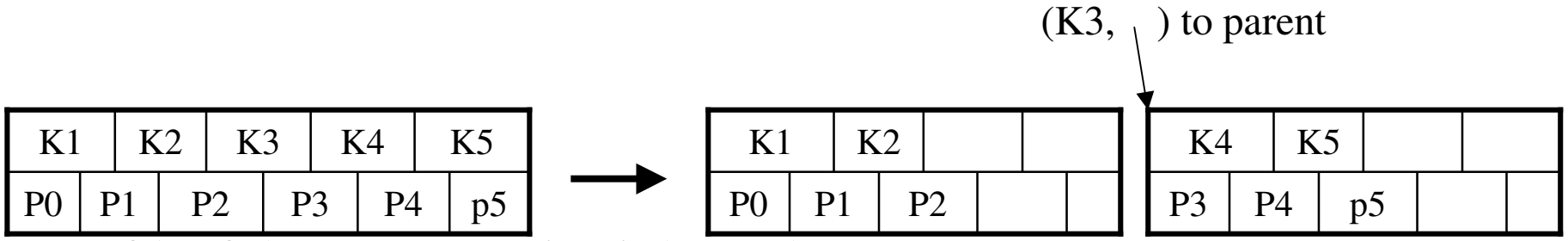
B+ Trees in Practice

- 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

Insertion in a B+ Tree

Insert (K, P)

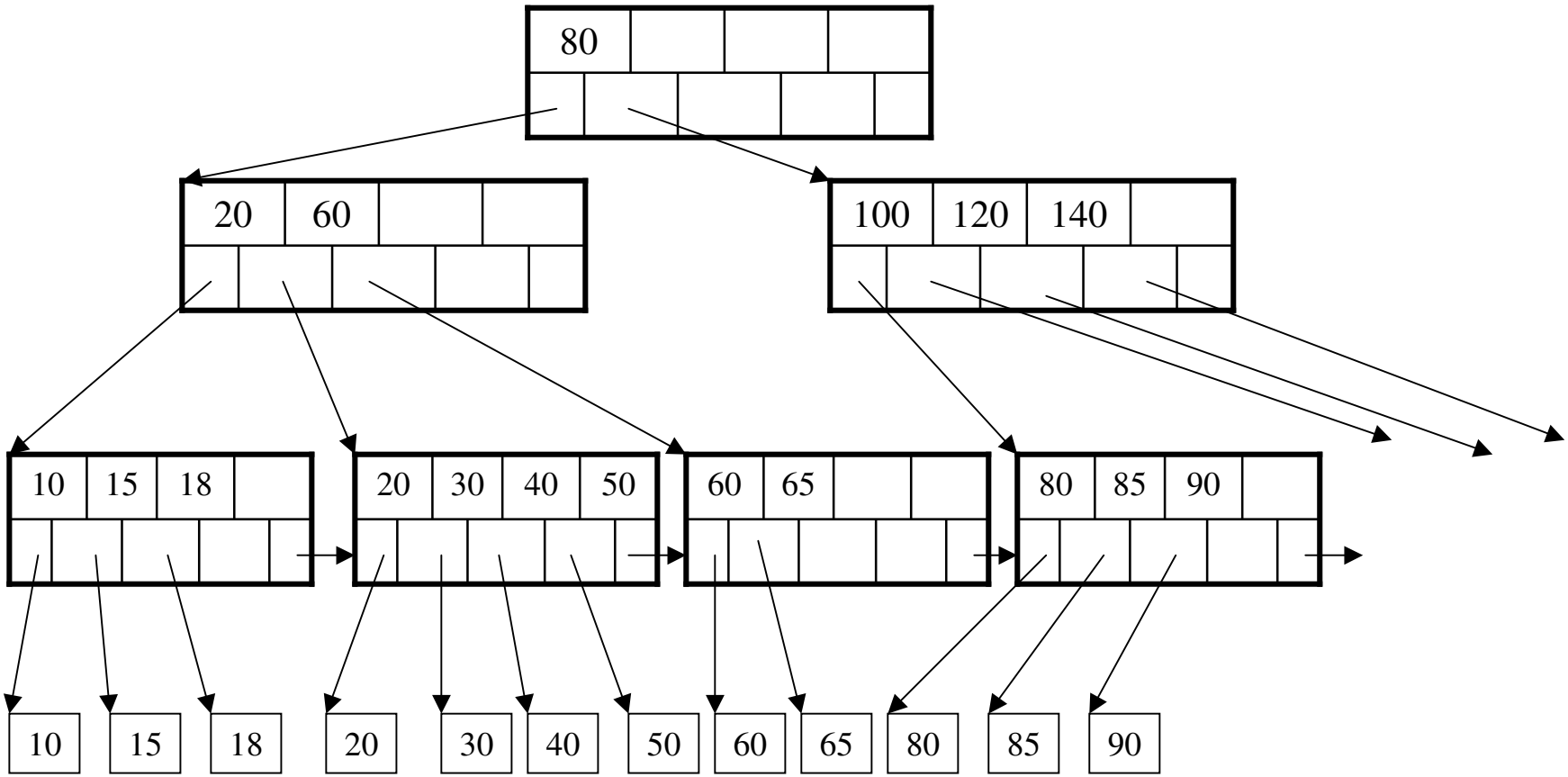
- Find leaf where K belongs, insert
- If no overflow (2d keys or less), halt
- If overflow (2d+1 keys), split node, insert in parent:



- If leaf, keep K3 too in right node
- When root splits, new root has 1 key only

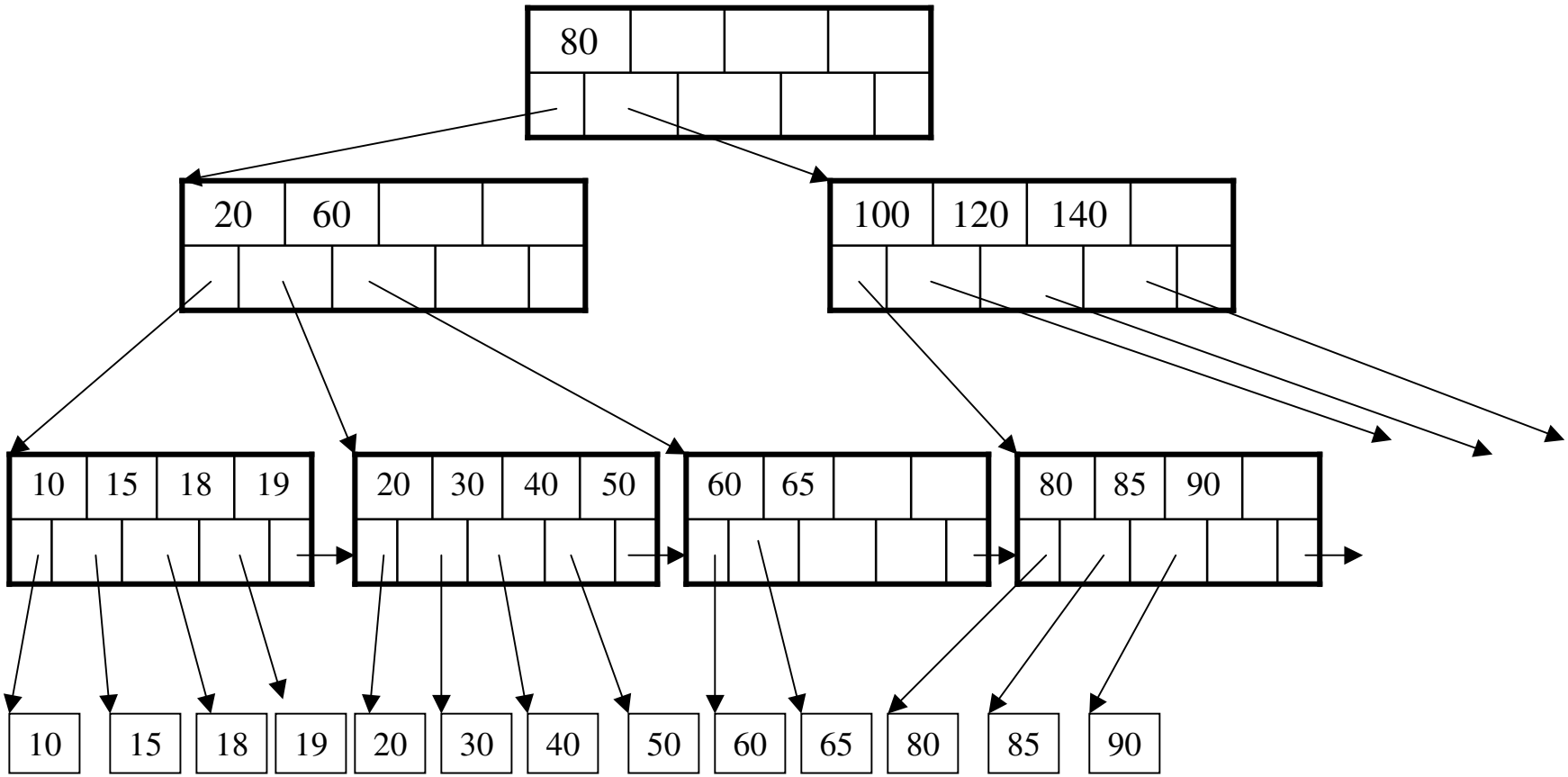
Insertion in a B+ Tree

Insert K=19



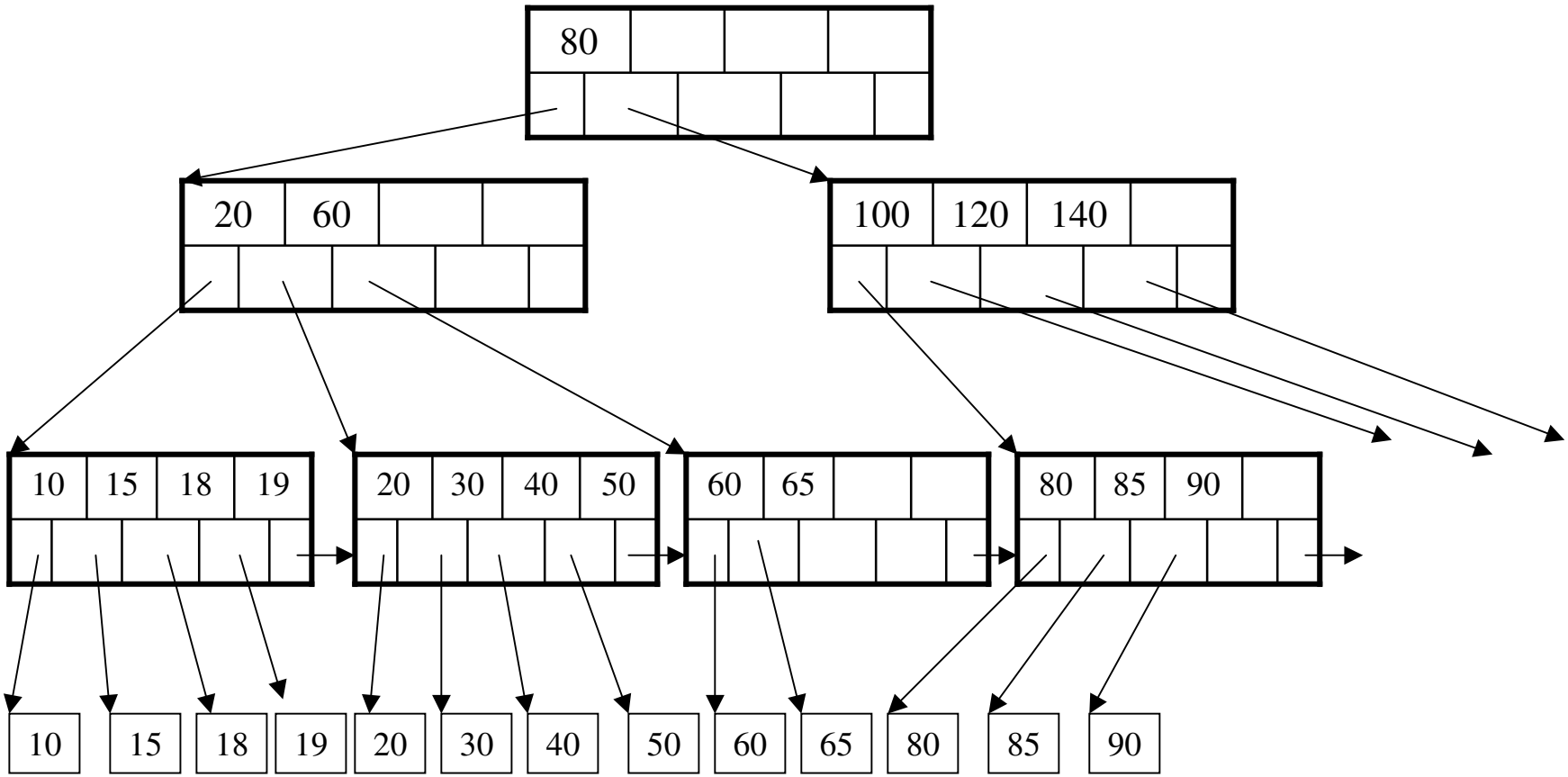
Insertion in a B+ Tree

After insertion



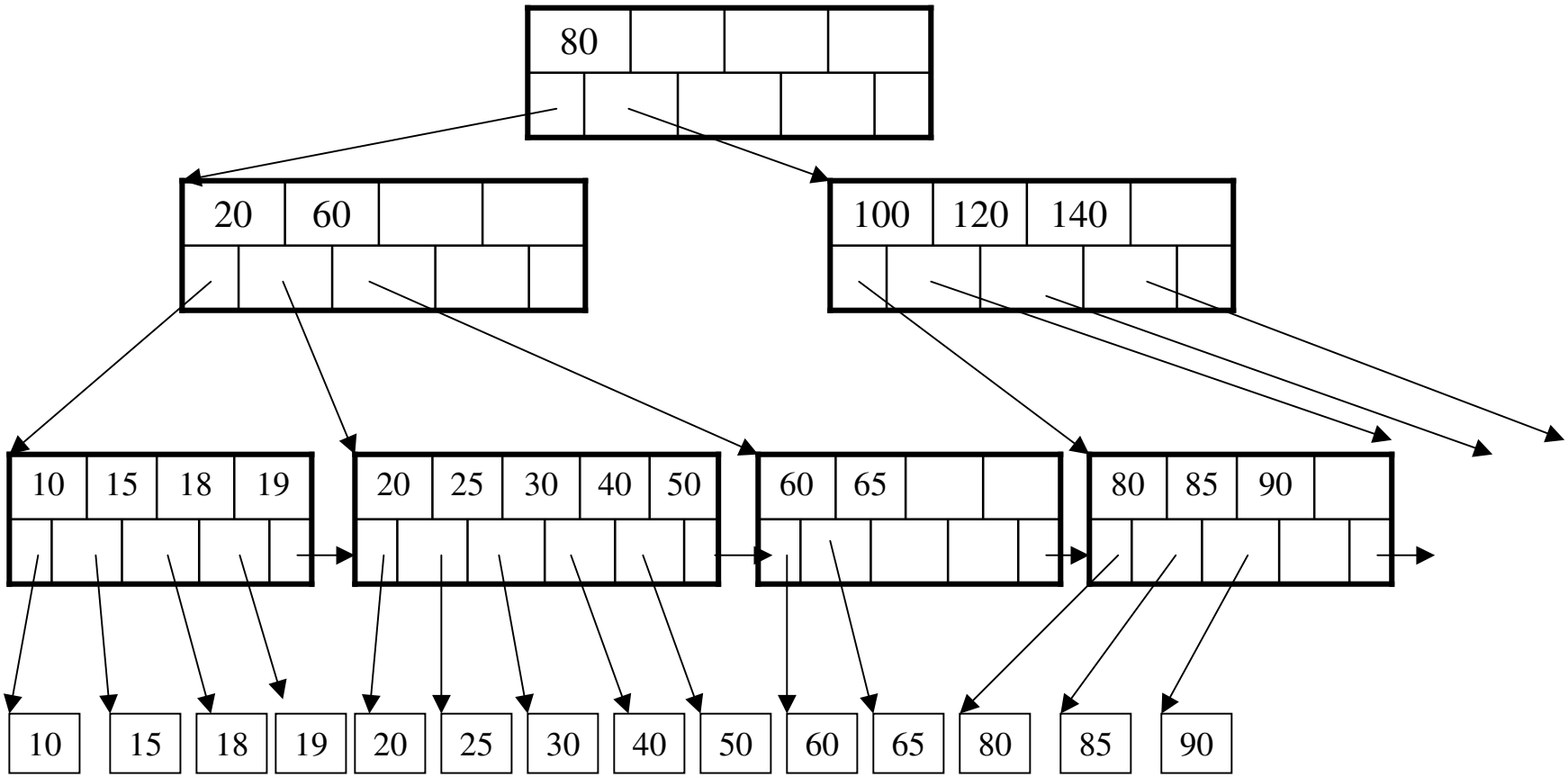
Insertion in a B+ Tree

Now insert 25



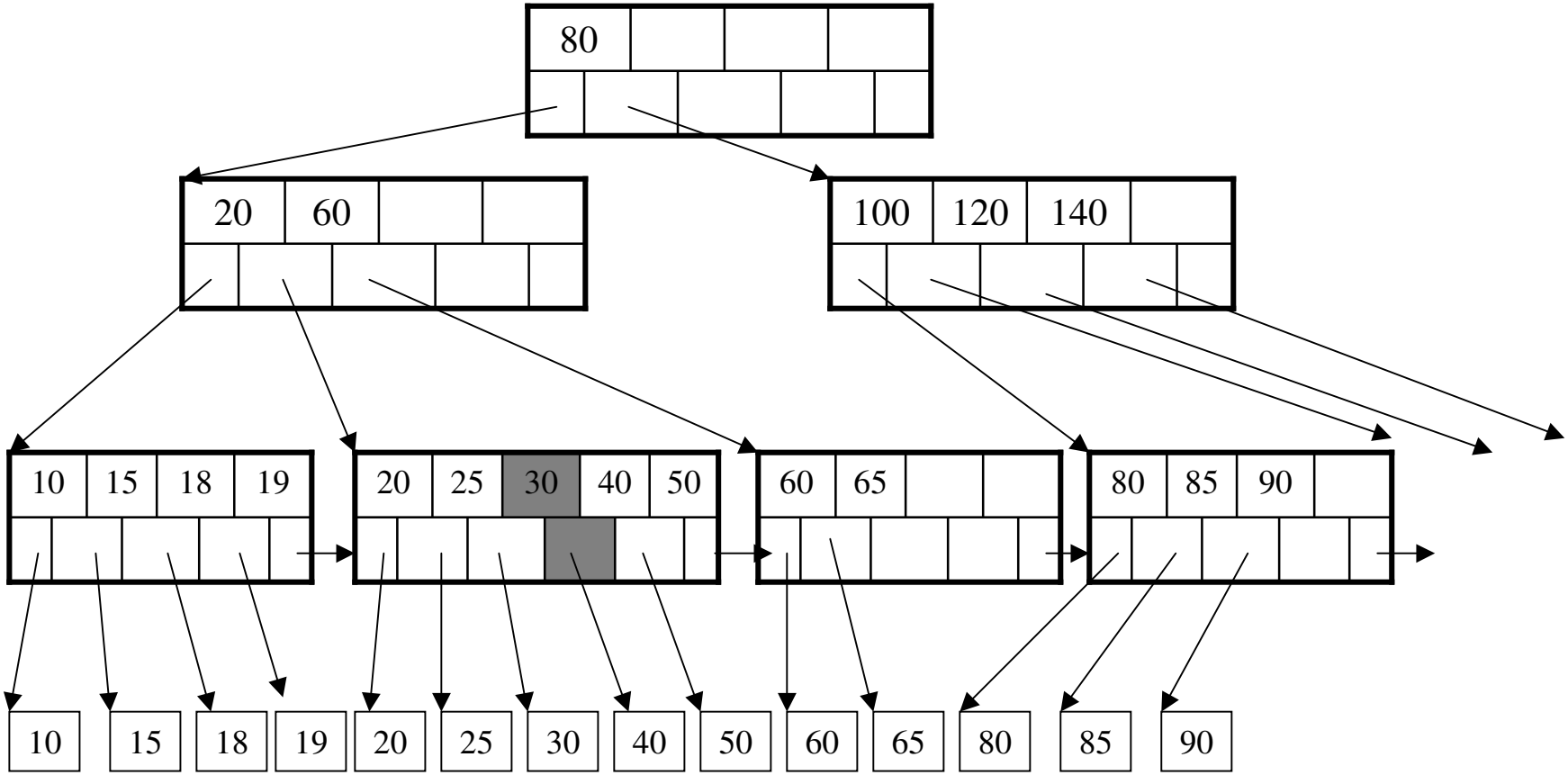
Insertion in a B+ Tree

After insertion



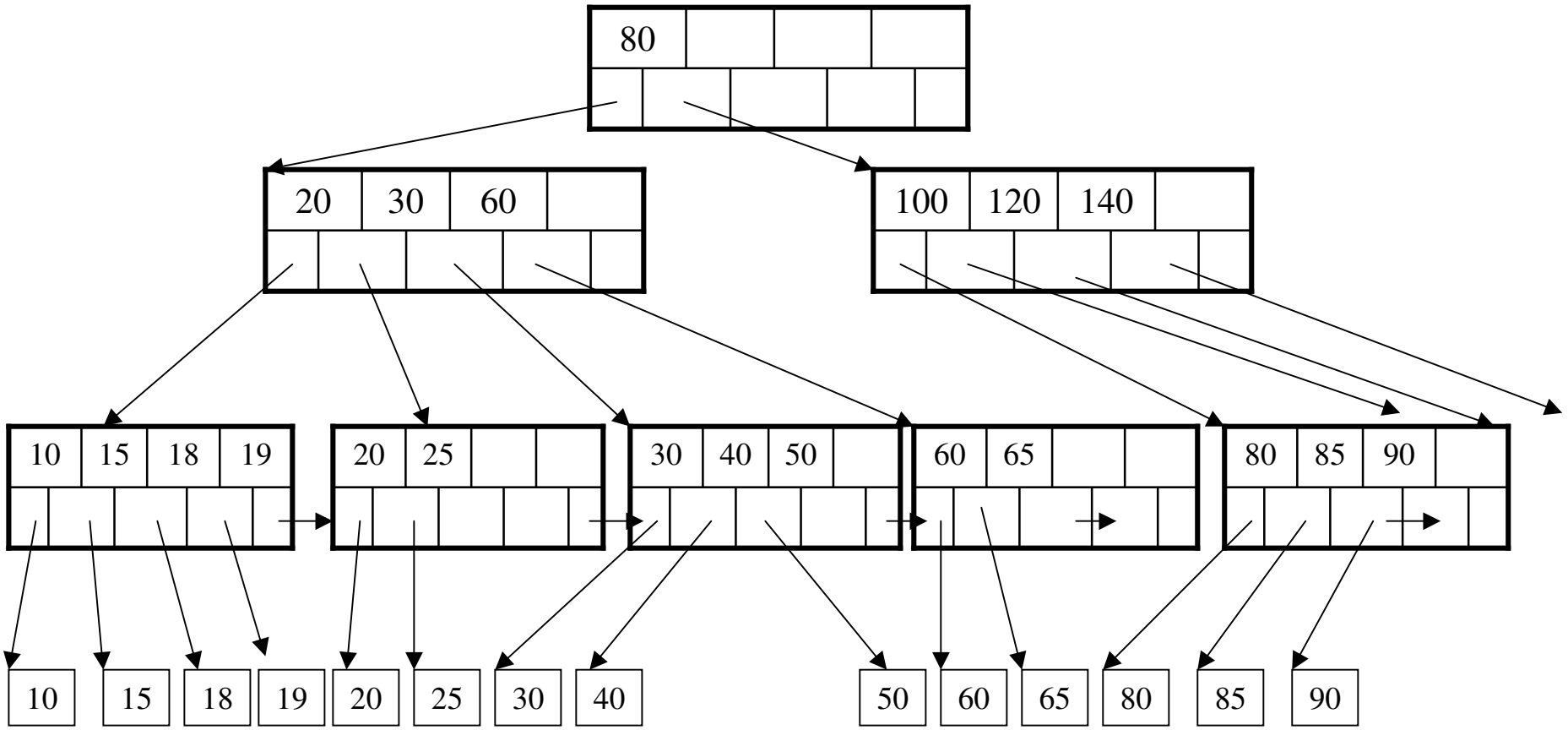
Insertion in a B+ Tree

But now have to split !



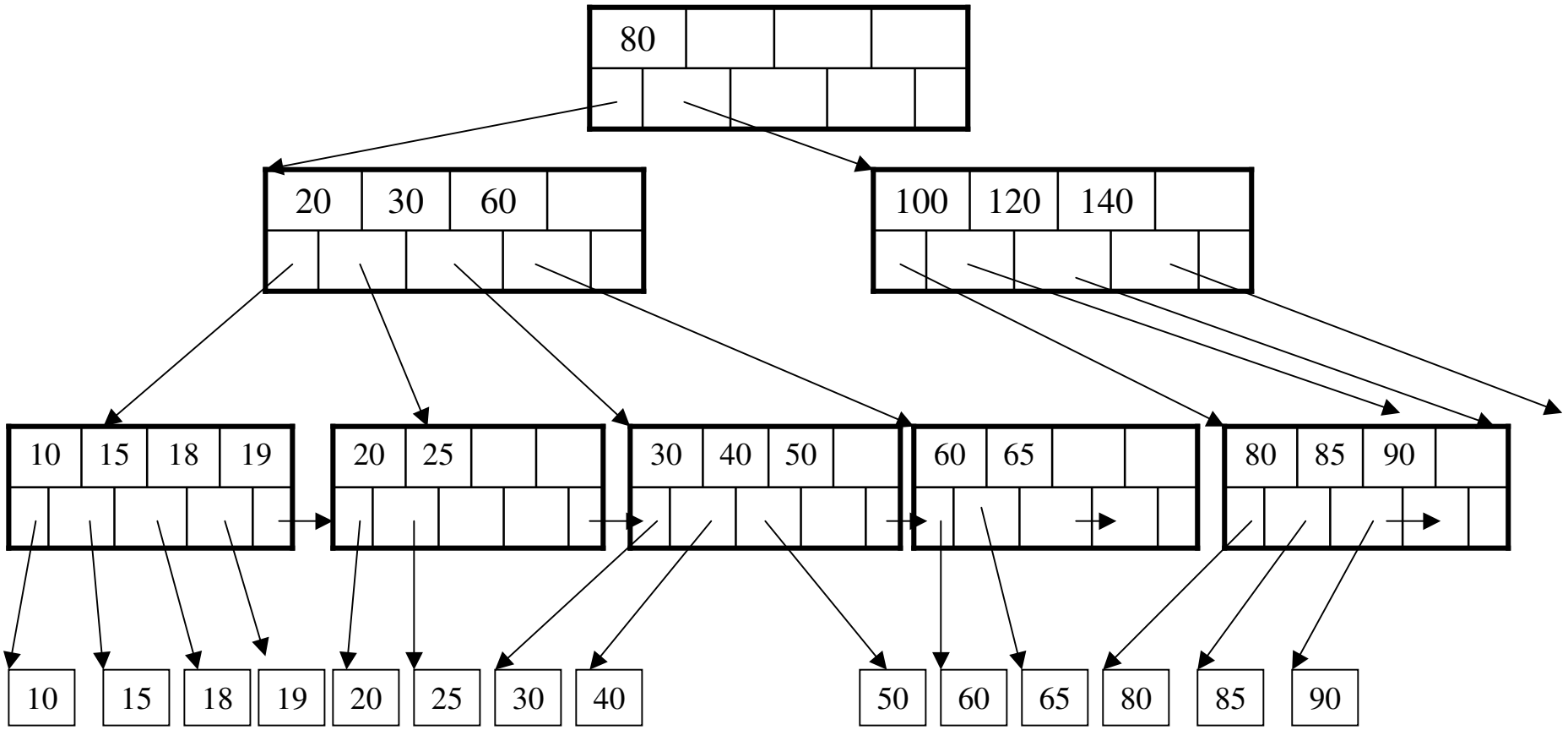
Insertion in a B+ Tree

After the split



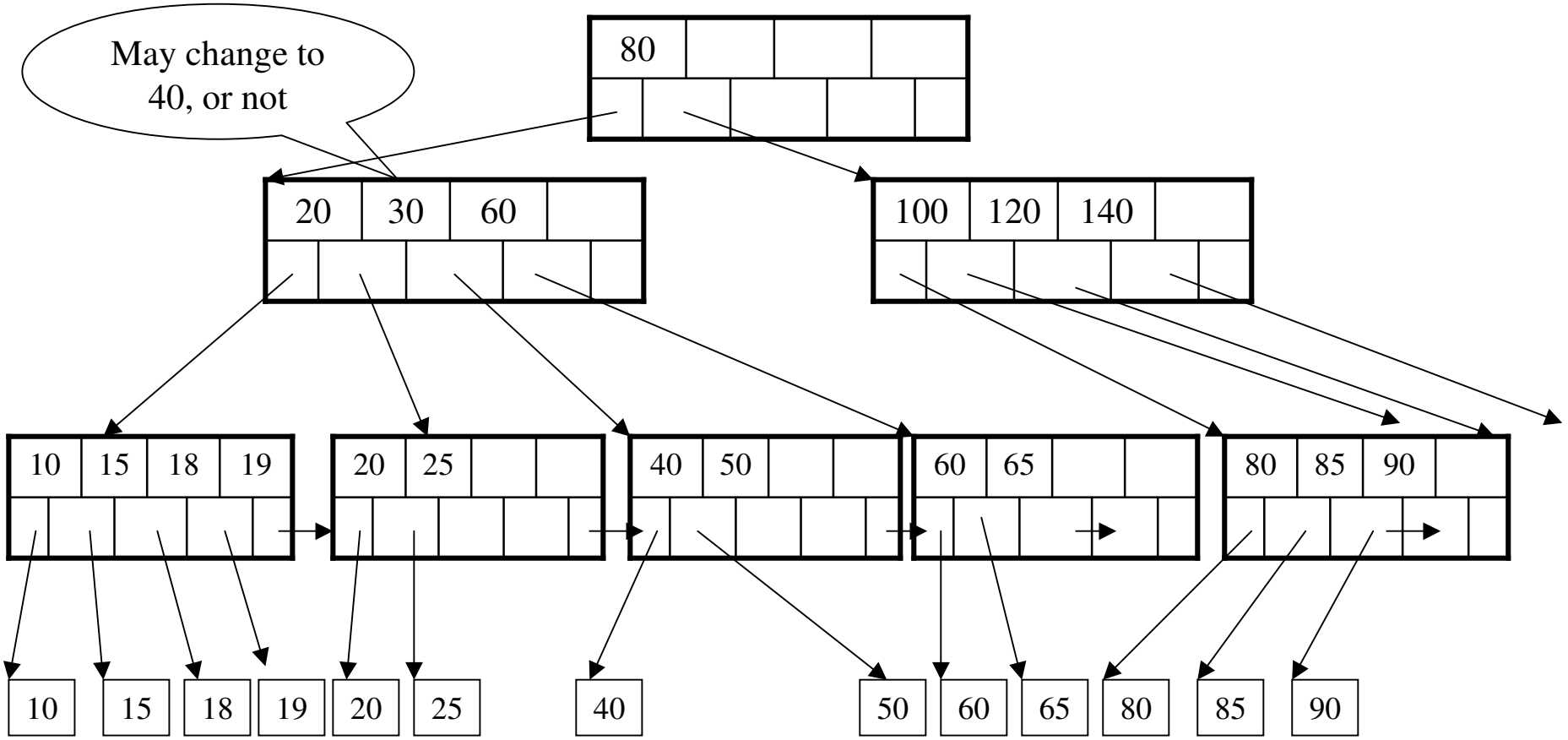
Deletion from a B+ Tree

Delete 30



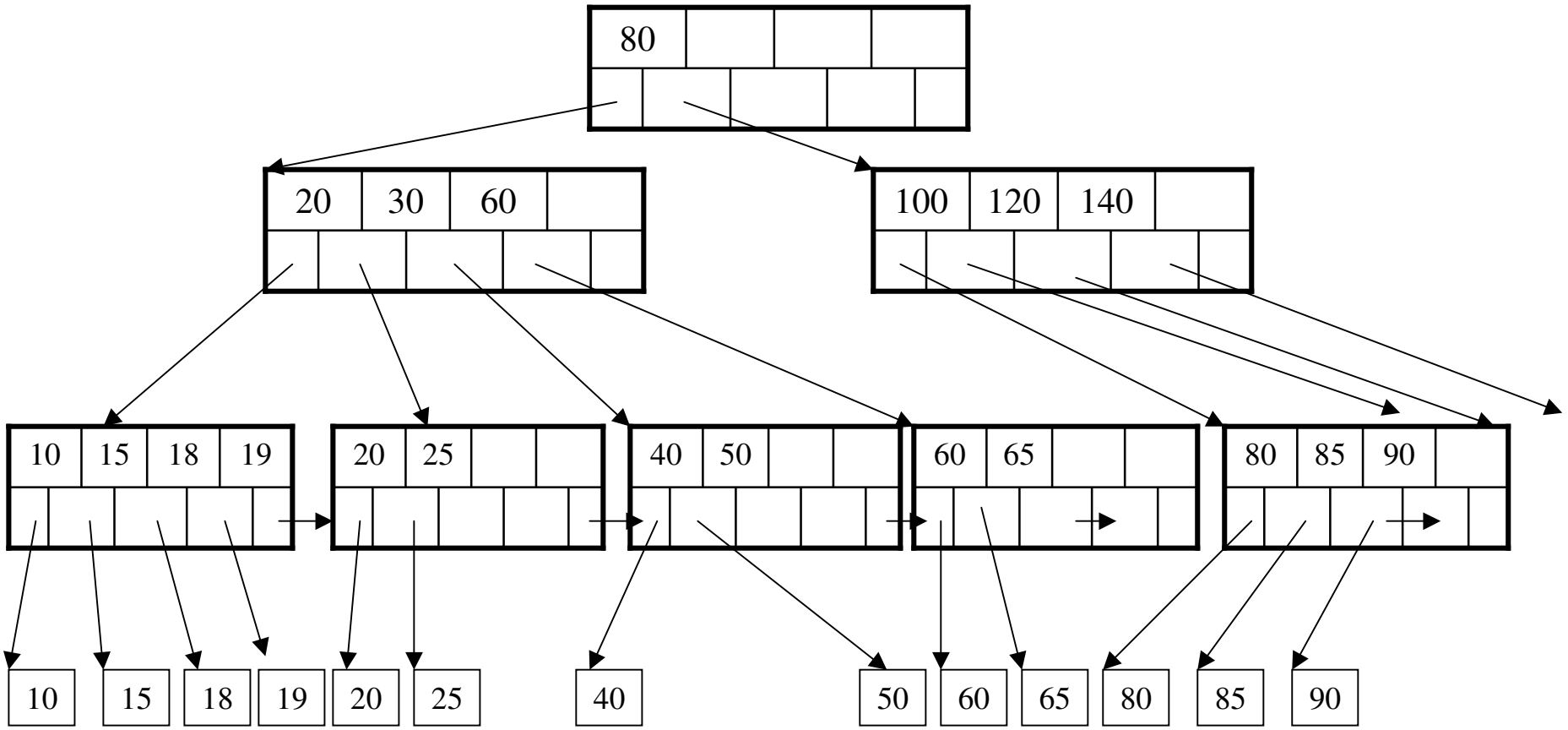
Deletion from a B+ Tree

After deleting 30



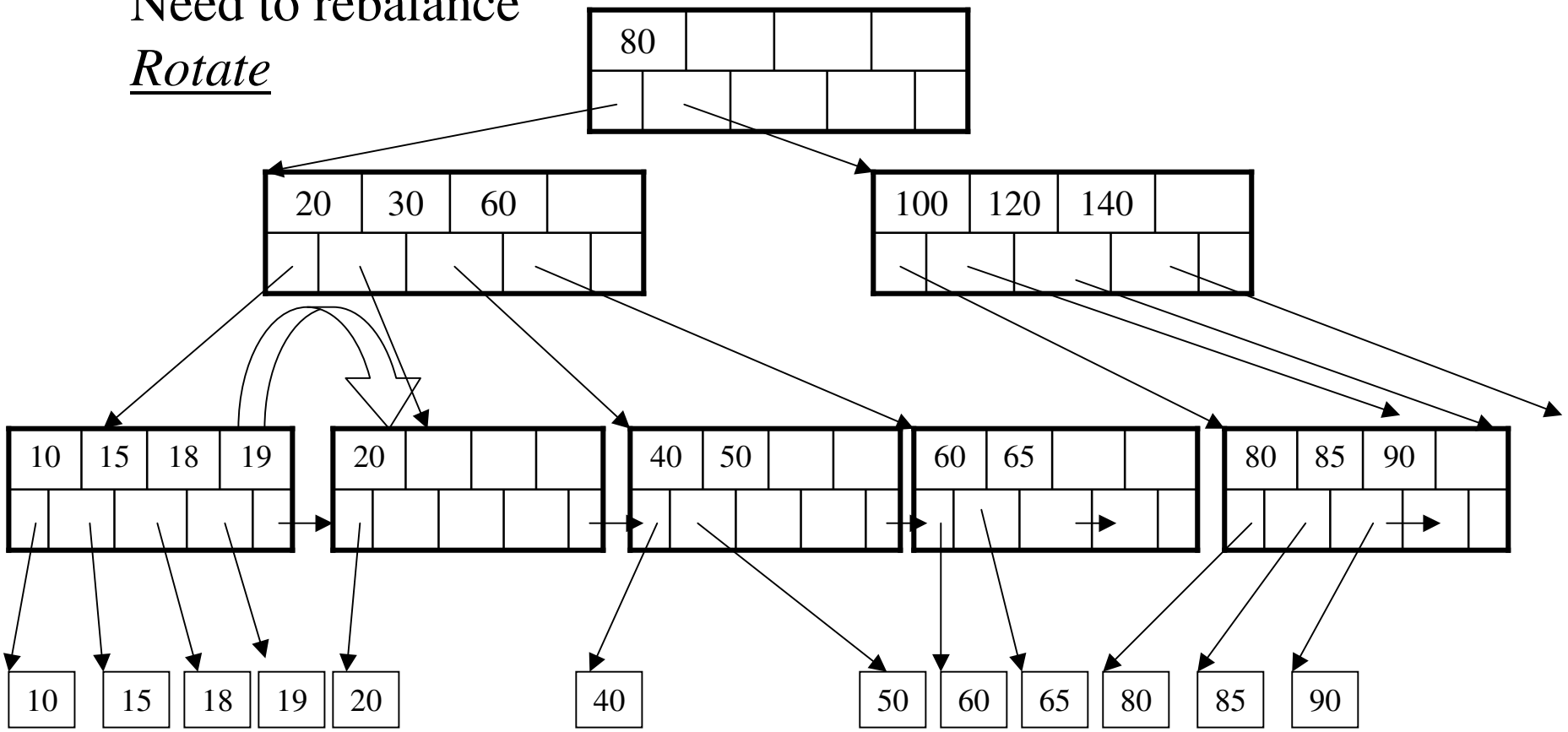
Deletion from a B+ Tree

Now delete 25



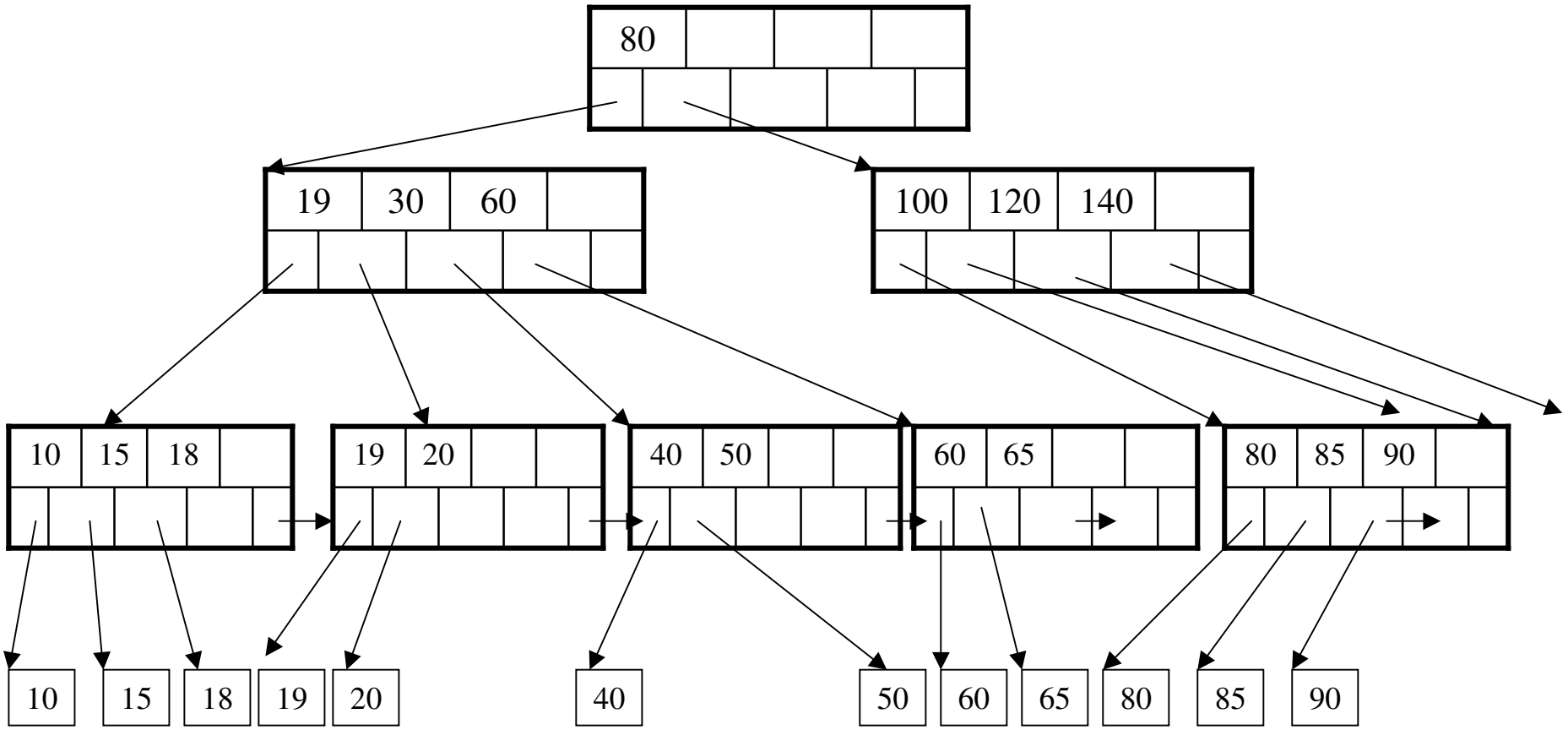
Deletion from a B+ Tree

After deleting 25
Need to rebalance
Rotate



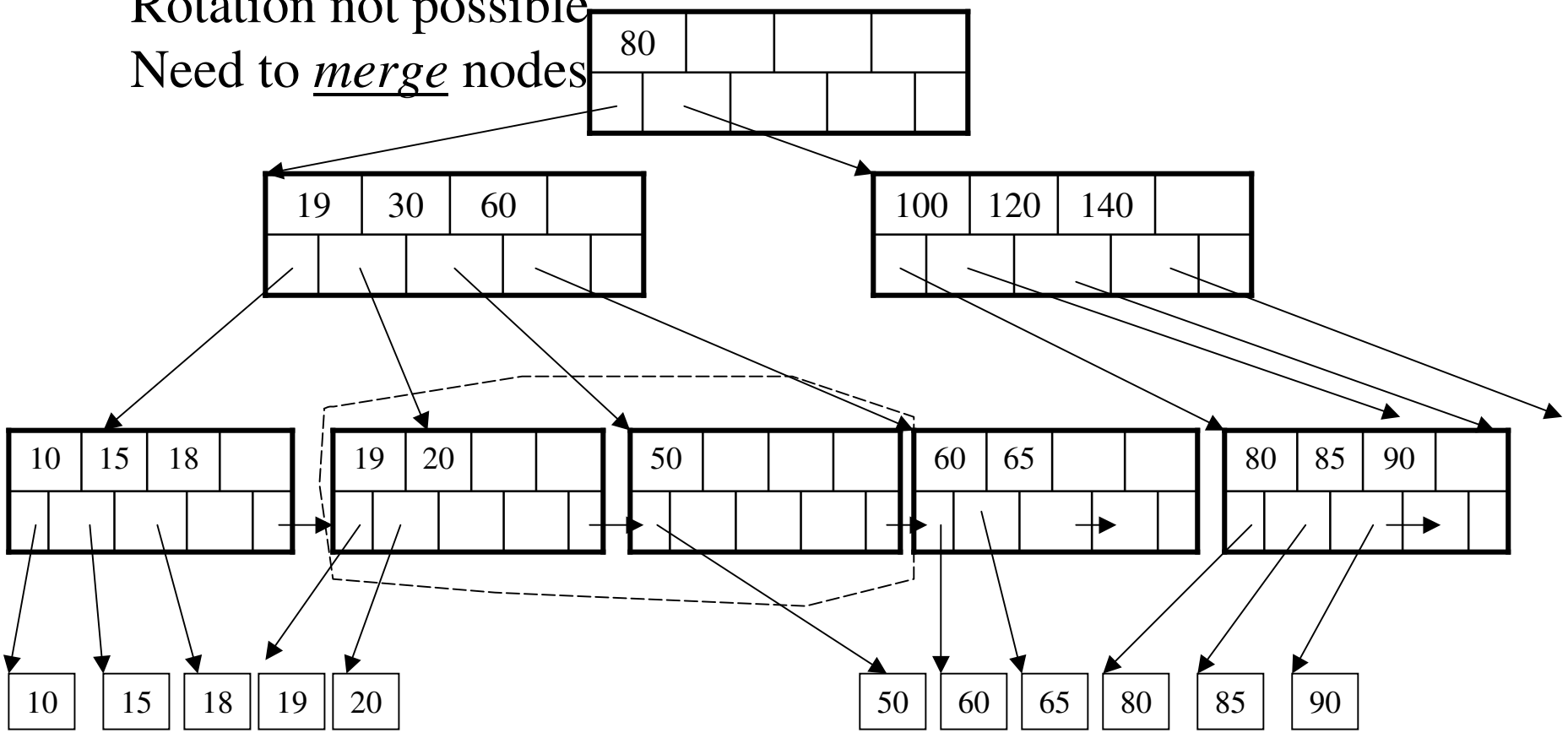
Deletion from a B+ Tree

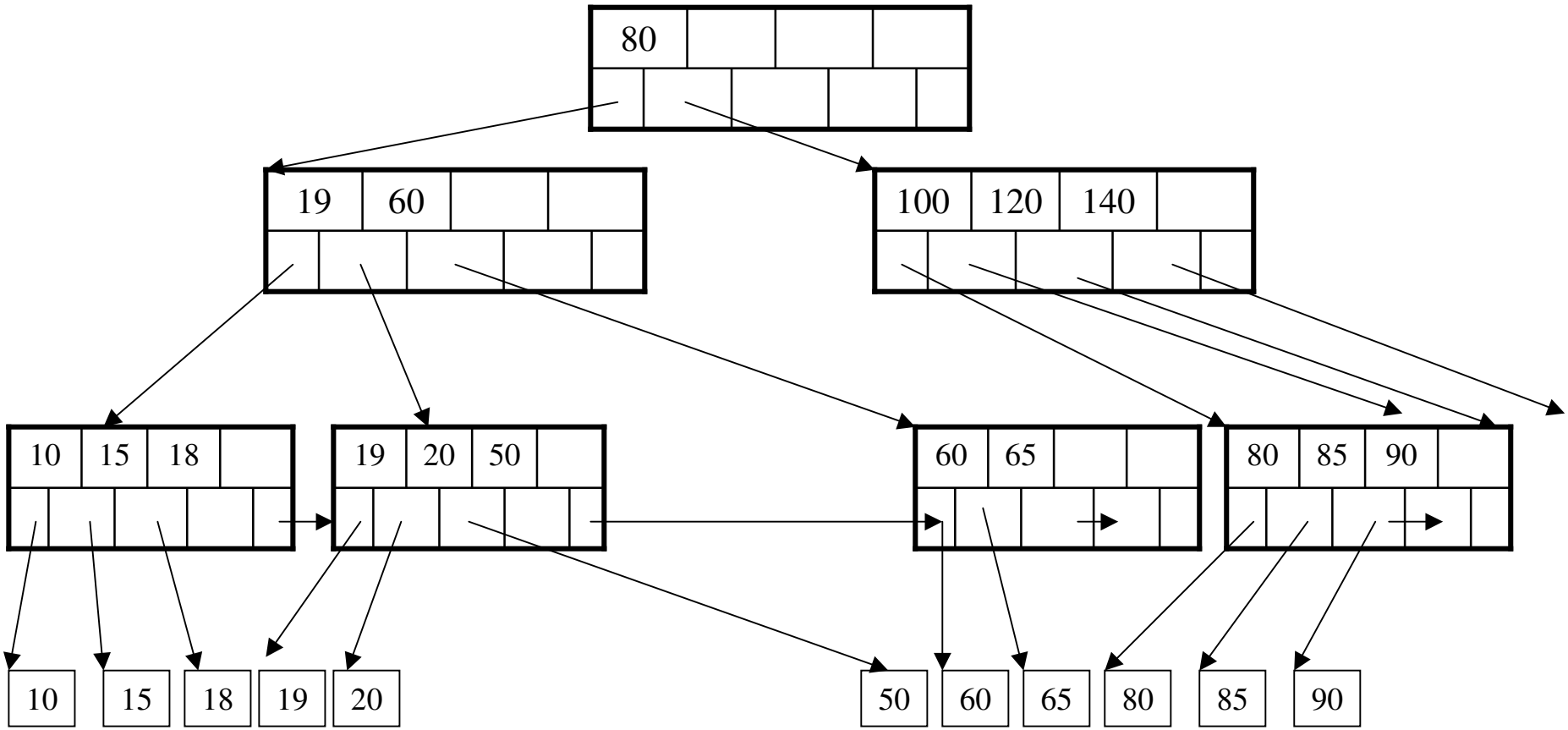
Now delete 40



Deletion from a B+ Tree

After deleting 40
Rotation not possible
Need to *merge* nodes



Deletion from a B+ Tree

Final tree